AFFINITY IN DISTRIBUTED SYSTEMS

A Dissertation

Presented to the Faculty of the Graduate School

of Cornell University

in Partial Fulfillment of the Requirements for the Degree of

Doctor of Philosophy

by

Ymir Vigfusson

February 2010

© 2010 Ymir Vigfusson
ALL RIGHTS RESERVED

AFFINITY IN DISTRIBUTED SYSTEMS

Ymir Vigfusson, Ph.D.

Cornell University 2010

In this dissertation we address shortcomings of two important group communication layers, IP Multicast and gossip based message dissemination, both of which have scalability issues when the number of groups grows.

We propose a transparent and backward-compatible layer called *Dr. Multicast* to allow data center administrators to enable IPMC for large numbers of groups without causing stability issues. Dr. Multicast optimizes IPMC resources by grouping together similar groups in terms of membership to minimize redundant transmissions as well as cost of filtering unwanted messages.

We then argue that when nodes belong to multiple groups, gossip based communication loses its appealing property of using fixed amount of bandwidth. We propose a platform called **GO** (for Gossip Objects) that bounds the node's bandwidth use to a customizable limit, prohibiting applications from joining groups that would cause the limit to be exceeded.

Both systems incorporate optimizations that are based on group similarity or *affinity*. We explore group affinity in real data-sets from social networks and a trace from an industrial setting. We present new models to characterize overlaps between groups, and discuss our results in the context of Dr. Multicast and **GO**.

The chapters on Dr. Multicast and **GO** are self-contained, extended versions of papers that appeared respectively in the ACM Hot Topics in Networks (HotNets) Workshop 2008 [85] and the International Peer-to-Peer (P2P) Conference 2009 [87].

BIOGRAPHICAL SKETCH

Ymir Vigfusson became absent from outdoors games with other kids on the block at early age due to "coding projects". He did a research internship on software vulnerabilities with an Internet security company in Paris in Summer 2001, and a year later he enrolled in Mathematics with Computer Science as a minor at the University of Iceland. He worked for Prof. Magnus Halldorsson at the Icelandic Genomics Corporation in Summer 2004 to design and implement software to estimate binding sites for short primers in-silico. He then worked with Halldorsson on approximation algorithms for sum coloring in interval graphs during the following year.

After completing his B.Sc. degree in Spring 2005, Ymir joined the Computer Science department at Cornell in the Fall to pursue theory driven by systems, later switching to systems driven by theory. He began working with Prof. Ken Birman in 2007, and spent the summer of 2008 working on distributed databases at Yahoo! Research. His focus for the following year was on the three pieces of research presented here, which Ymir defended as his Ph.D. dissertation in August 2009.

To my sisters, Embla and Röskva.

Í minningu Bjargar Svavarsdóttur.

ACKNOWLEDGEMENTS

I would first and foremost like to thank my co-authors of the work presented here for the many hours of stimulating, productive and fun conversations. These people are (in alphabetical order) Hussam Abu-Libdeh, Mahesh Balakrishnan, Ken Birman, Gregory Chockler, Qi Huang, Jure Leskovec, Deepak Nataraj and Yoav Tock. I thank Danny Dolev, Krzys Ostrowski, Anne-Marie Kermarrec, Davide Frey and Martin Bertier for their contributions at an earlier stage of the **GO** project [38].

Furthermore, I want to acknowledge Hitesh Ballani, Idit Keidar, Jon Kleinberg, Tudor Marian, Robbert van Renesse and Alexey Roytman for valuable input and assistance, and Mike Spreitzer for producing the WEBSPHERE trace. I feel privileged to have had a chance to work with these great people, and look forward to continue cooperating with them in the future. The same is true for all the other people I worked with on projects that are not included in this dissertation.

I am deeply thankful to my advisor, Ken Birman. I truly benefited from his well of wisdom, and I leave Cornell twice as strong as when I entered, in large part owing to his undying enthusiasm and energy. I am also grateful to the rest of the committee, Jon Kleinberg, Danny Dolev and Steven Stucky, for their advice and encouragement in all things computational and musical.

My friends in Ithaca made my stay at Cornell in all aspects more exciting, engaging and enjoyable. Whether it was for world travel, gorge jumping, dinner parties, movie nights, board games, dancing or deep philosophical conversations, I was always honored to have their company. There are too many of you to enumerate, but I trust you know who you are and that you forgive my laziness.

Since I am half-way into an Academy Awards speech here anyway, I also want to thank my family for their endless source of support, encouragement and smiles in times both sweet and sour. I truly appreciate each and every one of you.

Last, but certainly not least, I want to thank my lovely Becky for the wonderful years we have had together. Her astonishing energy and drive is contagious, and without her I might still be aimlessly awaiting directions from my future self. (I also might be ignoring invisible stop signs, wearing sneakers with a dress suit and inhaling spaghetti in job interviews.) Thank you for what you have done for me.

TABLE OF CONTENTS

Biographical Sketch iii
Dedication iv
Acknowledgements v
Table of Contents vii
List of Tables ix
List of Figures x

1 Introduction **1**
1.1 Modern Networking 1
1.2 Distributed Systems 3
1.3 Group Communication 4
1.4 Contributions 6

2 Dr. Multicast **9**
2.1 Introduction 9
2.2 Motivation 12
2.2.1 Multicast Scalability 13
2.2.2 Multicast Stability 16
2.3 Acceptable-Use Policy 17
2.3.1 Policy Primitives 17
2.3.2 Higher-Level Policy 20
2.3.3 Policy Examples 21
2.4 Design and Implementation 22
2.4.1 Library Module 23
2.4.2 The MCMD Agent 24
2.5 Theoretical Considerations 29
2.6 Experimental Results 31
2.6.1 Implementation 31
2.6.2 Real-World Application 37
2.7 Related Work 40
2.7.1 Stability and Security 40
2.7.2 Scalability 41
2.8 Conclusion 42

3 Gossip Objects **43**
3.1 Introduction 43
3.2 Gossip Algorithms 49
3.2.1 Model 49
3.2.2 Random Dissemination 50
3.2.3 Optimized Dissemination 50
3.2.4 Traffic Rates and Memory Use 58
3.3 Platform Implementation 59

3.3.1 Bootstrapping 60
3.3.2 Gossip Mechanism 60
3.3.3 Membership Component 61
3.3.4 Rumor Queue 62
3.4 Evaluation 63
3.4.1 Rumor Stacking and Message Indirection 63
3.4.2 Real-World Scenarios 67
3.4.3 Discussion 69
3.5 Future Directions 70
3.6 Related Work 72
3.7 Conclusion 72

4 Affinity **74**
4.1 Social Data Sets 76
4.1.1 Yahoo! Groups 77
4.1.2 Wikipedia Editors 78
4.1.3 Amazon.com Recommendations 79
4.1.4 LiveJournal Communities 79
4.2 Modeling Social Interactions 80
4.2.1 Power-law Distributions 81
4.2.2 Mutual Interest Model 82
4.3 Systems Data Set 84
4.3.1 WebSphere Bulletin Boards 84
4.4 Modeling Systems Communication Channels 87
4.4.1 Hierarchical Components 88
4.5 Analysis 90
4.5.1 Visualizing Affinity 91
4.5.2 Baseline Overlap 96
4.6 Dr. Multicast 102
4.6.1 Formal Model 103
4.6.2 The MCMD Heuristic 106
4.6.3 Evaluation on WEBSPHERE 107
4.6.4 Evaluation on other graphs 111
4.7 Related Work 112
4.8 Conclusion 115

5 Conclusion **116**

Bibliography **120**

LIST OF TABLES

4.1 Statistics for the bipartite graphs of the data sets and models. . . 92
4.2 Value of Δ averaged over all cells of each color plot. 102

LIST OF FIGURES

2.1 Receiver NIC Scalability: Probability of false positives in NIC imperfect hash filter vs. number of addresses. 12
2.2 Receiver NIC Scalability: Packet loss rate vs. number of IPMC groups joined. 13
2.3 Two under-the-hood mappings in MCMD, a direct IPMC mapping (left) and point-to-point mapping (right). 18
2.4 Overview of the MCMD architecture. 22
2.5 Application Overhead: Maximum throughput for a sender using regular IPMC and MCMD with direct IPMC mapping, and MCMD unicast to 5 or 10 receivers per group. 32
2.6 Application Overhead: Average CPU utilization for the sender application with and without MCMD. 33
2.7 Application Overhead: The overhead at a receiver caused by raw IPMC and collapsed IPMC usage by MCMD. The overhead spike in raw IPMC is associated with the packet loss shown in figure 2.2. 33
2.8 Network overhead: Traffic due to MCMD with and without an urgent notification channel. 35
2.9 Latency: Update latency using regular gossip (epochs) and with the urgent notification channel enabled (ms). 38
2.10 Policy Control: CPU utilization at a normal receiver. A malfunctioning node bombards the group at time 20, and the administrator restricts policy at time 40. 38
2.11 Policy Control: Average throughput between two nodes in separate subnets. At time 20, a node erroneously joins a high-traffic IPMC group in the other subnet, and the administrator corrects the access control policy at time 40. 39
2.12 Real-World Application: CPU utilization of WEBSPHERE Bulletin Board at a regular node vs. per-node send rate x with and without MCMD support with real subscription patterns. 39

3.1 The **GO** Platform. 57
3.2 Membership information maintained by **GO** nodes. The topology of the whole system on the left is modeled by the node in center as (i) the set of groups to which it belongs and neighbor membership information (local state), and (ii) the overlap graph for other groups, whose nodes are depicted as squares and edges are represented by thick lines (remote state). 61

3.3 Rumor Stacking and Indirection. Different heuristics running on the **GO** platform over the topology from figure 3.4. The plots show the number of new rumors received by nodes in the system over time (a) and as a function of messages sent (b). A vertical line is drawn at the time when all 2,000 rumors have been generated. 64
3.4 The topology used in first experiment. Each edge corresponds to a gossip group, the members of which are the two endpoints. . . 65
3.5 WebSphere trace: The number of new rumors received by nodes in the system and the number of messages sent (a), also plotted as a ratio of new rumors per message over time (b). The nodes using the random heuristics gossip per-group every round, whereas **GO** sends a single gossip message. 66

4.1 Y-GROUPS: complementary CDF for group size (left) and user degree (right). 77
4.2 WIKIPEDIA: complementary CDF for group size (left) and user degree (right). 78
4.3 AMAZON: complementary CDF for product degree (left) and user degree (right). 79
4.4 LIVEJOURNAL: complementary CDF for community size (left) and user degree (right). 80
4.5 MUTUAL-INTEREST: complementary CDF for group size (left) and user degree (right). 83
4.6 WEBSPHERE: complementary CDF for group size (left) and user degree (right). 85
4.7 WEBSPHERE: Publication (a) and subscription (b) matrices. Each dot represents a subscriber or publisher on a specific group. The traffic rate (bottom) on groups in the trace is expressed both in messages/sec (left) and bytes/sec (right). 86
4.8 WEBSPHERE: Communication patterns. A marker is plotted for each group, in the spatial location representing its number of subscribers and publishers. 86
4.9 HIERARCHY: PDF of component degrees. 90
4.10 Affinity matrices for 1,000 group samples from the WIKIPEDIA and AMAZON data sets. 93
4.11 Affinity matrices for 1,000 group samples from the LIVEJOURNAL social data set and the MUTUAL-INTEREST model. 94
4.12 Affinity matrices for 1,000 group samples from the systems data set WEBSPHERE and HIERARCHY model. 95
4.13 Δ plot for the WIKIPEDIA and AMAZON graphs. 98
4.14 Δ plot for the Y-GROUPS and MUTUAL-INTEREST model graphs. 99
4.15 Δ plot for the WEBSPHERE and HIERARCHY model graphs. . . . 100

4.16 WEBSPHERE: The cost of a single multicast with the MCMD heuristic vs. number of physical IPMC groups. 108
4.17 WEBSPHERE: Trade-off between filtering cost and transmission cost for a single multicast using the MCMD heuristic for a fixed number of physical groups. 109
4.18 Cost of a single multicast using the MCMD heuristic on samples from the data sets and models vs. number of physical groups. . . 110
4.19 Percentage of total cost savings achieved using the MCMD heuristic as a function of the number of physical groups. 111

CHAPTER 1
INTRODUCTION

1.1 Modern Networking

In the past decade we have witnessed a paradigm shift for client-server computing. The greatest impetus for the change has been cost: an expensive but monolithic server can often be replaced by a cheaper but higher performance *data center*, a farm of cheap commodity machines strung together in a fast network. Data centers are well matched to the highly parallel, loosely coupled streams of requests that arise in Internet applications, where vast numbers of clients independently interact with web services such as Google Search [5], Twitter [18], Facebook [3], YouTube [19], interactive multiplayer games [15], and so forth. The requests are not only logically independent and concurrent, but the actions taken to service them are in large part independent and have only loose consistency requirements. One could argue that the affordability of cheap computing power has been as much of a driver as the rapid growth of the network itself in enabling the diverse collection of "online" companies we see today.

Scalability, the ability to accommodate growth of computational requirements by adding cheap hardware, is a consideration that favors a data centers over giant servers. The importance of scalability is evident when one recognizes that computationally intensive yet interactive Internet applications, such as web search, can often precompute high-value data such as indices that will later support queries by hundreds of millions of users.

Scale also brings its own challenges. When a monolithic server crashes, the

system is down until it restarts. But large data centers need to be fault tolerant because, at any point in time, many components will be faulty or in the midst of upgrade. Google, for example, has data centers that comprise tens to hundreds of thousands of low-cost servers. Such a data center will inevitably experience a significant number of failures in a year [20].

Some applications are suited to having clients provide and exchange resources and information in a *peer-to-peer* fashion rather than burdening the servers in the data center. The initial reasons behind avoiding centralization were to evade the law — file exchange sites such as KaZaA [9] hoped to avoid the fate of the centralized Napster music sharing site, which was shut down by court order for promoting copyright infringements [10]. Later, the advantages of decentralized peer-to-peer computing were characterized more carefully, resulting in a surge of research on the topic that has now endured for more than a full decade [74]. An example of a prominent peer-to-peer system is Skype's voice-over-IP telephone service [16], in which the system circumvents delay and dropouts and a potential bandwidth bottleneck at the Skype data center by having users communicate directly with one another. The clients also maintain an overlay network for users currently online, allowing the service to scale dynamically without any additional cost in Skype's data centers.

The current trend is towards an intriguing mixture of data center and peer-to-peer systems called *cloud computing*. Users continuously interact with servers in some data centers (in the "cloud"), computation is done by both the client (the "edge") and the cloud servers and the cloud stores both private and public data (e-mail, documents, web pages, blogs, *etc.*). The data centers in the cloud employ peer-to-peer technology to ensure that critical data is replicated, and to

balance load by distributing it geographically across data centers while optimizing latency [45, 62].

1.2 Distributed Systems

Distributed computing is the paradigm of solving a computation problem in parallel on multiple machines that are connected by a network, for instance in a data center or in a peer-to-peer network, and a system that performs distributed computing is a *distributed system*.

Developing a modern distributed system is a complex and error-prone task. For example, despite careful design and skilled engineering, in one highly publicized event the Skype peer-to-peer overlay fragmented beyond what the recovery procedures could handle, rendering the system unusable for users for approximately two days [31]. Amazon's peer-to-peer storage balancing algorithms, in the S3 platform, malfunctioned in a way that directed all traffic to a single server [79]. And some content delivery overlays, including KaZaA [9], are notorious for serving up contaminated results (such as old Frank Sinatra songs that music industry operatives have uploaded under the titles of popular recent releases). This argues for a more principled approach to designing and reasoning about distributed systems.

Much as the complexity of writing regular software can be reduced by concise modular programming, the complexity of designing a distributed system can be reduced by constructing a *stack* of thin *layers*, each of which has a clearly defined interface and purpose. For example, Yahoo's PNUTS [45] and Google's BigTable [43] are thin layers that are used by the respective companies for sim-

ple but efficient distributed database functionality in their search engines. These layers are lean in terms of complexity and code, for instance the BigTable layer delegates all distributed data storage and redundancy issues to the layer sitting beneath it, the Google File System [55].

Upper layers in the software stack depend on the correctness of the lower layers. Problems tend to arise if the interface of the layer and/or functionality are not properly specified or implemented. Fortunately, the correctness of the lower layers has been scrutinized heavily for long periods of time. However, as the needs of the applications in the upper parts of the stack change with time, the assumptions that have been made in the lower layers sometimes fail and appear as frustrating bugs.

In this dissertation we identify and remedy scalability problems in the lower layers of modern distributed systems, specifically issues with two popular *group communication* paradigms. As the number of groups scale up, both layers start behaving badly, but for quite different reasons. Before delving deeper into these issues in section 1.4, we will first define and discuss the role of group communication in distributed systems.

1.3 Group Communication

Because the numerous machines hosting the layers of a distributed system could be connected by a potentially lossy network, scalable and reliable communication is a fundamental requirement addressed when designing those layers. A number of communication paradigms exist to accommodate these needs. We will discuss the merits and drawbacks of the ones central to this thesis.

Point-to-point unicast. A simple scheme is to maintain TCP connections between every pair of nodes that talk to one another, called *point-to-point unicast*. The reliability properties of the underlying TCP protocol ensure that temporary message loss or corruption of network links do not affect the system. While point-to-point unicast via TCP is feasible at smaller scales, for instance Yahoo!'s PNUTS [45] is designed in this fashion, there is much overhead associated with initializing and maintaining point-to-point unicast connections at larger scales, particularly if nodes have many communication partners.

Multicast. Nodes in distributed systems sometimes *broadcast* updates or information, meaning that they transmit the same packet simultaneously to all receivers. The nodes may not have the capabilities to maintain up to $n - 1$ TCP connections if the number of nodes n is large, and so a different scheme from the point-to-point unicast over TCP is needed. Node could also transmit packets to a more specific set of nodes, a paradigm known as application-level *multicast* or *one-to-many* communication. Because target sets frequently do not change, they are commonly specified as a multicast *group*.

Publish-subscribe. Another abstraction for multicast is *publish/subscribe* communication, in which *publishers* and *receivers* respectively send and receive messages to and from a *topic* of interest [50]. We will use the terms groups and topics interchangeably.

Gossip. One popular idea to send message one-to-many or one-to-all is to use *gossip* based protocols. Gossip was originally used to disseminate updates for replicas in a database system [49]. Each node exchanges the set of information it has learned with a random node during every time epoch. Nodes now communicate only with a small subset of other nodes and they tolerate substan-

tial node and message loss, at the cost of higher latency of dissemination.

IP Multicast. Another idea for one-to-many communication called *IP Multicast* (IPMC) was popularized in the early 1990s [48]. Nodes can join or leave a given group, and they can send and receive all messages sent within the group. IPMC operates on a network-level, leaving group membership and dissemination to the operating system kernel and networking hardware instead of the application itself. The semantics of IPMC is similar to that of IP unicast, best-effort guarantee with respect to routing but without any intrinsic reliability mechanism. Reliable IPMC has been the subject of much work in the past [33, 34, 53, 39]. IPMC is supported on most major routers and switches, and has in fact became the only option for multicast transport offered by operating systems and networks. Remarkably, IP Multicast rarely sees much use, for reasons we describe in chapter 2.

1.4 Contributions

In this dissertation we address shortcomings of two important group communication layers, IP Multicast and gossip based message dissemination, both of which have scalability issues when the number of groups grows, as stated earlier.

In chapter 2, we demonstrate that modern hardware limitations can trigger major stability problems with IPMC when too many groups are deployed. We propose a transparent and backward-compatible layer called *Dr. Multicast* to allow data center administrators to enable IPMC for large numbers of groups without causing stability issues. Dr. Multicast further optimizes the use of IPMC

resources by grouping together similar groups in terms of membership to minimize redundant transmissions as well as cost of filtering unwanted messages.

In chapter 3, we argue that when nodes belong to multiple groups, gossip based communication loses its appealing property of using fixed amount of bandwidth. We propose a platform called **GO** (for Gossip Objects) that bounds the node's bandwidth use to a customizable limit, prohibiting applications from joining groups that would cause the limit to be exceeded. We make the observation that gossip rumors are small so multiple rumors can be packed in a single packet, and thus rumors could be delivered indirectly via nodes in other gossip groups. **GO** exploits this observation by computing the *utility* of including a rumor in a message, taking into account the layout of gossip groups and the properties of the rumors.

Both Dr. Multicast and **GO** incorporate optimizations that are based on group similarity. The natural next question is "*How similar are subscriptions between groups?*" We explore the *affinity* or similarity between groups in chapter 4 by considering real data-sets from social networks and an industrial setting. We present and analyze models to characterize overlaps between groups, and discuss our results in the context of Dr. Multicast and **GO**. We discuss future questions and conclude the dissertation in chapter 5.

The chapters on Dr. Multicast and **GO** are self-contained, extended versions of papers that have appeared in print in peer-reviewed conferences. The work on Dr. Multicast appeared in the ACM Hot Topics in Networks (HotNets) workshop 2008 [85] as well as the Large-Scale Distributed Systems and Middleware (LADIS) workshop the same year [86]. The paper is in preparation for submission to the USENIX Symposium on Networked Systems Design and Implemen-

tation (NSDI) in 2010. The work on **GO** will appear in the Large-Scale Distributed Systems and Middleware (LADIS) in 2009 [88], and was invited to the International Peer-to-Peer (P2P) Conference 2009 [87].

CHAPTER 2
DR. MULTICAST

Data centers avoid IP Multicast due to scalability and stability concerns. In this chapter, we introduce *Dr. Multicast* (MCMD), a system that maps IPMC operations to a combination of point-to-point unicast and traditional IPMC transmissions. MCMD optimizes the use of IPMC addresses within a data center, while simultaneously respecting an administrator-specified acceptable-use policy. We argue that with the resulting range of options, IPMC no longer represents a threat and can therefore be used much more widely.

2.1 Introduction

As data center networks scale out, the software stack running on them is increasingly oriented towards multicast communication patterns. Publish-subscribe layers [14, 17] push data to large numbers of receivers simultaneously, clustered application servers [1, 8, 7] replicate state updates and heartbeats between server instances, and distributed caches [4, 11] invalidate and update cached information on large numbers of nodes. IP Multicast (IPMC) [48] is included by many of these products as a communication option — it permits each message to be sent using a single I/O operation, reducing latency and load at end-hosts and in the network.

Unfortunately, IPMC has earned a reputation as a poor citizen. Part of the problem relates to scalability: multicast filtering at switches and end-host NICs

does not scale well to large numbers of groups, defaulting to system-wide flooding beyond a threshold limit. Additionally, IPMC is perceived as an unstable technology — the intrinsic asymmetry between sending and receiving rates that makes it such a powerful communication option also renders it extremely dangerous if misused. Reliability and flow control protocols layered over IPMC are prone to 'storms' that can disrupt the entire data center. With the management of IPMC usage practically unsupported, administrators choose to banish it from their data centers, forcing applications to resort instead to redundant unicast transmissions.

This chapter introduces MCMD, a technology that permits data center operators to selectively enable IPMC while maintaining tight control on its use. The key insight behind MCMD is that IPMC addresses are scarce and sensitive resources. Accordingly, MCMD allows administrators to define fine-grained policies that dictate access control and IPMC usage rules within the data center — for example, by disallowing the use of IPMC by specific nodes, or by setting a limit on the number of IPMC groups in the data center. Taking into account these policies as well as multicast usage patterns, MCMD uses a clustering to efficiently allocate a limited number of IPMC addresses to selected groups (or sets of groups), and uses unicast communication for the rest.

To enforce access control and implement IPMC address allocations to groups, MCMD resides between the application and the OS network stack and intercepts standard IPMC system calls. Each IPMC address used by the application is translated into a combination of IPMC and unicast addresses. This translation spans two extremes:

- A true IPMC address is allocated to the group.

- Communication to the group is performed using point-to-point unicast messages to individual receivers.

Importantly, MCMD is completely *transparent*, requiring no modification to either the application or the network; this is a crucial property given the reliance of data centers on commodity hardware and software. Consequently, MCMD is extremely easy to deploy and use, supporting legacy IP Multicast applications over existing data center networks. MCMD is robust, timely and scalable in the number of groups in the system, running a gossip-based control plane across end-hosts to track membership and distribute address translations to senders.

The contributions of this chapter are the following:

- An approach to fine-grained policy control of IPMC within data centers that mitigates vulnerabilities and optimizes the use of multicast resources.
- A scalable and robust implementation that resides transparently between the application and the network stack.
- An evaluation using real-world subscription patterns based on a trace collected from a widely deployed commercial application server.

We will consider the problems of IPMC in data centers in the following section. The acceptable-use policy primitives and architecture of MCMD are discussed respectively in sections 2.3 and 2.4. We analyze the trace and experimentally evaluate components of MCMD in section 2.6. We will discuss related work in section 2.7, and conclude the chapter with section 2.8.

Later, we formalize the optimization problem of allocating a limited number of IPMC addresses, and provide and evaluate an effective heuristic for solving

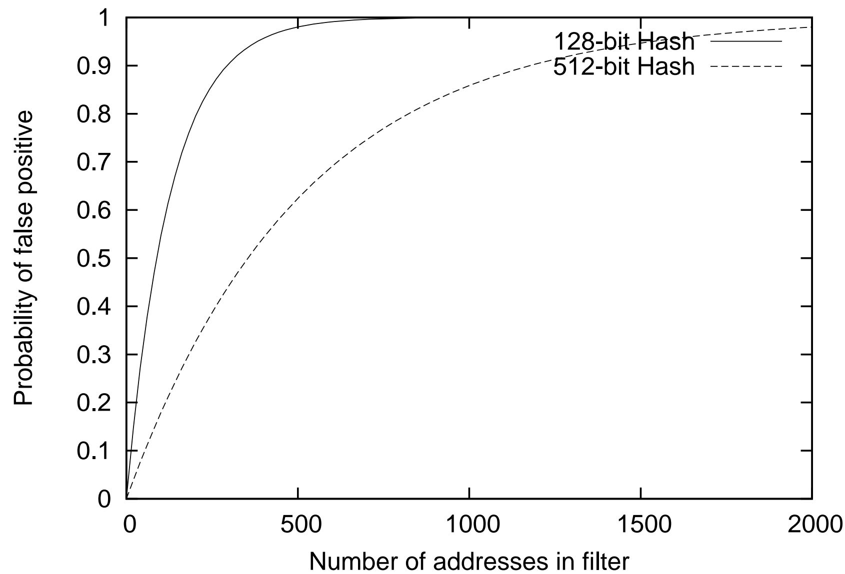


Figure 2.1: Receiver NIC Scalability: Probability of false positives in NIC imperfect hash filter vs. number of addresses.

it in chapter 4.

2.2 Motivation

In this chapter, we focus on the use of IP Multicast by trusted applications running within an administratively homogeneous data center. Applications are assumed to be non-malicious, but subject to misconfiguration and bugs. We assume the data center network to be primarily switched, with multiple levels of switching hierarchy and a top-level gateway router. Our target setting spans a range of application domains — large-scale Internet services, financial clusters, and even cloud computing platforms where back-end components use multicast to implement highly available infrastructural services.

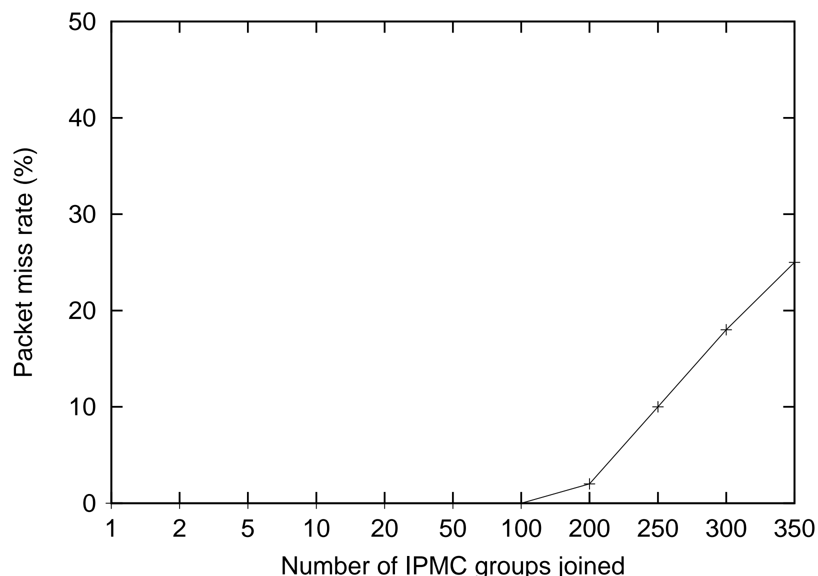


Figure 2.2: Receiver NIC Scalability: Packet loss rate vs. number of IPMC groups joined.

Modern data centers avoid IPMC, for two important reasons. First, IPMC is perceived as an *unscalable* technology — switches and end-host NICs can be overloaded by large numbers of groups and fail to effectively filter multicast traffic. Second, applications using IPMC are famously *unstable*, potentially exposing the data center to DoS scenarios and chaotic multicast storms.

2.2.1 Multicast Scalability

Layer 2 devices such as switches and NICs store membership information in the form of Ethernet multicast addresses (effectively 23 bits long), dropping the high-order 5 bits of each 28 bit class D IP address in the process [48]. Consequently, 32 IP addresses map to a single Ethernet MAC address, creating the possibility for expensive collisions if the data center uses thousands of ran-

domly chosen IPMC addresses. In practice, collisions can occur through poor address selection by applications even when a small number of groups is involved [6, 75]. For example, the default multicast group used by early versions of BEA WebLogic was 237.0.0.1 [2], an address that collided with the special all-hosts group 224.0.0.1 [48]; following versions of WebLogic changed the default to 239.192.0.0 [13] and added an injunction in the documentation to never use addresses in the x.0.0.1 range. Collisions can be extremely disruptive, allowing unwanted traffic to percolate through switches and NICs to end-host kernels, which must perform expensive discards in software.

A more fundamental problem is the limited capacity available on devices for storing membership information. To filter incoming multicast packets, a typical end-host NIC uses a combination of a perfect check against a small set of addresses, as well as an imperfect check against a hashed location within a table (effectively, a single-hash Bloom filter [40]). Stevens et al. [80] cite one commercial NIC as having a perfect matching set of 16 addresses and an imperfect matching table of 512 bits, another NIC as having a perfect matching set of 80 addresses with no imperfect matching table, and older NICs as supporting only imperfect matching with a 64 bit table. Figure 2.1 shows the probability of false positives for imperfect matching tables of size 128 bits and 512 bits.

Figure 2.2 illustrates the problem in practice. In this experiment, a multicast sender transmits on $2k$ multicast groups, whereas the receiver listens to k multicast groups. We varied the number of multicast groups k and measured the packet loss at the receiver. All group IP addresses were carefully chosen to avoid Ethernet address collisions. The sender transmits at a constant rate of 15,000 packets/sec, with a packet size of 8,000 bytes spread across all the

groups. The receiver thus expects to receive half of that, i.e. 7,500 packets/sec. The receiver and transmitter have 1Gbps NICs and are connected by a switch with IP routing capabilities. The experiments were conducted on a pair of single core Intel® Xeon™ 2.6GHz machines. The figure shows that the critical threshold that this particular NIC can handle is roughly 100 IPMC groups, after which throughput begins to fall off.

Switches perform only marginally better. They have been known to silently discard membership information beyond a threshold number of groups [12]; more commonly, they resort to flooding data on all ports. The performance of modern 10Gbps switches was evaluated in a recent review [68] which found that their group capacity ranged between as little as 70 and 1500. Less than half of the switches tested were able to support 500 multicast groups under stress without flooding receivers with all multicast traffic.

To summarize, multicast does not scale to large data centers with thousands of groups for two reasons — the imperfect mapping between IP and Ethernet multicast addresses on NICs and switches, and the limited capacity available on such devices for storing membership state. Collisions due to the imperfect mapping between IP addresses and Ethernet MAC addresses are easy to avoid in principle — by simply fixing the first five bits of all IPMC addresses in use — but occur in practice due to arbitrary address selection by applications. Capacity constraints within switches and NICs are much harder to resolve — for instance, each make of NIC can have a different hashing scheme and collisions are likely even with a few dozen groups. As a result, data centers are severely constrained in the number of IPMC groups they can support simultaneously.

2.2.2 Multicast Stability

The perception that IPMC is an unstable technology is harder to demonstrate in simple experiments. Below are some common scenarios encountered in modern data center deployments:

- *Multicast Storms* — A slow receiver running a publish-subscribe product with a built-in reliability layer drops packets and continuously multicasts retransmission requests to the group, provoking a multicast storm of retransmissions by other receivers that slows down the entire group and causes further packet loss — and potentially creates a cascading effect that brings the entire data center to a standstill [13, 75].
- *Multicast DoS* — An incorrectly parametrized loop results in a sender transmitting data to an IPMC group at very high speeds, overloading all the receivers in the group.
- *Traffic Magnets* — A receiver in a particular cluster within the data center inadvertently subscribes to one or more high data-rate groups used by a different cluster within the data center; the resulting flood of incoming traffic saturates the bandwidth connecting this cluster to the main data center topology.
- *Scattershot Senders* — A programming error causes an application to send data to the wrong IPMC address spamming machines subscribed to that group with packets that need to be discarded.
- *IGMP Churn* — A faulty receiver joins and leaves groups at a very high rate, overloading the networking back plane.

The root causes of multicast instability are two-fold — the skewed balance of power between senders and receivers in IPMC, and the free-for-all nature of its usage. Any machine can join or send data at any speed to any group in the system, with IPMC providing absolutely no regulatory mechanisms for multicast usage.

2.3 Acceptable-Use Policy

The basic operation of MCMD is simple. It translates an application-level multicast address used by an application to a set of unicast addresses and network-level multicast addresses, as shown in figure 2.3. The translation is governed by an acceptable-use policy for the data center as defined by the system administrator.

In this section we describe the policy primitives supported by MCMD, and demonstrate how scalability and stability concerns can be mitigated by constructing a high-level acceptable-use policy made from those building blocks.

2.3.1 Policy Primitives

In this section the terms logical and application-level groups are used interchangeably; the same goes for physical and network-level groups. An arbitrary node is denoted by the letter n.

We use the following notation while describing the primitive operations:

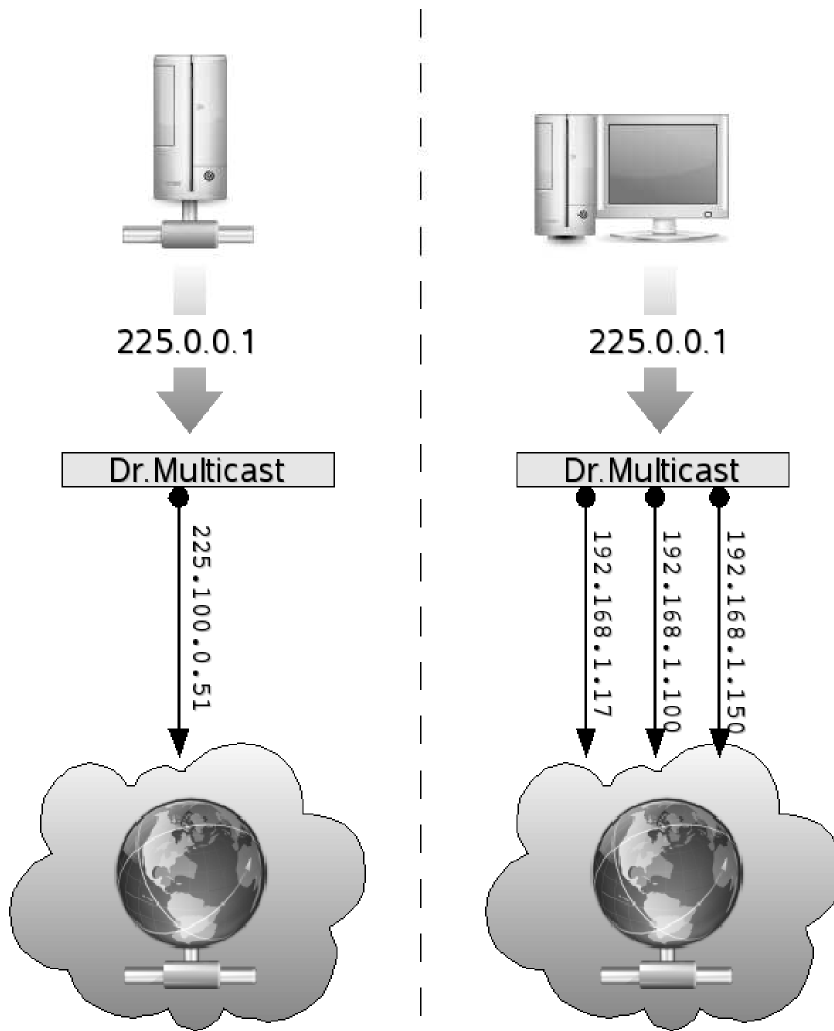


Figure 2.3: Two under-the-hood mappings in MCMD, a direct IPMC mapping (left) and point-to-point mapping (right).

- Logical groups by upper-case letters: A, B, C ...
- Physical groups by lower-case letters: a, b, c ...

If the physical group a is included in the set of unicast and multicast addresses that a logical group A is translated into by MCMD, we say that the physical group a is a *transport* for the logical group A.

In reality, identifiers for both logical and physical groups are independently

drawn from the set of class D IP addresses. For convenience, we assume that the physical and logical groups represented by the same letter are mapped to the same IP address; for example, logical group A and physical group a are both identified by the IP address 239.255.0.1. In addition, while discussing unmodified IPMC, we ignore the existence of logical groups and deal only with nodes and physical groups.

By default, no node in the data center is allowed to send to or join any logical group. The primitives serve the purpose of selectively allowing nodes to join and send to logical groups, as well as mandating when physical IPMC groups can be used as transports for logical groups.

MCMD understands a small set of *primitives*:

- **allow-join(n, A)** — Node n is allowed to join the logical group A.
- **allow-send(n, A)** — Node n is allowed to transmit data to logical group A.
- **allow-IPMC(n, A)** — Node n is allowed to use physical IPMC groups as transports for the logical group A.
- **max-rate(n, A, X)** — n is allowed to send data at a maximum rate of X KB/s to any of the physical addresses that are mapped to the logical group A.
- **max-IPMC(n, M)** — n is allowed to join at most M physical IPMC groups.
- **limit-IPMC(M)** — A maximum of M IPMC groups can be used within the data center.
- **max-churn(M)** — Each node in the system is limited to M membership change events per second.
- **limit-filtering(α)** — The fraction of unwanted traffic that can be tolerated by a receiver (i.e, $\alpha = 0.05$ implies that up to 5% of the traffic a node

receives can be in logical groups it did not join).

Our system implements these primitives efficiently; by intercepting socket system calls and controlling the mapping from logical groups to physical addresses, it can prevent nodes from joining or sending to logical groups, as well as limit the sending rate to these groups. Further, the use of IPMC can be enabled selectively on a per-group and per-node basis. We believe that this compact set of primitives is sufficient to mitigate most if not all the vulnerabilities of multicast communication within data centers.

2.3.2 Higher-Level Policy

The primitives allow a data center administrator to express policies on a spectrum that spans two extremes. On one hand, he or she can ban the use of IPMC completely, in which case all application-level multicast addresses are translated into a set of unicast addresses; alternatively, he or she can mandate the use of raw IPMC, in which case each application-level multicast group is directly mapped to a corresponding network-level multicast group. Actual mappings lie between these two cases, with each application-level multicast group mapped to a combination of unicast addresses and network-level multicast addresses, depending on the use of the primitives and subscription patterns.

While MCMD itself is controlled by policies expressed in terms of its primitive operations, we expect data center administrators to use higher level tools to define acceptable-use policies in a user-friendly manner. These policies would 'compile' into the lower level primitives that MCMD understands. In this chapter, we restrict ourselves to a very simple scheme for generating these lists of

primitive operations — we expect the data center administrator to map applications within the data center to the specific nodes they run on, define the set of logical groups to which these applications send or receive data in, and consequently generate a mapping from the nodes to the logical groups they use.

2.3.3 Policy Examples

The policies defined by the administrator resolve the stability problems of IPMC by implementing a form of access control for groups. In addition, they mitigate the scalability concerns of IPMC by placing a limit on the total number of IPMC addresses in use within the data center and by each node individually.

Are these simple primitives sufficient to prevent the stability problems of IPMC? Let us consider the instability scenarios outlined earlier:

- Cure for the *Multicast Storm* scenario: While it is difficult to prevent unstable reliability protocols running within a group from impacting the receivers in that group, MCMD can isolate the slowdown to just that group by either disabling IPMC transports for it or placing a rate cap.
- Cure for the *Multicast DoS* scenario: By limiting the maximum rate at which any sender is allowed to transmit data to a particular group, we can prevent the scenario where a single machine launches a DoS attack on a group by sending data to it as fast as possible.
- Cure for the *Traffic Magnet* and *Scattershot Sender* scenarios: Access control mechanisms for regulating joins and sends are sufficient to prevent both these cases. Nodes can longer join or send data to arbitrary multicast groups.

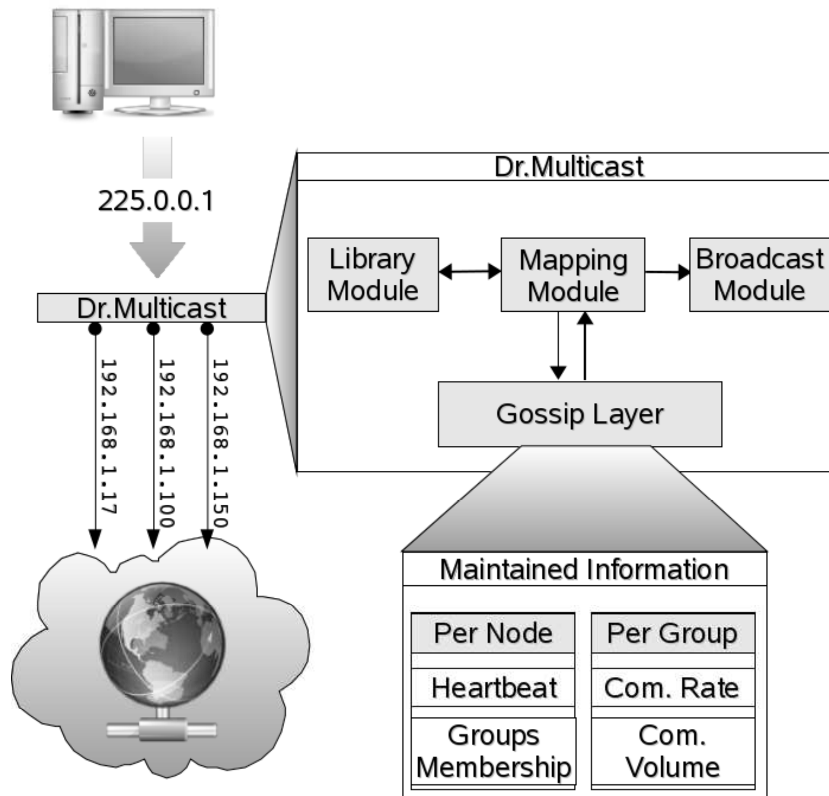


Figure 2.4: Overview of the MCMD architecture.

- Cure for the *IGMP Churn* scenario: By setting the maximum churn rate of any single node, we can rate-limit joins and leaves to prevent network overload.

2.4 Design and Implementation

We built MCMD in two main components; a *library* module that overloads the standard socket interface and allows MCMD to be transparently loaded into applications, and an *agent* daemon that is responsible for implementing the user-defined policy and the application-level multicast mapping. Each node in the

system has a running agent, and one of these agents is designated as a *leader* that periodically issues multicast group mappings. The mapping information is replicated across all the agents via a gossip layer, and an additional urgent broadcast channel is used to quickly disseminate urgent updates. Figure 2.4 highlights the different components of MCMD.

In the remainder of this section we will discuss the design and implementation of each of these components in detail.

2.4.1 Library Module

The library module exports a `netinet/in.h` library to applications, with interfaces identical to the standard POSIX version. By overloading the relevant socket operations, MCMD can intercept join, leave and send operations. For example:

- In the overloaded version of `setsockopt()`, invocations with e.g. the `IP_ADD_MEMBERSHIP` parameter will be intercepted by the mapping module. An IGMP JOIN message will only be sent if the logical IPMC address is mapped to a new physical IPMC address.
- `sendto()` is overloaded so that a send to a class D group address is intercepted and converted to multiple sends to a list of addresses. The acceptable-use policy can limit the rate of sends.

The library module periodically interacts with the mapping module of the agent daemon via a UNIX socket to pull — and cache — the list of IP Multicast groups it is supposed to join, and the translations for application-level groups

it wants to send data to. The library module can receive invalidation messages from the mapping module, causing it to refresh its cached entries. Simultaneously, it pushes information and statistics about grouping and traffic patterns used by the application to the mapping module. This includes an exponential-average of the message rate for the application-level group.

Multi-send Optimization

As performance optimization, the library module uses a custom multi-send system call implemented in the Linux 2.6.24 kernel — a variant of the `sendto()` call that accepts a list of destinations for the message. As a result, when the application sends a message to an application-level group and the library module translates the operation into a multi-send to a set of physical addresses, it can send the message to these addresses in a single efficient system call.

2.4.2 The MCMD Agent

The *agent* is a background daemon process running on every node in the system. Each agent instance acts as a *mapping module*, maintaining four pieces of information that are globally replicated on every agent in the system — we refer to these collectively as the *agent state*:

- Membership sets for all the nodes in the system — essentially, a map from nodes to the application-level groups they are receivers in.
- Sender sets for all the nodes in the system — a map from nodes to the application-level groups they are senders to.

- Group translations — a map from application-level groups to sets of unicast and multicast network addresses.
- Access control lists — two separate maps determining which application-level groups each node in the system is allowed to receive data in and send data to, respectively.

Each agent in the system has read-access to a locally replicated copy of the agent state. However, write-access to the agent state is strictly controlled. The first two items of the agent state can be written to only by the nodes concerned — a node can change only its own membership set or its sender set. The last two items of the agent state can be modified only by the leader agent. When an agent — leader or otherwise — writes to its local copy of the agent state, the change is propagated to other agents in the system via a gossip layer, which guarantees eventual consistency of agent state replicas. Since each item in the agent state has exactly one writer, there are no conflicts over multiple concurrent updates to the agent-state.

The leader agent allocates IPMC addresses to different sets of machines in the data center, using the group membership information, sender information and access control lists in its local state to determine the best set of translations for the system. Once it writes these translations to its local state, the gossip layer disseminates the updates to other agents in the system, which read the translations off their local replicated copy of the agent state and direct their corresponding library modules to join and leave the appropriate IGMP groups. The process followed by the leader while allocating network-level IPMC resources to application-level multicast groups is the subject of section 2.5.

State Replication via Gossip

A gossip-based failure detector identical to the one described by van Renesse [83] is used to replicate the agent state across all the agents. Each node maintains its own version of a global table, mapping every node in the data center to a time-stamp or heartbeat value. Every T milliseconds, a node updates its own heartbeat in the map to its current local time, randomly selects another node and reconciles maps with it. The reconciliation function is extremely simple – for each entry, the new map contains the highest time-stamp from the entries in the two old maps. As a result, the heartbeat timestamps inserted by nodes into their own local maps propagate through the system via gossip exchanges between pairs of nodes.

When a node notices that the time-stamp value for some other node in its map is older than T_1 seconds, it flags that node as 'dead'. It does not immediately delete the entry, but instead maintains it in a dead state for T_2 more seconds. This is to prevent the case where an entry is reintroduced into its map by some other node. After T_2 seconds have elapsed, the entry is deleted.

The comparison of maps between two gossiping nodes is highly optimized. The initiating node sends its peer a set of hash values for different portions of the map, where portions are themselves determined by hashing entries into different buckets. If the receiving node notices that the hash for a portion differs, it sends back its own version of that portion. This simple interchange is sufficient to ensure that all maps across the system are kept loosely consistent with each other. An optional step to the exchange involves the initiating node transmitting its own version back to the receiving node, if it has entries in its map that are more recent than the latter's.

Crucially, *the failure detector can be used as a general purpose gossip communication layer.* Nodes can insert arbitrary state into their entries to gossip about, not just heartbeat timestamps. For example, a node could insert the average CPU load or the amount of disk space available — or, more relevantly, its agent state — and eventually this information propagates to all other nodes in the system. The reconciliation of entries during gossip exchanges is still done based on which entry has the highest heartbeat, since that determines the staleness of all the other information included in that entry.

Urgent Broadcast Channel

Although gossip is useful to replicate agent state data across multiple nodes, it can be slow. For this reason, an urgent notifications broadcast channel is used to quickly disseminate urgent updates.

MCMD uses urgent notifications in three cases:

- When a new receiver joins a group, its agent updates the local version of agent state and simultaneously sends unicast notifications to every node listed in the agent state as a sender to that group. As a result, senders that are using multi-send unicast to transmit data to the group can immediately include the new receiver in their transmissions. In addition, the new receiver's agent contacts the leader agent for updates to the sender set of that group; if the leader reports back with new senders not yet reflected in the receiver's local copy of the agent state, the receiver's agent sends them notifications as well.
- When a new sender starts transmitting to a group, the agent running on it

updates the sender set of the group on its own local version of the global agent state, and simultaneously sends a notification to the leader agent. The leader agent responds with the latest version of the group membership information for that particular group.

- When the leader agent creates or modifies a translation, it sends notification messages to all the affected nodes — receivers who should join or leave IPMC groups to conform to the new translation, and senders who need to know the new translation to transmit data to the group. These messages cause their recipients to 'dial home' and obtain the new translation from the leader.

Notice that the first two cases involve a single unicast exchange with the leader, imposing load on it that increases linearly with the level of churn in the system. The task of updating other interested nodes in the system is delegated to the node that caused the churn event in the first place; this ensures that nodes can only disrupt themselves by changing membership and sender sets at a high rate. In addition, MCMD limits the rate of such events at any particular node. Also, since MCMD was explicitly designed for data centers, it assumes a low rate of membership change in the system, with no more than a few nodes joining or leaving groups per second. When large-scale membership changes do occur (due to correlated failure, for example), rate-limiting prevents a load spike on the notification channel, and the gossip layer eventually converges to a stable view of the system.

Robustness and Responsiveness

Replicating the agent state across all the nodes makes the system robust against leader failure. Once agents realize that the leader is no longer responsive, the leader is marked as dead and that information is disseminated to all the nodes via the control plane. A leader election protocol is started to appoint a new leader agent. Additionally, the size of the replicated global view is not prohibitive; for example, we can store the agent state for a 1,000-node cluster with a membership pattern based on our real-life trace within a few MB of memory. In addition to that, complementing the gossip layer with an urgent channel ensures that nodes are responsive to sudden changes in the state of the system.

2.5 Theoretical Considerations

The MCMD leader can map network-level IP multicast addresses to some of the application-level groups in the system, and command others to communicate via unicast. The mapping must adhere to the acceptable-use policy, but should also achieve scalability goals:

- *Minimize the number of network-level IPMC addresses*. NICs, routers and switches do not scale in the number of IPMC addresses, as discussed earlier.
- *Minimize redundant transmissions*. This reduces the rate of packets sent by publishers and alleviates network overhead.
- *Minimize receiver filtering*. End host filtering of unwanted traffic is expensive [42]. Furthermore, imposing high CPU loads on receivers can have

unanticipated consequences and potentially cause more trouble than the system solves.

These goals are in tension. For instance, one could assign a single physical IPMC address to multiple application-level groups that have many common receivers at the cost of filtering of superfluous traffic for some receivers. Alternatively, using point-to-point unicast for these groups will minimize filtering and the number of IPMC addresses, but forces senders to transmit each packet multiple times.

In MCMD, we put a hard limit M on the total number of physical IPMC groups that are allowed in the system, but make redundant transmissions and receiver filtering soft. At a high-level, our heuristic works as follows. We cluster logical groups that are similar in terms of membership using the k-means clustering heuristic with $k = M$. We could directly promote the groups in these M clusters to physical IPMC groups, as this would minimize transmission costs while keeping the hard limit M. However, the filtering costs could be prohibitive, so we gradually remove groups from clusters that have the highest filtering costs, and have them use point-to-point unicast. When filtering costs are below the **limit-filtering** threshold, we promote the current clusters to use IP multicast.

Chapter 4 has detailed discussion of the MCMD optimization problem and heuristics, as well as an evaluation of those heuristics on various real-world and synthetic data sets.

2.6 Experimental Results

We tested an implementation of MCMD to measure its overhead, policy application, and scalability properties. We evaluated our implementation in some of the bad-case scenarios outlined in section 2.2. We also simulated the MCMD heuristic on a trace of IBM WebSphere Application Server to measure its potential effects on multicast usage in the data center. The details of the trace are provided in Chapter 4.3.1. In section 4.6 we will analyze the trace to better understand the types of subscription patterns that arise in a real data center. Our results suggest that MCMD provides fast and scalable control of IP multicast with negligible overhead.

2.6.1 Implementation

We implemented a prototype of MCMD in C/C++, and deployed it on the Emulab testbed. All nodes have an Intel Pentium 3.0GHz processor and 1GB of RAM. Unless explicitly mentioned, the network configuration is a star topology with 100Mbps links between nodes. Each node in the testbed ran the MCMD agent, along with one of the following applications:

- A *sender application* joins k logical groups, waits for 2 seconds, then transmits 100,000 1KB packets using `sendto()` to the k groups in a round-robin fashion as fast as possible. The application can be configured to rate-limit the send to 5,000 packets/sec.
- A *receiver application* joins the same k logical groups, and waits for incoming packets in a `recv()` loop.

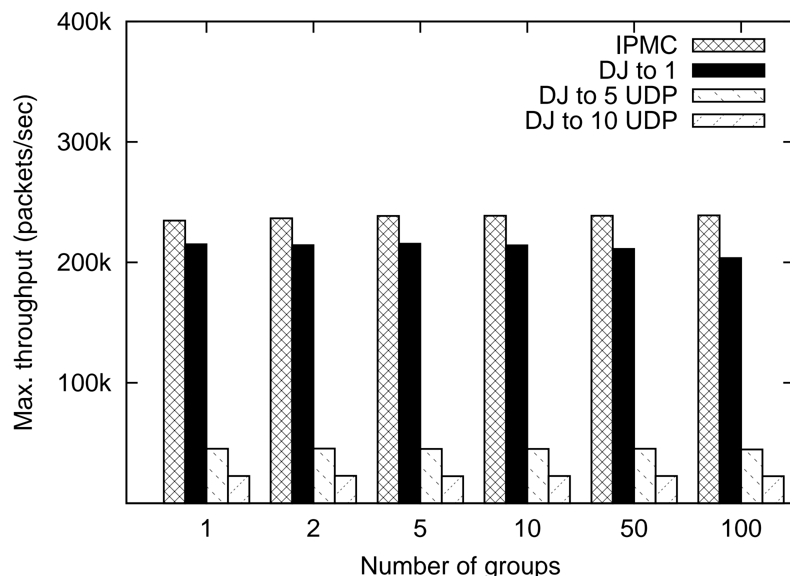


Figure 2.5: Application Overhead: Maximum throughput for a sender using regular IPMC and MCMD with direct IPMC mapping, and MCMD unicast to 5 or 10 receivers per group.

The rate of gossip or *epoch length* for the agent was set to 1 exchange per second, unless otherwise specified. Error bars represent one standard deviation, and are omitted if they are too small to be clearly visible.

Application Overhead

We measured the difference in maximum throughput for the sender application for varying k with and without the MCMD library. We considered both the case where MCMD maps each application-level address to a single network-level IPMC address, and also the case where each address resolves to 5 or 10 unicast addresses. The average group size in the WEBSPHERE trace was around 12.

As shown in figure 2.5, there is merely a 10-15% reduction of maximum throughput by running the sender application with MCMD over one-to-one

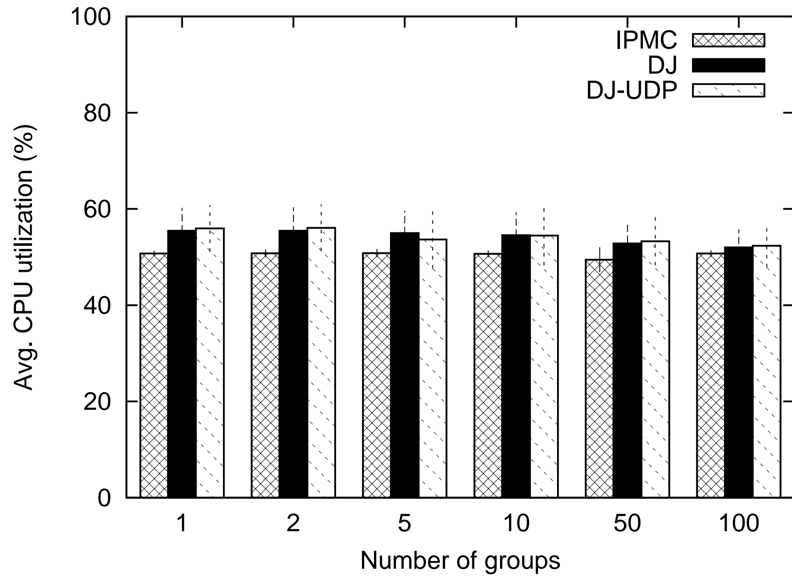


Figure 2.6: Application Overhead: Average CPU utilization for the sender application with and without MCMD.

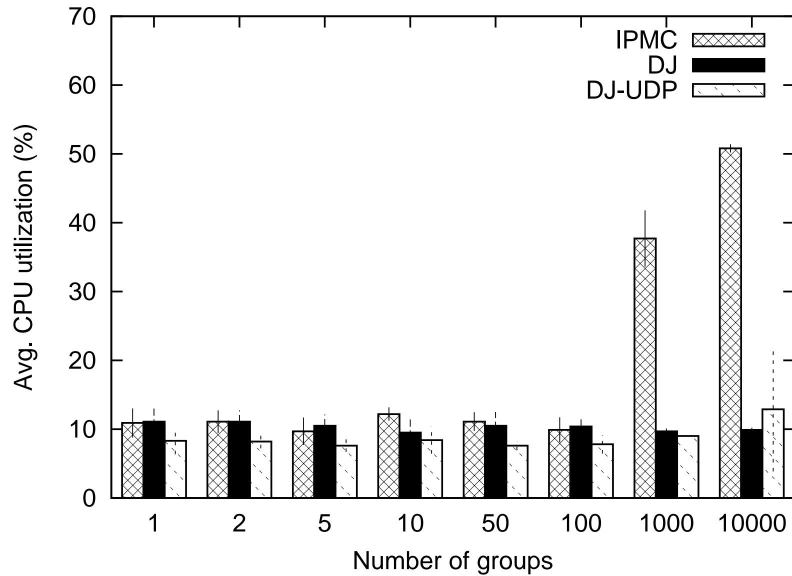


Figure 2.7: Application Overhead: The overhead at a receiver caused by raw IPMC and collapsed IPMC usage by MCMD. The overhead spike in raw IPMC is associated with the packet loss shown in figure 2.2.

address mapping, depending on number of logical groups. We also measured CPU utilization for the sender application with and without MCMD active. Figure 2.6 shows an increase of no more than 10% independent of the number of groups. Our system supports sending over 200,000 packets per second. This was made possible by moving system calls from the critical path of overloaded routines to a separate thread. Collisions in hash-maps account for the slight increase in MCMD look-up time.

Figure 2.7 shows the effect of collapsing IPMC groups on the CPU utilization at receivers. With raw IPMC CPU utilization spikes when a large number of groups is joined; the same spike in packet loss that was previously exhibited by IPMC in figure 2.2. As figure 2.7 shows, collapsing IPMC groups by MCMD causes a consistent CPU utilization even when a large number of groups is joined.

The performance of point-to-point unicast meets our expectations, realizing approximately $1/r$ of the maximum possible throughput when each application-level group is mapped to r physical addresses.

We experimented with the multi-send kernel system-call separately, revealing a consistent 17% increase in throughput over a `sendto()` loop in user-space. The improvements stem from fewer context switches taking place when using our kernel calls, since substantially less data is copied from user space to kernel space.

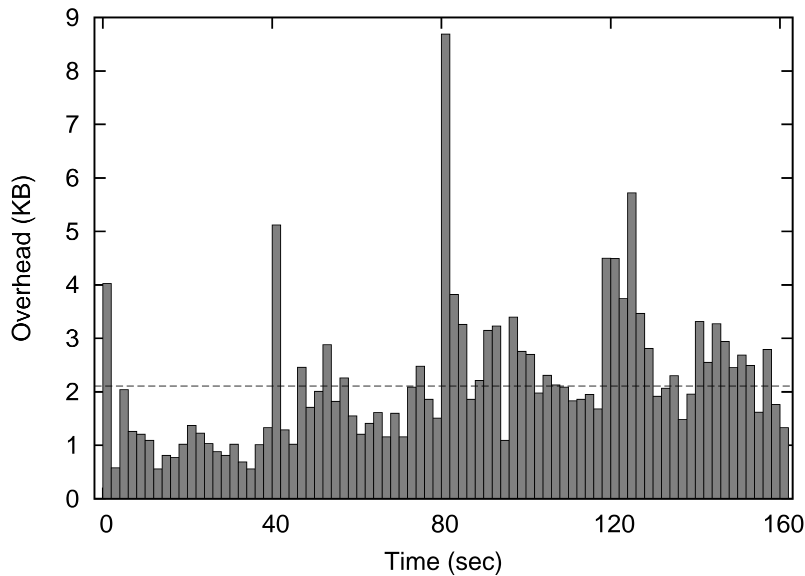


Figure 2.8: Network overhead: Traffic due to MCMD with and without an urgent notification channel.

Network Overhead

The network overhead of the MCMD protocol is shown in figure 2.8. In this experiment the network constitutes 16 nodes. The graph shows the amount of traffic transmitted and received by the most loaded node with respect to network traffic, namely the leader. Initially, there are 6 nodes running both a sender and receiver, joined by 5 more nodes at time 40 and 6 more at 80. At time 120, a new translation is computed by the leader, and an urgent notification is transmitted to the appropriate nodes.

By design, the gossip module in the MCMD agent produces configurable constant background traffic. At no point does MCMD traffic exceed 10 KB/sec, even when the urgent notification channel is enabled.

Latency

We now measure the latency of updates between nodes, namely membership and mapping changes. This determines, for instance, how fast a new receiver starts receiving messages from senders, or how long it keeps receiving messages after leaving a logical group. As discussed earlier, solutions need to trade-off latency and scalability. We compare the scalable gossip control plane per se to the fast version of MCMD that deploys an urgent notification channel on top of the gossip mechanism. In figure 2.9 we can see how fast new updates propagate through a 32-node network with and without the urgent notification channel. In this experiment, the gossip module has propagated the update everywhere after 10 epochs, and follows the well-studied epidemic replication curve [83]. When urgent notifications are used, the latency drops to at most 15 ms.

Policy Control

We revisit the *Multicast DoS* scenario from section 2.2 in which a malfunctioning sender suddenly starts sending large amounts of traffic in a loop to a logical group, thus overloading the receivers. Consider a network of 16 nodes that are sending and receiving low rates of traffic over a single IPMC group. At time 20, one of the senders starts bombarding the group with traffic. The administrator changes the policy at time 40 to remove the faulty sender from the group. Alternatively, the administrator could have put a rate-limit on sends to this particular group.

In figure 2.10 we see the CPU utilization of a receiver in the group, averaged over 10 trials of running this experiment. The CPU utilization increases sub-

stantially when the DoS begins, and decreases almost instantly after the new administrative policy is issued. In effect, the sender was commanded to leave the group via an urgent notification from the leader.

We also looked at the *Traffic Magnet* scenario where an unsuspecting node in cluster B joins a high-traffic multicast group in cluster A, increasing the load on the router between the two clusters substantially. We set up an experiment where 12 nodes in A each transmit 20 KB/s to a logical group that is mapped to a network-level IPMC by MCMD. We measured the average throughput over 10 trials between two regular nodes, one in each cluster, and show the results in figure 2.11. At time 20, a node n in cluster B joins the IPMC group, causing the throughput between the regular nodes to plummet to 2.5Mbps a 75% drop. At time 40, the administrator changes the access policy and disallows node n from belonging to the logical group, causing MCMD to make n leave the network-level IPMC group. The network has recovered 5 seconds later.

Naturally, both of these episodes could have been prevented by specifying a complete administrative policy with access restrictions and rate-limits for senders, as described in section 2.3.

2.6.2 Real-World Application

To test the MCMD implementation against a real application, we modified the Bulletin Board (BB) part of the IBM Websphere Application Server to disseminate messages using MCMD. The original version uses an unstructured application-level overlay network to broadcast each message by flooding. Since BB has no IPMC support, we modified it to use groups through an IPMC inter-

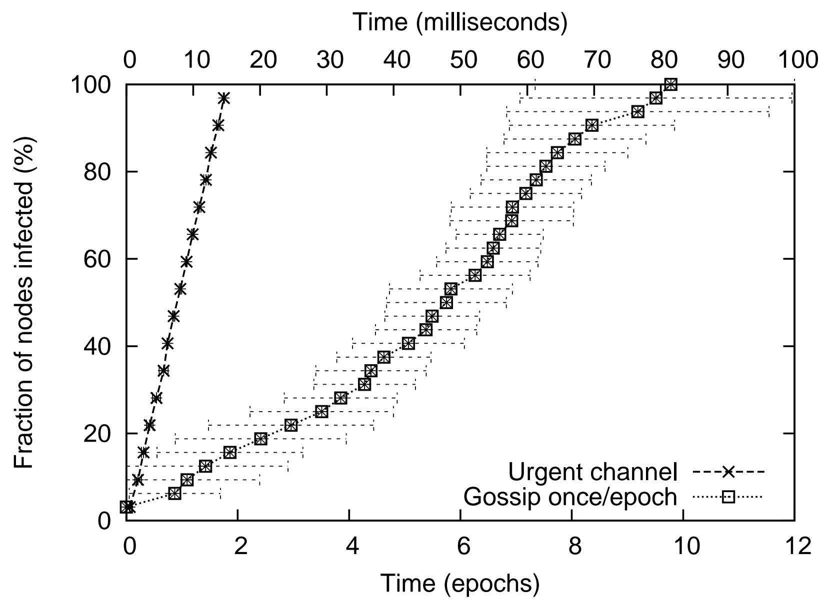


Figure 2.9: Latency: Update latency using regular gossip (epochs) and with the urgent notification channel enabled (ms).

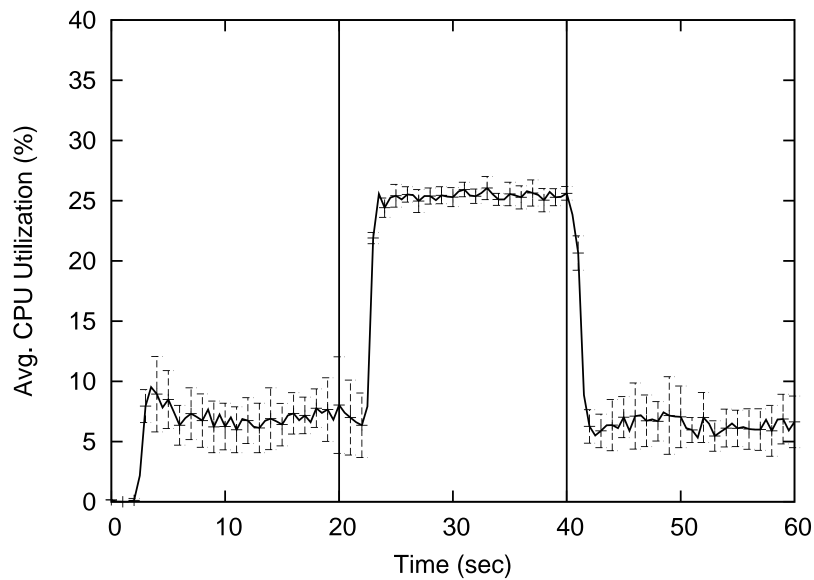


Figure 2.10: Policy Control: CPU utilization at a normal receiver. A malfunctioning node bombards the group at time 20, and the administrator restricts policy at time 40.

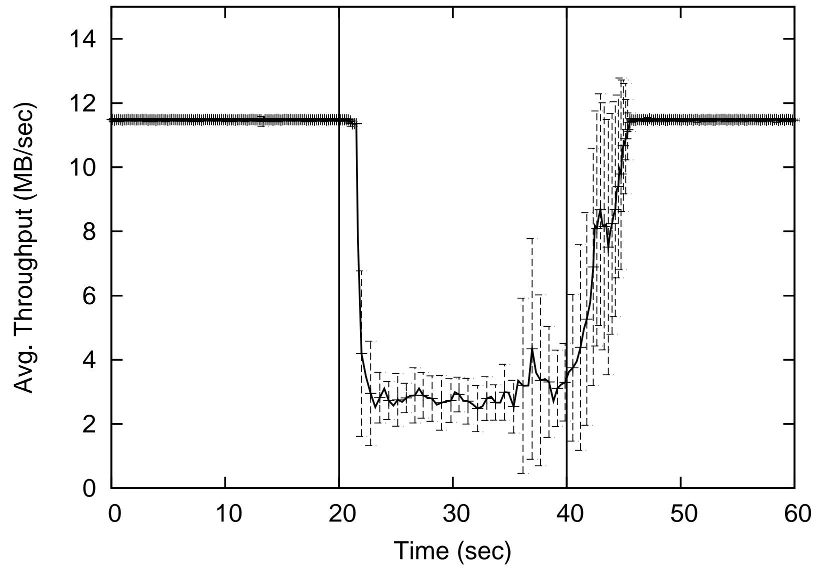


Figure 2.11: Policy Control: Average throughput between two nodes in separate subnets. At time 20, a node erroneously joins a high-traffic IPMC group in the other subnet, and the administrator corrects the access control policy at time 40.

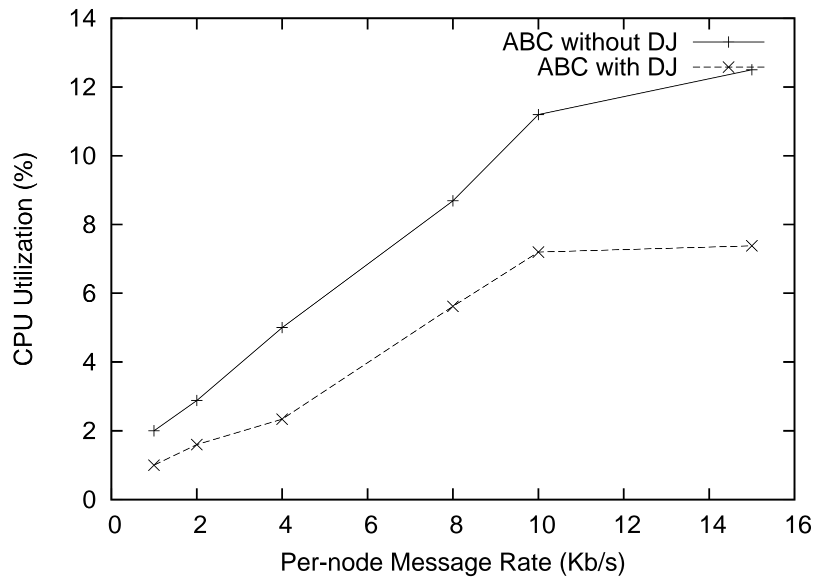


Figure 2.12: Real-World Application: CPU utilization of WEBSPHERE Bulletin Board at a regular node vs. per-node send rate x with and without MCMD support with real subscription patterns.

face.

We measured the performance of WAS with and without MCMD on a virtualized cluster of 97 nodes using the subscription patterns from a system trace that has thousands of non-empty logical groups. For the evaluation, we used the subscription patterns from the trace in section 4.3.1 and synthesized the traffic rates. Each process first joins the logical groups it ever subscribes to, and then sends traffic to those logical groups it publishes on at a total rate of x Kb/s. Figure 2.12 shows the CPU utilization at a non-leader node as the per-node send rate x varies. MCMD's use of network-level multicast clearly alleviates the burden of application-level packet forwarding and filtering required by the overlay.

2.7 Related Work

In the two decades since IP Multicast was first introduced [48], researchers have extensively examined its security, stability and scalability characteristics. Much of this work has attempted to scale and secure multicast in the wide area.

2.7.1 Stability and Security

Work on secure multicast has focused on achieving two properties in the wide-area: secrecy and authentication [41, 67]. Secrecy implies that only legal receivers in the group can correctly receive data sent to the group, and authentication implies that only legal senders can transmit data to the group [67]. Both these properties are typically obtained by using cryptographic keys, and much of the work in this area has focused on the task of group key management.

The security issues examined by MCMD are orthogonal to this existing body of work — within a data center, we are not concerned with either secrecy or authentication. Achieving these properties would not alleviate the performance problems of IP Multicast; for instance, a sender could still spam a group with nonsense data that fails to authenticate but nevertheless overloads receivers.

Access control for multicast has been proposed before as a solution for achieving secure multicast [35, 58]; once again, this work is aimed at wide-area scenarios and focuses on the secure implementation of the access control mechanism. SSM [37] is an IP Multicast variant that allows receivers to subscribe to individual senders within multicast groups, eliminating the problem of arbitrary machines launching DoS attacks on a group.

Reliable multicast is a research sub-area in itself, and many papers have looked specifically at the stability of reliability mechanisms. SRM [53] — a well-known protocol with many widely deployed variants including PGM [54] — is known to be susceptible to storms of recovery traffic in certain conditions [39]. MCMD operates at the routing layer and is oblivious to end-to-end reliability mechanisms, but can help mitigate the ill-effects of these protocols, as described previously.

2.7.2 Scalability

The scalability of IP Multicast in the number of groups in the system is limited by the space available in router tables [77]. The impact of adding IPMC state to network routers has been analyzed by Wong, Katz and McCanne [90, 91]. Prior work on algorithmic issues and the *channelization* problem that is at the heart of

the MCMD heuristic is discussed in section 4.7.

2.8 Conclusion

Many major data center operators legislate against the use of IP multicast: the technology is perceived as disruptive and insecure. Yet IPMC offers very attractive performance and scalability benefits. This chapter proposes MCMD, a remedy to this conundrum. By permitting operators to define an acceptable use policy (and to modify it at run-time if needed), MCMD permits active management of multicast use. Moreover, by introducing a novel scheme for sharing scarce IPMC addresses among logical groups, MCMD can reduce the number of IPMC addresses needed sharply, and ensures that the technology is only used in situations where it offers significant benefits.

CHAPTER 3
GOSSIP OBJECTS

Gossip-based protocols are increasingly popular in large-scale distributed applications that disseminate updates to replicated or cached content. **GO** (Gossip Objects) is a per-node gossip platform that we developed in support of this class of protocols. In addition to making it easy to develop new gossip protocols and applications, **GO** allows nodes to join multiple gossip groups without losing the appealing fixed bandwidth guarantee of gossip protocols, and the platform optimizes rumor delivery latency in a principled manner. Our heuristic is based on the observations that multiple rumors can often be squeezed into a single IP packet, and that indirect routing of rumors can speed up delivery. We formalize these observations and develop a theoretical analysis of this heuristic. We have also implemented **GO**, and study the effectiveness of the heuristic by comparing it to the more standard random dissemination gossip strategy via simulation. We also evaluate **GO** on a trace from a popular distributed application.

3.1 Introduction

Gossip-based communication is commonly used in distributed systems to disseminate information and updates in a scalable and robust manner [49, 60, 39]. The idea is simple: At some fixed frequency, each node sends or exchanges information (known as *rumors*) with a randomly chosen peer in the system, al-

lowing rumors to propagate to everybody in an "epidemic fashion".

The basic gossip exchange can be used for more than just sharing updates. Gossip protocols have been proposed for scalable aggregation, monitoring and distributed querying, constructing distributed hash tables and other kinds of overlay structures, orchestrating self-repair in complex networks and even for such prosaic purposes as to support shopping carts for large data centers [47]. By using gossip to track group membership, one can implement gossip-based group multicast protocols.

When considered in isolation, gossip protocols have a number of appealing properties.

P1. **Robustness**. They can sustain high rates of message loss and crash failures without reducing reliability or throughput [39], as long as several assumptions about the implementation and the node environment are satisfied [28].

P2. **Constant, balanced load**. Each node initiates exactly one message exchange per round, unlike leader-based schemes in which a central node is responsible for collecting and dispersing information. Since message exchange happens at fixed intervals, network traffic overhead is bounded [84].

P3. **Simplicity**. Gossip protocols are simple to write and debug. This simplicity can be contrasted with non-gossip styles of protocols, which can be notoriously complex to design and reason about, and may depend upon special communication technologies, such as IP multicast [48], or embody restrictive assumptions, such as the common assumption that any node can communicate directly with any other node in the application.

P4. **Scalability**. All of these properties are preserved when the size of the system increases, provided that the capacity limits of the network are not reached and the information contained in gossip messages is bounded.

However, gossip protocols also have drawbacks. The most commonly acknowledged are the following. The basic gossip protocol is probabilistic meaning that some rumors may be delivered late, although this occurs with low probability. The expected number of rounds required for delivery in gossip protocols is logarithmic in the number of nodes. Consequently, the latency of gossip protocols is on average higher than can that provided by systems using hardware accelerated solutions like IP Multicast. Finally, gossip protocols support only the weak guarantee of *eventual consistency* — updates may arrive in any order and the system will converge to a consistent state only if updates cease for a period of time. Applications that need stronger consistency guarantees must employ more involved and expensive message passing schemes [39]. We note that weak consistency is not *always* a bad thing. Indeed, relaxing consistency guarantees has become increasingly popular in large-scale industrial applications such as Amazon's Dynamo [47] and Yahoo!'s PNUTS [45].

Gossip also has a less-commonly recognized drawback. An assumption commonly seen in the gossip literature is that all nodes belong to a single gossip *group*. Since such a group will often exist to support an application component, we will also call these *gossip objects*. While sufficient in individual applications, such as when replicating a database [49], an object-oriented style of programming would encourage applications to use multiple objects and hence the nodes hosting those applications will belong to multiple gossip groups. The trends seen in other object oriented platforms (e.g., Jini and .NET) could carry over to

gossip objects, yielding systems in which each node in a data center hosts large numbers of gossip objects. These objects would then contend for network resources and could interfere with one-another. The gossip-imposed load on each node in the network now depends on the number of gossip objects hosted on that node, which violates property P2.

We believe that this situation argues for a new kind of operating system extension focused on nodes that belong to multiple gossip objects. Such a platform can play multiple roles. First, it potentially simplifies the developer's task by standardizing common operations, such as tracking the neighbor set for each node or sending a rumor, much as a conventional operating system simplifies the design of client-server applications by standardizing remote method invocation. Second, the platform can implement fair-sharing policies, ensuring that when multiple gossip applications are hosted on a single node, they each get a fair share of that node's communication and memory resources. Finally, the platform will have opportunities to optimize work across independently developed applications – the main focus of the present chapter. For example, if applications A and B are each replicated onto the same sets of nodes, any gossip objects used by A will co-reside on those nodes with ones used by B. To the extent that the platform can sense this and combine their communication patterns, overheads will be reduced and performance increased.

With these goals in mind, we built a per-node service called the Gossip Objects platform (**GO**) which allows applications to join large numbers of gossip groups in a simple fashion. The initial implementation of **GO** provides a multicast-like interface: local applications can join or leave gossip objects, and send or receive rumors via callback handlers that are executed at particular

rates. Down the road, the **GO** interfaces will be extended to support other styles of gossip protocols, such as the ones listed earlier. In the spirit of property P2, the platform enforces a configurable per-node bandwidth limit for gossip communication, and will reject a join request if the added gossip traffic would cause the limit to be exceeded. The maximum memory space used by **GO** is also limited and customizable.

GO incorporates optimizations aimed at satisfying the gossip properties while maximizing performance. Our first observation is that gossip messages are frequently short: perhaps just a few tens of bytes. Some gossip systems push only rumor version numbers to minimize waste [84, 34], so if the destination node does not have the latest version of the rumor, it can request a copy from the exchange node. An individual rumor header and its version number can be represented in as little as 12-16 bytes. The second observation is that there is negligible difference in operating system and network overhead between a UDP datagram packet containing 10 bytes or 1000 bytes, as long as the datagram is not fragmented [89]. It follows from these observations that *stacking* multiple rumors in a single datagram packet from node s to d is possible and imposes practically no additional cost. The question then becomes: *Which rumors should be stacked in a packet?* The obvious answer is to include rumors from all the gossip objects of which both s and d are members. **GO** takes this a step further: s will sometimes include rumors for gossip objects that d is not interested in, and when this occurs, d will attempt to forward those rumors to nodes that will benefit from them. We formalize rumor stacking and *message indirection* by defining the *utility* of a rumor in section 3.2.

We envision a number of uses for **GO**. Within our own work, **GO** will be

the WAN communication layer for Live Distributed Objects, a framework for abstract components running distributed protocols that can be composed easily to create custom and flexible live applications or web pages [73, 38]. This application is a particularly good fit for **GO**: Live Objects is itself an object-oriented infrastructure, and hence it makes sense to talk about objects that use gossip for replication. The **GO** interface can also be extended to resemble a gossip-based publish/subscribe system [50]. Finally, **GO** could be used as a kind of IP tunnel, with end-to-end network traffic encapsulated, routed through **GO**, and then de-encapsulated for delivery. Such a configuration would convert a conventional distributed protocol or application into one that shares the same gossip properties enumerated earlier, and hence might be appealing in settings where unrestricted direct communication would be perceived as potentially disruptive.

This chapter focuses on the initial implementation of **GO**, and makes the following contributions:

- A natural extension of gossip protocols in which multiple gossip objects can be hosted on each node.
- A novel heuristic to exploit the similarity of gossip groups to improve propagation speed and scalability.
- An evaluation of the **GO** platform on a real-world trace by simulation.

3.2 Gossip Algorithms

3.2.1 Model

Our model focuses on push-style gossip, but can easily be extended to the push-pull or pull-only cases.

Consider a system with a set N of n nodes and a set M of m gossip objects denoted by $\{1, 2, \ldots, m\}$. Each node i belongs to some subset A_i of gossip objects. Let O_j denote *member set* of gossip object j, defined as $O_j := \{i \in N : j \in A_i\}$. We let N_i denote the set of *neighbors* of i, defined as $\bigcup_{j \in A_i} O_j$.

A subset of nodes in a gossip object generate *rumors*. Each rumor r consists of a payload and two attributes: (i) $r.dst \in M$: the destination gossip object for which rumor r is relevant, and (ii) $r.ts \in \mathbb{N}$: the timestamp when the rumor was created. A gossip *message* between a pair of nodes contains a collection of at most L stacked rumors, where L reflects the maximum transfer unit (MTU) for IP packets before fragmentation kicks in. For example, if each rumor has length of 100 bytes and the MTU is 1500 bytes, L is 15.

We will assume throughout this chapter that each node i knows the full membership of all of its neighbors N_i. This assumption is for theoretical clarity, and can be relaxed using peer sampling techniques [57] or remote representatives [82]. The types of applications for which **GO** is appropriate, such as pub-sub systems or Live Objects, will neither produce immensely large groups nor sustain extreme rates of churn.

3.2.2 Random Dissemination

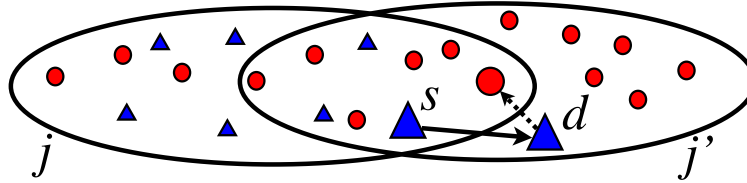
A gossip algorithm has two stages: a *recipient selection* stage and a *content selection* stage [60]. The content is then sent to the recipient. For baseline comparison, we will consider the following straw-man gossip algorithm RANDOM-STACKING running on each node i.

- **Recipient selection:** Pick a recipient d from N_i uniformly at random.
- **Content selection:** Pick a set of L unexpired rumors uniformly at random.

If there are fewer than L unexpired rumors, RANDOM-STACKING will pick all of them. We will also evaluate the effects of rumor stacking; RANDOM is a heuristic that packs only *one* random rumor per gossip message, as would occur in a traditional gossip application that sends rumors directly in individual UDP packets.

3.2.3 Optimized Dissemination

As mentioned earlier, the selection strategy in RANDOM can be improved by sending rumors indirectly via other gossip objects. In the following diagram, a triangle representing a rumor specific to gossip object j is sent from node s to a node d only in j'. Node d in turn infects a node in the overlap of the two gossip objects.



We will define the *utility* of including a rumor in a gossip message, which informally measures the “freshness” of the rumor once it reaches the destination gossip object, such that a “fresh” rumor has higher probability of infecting an uninfected node. If rumor r needs to travel via many hops before reaching a node in $r.dst$, by which time r might be known to most members of $r.dst$, the utility of including r in a message is limited. Ideally, rumors that are “young” or “close” should have higher utility.

Hitting Time

We make use of results on gossip within a single object. Define an *epidemic on n hosts* to be the following process: One host in a fully-connected network of n nodes starts out infected. Every round, each infected node picks another node uniformly at random and infects it.

Definition 1 *Let $S(n,t)$ denote the number of nodes that are* susceptible *(uninfected) after t rounds of an epidemic on n hosts.*

To the best of our knowledge, the probability distribution function for $S(n,t)$ has no closed form. It is conjectured in [49, 59] that $\mathbb{E}[S(n,t)] = n\exp(-t/n)$ for push-based gossip and large n using mean-field equations, and that $\mathbb{E}[S(n,t)] =$

$n\exp(-2^t)$ for push-pull. Here, we will assume that $S(n,t)$ is sharply concentrated around this mean, so $S(n,t) = n\exp(-t/n)$ henceforth. Improved approximations, such as using look-up tables for simulated values of $S(n,t)$, can easily be plugged into the heuristic code.

Definition 2 *The* expected hitting time $H(n,k)$ *is the expected number of rounds in an epidemic on* n *hosts until we infect some node in a given subset of* k *special nodes assuming* $S(n,t)$ *nodes are susceptible in round* t*.*

If a gossip rumor r destined for some gossip object j ends up in a different gossip object j' that overlaps with j, then the expected hitting time roughly approximates how many rounds elapse before r infects a node in the intersection of O_j and $O_{j'}$. Two simplifying assumptions are at work here, first that each node in j contacts only nodes within j in each round, and second that r has high enough utility to be included in all gossip messages exchanged within the group.

Let $p(n,k,t) = 1 - \left(1 - \frac{k}{n}\right)^{n-S(n,t)}$ denote the the probability of infecting at least one of k special nodes at time t when $S(n,t)$ are susceptible. We derive an expression for $H(n,k)$ akin to the expectation of a geometrically distributed random variable.

$$H(n,k) = \sum_{t=1}^{\infty} t p(n,k,t) \prod_{\ell=1}^{t-1} (1 - p(n,k,\ell)),$$

which can be approximated by summing a constant number *max-depth* of terms from the infinite series, and by plugging in $S(n,t)$ from above, as shown in Algorithm 1.

Algorithm 1: $H(n, k, t)$: approximate the expected hitting time of k of n at time t.

if $t \geq$ *max-depth* **then**

  return 1.0 {Prevent infinite recursion.}

end if

$p \leftarrow \exp(\log(1.0 - k/n) \cdot S(n, t)$

return $t \cdot (1.0 - p) + H(n, k, t+1) \cdot p$

Algorithm 2: *Compute-graph*: determine the overlap graph, hitting times and shortest paths between every pair of nodes.

Require: *overlap*$[j][j'] = w(j, j')$ has been computed for all groups j and j'.

for $j \in$ *groups* **do**

  for $j' \in$ *groups* **do**

    if *overlap*$(j, j') > 0$ **then**

      graph$[j][j'] \leftarrow H($*overlap*$(j, j'), j.$*size*$, 0)$

    else

      graph$[j][j'] \leftarrow \infty$

    end if

  end for

end for

Run an all-pairs shortest path algorithm [52] on *graph* to produce *graph-distance*.

Utility

Recall that each node i only tracks the membership of its neighbors. What happens if i receives gossip message containing a rumor r from an unknown gossip object j? To be able to compute the utility of including r in a message to a given neighbor, we will have nodes track the size and the connectivity between every pair of gossip objects. Define an *overlap graph* for propagation of rumors across gossip objects as follows:

Definition 3 *An* overlap graph $G = (M, E)$ *is an undirected graph on the set of gossip objects, and* $E = \{\{j, j'\} \in M \times M : O_j \cap O_{j'} \neq \emptyset\}$. *Define the* weight *function* $w : M \times M \to \mathbb{R}$ *as* $w(j, j') = |O_j \cap O_{j'}|$ *for all* $j, j' \in M$. *Let* $\mathcal{P}_{j,j'}$ *be the set of simple paths between gossip objects* j *and* j' *in the overlap graph* G.

We can now estimate the propagation time of a rumor by computing the expected hitting time on a path in the overlap graph G. A rumor may be diffused via different paths in G; we will estimate the time taken by the *shortest* path.

Definition 4 *Let* $P \in \mathcal{P}_{j,j'}$ *be a path where* $P = (j = p_1, \ldots, p_s = j')$. *The* expected delivery time on P *is*

$$D(P) = \sum_{k=1}^{s-1} H\left(|O_{p_k}|, w\left(p_k, p_{k+1}\right)\right).$$

The expected delivery time *from when a node* $i \in N$ *includes a rumor* r *in an outgoing message until it reaches another node in* $r.dst$ *is*

$$D(i, r) = \min_{j \in A_i} \min_{P \in \mathcal{P}_{j,r.dst}} D(P).$$

Algorithm 2 shows pseudo-code for computing the expected delivery time between every pair of groups.

We can now define a utility function U to estimate the benefit from including a rumor r in a gossip message.

Algorithm 3: $U_s(d, r, t)$: utility of sending rumor r from s to d at time t.

Require: *compute-graph* must have been run.

distance $\leftarrow \infty$

for $j \in d.groups$ **do**

distance $\leftarrow \min\{distance, graph\text{-}distance[j][r.dst]\}$

end for

if *distance* $= \infty$ **then**

return 0.0

end if

return $S(j.size, t - r.ts + dist)/j.size$

Definition 5 *The* utility $U_s(d, r, t)$ *of including rumor r in a gossip message from node s to d at time t is the expected fraction of nodes in gossip object $j = r.dst$ that are still susceptible at time $t' = t - r.ts + D(s, r)$ when we expect it to be delivered. More precisely,*

$$U_s(d, r, t) = \frac{S(|O_j|, t')}{|O_j|}.$$

Pseudo-code for approximating the utility function is shown in Algorithm 3. The code is optimized by making use of the overlap graph computed by Algorithm 2.

Algorithm 4: *Sample*(u, R, L): sample L rumors without replacement from R with probability proportional to u.

```
S ← ∅ {The set of rumors in the sample}
sum ← Σ_{r∈R} u(r)
Let r_1, r_2, ..., r_k be a random permutation of R.
z ← random(0, 1) {Uniformly random number in [0, 1)}
ζ ← 0
for ℓ = 1 to k do
  ζ ← ζ + u(r_ℓ) · L/sum
  if ζ ≥ z then
    S ← S ∪ {r_ℓ} and ζ ← ζ − 1.0
  end if
end for
return S
```

The GO Heuristic

The following code is run by client on node s at time t.

- **Recipient selection:** Pick a recipient d uniformly at random from N_s.
- **Content selection:** Let R denote the set of unexpired rumors. Calculate the utility $u(r) = U_s(d, r, t)$ for each $r \in R$ using Algorithm 3. Call *Sample*(u, R, L) (Algorithm 4) to pick L rumors at random from R so that the probability of including rumor $r \in R$ is proportional to its utility $u(r)$.

Algorithm 4 for sampling without replacement while respecting probabilities on the elements may be of independent interest. We include it here without proof for the curious reader.

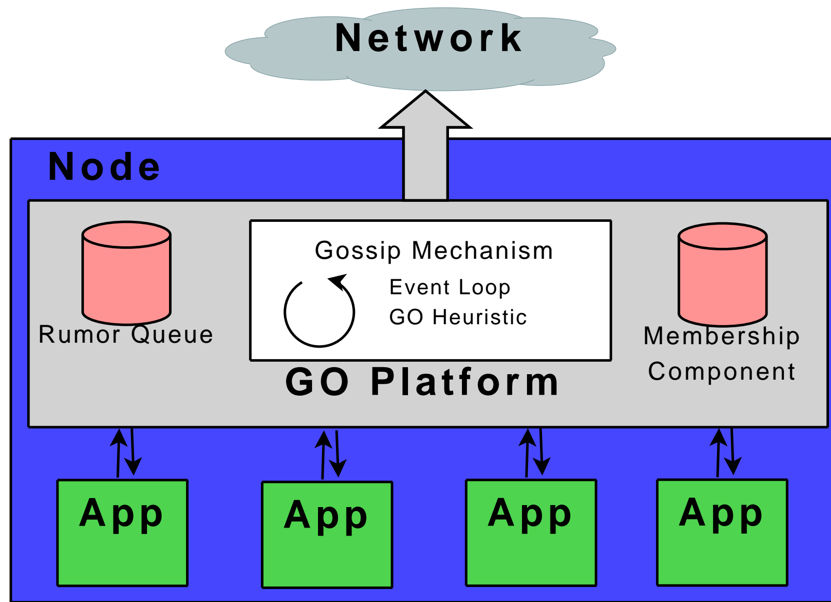


Figure 3.1: The **GO** Platform.

In order to compute the utility of a rumor, each node needs to maintain complete information about the overlap graph and the sizes of gossip objects. We describe the protocol that maintains this state in section 3.3.3.

The cost of storing and maintaining such a graph may become prohibitive for very large networks. We intend to remedy this potential scalability issue by maintaining only a *local view* of the transition graph, based on the observation that if a rumor belongs to distant gossip object with respect to the overlap graph, then its utility is automatically low and the rumor could be discarded. Evaluating the trade-off between the view size and the benefit that can be achieved by the above optimizations is a work in progress.

Consider the content selection policies for the RANDOM-STACKING and the **GO** heuristic. A random policy will often include rumors in packets that have no chance of being useful because the recipient of the packet has no "route" to the group for which the rumor was destined. **GO** will not make this error: if it

includes a rumor in a packet, the rumor has at least some chance of being useful. We evaluate the importance of this effect in section 3.4.

3.2.4 Traffic Rates and Memory Use

The above model can be generalized to allow gossip objects to gossip at different *rates*. Let λ_j be the rate at which new messages are generated by nodes in gossip object j, and R_i the rate at which the **GO** platform gossips at node i.

For simplicity, we have implicitly assumed that all platforms gossip at the same fixed rate R, and that this rate is "fast enough" to keep up with all the rumors that are generated in the different gossip objects. Viewing a gossip object as a queue of rumors that arrive according to a Poisson process, it follows from Little's law [64] that the average rate at which node i sends and receives rumors, R_i, cannot be less than the rate λ_j of message production in j if rumors are to be diffused to all interested parties in finite time with finite memory. In the worst case there is no exploitable overlap between gossip objects, in which case we require R to be at least $\max_{i \in N} \sum_{j \in A_i} \lambda_j$. Furthermore, the amount of memory required is at least $\max_{i \in N} \sum_{j \in A_i} \mathcal{O}\left(\log |O_j|\right) \lambda_j$ since rumors take logarithmic time on average to be disseminated within a given gossip object.

The **GO** platform enforces customizable upper bounds on both the memory use and gossip rate (and hence bandwidth), rejecting applications from joining gossip objects that would cause either of these limits to be violated. Rumors are stored in a priority queue based on their maximum possible utility; if the rumors in the queue exceed the memory bound then the least beneficial rumors are discarded.

3.3 Platform Implementation

As noted earlier, **GO** was implemented using Cornell's Live Distributed Objects technology, and inherits many features from the Live Objects system. For reasons of brevity, we limit ourselves to a short summary. Each **GO** application runs as a small component, coded in any of the 40 or so languages supported by Microsoft .NET, and implements a standard interface defined by the Live Objects framework. At run-time, an "end user" application can link to **GO** applications through simple library interfaces. Moreover, gossip objects can be composed into graphs, with one object talking to another through typed endpoints over which events are passed. The resulting architecture is rich, flexible, and quite easy to extend.

The **GO** platform runs on all nodes in the target system, and currently supports applications via an interface focused on group membership and multicast operations. The platform consists of three major parts: the membership component, the rumor queue and the gossip mechanism, as illustrated in figure 3.1.

GO exports a simple interface to applications. Applications first contact the platform via a client library or an IPC connection. An application can then `join` (or `leave`) gossip objects by providing the name of the group, and a poll rate R. Note that a `join` request might be rejected. An application can start a rumor by adding it to an outgoing rumors queue which is polled at rate R (or the declared poll rate in the gossip object) using the `send` primitive. Rumors are received via a `recv` callback handler which is called by **GO** when data is available.

Rumors are garbage collected when they expire, or when they cannot fit in memory and have comparatively low utility to other rumors as discussed in

section 3.2.4.

3.3.1 Bootstrapping

We bootstrap gossip objects using a rendezvous mechanism that depends upon a directory service (**DS**), similar to DNS or LDAP. The **DS** tracks a random subset of members in each group, the size of which is customizable. When a **GO** node i receives a request by one of its applications to join gossip object j, i sends the identifier for j (a string) to the **DS** which in turn returns a random node $i' \in O_j$ (if any). Node i then contacts i' to get the current state of gossip object j: (i) the set O_j, (ii) full membership of nodes in O_j, and (iii) the subgraph spanned by j and its neighbors in the overlap graph G along with weights. If node i is booting from scratch, it gets the full overlap graph from i'.

3.3.2 Gossip Mechanism

GO's main loop runs periodically, receives gossip messages from other messages and performing periodic upcalls to applications, which may react by adding rumors to the *rumor queue*. Each activity period ends when the platform runs the **GO** heuristic (from section 3.2.3) to send a gossip message to a randomly chosen neighbor. The platform then discards old rumors.

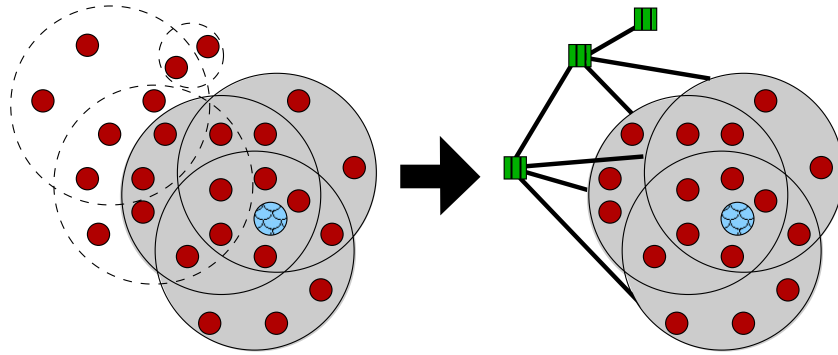


Figure 3.2: Membership information maintained by **GO** nodes. The topology of the whole system on the left is modeled by the node in center as (i) the set of groups to which it belongs and neighbor membership information (local state), and (ii) the overlap graph for other groups, whose nodes are depicted as squares and edges are represented by thick lines (remote state).

3.3.3 Membership Component

Each **GO** node i maintains the membership information for all of its neighbors, N_i (*local* state). It also tracks the overlap graph G and gossip group sizes (*remote* state), as discussed in section 3.2. Figure 3.2 illustrates an example of system-wide group membership (left) and the local and remote state maintained by the center node (right). The initial implementation of **GO** maintains both pieces of state via gossip.

Remote state

After bootstrapping, all nodes join a dedicated gossip object j^* on which nodes exchange updates for the overlap graph. Let P be a global parameter that controls the rate of system-wide updates, that should reflect both the anticipated level of churn and membership changes in the system, and the $\mathcal{O}(\log n)$ gossip

dissemination latency constant. Every $P \log |O_j|$ rounds, some node i in j starts a rumor r in j^* that contains the current size of O_j and overlap sizes of O_j and j's neighboring gossip objects. The algorithm is leaderless and symmetric: each node in O_j starts their version of rumor r with probability $1/|O_j|$. In expectation, only one node will start a rumor in j^* for each gossip object.

Local state

GO tracks the time at which each neighboring node was last heard from; a node that fails will eventually be removed from the membership list of any groups to which it belongs. When node i joins or changes its membership, an upcall is issued to each gossip object in A_i as a special system rumor. We rate-limit the frequency of membership changes by allowing nodes to only make special system announcements every P rounds.

3.3.4 Rumor Queue

As mentioned in section 3.2.4, **GO** tracks a bounded set of rumors in a priority queue. The queue is populated by rumors received by the gossip mechanism (remote rumors), or by application requests (local rumors). The priority of rumor r in the rumor queue for node s at time t is $\max_{d \in N_i} U_s(d, r, t)$, since rumors with lowest maximum utility are least likely to be included in any gossip messages. As previously discussed, priorities change with time so we speed up the recomputation by storing the value of $\text{argmax}_{d \in N_i} D(s, r)$.

3.4 Evaluation

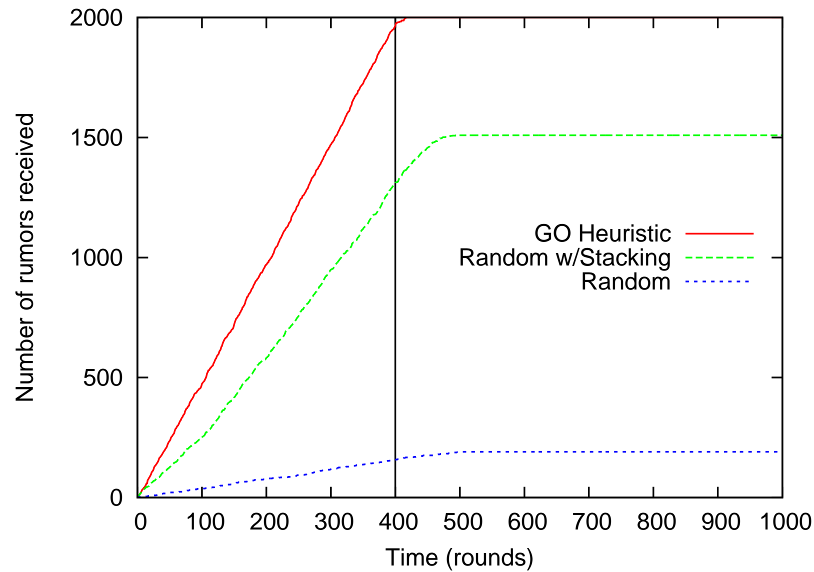
We evaluate the **GO** platform using a discrete time-based simulator. The focus of our experiments is on quantifying the effectiveness of **GO** in comparison to implementations in which each gossip object runs independently without any platform support at all.

Our first experiment explores the usefulness of rumor stacking, and evaluates the benefits of computing utility for rumors. We compare the three different gossip algorithms (the **GO** heuristic, RANDOM and RANDOM-STACKING) running in a simple topology.

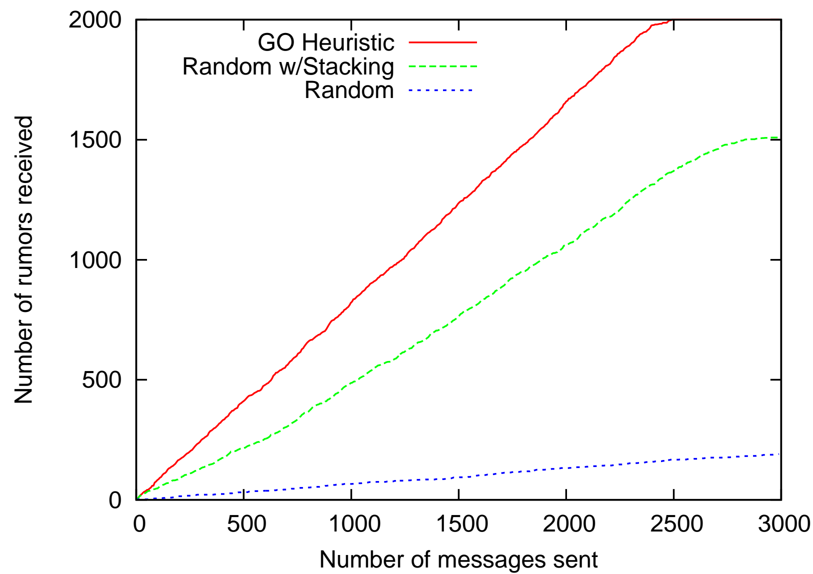
We then evaluate **GO** on a trace of a widely deployed web-management application, IBM WebSphere. This trace shows WebSphere's patterns of group membership changes and group communication in connection with a whiteboard abstraction used heavily by the product, and thus is a good match with the kinds of applications for which **GO** is intended. We provide a detailed exposition of the WebSphere trace and its connectivity patterns in section 4.3.1, as well as other topologies relevant to **GO**.

3.4.1 Rumor Stacking and Message Indirection

We evaluated the benefits of message indirection used by the **GO** heuristic using the topology shown in figure 3.4. The scenario constitutes a group j that contains nodes s and d in which s sends frequent updates for d. Both nodes also belong to a number of other gossip objects that overlap, so that they share some set of common neighbors, in this case four. Assuming the **GO** platform at s only



(a)



(b)

Figure 3.3: Rumor Stacking and Indirection. Different heuristics running on the **GO** platform over the topology from figure 3.4. The plots show the number of new rumors received by nodes in the system over time (a) and as a function of messages sent (b). A vertical line is drawn at the time when all 2,000 rumors have been generated.

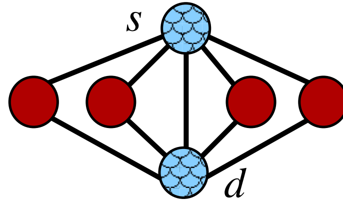


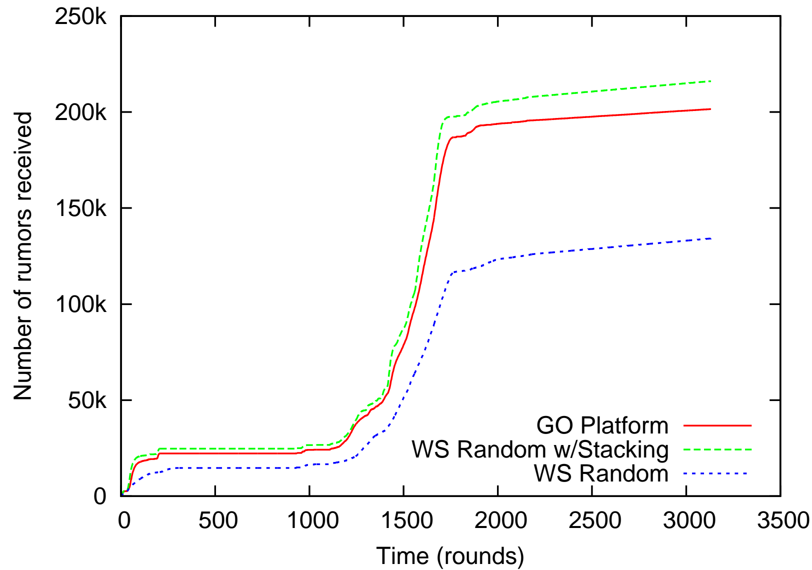
Figure 3.4: The topology used in first experiment. Each edge corresponds to a gossip group, the members of which are the two endpoints.

sends one gossip message per round, the shared neighbors are in a position to propagate messages intended for other gossip objects.

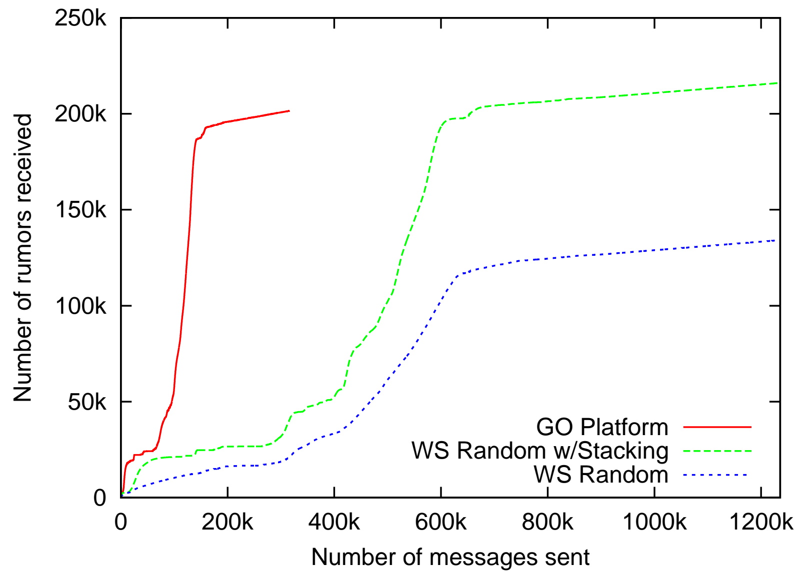
We measured the speed of propagation of messages in group j using our simulator. All nodes simulate the **GO** platform with a message rate of 1 message per round, using one of the three gossip algorithms discussed earlier. During each time step until time 400 (vertical line), node s generates a new rumor for each group in A_s, after which rumor generation stops. We assume that 15 rumors can be stacked in each packet, and that nodes can fit at most 100 rumors in memory.

Figure 3.3 shows the total number of distinct rumors node d has received for group j. The benefits of rumor stacking are evident when one compares the results of the RANDOM-STACKING algorithm to the RANDOM one. RANDOM-STACKING diffuses rumors more than 5 times faster than the single-message RANDOM.

Next, compare the **GO** heuristic results to those of the RANDOM-STACKING algorithm. The **GO** heuristic delivers rumors efficiently: nodes are on average only 11.5 rumors behind an optimal delivery, compared to 460 for RANDOM-STACKING and 1,460 for RANDOM.



(a) WebSphere, new rumors vs. number of messages.



(b) WebSphere, ratio of new rumors per message vs. time.

Figure 3.5: WebSphere trace: The number of new rumors received by nodes in the system and the number of messages sent (a), also plotted as a ratio of new rumors per message over time (b). The nodes using the random heuristics gossip per-group every round, whereas **GO** sends a single gossip message.

3.4.2 Real-World Scenarios

As noted earlier, IBM WebSphere [7] is a widely deployed commercial application for running and managing web applications. A WebSphere cell may contain hundreds of servers, on top of which application clusters are deployed. Cell management, which entails workload balancing, dynamic configuration, inter-cluster messaging and performance measurements, is implemented by a form of built-in whiteboard, which in turn interfaces to the underlying communication layer via a pub-sub [50] interface. To obtain a trace, IBM deployed 127 WebSphere nodes constituting 30 application clusters for a period of 52 minutes, and recorded topic subscriptions as well as the messages sent by every node. An average process subscribed to 474 topics and posted to 280 topics, and there were a total of 1,364 topics with at least two subscribers and at least one publisher. The topic membership is strongly correlated, in fact 26 topics contain at least 121 of the 127 nodes. On the other hand, none of the remaining topics contained more than 10 nodes. Further details about the WebSphere trace and its connectivity patterns are in section 4.3.1.

We used the WebSphere trace to drive our simulation by assigning a gossip group to each topic. All publishers and subscribers for the topic are members of the corresponding gossip group. We limited the memory and bandwidth requirements by expiring rumors 100 rumors after they were first generated. Again, we compare the **GO** heuristic with RANDOM and RANDOM-STACKING. However, in contrast to the experiment of section 3.4.1, in which the **GO** platform itself used the specified stacking policy, this WebSphere experiment is slightly different: it compares a simulated "port" of WebSphere to run over **GO** with a simulation of WebSphere running over independent gossip groups that

exhibit the same membership and communication patterns, but do not benefit from any form of platform support. To emphasize that these group policies are not identical to RANDOM-STACKING and RANDOM, as used internally by the **GO** platform itself in the first experiment, we designate the policies as WS-RANDOM-STACKING and WS-RANDOM in what follows.

We expect the naïve approaches to disseminate rumors faster than **GO** because each WebSphere group is operated independently and in a "greedy" manner. As a consequence, each node sends one gossip message per group per round, as opposed to only one message per round as the **GO** platform does. As can be seen in figure 3.5(a), the delivery speed of the **GO** platform is 6.7% percent lower on average than the naïve WS-RANDOM-STACKING approach. **GO**, however, beats WS-RANDOM by a factor of 2. An even bigger win for **GO** can be seen in figure 3.5(b), which shows the number of new rumors delivered versus the number of messages exchanged. The **GO** platform sends 3.9 times fewer messages than the naïve approaches, thus keeping bandwidth bounded, while disseminating rumors almost as fast.

At the end of the trace, the total number of rumors received by all nodes was 8% lower when using **GO** than WS-RANDOM-STACKING, meaning that some rumors had not reached all intended recipients. We traced this loss to a specific point in the execution at which WebSphere generates a burst of communication, exceeding the **GO**-imposed bandwidth limit. One reasonable inference is that such loss is an unavoidable consequence of our approach, in which a single platform handles communication on behalf of all gossip groups. However, it is interesting to realize that the WebSphere traffic burst was brief and that averaged over even a short window, need not have overwhelmed **GO**. This ob-

servation is motivating us to explore dynamically adjusting the platform gossip rate to cope with bursty senders, but in ways that would still respect operator-imposed policies over longer time periods.

3.4.3 Discussion

There are two take-away messages from the first experiment. First, rumor stacking is inherently useful even when using RANDOM-STACKING without a utility-driven rumor selection scheme. Nonetheless, we see a substantial gain when using the **GO** heuristic to guide the platform's stacking choices. Although not reported here, we have conducted additional experiments that confirm this finding under a wide range of conditions. Second, if processes exhibit correlated but not identical group membership, then there may often be indirect paths that can be exploited using message indirection. **GO** learns these paths by exploring membership of nearby groups, and can then ricochet rumors through those indirectly accessible groups. The RANDOM-STACKING policy lacks the information needed to do this. While the topology in the first experiment is deliberately adversarial, it is also extremely simple. For this reason, we believe that patterns of this sort may be common in the wild, where correlated group membership is known to be a pervasive phenomenon.

The WebSphere experiment supports our belief that the **GO** platform is able to cope with real-world message dissemination at a rate close to that of a naïve implementation without losing the fixed bandwidth guarantee discussed in the introduction, and in fact using substantially fewer messages than a non-platform approach.

We believe that the scenarios we evaluated illustrate the potential benefit of the **GO** methodology in a reasonably general way. If a large number of groups overlap at a single node, conditions could arise that would favor the **GO** heuristic to an even greater degree than in our examples. For example, this would be the case if a large number of groups overlap, generating high volumes of gossip traffic, and yet the pattern of membership is such that relatively few rumors are legitimate candidates for stacking in any particular gossip message. **GO** has the information to optimize for such cases, including only high-value rumors; random stacking would tend to fill packets with useless content, missing the opportunity.

3.5 Future Directions

Recall that **GO** nodes maintain membership of all groups to which they belong. To address scalability concerns, large groups can likely be fragmented at a cost of higher latency.

In ongoing work, we are changing the **GO** membership algorithm to bias it in favor of accurate *proximal* information at the expense of decreased accuracy about membership of remote groups. The rationale for this reflects the value of of having accurate information in the utility computation. As observed earlier, rumors have diminishing freshness with time, which also implies that the expected utility of routing a rumor very indirectly is low. In effect, a rumor sent indirectly still needs to reach a destination quickly if it is to be useful. We conjecture that the **GO** heuristic can be proved to be insensitive to information about groups and membership very remote (*i.e.*, several hops from a sender node),

but highly sensitive to what might be called proximal topology information. It would follow that proximal topology suffices.

At present, **GO** rejects gossip join requests if the resulting additional gossip load would overflow its rumor buffers. One might imagine a more flexible scheme that would allocate rumor buffer space among applications in an optimized manner, so as to accommodate applications with varied data production rates. If we then think about information flow rates within individual groups, and compare this with those achievable using the **GO** (where groups carry traffic for one-another), it would be possible to demonstrate an increase in the peak data rates when using **GO** relative to systems that lack this cooperative behavior.

A second direction for future investigation concerns other potential uses for **GO**. As noted earlier, our near term plan is to extend **GO** so that it can support a wider range of gossip styles. Beyond this, we are considering hosting non-gossip protocols "over" **GO**, tunneling their communication traffic through **GO** so as to gain the properties of those protocols (such as consistency, tolerance of application-level Byzantine faults, *etc.*) while also benefiting from **GO**'s simple worst-case communication loads.

Yet a third open topic concerns security. The **GO** rumor stacking scheme does not currently provide true performance isolation: an aggressive application may be able to dominate a less aggressive one, seizing an unfair share of stacking space. A thorough exploration of this form of fairness, and of other security issues raised by **GO**, would represent an appealing subject for further study.

In summary, **GO** is a work in progress. While gossip protocols for individual applications are a relatively mature field, it is interesting to realize that by building a platform – an operating system – to support multiple gossip applications, one encounters such a wide range of challenging problems. We conjecture that practitioners who use gossip aggressively will encounter these problems too, and that in the absence of good solutions, might conclude that gossip is not as effective a technology as generally believed. Yet there seems to be every reason to expect that these problems can be solved. By doing so we advance the theory, while also enlarging the practical utility of gossip in large data centers and WAN peer-to-peer settings, where gossip seems to be a good fit to the need.

3.6 Related Work

The pioneering work by Demers *et al.* [49] used gossip protocols to enable a replicated database to converge to a consistent state despite node failures or network partitions. The repertoire of systems that have since employed gossip protocols is impressive [34, 84, 82, 50, 47, 76], although most work is focused on application-specific use of gossip instead of providing gossip communication as a fundamental service.

3.7 Conclusion

The **GO** platform generalizes gossip protocols to allow them to join multiple groups without losing the appealing fixed bandwidth guarantee of gossip protocols, and simultaneously optimizing latency in a principled way. Our heuris-

tic is based on the observations that a single IP packet can contain multiple rumors, and that indirect routing of rumors can accelerate delivery. The platform has been implemented, but remains a work in progress. Our vision is that **GO** can become an infrastructure component in various group-heavy distributed services, such as a robust multicast or publish-subscribe layer, and an integral layer of the Live Distributed Objects framework.

CHAPTER 4

AFFINITY

In this chapter, we investigate the level of *affinity* present in a number of different data sets and models, by which we mean a high degree of pairwise overlap between groups. Group affinity depends on what a *group* is supposed to abstract, and the underlying process for how these groups are populated by users or processes.

We will consider two general categories of group abstractions.

- **Social interactions:** On the one hand, we have data sets in which groups contain *real people*, and affinity between groups is determined by processes that depend on the attributes of the users, such as interests, preferences, social interactions, and so forth.
- **System communication channels:** On the other hand, we consider data sets in which a group denotes a *communication channel of a system*, and affinity between groups is driven by properties of the system. For instance, a system that replicates state across different components could create a group to handle the replication, and is thus bound to have groups with similar or identical membership.

We create two models, one for each category of group abstractions, each of which capture some statistics or process believed to influence group membership among people or in real systems. From a scientific standpoint, models like these allow us to get insights by exploring fundamental processes underlying reality in isolation. From an engineering standpoint, these models provide a

means to experiment with a system at arbitrary scales and to identify opportunities for improvement within the system.

Understanding group affinity has clear practical benefits. The optimizations used in both Dr. Multicast (MCMD) and the **GO** platform depend on the amount of overlap between groups. In MCMD, merging groups with similar membership into single physical IPMC group saves the limited IPMC resources at the cost of filtering unwanted packets. In **GO**, indirection opportunities directly correlate with the size of the overlap between pairs of groups. Opportunities to exploit group similarity arise in various other types of work, such as general publish-subscribe systems [50], dissemination overlays [56] and deduplication of Internet traffic [30, 29].

Our road map for this chapter is as follows. We will first explore the properties and perform some analysis of the data sets and models for social interactions in sections 4.1 and 4.2, and for system communication channels in sections 4.3 and 4.4. We then investigate the level of affinity present in the different data sets and models in section 4.5, posing and answering the question "*How random are group overlaps?*". We will formalize the optimization problem present in MCMD,and derive and evaluate a heuristic for it on the data sets and models in sections 4.6. We will discuss some related work in section 4.7, and finally conclude in section 4.8.

The contributions of this chapter are the following.

- We present and analyze data sets and models for group overlap for both social groups and systems communication channels.
- We investigate the level of affinity present in these data beyond what

would exist by chance.

- We formalize the optimization problems in MCMD, and evaluate a novel algorithm on the data sets and the models.

4.1 Social Data Sets

We obtained a number of real-world data sets in which human interactions can be abstracted as groups and users. The data sources are Yahoo! Groups users, Amazon.com product recommendations, Wikipedia editors and LiveJournal communities. For each data set, we will produce a *bipartite graph* between the sets of groups and users in which group membership is denoted by an edge between a user and a group.

Understanding the patterns of overlap in these data sets is useful to systems design, since all systems ultimately interface with humans at some level. Trends or attributes associated with human behavior at higher levels of a system will trickle down to the system's lower layers, where it influences the communication patterns of the components. For instance, the PNUTS data storage system at Yahoo! [45] has a publish/subscribe layer that maintains and replicates state geographically. If PNUTS were used to store Yahoo! Groups, then the communication patterns of the publish/subscribe layer would be heavily influenced by the human behavior evident in the data set.

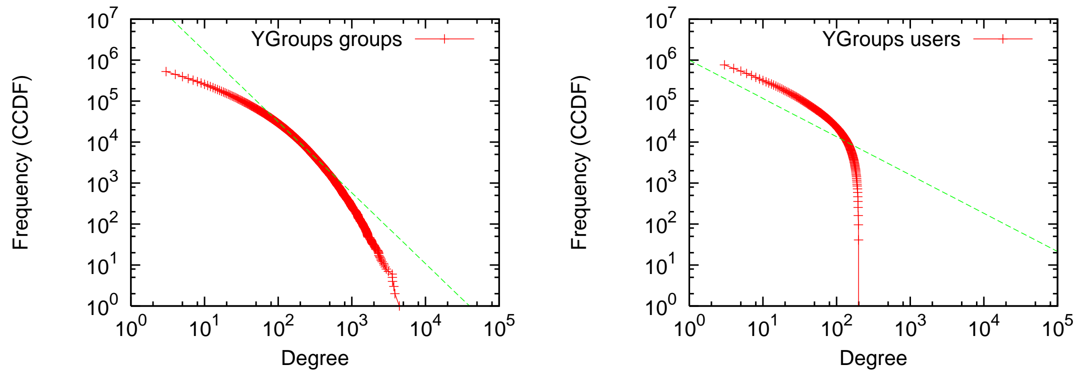


Figure 4.1: Y-GROUPS: complementary CDF for group size (left) and user degree (right).

4.1.1 Yahoo! Groups

Yahoo! Groups is an on-line community-driven forum [24]. Members can subscribe to groups and receive posts and updates to those groups via e-mail or using a web interface. Groups tend to be either discussion venues or announcement lists, and some groups are moderated.

The Y-GROUPS data set contains 640,000 groups and 1 million users and edges corresponding to group membership [21]. There are 15 million edges in the graph.

Unfortunately, figure 4.1 indicates that the trace does not contain any groups with more than 200 members. The cap may stem from artificial limits imposed when the trace was gathered.

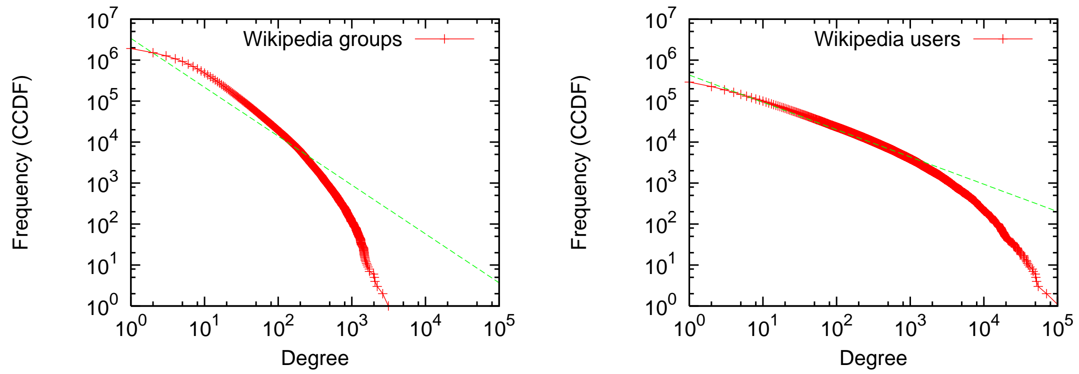


Figure 4.2: WIKIPEDIA: complementary CDF for group size (left) and user degree (right).

4.1.2 Wikipedia Editors

Wikipedia [23] is an on-line encyclopedia that can be edited freely by anyone. We consider the edit history of all articles on Wikipedia by registered users through April 1st, 2007 [46]. The edits performed by unregistered users (*i.e.*, anonymous IP addresses) were discarded. The nodes of the bipartite Wikipedia graph WIKIPEDIA constitutes 430,000 registered editors and 3.4 million articles, and we place an edge between an editor and an article if the editor has ever edited the article. There are 23 million edges in the Wikipedia graph.

It should be noted that the users who edited the greatest number of articles are robots, programs made to automatically adjust the style and references of Wikipedia pages, and to quickly revert obvious vandalism.

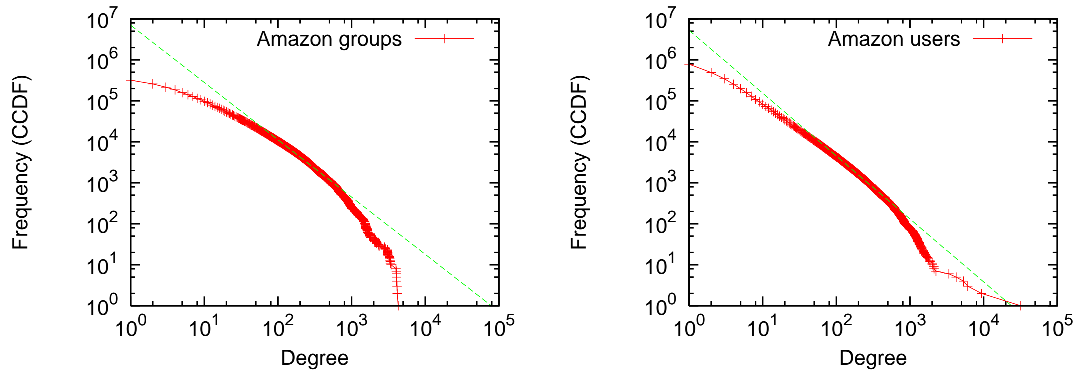


Figure 4.3: AMAZON: complementary CDF for product degree (left) and user degree (right).

4.1.3 Amazon.com Recommendations

Amazon.com is a large, popular on-line retailer for books, DVDs, electronics, apparel, and more [22]. Customers can rate and write reviews for the items they purchase, conveying information about product quality and service to other potential customers. The bipartite graph AMAZON of product reviews contains between 400,000 products and reviews by over 1.5 million users through July 2005 [63]. The data set contains 64 million reviews, represented by edges in the graph.

4.1.4 LiveJournal Communities

LiveJournal is a large on-line web-site where users can keep an electronic diary, blog or journal. Users can create "communities" and identify to which of those communities they belong. Joining a community gives a user the right to write new posts in that community and the access to other people's posts is improved.

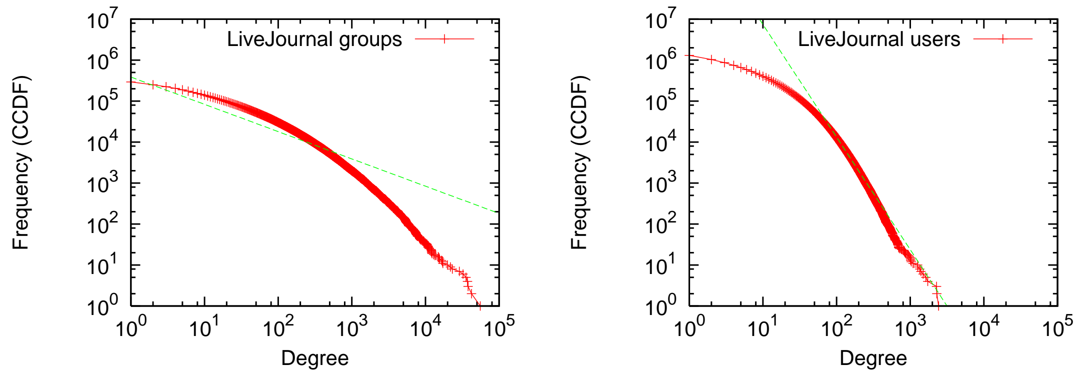


Figure 4.4: LIVEJOURNAL: complementary CDF for community size (left) and user degree (right).

We obtained a snapshot from 2005 [32] of 390,000 communities populated by 1.9 million self-identified users. We processed the snapshot to produce a bipartite graph LIVEJOURNAL of edges between users to communities which contains 16.9 million edges.

4.2 Modeling Social Interactions

We wish to create a model for how humans join groups such that we match important statistics observed in real-world data such as the data sets presented above. Let us begin by describing a common pattern that arises in social data, namely power-laws.

4.2.1 Power-law Distributions

A random variable X taking integer values $1, 2, 3, \ldots$ follows a *power-law* distribution if $\Pr[X = k] \propto k^{-\alpha}$ for some constant $\alpha > 0$. The distribution is *heavy-tailed*, meaning that tail is heavier than the exponential distribution (*i.e.*, $\int_0^\infty x^\beta \Pr[X = x]dx$ diverges for some $\beta > 0$).

Power-law distributions are ubiquitous in nature [71, 72, 66], where the value of α is usually observed to be between 2.0 and 3.0 with few exceptions [44]. Power-law distributions have also been observed in various social graphs, for example the the number of routers with a given degree k in the inter-domain topology on the Internet is roughly $k^{-2.48}$ [51], the fraction of web pages with k in-links is roughly $k^{-2.1}$ [61, 36], and the popularity of RSS feeds also follows a power-law with $\alpha = 2.37$ [65]. In fact, there was a "power-law craze" for a while during which people even looked at the social network of Marvel comic characters [26].

Letting $y = cx^{-\alpha}$ for some constant c, we see that $\log y = -\alpha \log x + \log c$ is linear and thus the PDF of a power-law distribution has a linear fit on a log-log plot. The complement of the CDF is also linear in log-log space for the same reason, since

$$\Pr[X \geq x] = \int_x^\infty cx^{-\alpha}dx = \left[\frac{c}{-\alpha}x^{-(\alpha-1)}\right]_{x=x}^\infty = \frac{c}{\alpha}x^{-(\alpha-1)}.$$

Consider the complementary CDF for the social data sets from WIKIPEDIA, AMAZON and LIVEJOURNAL presented above in figures 4.2, 4.3 and 4.4. For one to two orders of magnitude on the x-axis, the curves appear to follow a power-law (the fitted lines) decently, and then have an exponential drop-off. What might generate this behavior for such different types of data?

4.2.2 Mutual Interest Model

Much research effort has been put into understanding what causes power-laws, and why they arise in so many different domains [66]. One of the best known generative models for graphs with power-laws degree distributions is the *preferential attachment* model by Barabási and Albert [36]. In the model, a graph is constructed by gradually introducing new nodes, and each new node builds an edge to an earlier node with probability proportional to that node's degree. The idea behind the model is that "the rich get richer". The number of vertices with degree k is roughly proportional to k^{-3} [71]. An exponent between 2 and 3 can be produced by an algorithm which mixes between a preferential attachment phase and a phase in which nodes connect to other nodes uniformly at random [27].

We devised what we call the MUTUAL-INTEREST model, based on preferential attachment processes, to generate group memberships such that both group and user popularity follow power-laws as suggested by the data sets. The original preferential attachment model produces graphs with a single type of nodes, whereas we are interested in generating a bipartite graph with users and groups.

We start with a single group with one user, and then repeat the following until the desired number of users and groups is reached. The probability parameters p and q respectively control the density of the graph, and the desired fraction of users to groups.

- Pick a user u among current users with probability proportional to their degree (number of group memberships) as in the preferential attachment model.

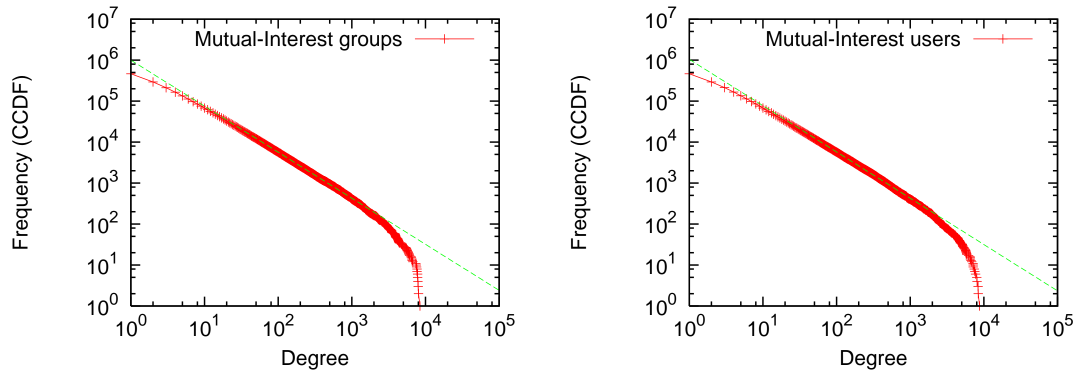


Figure 4.5: MUTUAL-INTEREST: complementary CDF for group size (left) and user degree (right).

- Similarly, pick a group g among current groups with probability proportional to the group size (number of users).
- With probability pq, add a new user u' and make u' join g.
- With probability $p(1-q)$, add a new group g', and make u join g'.
- Otherwise, with probability $1-p$, make u join g.

The intuition is that users prefer popular groups, and new groups are more likely to interest those with many interests.

The MUTUAL-INTEREST model is composition of several processes, so even though it has been proven that the preferential attachment model produces a power-law [71], this proof does not carry over to the MUTUAL-INTEREST model unchanged. However, in figure 4.5 we see that a run of the model with density $p = \frac{1}{10}$ and $q = \frac{1}{2}$ generates a graph which produces good fits to power-laws for both user and group popularity.

4.3 Systems Data Set

Understanding the social aspects of group subscriptions is helpful to systems design because ultimately humans influence the way systems are used. However, the systems we consider in this thesis, Dr. Multicast and **GO**, are designed to accommodate and improve *existing* systems. As such, understanding of the communication patterns of real-world systems has more relevance and applicability to our optimization algorithms than speculations about hypothetical systems based on the social data sets presented before.

Unfortunately, obtaining traces from real-world applications is not an easy task. The key sources for such information are usually industry players who face intense pressure on not releasing data outside the company due to competition and privacy concerns, and because the financial return of their investment may be limited.

However, we did obtain a trace of the communication patterns of a large industrial system, IBM WebSphere. We have already used this trace for evaluation of our systems in chapters 2 and 3. In what follows, we describe the details of the trace and the bipartite graph of its group membership.

4.3.1 WebSphere Bulletin Boards

We obtained a trace from IBM WebSphere Application Server, a popular commercial distributed system for managing web applications [7]. As we mentioned for the evaluation of **GO** in section 3.4, IBM deployed 127 WAS processes constituting 30 application clusters for a period of 52 minutes in January 2009,

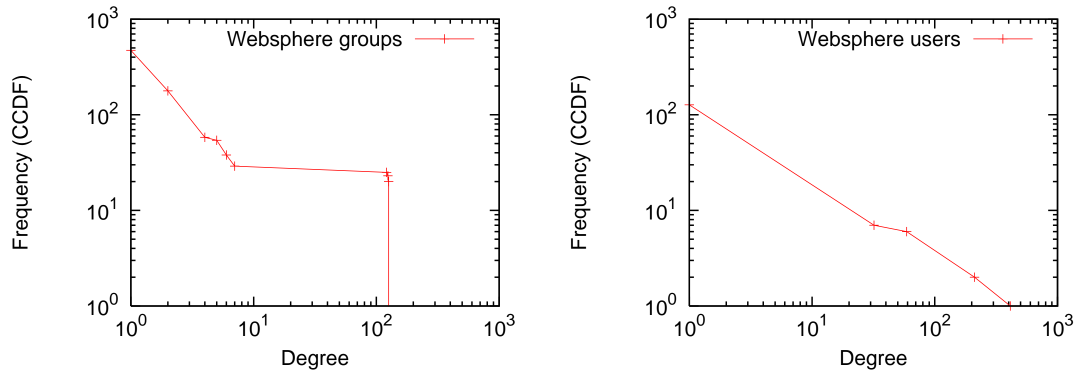


Figure 4.6: WEBSPHERE: complementary CDF for group size (left) and user degree (right).

and recorded the publications and group subscriptions for each process. There were 2,886 logical groups with both subscribers and publishers, and 1,364 of these groups were used to disseminate messages during the tracing period.

We produced a bipartite graph WEBSPHERE using the trace by including all *subscriptions* to the non-idle groups even if processes later decided to unsubscribe. Note that the graph does not contain information about publishers. The graph contains 5,789 subscriptions by 127 processes to 1,364 logical groups.

The publishing and subscription matrices of processes to logical groups in figure 4.3.1 show that interests are highly structured. The spatial distribution of logical groups according to their number of subscribers (x-axis) and publishers (y-axis) in figure 4.8 suggests that there are four types of communication patterns in the trace: few-to-few (F2F), few-to-many (F2M), many-to-few (M2F), and many-to-many (M2M). Here, *few* means no more than 10 processes, and *many* implies all 127 processes except at most 10.

Interestingly, each group in the trace fits one of the four categories. Some of

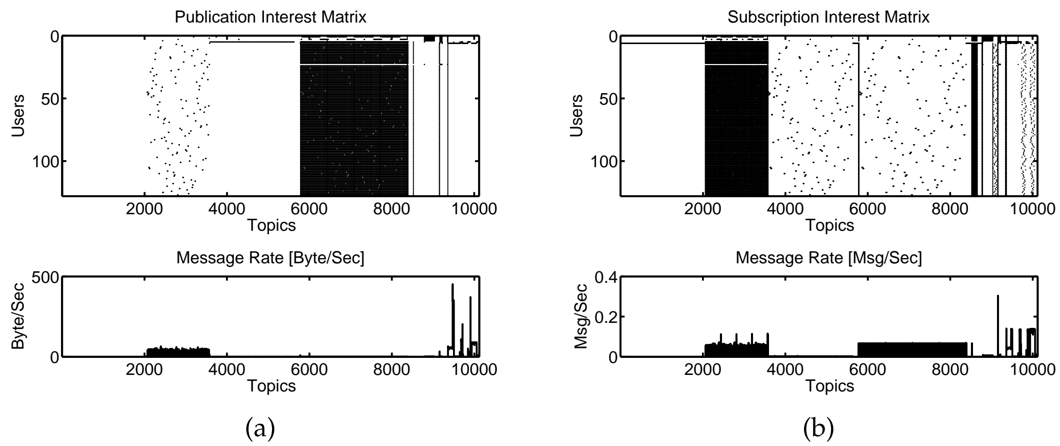


Figure 4.7: WEBSPHERE: Publication (a) and subscription (b) matrices. Each dot represents a subscriber or publisher on a specific group. The traffic rate (bottom) on groups in the trace is expressed both in messages/sec (left) and bytes/sec (right).

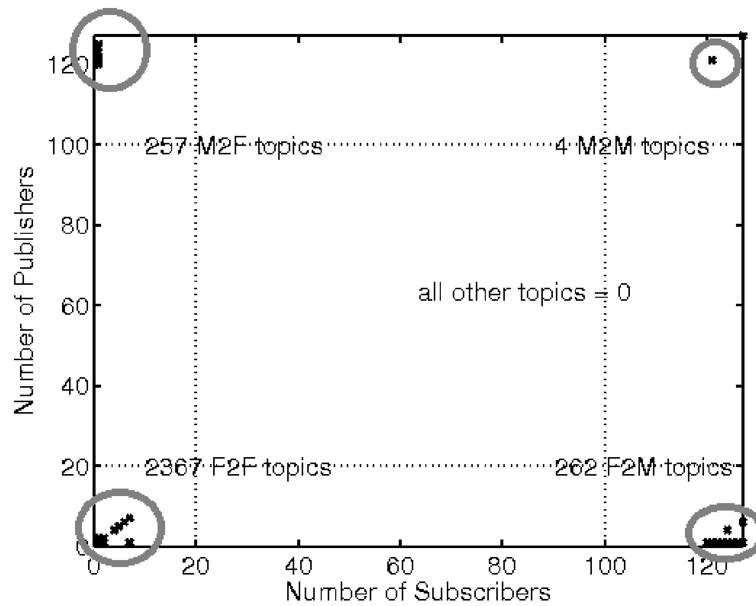


Figure 4.8: WEBSPHERE: Communication patterns. A marker is plotted for each group, in the spatial location representing its number of subscribers and publishers.

this behavior can be directly attributed to design decisions made in particular components of WAS — some of the M2F groups, for instance, were used for gathering statistical reports — but other behavior is harder to characterize.

Unlike the previous data sets in which connectivity is directly influenced by user interactions, the WEBSPHERE data set is produced by the components of an *actual* system. The distinction has important consequences, since it is reasonable to expect subscription patterns made by mechanical components to look more homogeneous and hierarchical than those of their user driven counterparts. We will address this issue in our modeling work.

4.4 Modeling Systems Communication Channels

We saw that the WEBSPHERE trace consists of components that produce highly correlated subscription patterns. Several factors in systems design contribute to high group affinity. In many data centers, multicast (or publish/subscribe) is used to replicate data so that load can be spread over multiple computing nodes. Since many applications are built from multiple subsystems, for example a web-page generator as a front-end to an inventory service, a pricing service, a popularity ranking service, *etc.*, we end up with a set of components, identically replicated on each node where the application is cloned.

Some distributed systems have multicast or publish/subscribe communication occurring at every layer. An example is the Live Objects framework [73], developed at Cornell, which produces hierarchically structured applications or *documents*. Each document is a set of *objects*, and each object can in turn consist of sub-objects, and so forth. The idea is that changes to any object reflected in

all instances in a synchronous fashion, for instance a string of text written in a notepad object should appear in all documents that embed the object. The gossip object mechanism from chapter 3 was created for Live Objects, and provides it with an optimized communications channel to synchronize object contents over wide-area networks.

4.4.1 Hierarchical Components

To model the high similarity and hierarchical structure in components of a system like Live Objects, we start with a simple binary tree. Each node x in the tree corresponds to a *component*, and each component must synchronize itself with all users who use that component. When a user picks node x, he or she needs to instantiate x as well as *all* sub-components in the subtree of x.

The selection process is as follows. Given a binary tree T with $n = 2^{h+1} - 1$ nodes, a user first picks a *level* in the tree between 0 and h uniformly at random, and then picks a node in T at that level uniformly at random. We call this the HIERARCHY model.

Let $S(x)$ for $x \in T$ denote the subtree spanned by x, *i.e.*, the set of nodes who have x as an ancestor (including x itself).

Theorem 1 *Each user x belongs to $\dfrac{2n}{\log(n+1)} - 1$ components in expectation.*

Proof 1 *We have*

$$\mathbb{E}[|S(x)|] = \sum_{i=0}^{h} \left(2^{i+1} - 1\right) \frac{1}{h+1} = \frac{2^{h+2} - 2 - (h+1)}{h+1} = \frac{2n}{\log(n+1)} - 1.$$

□

Theorem 2 *For each pair of users x and y, the expected number of overlapping components $O(x, y)$ between the users is*

$$\mathbb{E}[O(x,y)] = \Theta\left(\frac{n}{\log^2 n}\right).$$

Proof 2 *A binary tree of height h (with levels starting from 0) has 2^{h-i} nodes at level i.*

Define $L : T \to \{0, 1, \ldots, h\}$ to be the level *of vertex x in the tree T. Define $O(x, y)$ to be the* overlap size *of the subtrees spanned by $x, y \in T$, that is $|S(x) \cap S(y)|$. We note that in a tree,*

$$|S(x) \cap S(y)| = \begin{cases} \min\{|S(x)|, |S(y)|\} & \textit{if } S(x) \cap S(y) \neq \emptyset \\ 0 & \textit{otherwise.} \end{cases}$$

Let x and y be two nodes in a tree of size h, chosen by the user selection process above independently. We have

$$\begin{aligned}
\mathbb{E}[O(x,y) \mid L(x) = a] &= \sum_{i=0}^{a} \left(2^{i+1} - 1\right) \frac{2^{a-h}}{h+1} + \sum_{i=a+1}^{h} \left(2^{a+1} - 1\right) \frac{2^{i-h}}{h+1} \\
&= \frac{2^{-h}}{h+1} \left(2^{2a+1} - 2^a + \left(2^{a+1} - 1\right)\left(2^{h+1} - 2^{a+1}\right)\right) \\
&= \frac{2^{-h}}{h+1} \left(2^a + 2^{a+h+2} - 2^{2a-1} - 2^{h+1}\right)
\end{aligned}$$

Since $\mathbb{P}[L(x) = a] = 1/(h+1)$ we get

$$\begin{aligned}
\mathbb{E}[O(x,y)] &= \sum_{a=0}^{h} \mathbb{E}[O(x,y) \mid L(x) = a]\mathbb{P}[L(x) = a] \\
&= \frac{2^{-h}}{(h+1)^2} \left(\frac{2^{2h+3} - 2}{3} + 2^{h+1} - 1 + 2^{h+2}\left(2^{h+1} - 1\right) - (h+1)2^{h+1}\right) \\
&= \frac{2^{-h}}{(h+1)^2} \left(2^{2h+3} - \frac{2^{2h+3}}{3} - \frac{5}{3} - h2^{h+1} - 2^{h+1} - 2^{h+1}\right) \\
&= \frac{1}{(h+1)^2} \left(\frac{2}{3}2^{h+3} - \frac{5}{3}2^{-h} - 2(h+2)\right).
\end{aligned}$$

Then $\lim_{h \to \infty} \frac{(h+1)^2 \mathbb{E}[O(x,y)]}{n} = \frac{8}{3}$ *so* $\mathbb{E}[O(x,y)] = \Theta\left(\frac{n}{\log^2 n}\right)$. □

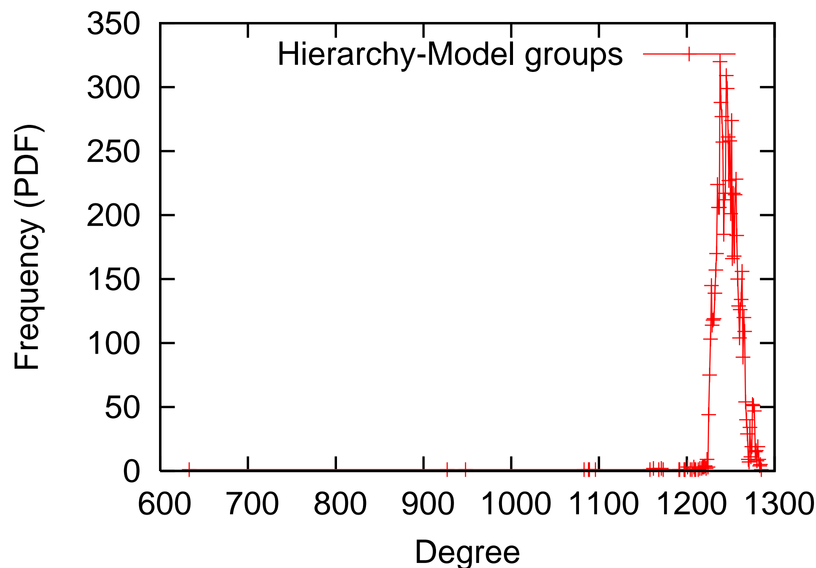


Figure 4.9: HIERARCHY: PDF of component degrees.

We ran the HIERARCHY model with $2^{13} = 8192$ components in a binary tree, and equally many independent users. The user degree distribution is such that roughly $\frac{1}{13}$ fraction of the users has degree $2^d - 1$ for $d = 1, 2, \ldots, 13$, as anticipated. Figure 4.9 shows the PDF of the component degree distribution in the graph. The mean is $1,246 \pm 15.3$ which is close to the value of $\frac{2^{14}}{13} - 1 \simeq 1,259$ predicted by Theorem 1.

4.5 Analysis

Unfortunately, data becomes hard to analyze visually at large scales. For example, if we were to draw a single pixel for every pair of groups in the WIKIPEDIA data set, we could only display 0.00003% of the data on a 1600×1200 pixel monitor. Algorithms used in data analysis often have super-linear running times, which naturally is also prohibitive at large scale. These obstacles have spawned

the growing field of data mining, which is the process of extracting information and non-obvious patterns from vast quantities of data. We will employ some data mining techniques and optimizations to help understand the degree of affinity present in the data sets and models.

4.5.1 Visualizing Affinity

First, we will show pairwise overlaps between groups in small samples of the data sets. We sample up to 10,000 groups uniformly at random, and include each user that belong to some group in the sample such that all edges incident on the groups are retained.

Let G_j denote the set of users in group j.

Definition 6 *The* similarity *of two groups j, j' is defined as*

$$\textsc{sim}(j, j') = \frac{|G_j \cap G_{j'}|}{\max\{|G_j|, |G_{j'}|\}}.$$

Various similarity metrics are used in different contexts, for example $\frac{|G_j \cap G_{j'}|}{\min\{|G_j|, |G_{j'}|\}}$, $\frac{|G_j \cap G_{j'}|}{|G_j \cup G_{j'}|}$ or cosine similarity to list a few. For our purposes, using the max-metric is convenient.

The *affinity matrices* in figures 4.10, 4.11 and 4.12 show the degree of overlap between pairs of groups in the samples. Groups are sorted by their size from left to right, and from bottom to top. The origin $(0, 0)$ is in the bottom-left part of the graph. The color of each cell (j, j') denotes the value of $\textsc{sim}(j, j')$ such that white means no overlap, and black means that $\textsc{sim}(j, j') = 1$, which implies

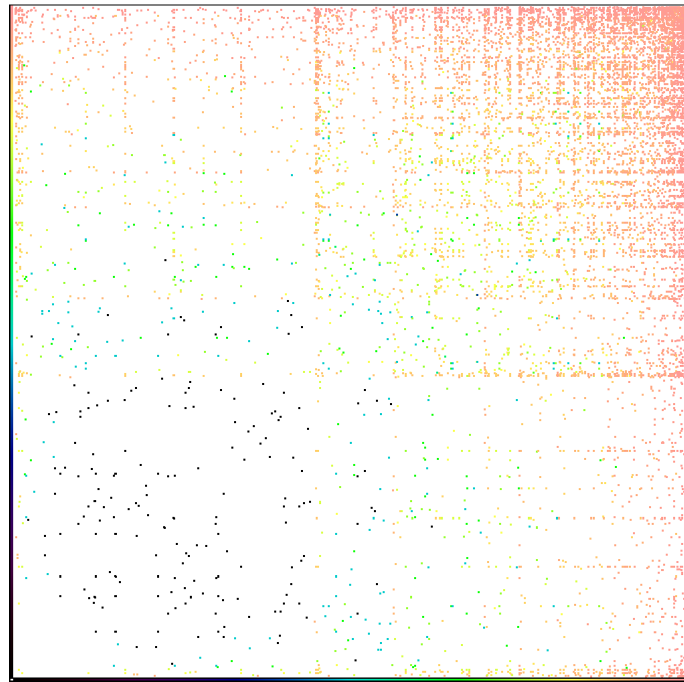
Table 4.1: Statistics for the bipartite graphs of the data sets and models.

Data set or model	# groups	# users	# edges
Y-GROUPS	638,124	999,744	15,205,016
WIKIPEDIA	3,389,252	432,533	22,863,096
AMAZON	402,724	1,555,170	6,359,182
LIVEJOURNAL	385,959	1,877,738	16,932,231
MUTUAL-INTEREST model[1]	81,991	400,000	5,280,760
WEBSPHERE	1,364	128	5,789
HIERARCHY model	8,191	8,192	10,210,872

that the two groups fully overlap. The color spectrum we use is the following, ranging from no overlap (left) to full overlap (right).

The intensity of the right end of the spectrum is biased to ensure that every non-zero similarity value is visible on the affinity matrices.

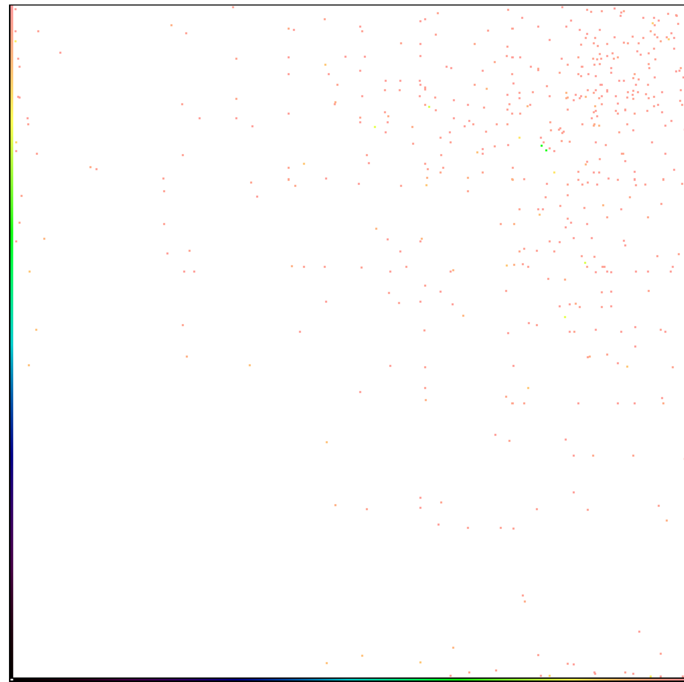
The social data sets and models in figures 4.10 and 4.11 are substantially sparser in terms of non-zero pairwise overlap than the systems data sets and models in figure 4.12. The HIERARCHY model produces high group affinity, as predicted by theorem 2. The large area of minor overlaps visible in the AMAZON trace are largely due to a singleton overlap with the user named *"A Customer"*, which is presumably a default user identifier rather than a person responsible for over 123,000 product recommendations.



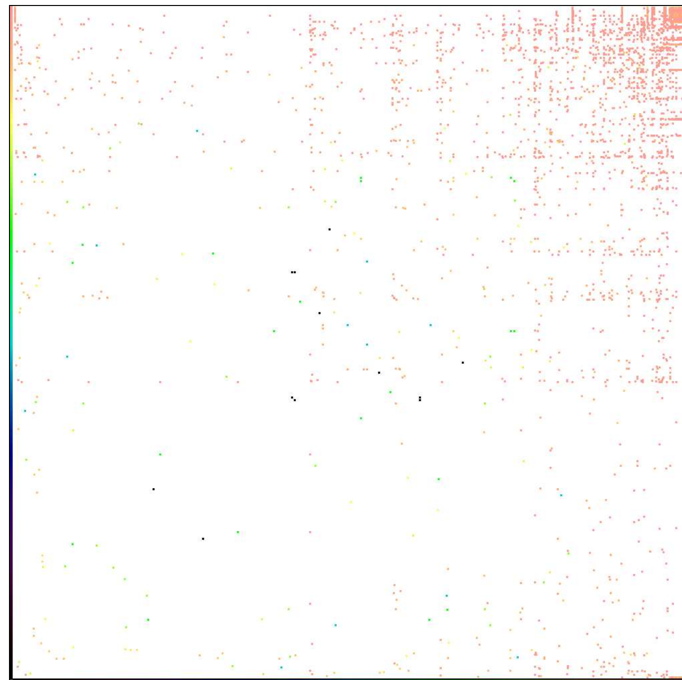
(a) WIKIPEDIA

(b) AMAZON

Figure 4.10: Affinity matrices for 1,000 group samples from the WIKIPEDIA and AMAZON data sets.

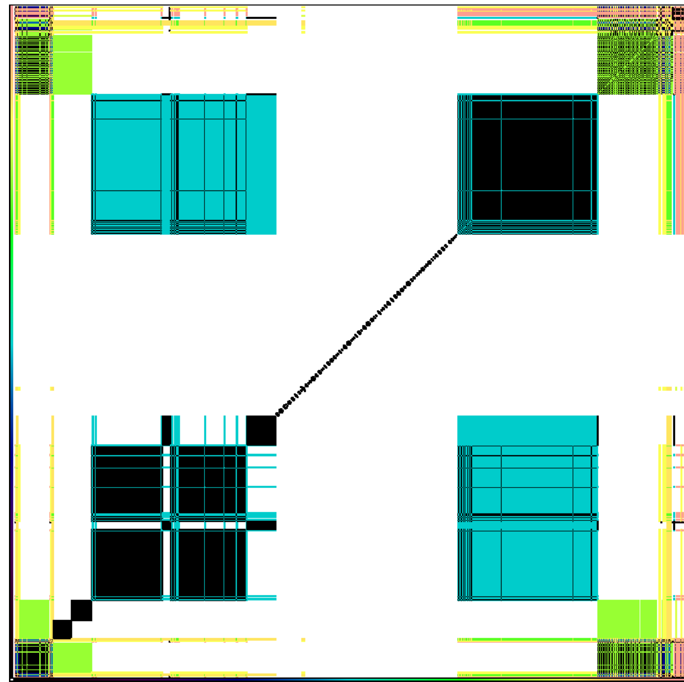


(a) LIVEJOURNAL

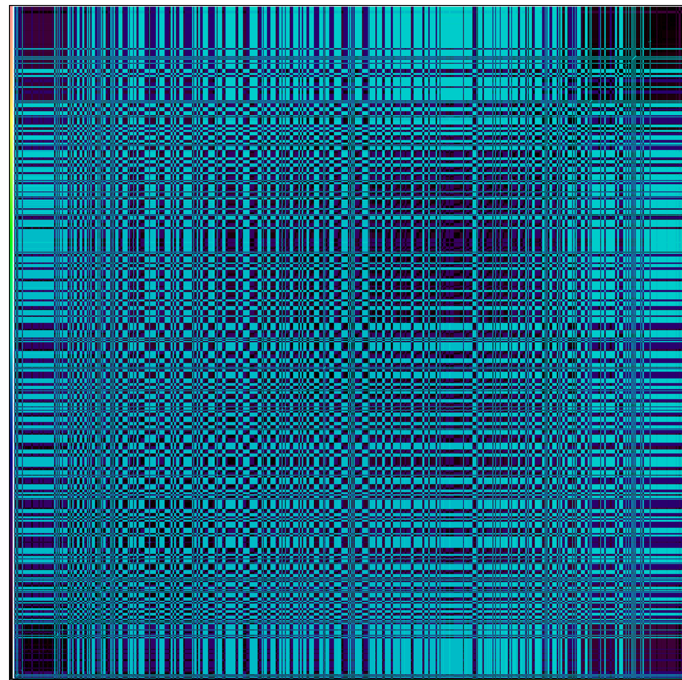


(b) MUTUAL-INTEREST model

Figure 4.11: Affinity matrices for 1,000 group samples from the LIVEJOURNAL social data set and the MUTUAL-INTEREST model.



(a) WEBSPHERE



(b) HIERARCHY model

Figure 4.12: Affinity matrices for 1,000 group samples from the systems data set WEBSPHERE and HIERARCHY model.

4.5.2 Baseline Overlap

We saw in the previous subsection that the affinity matrices produced by social models have similar characteristics, sparse with higher similarity at for large degrees, but different from the systems data set and HIERARCHY model which appear more structured and to embody higher levels of group overlap. Several interesting questions pop to mind. How random are these graphs? Is there a greater deal of preferential attachment associated with the large groups in the social data sets? Is the affinity evident in the systems graphs due to the low number of users and/or groups?

In this section, we will determine the extent of which group overlap arises *by chance* in the different data sets and models. To do so, we will need to define which random graphs can act as a baseline in this context. The first approach might be to generate a random graph that has the same number of users and groups as the graph whose randomness we wish to evaluate. One might even fix a distribution for the degree distributions, such as the power-law distribution from section 4.2.1, and keep the number of edges the same. The first problem with this approach is that real-world degree distributions are hard to emulate [71]. The second problem is that for group similarity the probability space in which we are interested should not include randomizing user or group degree information, but rather only the (user, group) pairs.

Our SPOKES model solves this conundrum. Informally, we take a bipartite graph and rewire its edges such that the node degrees are the same. Given a bipartite graph $\Gamma = (A \cup B, E)$ of users A and groups B, we define $\text{SPOKES}(\Gamma) = (A \cup B, \hat{E})$ where the random edge set $\hat{E}$ satisfies $\hat{E} \subseteq \{(a, b) : a \in A, b \in B\}$ such that $\deg_{\hat{E}}(v) = \deg_E(v)$ for $v \in A \cup B$. We now have a random set of (user, group)

pairs based on the input graph without tampering with the degree distributions or making other changes.

Define $N_\Gamma(v)$ to be the set of neighbors of node $v \in \Gamma$ for a graph Γ. Let

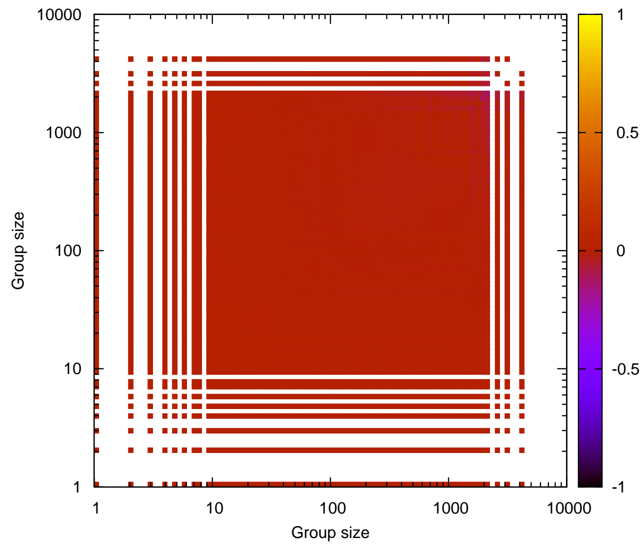
$$\textsc{sim}_\Gamma(j, j') = \frac{|N_\Gamma(j) \cap N_\Gamma(j')|}{\max\{|N_\Gamma(j)|, |N_\Gamma(j')|\}}.$$

Take a bipartite graph $\Gamma = (A \cup B, E)$ and produce $\hat{\Gamma} = \textsc{Spokes}(\Gamma)$. Note that $N_\Gamma(j)$ is the set of users in group $j \in B$. For every pair of groups j, j' in B, we want a function $\Delta(j, j')$ to quantify how much similarity there is between those groups *beyond* what arises randomly (*i.e.*, in the SPOKES variant). We will consider the difference between the similarity measures,

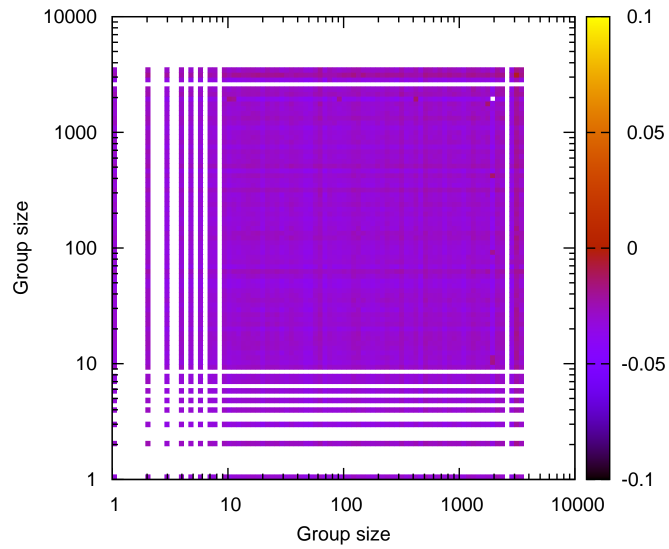
$$\Delta(j, j') = \textsc{sim}_\Gamma(j, j') - \textsc{sim}_{\hat{\Gamma}}(j, j').$$

The function could also be defined as the ratio between the quantities, as an ℓ_p norm or in various other ways. We decided to keep the definition simple to maintain a concise visual representation of group affinity. Note that a positive value of $\Delta(j, j')$ means that a data set produces more overlaps than a random model would.

In figures 4.13, 4.14 and 4.15 we provide a visual reference to the Δ function for the data sets and models discussed earlier. We experimented with multiple trials of the SPOKES randomization process, and show the output of a typical run. Intuitively, a cell denotes the average value of Δ for groups of similar size. More specifically, the color of each cell (d, d') in the color plot corresponds to the average value of $\Delta(j, j')$ over a random sample of at most 50 groups j and j' such that $|N_\Gamma(j)| \sim d$ and $|N_\Gamma(j')| \sim d'$, that is $\frac{d}{1+\varepsilon} \leq |N_\Gamma(j)| < d$ and $\frac{d'}{1+\varepsilon} \leq |N_\Gamma(j')| < d'$. We set the value of ε to be 0.1, producing an exponential

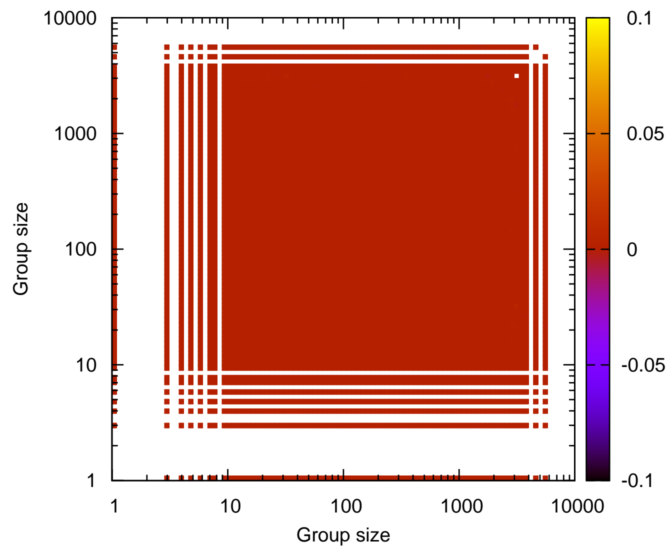


(a) WIKIPEDIA

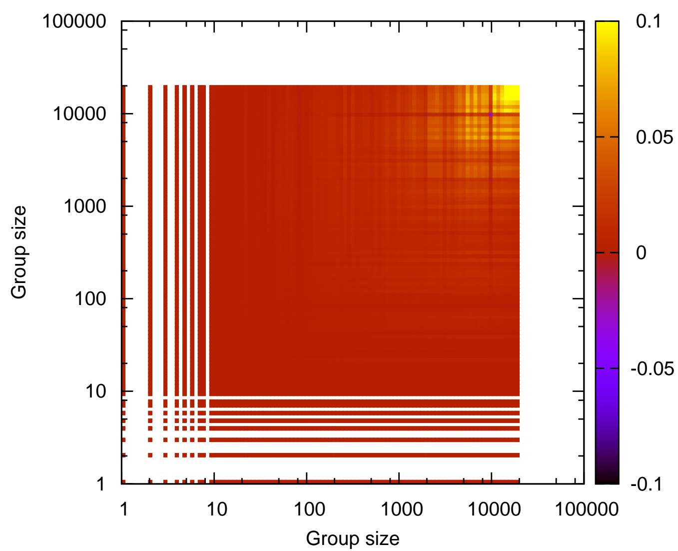


(b) AMAZON

Figure 4.13: Δ plot for the WIKIPEDIA and AMAZON graphs.

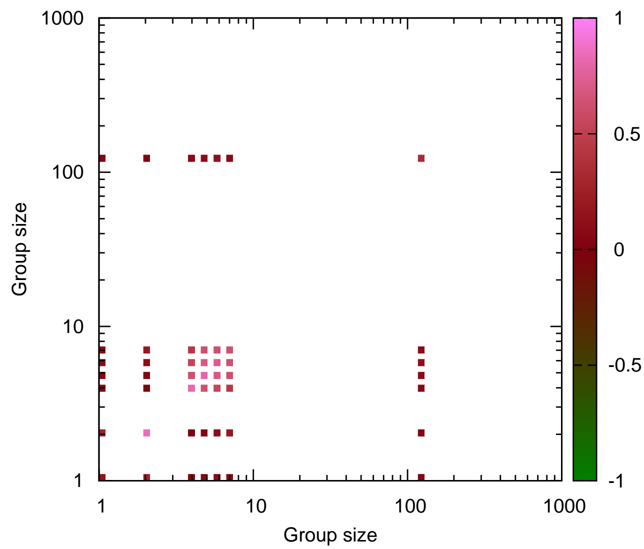


(a) Y-GROUPS

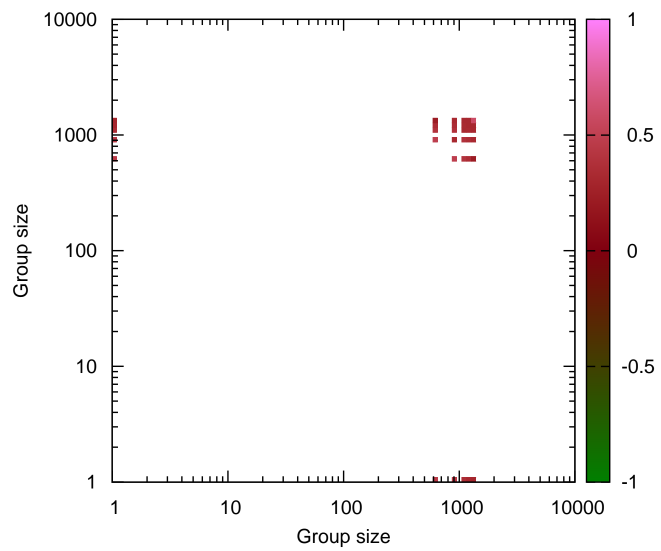


(b) MUTUAL-INTEREST model

Figure 4.14: Δ plot for the Y-GROUPS and MUTUAL-INTEREST model graphs.



(a) WEBSPHERE



(b) HIERARCHY model

Figure 4.15: Δ plot for the WEBSPHERE and HIERARCHY model graphs.

grid of size up to roughly 128×128 on log-log scale for the data sets. If no groups fit the size range, the color of the cell is white.

The figures indicate that the social data sets do not have any *general* overlap structure beyond that of the SPOKES baseline. In other words, the general group affinity present in those data sets appears to have a good fit with a random model. This does not imply that there are no pairwise overlaps that can be clustered (see section 4.6.4), but that overlaps beyond random allocation may be relatively rare independently of group sizes. Unlike the other social data sets, the MUTUAL-INTEREST model in figure 4.14 shows an increase in the similarity between large groups. This might imply that the generative procedure for the model needs to be refined to reduce mixing between high degree nodes. The model might also linger too long in its early phases, populating the first few groups with most of the first few users because of the density constraints.

Both systems graphs, that is the WEBSPHERE data set as well as the HIERARCHY model in figure 4.15, have a significantly higher level of affinity compared to a random baseline. It should be pointed out that the figure has a wider color range, ranging from $[-1, 1]$ since most Δ values in the plots for the systems exceed the upper limit of 0.1 imposed on the social graphs.

Table 4.2 shows the three most significant digits of Δ averaged over all cells of each color plot. Numbers close to zero imply that the data set or model does not have significant overlap structure beyond a random graph; numbers close to 1 imply that the graph is very structured, and a number close to -1 would imply that the graph has less overlap structure than that given by a randomly generated graph. We notice that the social data sets and MUTUAL-INTEREST model have values close to zero, whereas the WEBSPHERE model and the HI-

Table 4.2: Value of Δ averaged over all cells of each color plot.

Data set or model	Avg. Δ value
Y-GROUPS	0.000
WIKIPEDIA	-0.004
AMAZON	0.031
MUTUAL-INTEREST	0.006
WEBSPHERE	**0.284**
HIERARCHY	**0.358**

ERARCHY model both display significantly higher values, implying structure beyond the SPOKES baseline.

The take-away from this section is that group overlap in social data sets is close to random, whereas for systems data sets and models it appears to be more structured and quite substantial. While only a few data points have been presented for reasons already explained, it is reasonable to expect similar conclusions arising with other data sets in future work.

4.6 Dr. Multicast

In this section, we formalize the optimization problem that arises in Dr. Multicast, devise a heuristic to solve it and and evaluate it on the data sets and models we have presented so far.

Recall from chapter 2 that the MCMD leader can map network-level IP multicast addresses to some of the application-level groups in the system, and com-

mand others to communicate via unicast. The mapping must adhere to the acceptable-use policy, but should also achieve scalability goals:

- *Minimize the number of network-level IPMC addresses*. NICs, routers and switches do not scale in the number of IPMC addresses, as discussed earlier.
- *Minimize redundant transmissions*. This reduces the rate of packets sent by publishers and alleviates network overhead.
- *Minimize receiver filtering*. End host filtering of unwanted traffic is expensive [42]. Furthermore, imposing high CPU loads on receivers can have unanticipated consequences and potentially cause more trouble than the system solves.

The goals spur a family of optimization questions, some which have been previously addressed in the literature. We discuss previous work in section 4.7.

4.6.1 Formal Model

An overview of the scalability problem we will address in this section is as follows.

- Minimize duplicate transmissions by senders.
- Keep the number of IPMC addresses fixed at the configured limit.
- Guarantee that at most additional α fraction of network traffic needs to be filtered by receivers.

Let **limit-IPMC** and **limit-filtering** $= \alpha \in [0, \infty)$ denote the configurable knobs defined by the policy primitives in section 2.3.

Let $\mathbf{L} = \{1, 2, \ldots, K\}$ denote the set of logical (application-level) multicast groups. Let us assume that the message transmission rate on logical group $k \in \mathbf{L}$ is λ_k messages per second, and $\lambda = (\lambda_1, \cdots, \lambda_K)$.

Let $\mathbf{P} = \{1, 2, \ldots, N\}$ denote the set of processes in the system. Each process subscribes to some number of logical groups, represented by a binary *subscription vector* of length K, where a 1 in the kth position denotes the process receives traffic from logical group k.

Let us define the *subscription matrix* $W = (w_{nk})$, $k \in \mathbf{L}$, $n \in \mathbf{P}$, the rows of which are the processes' subscription vectors:

$$w_{nk} = \begin{cases} 1 & \text{process } n \text{ subscribes to logical group } k. \\ 0 & \text{otherwise.} \end{cases}$$

Logical groups can be mapped to one or more *meta-groups*, the set of which is denoted by $\mathbf{G}$ with $M = |\mathbf{G}|$. The logical group to meta-group mapping matrix, $X = (x_{km})$, $k \in \mathbf{L}$, $m \in \mathbf{G}$, is defined as:

$$x_{km} = \begin{cases} 1 & \text{logical group } k \text{ is mapped to } m. \\ 0 & \text{otherwise.} \end{cases}$$

Each meta-group can either be assigned a physical IPMC address, in which case each logical group mapped to the meta-group transmits to that address, or it can be made to use point-to-point unicast. A *transport* vector $\vec{T} = (t_m)_{m \in \mathbf{G}}$ is defined for the meta-groups as:

$$t_m = \begin{cases} 1 & \text{if meta-group } m \text{ uses physical IPMC.} \\ 0 & \text{if } m \text{ uses point-to-point unicast.} \end{cases}$$

Processes map to one or more meta-groups, depending on subscription patterns. The listening matrix, $Z = (z_{nm})$, $n \in \mathbf{P}, m \in \mathbf{G}$, specifies which meta-groups each process must join:

$$z_{nm} = \begin{cases} 1 & \text{process } n \text{ should join meta-group } m. \\ 0 & \text{otherwise.} \end{cases}$$

The formal optimization question we wish to solve is the following.

Definition 7 (MCMD's Optimization Problem) *Given a subscription matrix W, address bound **limit-IPMC** and $\alpha \geq 0$, find a set of mappings X, Z and a transport vector $\vec{T} = (t_m)_{m \in \mathbf{G}}$ such that:*

$$\min_{X,Z,\vec{T}} \sum_{m \in \mathbf{G}} \sum_{k \in \mathbf{L}} \lambda_k x_{km} \left(t_m + (1 - t_m) \sum_{n \in \mathbf{P}} z_{nm} \right) \tag{4.1}$$

subject to the constraints:

$$\sum_{m \in \mathbf{G}} z_{nm} \cdot x_{km} - w_{nk} \geq 0 \quad \forall n \in \mathbf{P}, \forall k \in \mathbf{L} \tag{4.2}$$

$$\sum_{m \in \mathbf{G}} \sum_{k \in \mathbf{L}} \sum_{n \in \mathbf{P}} \lambda_k z_{nm} x_{km} t_m \left(1 - (1 - \alpha) w_{nk}\right) \geq 0 \tag{4.3}$$

$$\sum_{m \in \mathbf{G}} t_m \leq \textbf{\textit{limit-IPMC}} \tag{4.4}$$

Primary objective (4.1) minimizes the aggregate rate of transmissions, reflecting our goal of minimizing packet duplicates. Equation (4.2) specifies that all subscribers should receive at least one copy of the traffic they are interested in. Inequality (4.3) guarantees that the aggregate rate of traffic needed to be filtered by receivers should never exceed more than α fraction of the total traffic flow. Finally, constraint (4.4) makes sure that at most **limit-IPMC** physical IPMC addresses are used. The problem can be further constrained to impose the **max-**

IPMC per-node limit on the number of physical IPMC addresses in a straightforward manner; we will assume these limits are reflected in **limit-IPMC** for sake of simplicity.

4.6.2 The MCMD Heuristic

We propose an algorithm for the above optimization problem which simulations suggest will perform well in practice.

First we cluster logical groups in the discrete space of user interests (using the vector $(w_{nk})_{n \in \mathbf{P}}$ for logical group $k \in \mathbf{L}$) into **limit-IPMC** clusters as to minimize total filtering cost incurred by receivers. This will automatically satisfies constraint (4.2). For this step, we use the k-means algorithm from [81] since it has been the most competitive in the channelization literature.

We could now take each cluster and assign a single physical IPMC address to all logical groups it contains, (set $t_m = 1$ for all logical groups m in the cluster), and then have the affected users join those groups ($z_{nm} = 1$ for affected users n). There will be no network transmissions costs and thus objective (4.1) is minimized. However, even though the clustering algorithm will attempt to minimize aggregate filtering costs, they might still exceed more than α portion of the network traffic, thus violating constraint (4.3).

To produce a feasible solution, the second step of the algorithm is to gradually alleviate filtering costs by determining which logical group m would maximally reduce the filtering cost without increasing the transmission cost by too much, and promote it to use point-to-point unicast (i.e. set $t_m = 0$). Specifically,

we pick the group m that maximizes the ratio

$$\frac{\{\text{reduction in filtering cost}\}}{\{\text{extra transmission cost}\}}$$

if it were made to use point-to-point unicast. This step is repeated until the relative filtering constraint (4.3) is satisfied.

The MCMD agent periodically reruns the algorithm to reflect changes due to membership changes. The k-means algorithm has the virtue of being incrementally stable when it is given previous mappings as input, implying that our heuristic will not produce disruptive updates following minor changes in subscription patterns.

4.6.3 Evaluation on WEBSPHERE

We simulated the MCMD heuristic on the data sets and models while varying **limit-IPMC**, the total number of physical IPMC groups. We will start by giving special attention to the WEBSPHERE graph, as we believe that the types of communication patterns seen in that real-world system are typical for data centers, confirming the validity of our approach. It illuminates the trade-off between filtering of superfluous traffic and a greater number of physical group that could be anticipated in a real deployment.

We first ran the algorithm on three different sets of topics from the WEBSPHERE graph: *(i)* All topics; *(ii)* Any-to-few (*i.e.*, few-to-few and many-to-few), and *(iii)* Any-to-many (*i.e.*, few-to-many and many-to-many).

The division of topics into clearly separated categories in figure 4.8 does not automatically imply that MCMD will work well with a handful of physi-

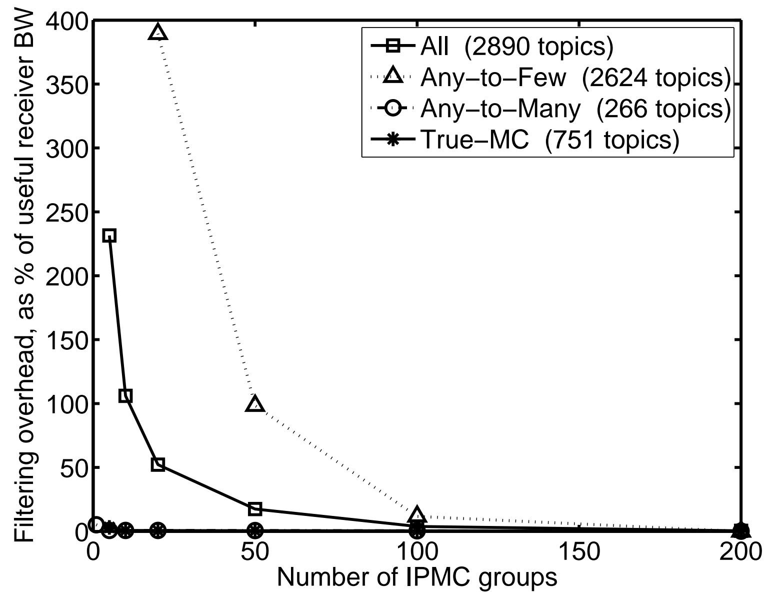


Figure 4.16: WEBSPHERE: The cost of a single multicast with the MCMD heuristic vs. number of physical IPMC groups.

cal IPMC addresses. The topics must exhibit significant affinity to be collapsed into a smaller number of physical IPMC multicast groups.

Figure 4.16 shows that there is significant reduction of filtering cost when the number of groups is increased. When clustering the any-to-few topics with 100 physical IPMC groups, the filtering overhead achieved topics is 12% of network bandwidth, whereas with 200 IPMC groups the overhead it becomes negligible. This means that the 2,624 any-to-few topics contain at most 200 exact user patterns.

The simulation results for WEBSPHERE indicate that a system comprising thousands of topics exhibiting complex subscription patterns, can be mapped to only 100 physical IPMC groups, a number that is entirely feasible on modern hardware, while incurring approximately 4% of filtering overhead. When

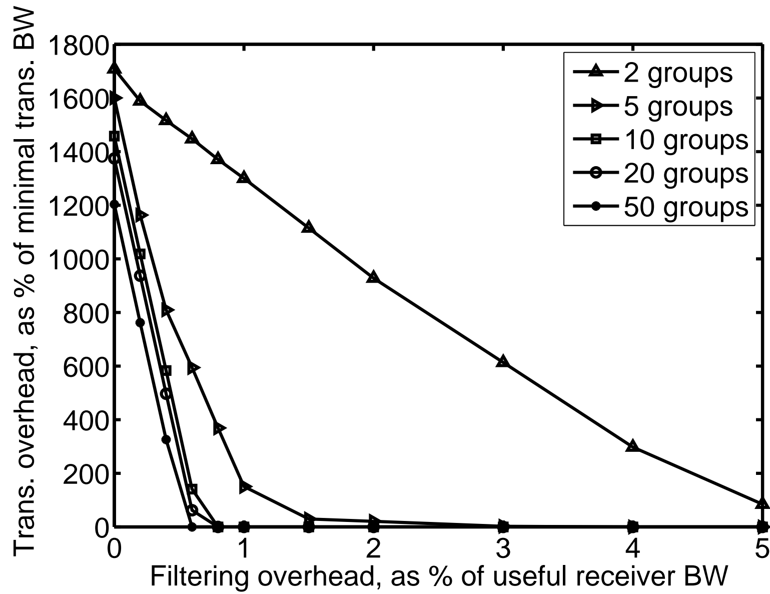
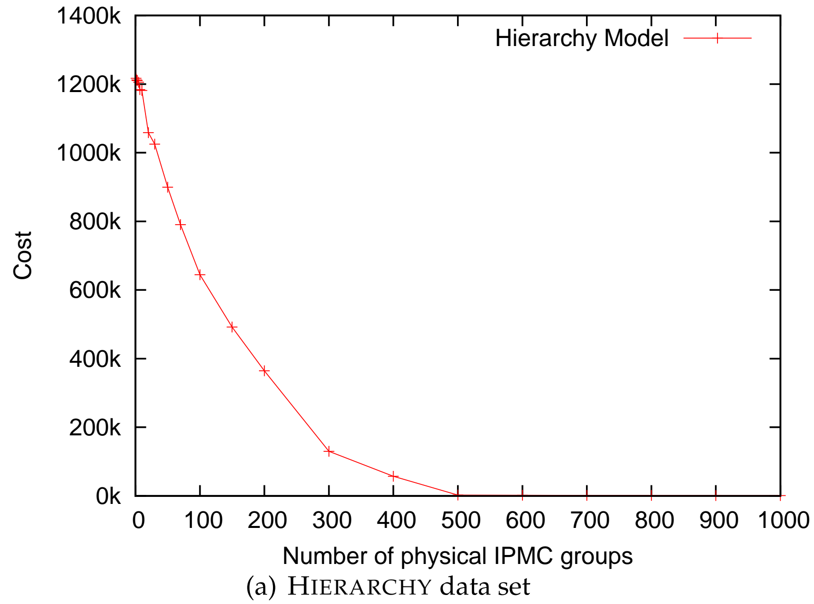


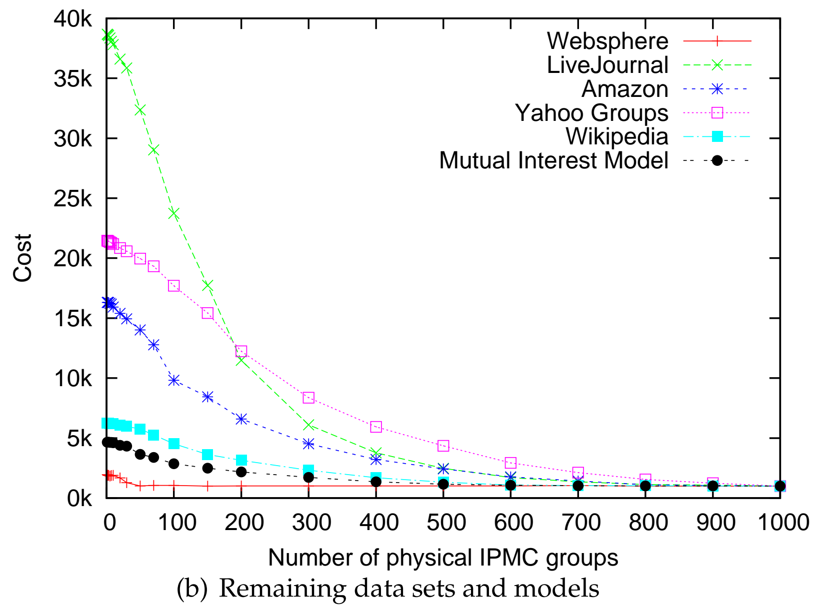
Figure 4.17: WEBSPHERE: Trade-off between filtering cost and transmission cost for a single multicast using the MCMD heuristic for a fixed number of physical groups.

limit-IPMC is increased to 200, the filtering overhead is a meager 0.5% of network traffic. Furthermore, figure 4.17 shows that with a filtering overhead of at least 3% and 5 IPMC groups results in no duplicate transmissions in the network.

Even though a systematic re-optimization of the WAS code base could reap the same reduction in network traffic as that by using the MCMD heuristic, such a process is both tedious and error-prone. The MCMD heuristic makes it possible to exploit the correlation across topics automatically.



(a) HIERARCHY data set



(b) Remaining data sets and models

Figure 4.18: Cost of a single multicast using the MCMD heuristic on samples from the data sets and models vs. number of physical groups.

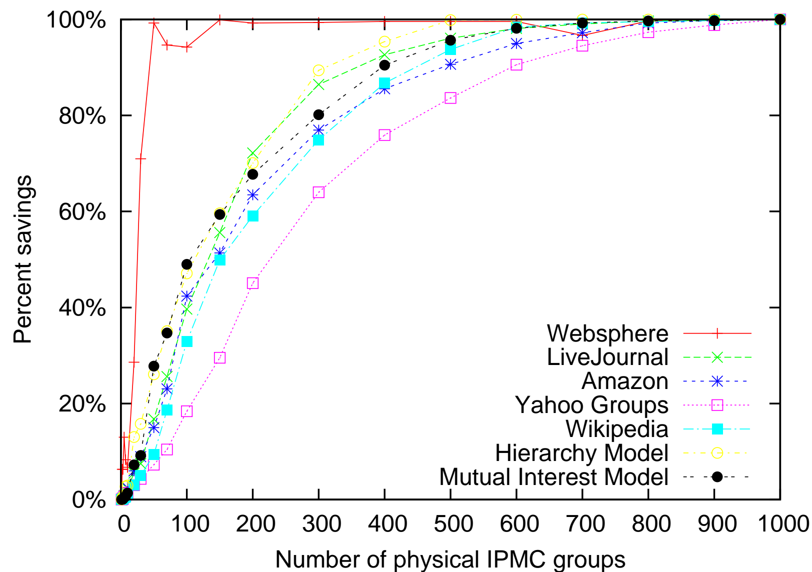


Figure 4.19: Percentage of total cost savings achieved using the MCMD heuristic as a function of the number of physical groups.

4.6.4 Evaluation on other graphs

We ran the MCMD heuristic on samples of 1,000 groups from each of the remaining data sets and models. We fixed the value of α, the proportion of superfluous traffic that can be filtered, to 20%. Figure 4.18(b) shows the cost function as a function of the total number of physical IPMC groups available in the system. Clearly, if there are 1,000 physical IPMC groups available there is no filtering cost, and transmission cost is one packet per group. We draw the HIERARCHY model separately on figure 4.18(a) because of the high costs associated with multicasts in graphs produced by the model.

The results show that if MCMD gets super-linear savings in the number of physical IPMC groups. Looking at figure 4.19 we see that with only 100 to 200 physical IPMC addresses, we save over 50% of the total cost of a single multicast call. These significant cost savings hold true for not only the systems data sets

and models, but also the social data sets.

We can conclude that clustering membership, for instance via the MCMD heuristics, is a viable and attractive option to optimize systems in large-scale data centers.

4.7 Related Work

Recently, de-duplication of Internet traffic has become a hot topic [29, 78]. Measurements show that in a trace of outgoing university traffic, some 12-15% of packet contents were redundant, whereas for a trace of a data center link the number is as high as 45% [30]. The idea is to avoid resending identical strings of information across routers by maintaining a fingerprint database for substrings encountered in recent packets, and instead send a shim packet which the destination router can inflate using its version of the database.

A natural question to ask is whether network de-duplication at the packet level will subsume efforts to enable IP multicast to minimize redundant unicast traffic, such as presented in this dissertation. Our work on Dr. Multicast is more focused on communication *within* a data center, and thus through switching hierarchies as opposed to sophisticated routers as targeted by Anand *et al.* [30]. The network de-duplication techniques could be deployed in the switches with some potential savings on regular traffic as well as identical application-level multicast packets. However, maintaining a fingerprint database is costly in terms of memory, requiring data centers to purchase new expensive hardware. IP multicast is backward-compatible, and using Dr. Multicast one also minimizes the traffic incurred by application-level multicast in the data center

at the packet level.

Several papers have been written about the optimization problem at the heart of Dr. Multicast. Work on the *channelization* problem [90, 81, 25] explores the following formulation: Allocate a fixed number of IPMC addresses to minimize a linear combination of sender transmission costs and receiver processing costs such that all subscribers receive all messages they are interested in at least once. The problem is unsurprisingly NP-complete [25], and the most competitive heuristic for a range of input is the k-means clustering algorithm [81]. The channelization problem does not address the fact that end-host NIC performance degrades with large numbers of multicast groups. Thus, an optimal solution to the channelization problem may require receivers to join a large number of groups. These papers focus only on allocating physical IPMC addresses, and while hybrid solutions using IPMC and point-to-point unicast are briefly mentioned, they are deferred to future work.

In an earlier version of Dr. Multicast [85], we consider the NP-complete problem of minimizing the number of IPMC addresses subject to *zero* receiver filtering, and minimize network traffic as a secondary objective. We provided a simple greedy algorithm for address allocation which forces zero receiver filtering. However, this approach is sensitive to minor perturbations in group membership and thus incrementally unstable. Since Dr. Multicast needs to be able to tolerate limited levels of churn in a stable fashion, we decided to relax the zero receiver filtering guarantee.

Among the systems that may directly benefit from understanding group affinity is Gravity [56]. The system is a small-world peer-to-peer overlay that dynamically clusters nodes in the overlay based on users' subscription pref-

erences. The goal is to minimize propagation cost for routing in the overlay, making it suitable for high-speed wide-area message dissemination such as a publish/subscribe service on the Internet.

The physics community has become interested in *assortative mixing* in networks in the recent years [70, 69]. Assortative mixing is the tendency of high-degree vertices to attach to other high-degree vertices, and similarly, disassortative mixing is when high-degree vertices connect to low-degree ones. An assortativity coefficient is defined by Newman in [70] takes values between -1 and 1 denoting respectively fully disassortative and full assortativity between vertices. This value is akin to the average value of Δ defined in section 4.5.2, although there are crucial differences. Newman shows that several social data sets show assortative mixing, whereas technological data sets he considers display disassortative mixing. He shows that processes such as preferential attachment [36] are incomplete because they fail to capture assortativity. An intriguing future direction is to generalize Newman's analysis to support bipartite graphs with two distinct vertex types (users and groups), and compute the assortativity coefficient or a related quantity for the data sets considered in this chapter. Such an analysis could give insight into the impact of vertex degrees on group affinity.

Recent papers in the data mining literature, for instance by Backstrom *et al.* [32] and Crandall et al. [46], analyze the evolution of social networks like LiveJournal and Wikipedia over time. They look at how groups grow with time and the process by which a user decides to join a community or contribute to an article. Analyzing the temporal characteristics of systems group membership graphs is a particularly compelling future direction towards understanding the

structure and opportunity of affinity in distributed systems.

4.8 Conclusion

Group affinity, the degree to which groups overlap, is a problem at the heart of MCMD, **GO** as well as other systems [56, 81, 29]. Despite the practical benefits arising from optimizations of group overlaps [25, 81], little work has been done to study affinity in *real-world* instances.

We analyzed four data sets from large-scale social settings, Y-GROUPS, WIKIPEDIA, AMAZON and LIVEJOURNAL, as well as a novel model based on preferential attachment (MUTUAL-INTEREST), and found that group affinity arising in this setting is limited and comparable to that which might happen by random chance.

We then analyzed a data set from a real-world system from IBM WEBSPHERE and discovered substantial and systematic overlaps between groups. We also presented a novel HIERARCHY model for the Live Objects platform that embodies similar characteristics.

We formalized the optimization problem at the core of MCMD, and devised a heuristic based on k-means to cluster similar logical groups into physical IPMC groups while minimizing the cost of sending and receiving a multicast message. Finally, we evaluated the heuristic on the data sets and models, and find that it has high potential for substantial cost savings for multicast in data centers.

CHAPTER 5

CONCLUSION

Let us summarize what we have discussed and accomplished in this thesis so far. We began this thesis by addressing scalability issues with two different group communication paradigms in distributed systems.

One of these paradigms, IP Multicast, runs into a wall of trouble beyond a certain number of multicast groups: switch and router state space become exhausted and the NIC filters saturate, so the nodes' kernels become responsible for filtering out unwanted traffic. An experiment (see figure 2.2) shows that nodes are unable to keep up with the IP Multicast traffic if more than roughly 100 groups use the technology. Since administrators have no control over the use or scale of IP Multicast in their data centers, they frequently opt to disable the technology to prevent catastrophes.

We designed Dr. Multicast to allow fine-grained control of the IP Multicast technology. Dr. Multicast makes use of the technology as far it can scale, but then transparently uses slower but safe group communication via point-to-point unicast. The system is fully backwards-compatible with both applications and networking hardware, meaning that no changes are required to deploy Dr. Multicast in a data center. The system is designed to be smart about its allocation of sparse IP Multicast resources by trying to merge groups with similar membership. The extent to which such resource sharing can be performed depends heavily on properties of the groups and their overlaps. This point prompted a more scientific inquiry into the structure of group *affinity* which we undertook in chapter 4.

The other paradigm we discussed is one where we use gossip protocols for group communication. We made the observation that as the number of groups scales up, protocols that gossip independently for each group lose a crucial property of gossip: to use fixed bandwidth for group communication.

We proposed the **GO** platform to solve this conundrum. Our system allows an administrator to specify a maximum bandwidth that can be used for gossip communication at a node, applications then declare the rate at which they intend to gossip and are allowed only if the bandwidth policy at the node can be respected. We make the observation that gossip rumors tend to be small relative to the size of an IP packet, and thus multiple rumors can be stacked into a single message at negligible cost to the operating system and network. The question becomes: what rumors should be stacked in a message to a neighbor? We developed a heuristic that aims to optimize delivery speed by maximizing the *utility* of rumor stacked in a message. The utility depends on the age of the rumor, the number of nodes who are interested in it as well as the "distance" of the rumor to its final destination. The distance here depends on the structure of group overlaps in the system, a second reason we chose to study group affinity.

Whereas the first two chapters were centered on the engineering aspects of group communication layers, we addressed the group affinity questions that arose in both systems from a scientific standpoint in chapter 4. We began by presenting a number of group membership data sets, both sociological data set and the IBM WEBSPHERE system, whose structure we set out to explore. Degree distributions in sociological data tend to be power-laws, so we devised a MUTUAL-INTEREST model based on preferential attachment, a popular generative model for power-law degree distributions. The advantage of using such a

model is that it captures the statistics we observed in the sociological data sets, and gives engineers a tool to synthesize realistic group membership to evaluate their systems at an arbitrary scale. We also presented a systems oriented HIERARCHY model, based on component hierarchies that arise from layered and distributed system design, such as in Live Objects.

We noticed that pairwise overlaps, or *affinity*, of the groups in the sociological data sets and the MUTUAL-INTEREST model was low. However, the systems data set (from WEBSPHERE) displays remarkable structure, and the MUTUAL-INTEREST model does as well by construction. We devised a group clustering heuristic to allocate the sparse IP Multicast resources in the Dr. Multicast setting, under a particular optimization model, and evaluated this heuristic on the data sets and models. We found that clustering is extremely helpful, even in the sociological data sets, and in the WEBSPHERE data set we are able to condense all groups into a small number of IP Multicast groups with negligible network overhead (figure 4.18(b)).

From the limited data we looked at, it is impossible to draw general conclusions about group affinity. Although human behavior often directly influences the groups arising in distributed systems, the sociological data sets are mainly interesting from a sociological perspective since performing optimization of any system incorporating human affinity depends on a different level of abstraction for groups. For instance, if we intend to optimize Amazon's product recommendations as a publish-subscribe system by using Dr. Multicast, the input group membership graph should be the system's *nodes* to groups, not *users* to groups, with the important difference that a system node abstracts the aggregate group membership of the hundreds or thousands of users that the node stores, provid-

ing very different results from the AMAZON users to groups data set.

The single systems data point we obtained (WEBSPHERE) embodies much of the group structure that we suspect is commonplace in large distributed systems. An intriguing question is whether other real systems produce membership with substantial group affinity. We hope to see progress made towards answering this question, but our ability to explore the problem will depend on the cooperation of the kinds of large companies and/or academic researchers, who will need to make group data from their systems available. As mentioned earlier, understanding the temporal aspects of system group membership would be rewarding, both from a scientific perspective (*"what fundamental processes drive the group structure?"*) as well as an engineering perspective (*"how should we decide on the right protocols and optimize group communication dynamically?"*). Our affinity investigation should be viewed as a first step towards addressing structure that has for the most part been ignored by engineers, and has important consequences as demonstrated by optimizations in the systems we presented.

We conclude that group scalability in distributed systems has been lacking, as evidenced by IP Multicast losing packets and gossip protocols, but that systems such as Dr. Multicast and **GO** can remedy the situation by giving administrators much needed control. The icing on the cake is that such systems can furthermore enhance network performance by exploiting group affinity present in those distributed systems.

BIBLIOGRAPHY

[1] BEA Weblogic. http://www.bea.com/framework.jsp?CNT=features.htm&FP=/content/products/weblogic/server/ (accessed in July 2009).

[2] Bea WebLogic Server 7.0 Documentation. http://e-docs.bea.com/wls/docs70/pdf/cluster.pdf (accessed in July 2009).

[3] Facebook. http://www.facebook.com/ (accessed in August 2009).

[4] GEMSTONE GemFire. http://www.gemstone.com/products/gemfire/enterprise.php (accessed in July 2009).

[5] Google search. http://www.google.com/ (accessed in August 2009).

[6] Guidelines for Enterprise IP Multicast Address Allocation. http://www.cisco.com/warp/public/cc/techno/tity/prodlit/ipmlt_wp.pdf.

[7] IBM WebSphere. http://www-01.ibm.com/software/webservers/appserv/was/ (accessed in July 2009).

[8] JBoss Application Server. http://www.jboss.org/ (accessed in July 2009).

[9] KaZaA. http://en.wikipedia.org/wiki/KaZaA (accessed in July 2009).

[10] Napster. http://en.wikipedia.org/wiki/Napster (accessed in July 2009).

[11] Oracle Coherence 3.4 User Guide. http://coherence.oracle.com/display/COH34UG/Coherence+User+Guide+(Full) (accessed in July 2009).

[12] Oracle Coherence 3.4 User Guide. http://coherence.oracle.com/display/COH34UG/Deployment+Considerations+-+Cisco+Switches (accessed in July 2009).

[13] Oracle WebLogic Server 10.3 Documentation. http://e-docs.bea.com/wls/docs103/pdf/cluster.pdf (accessed in July 2009).

[14] Real Time Innovations Data Distribution Service. http://www.rti.com/products/data_distribution/ (accessed in July 2009).

[15] Second life. http://en.wikipedia.org/wiki/Second_Life (accessed in July 2009).

[16] Skype. http://en.wikipedia.org/wiki/Skype (accessed in July 2009).

[17] TIBCO Rendezvous. http://www.tibco.com/software/messaging/rendezvous/default.jsp (accessed in July 2009).

[18] Twitter. http://www.twitter.com/ (accessed in August 2009).

[19] YouTube. http://www.youtube.com/ (accessed in August 2009).

[20] CNET News: Google spotlights data center inner workings. http://news.cnet.com/8301-10784_3-9955184-7.html (accessed in July 2009), May 2008.

[21] Yahoo! Research Webscope, Groups data-set version 1.0. http://research.yahoo.com/ (accessed in July 2009), Jan 2008.

[22] Amazon.com: Online shopping for electronics, apparel, computers, books, and more. http://www.amazon.com/ (accessed in July 2009), July 2009.

[23] Wikipedia, the free encyclopedia that everyone can edit. http://www.wikipedia.org (accessed in July 2009), July 2009.

[24] Yahoo! groups. http://groups.yahoo.com/ (accessed in July 2009), July 2009.

[25] Micah Adler, Zihui Ge, James F. Kurose, Donald F. Towsley, and Steve Zabele. Channelization problem in large scale data dissemination. In *ICNP 2001*.

[26] R. Alberich, J. Miro-Julia, and F. Rossello. Marvel universe looks almost like a real social network, 2002.

[27] Reka Albert and Albert-Laszlo Barabasi. Statistical mechanics of complex networks. *Reviews of Modern Physics*, 74:47, 2002.

[28] Lorenzo Alvisi, Jeroen Doumen, Rachid Guerraoui, Boris Koldehofe, Harry C. Li, Robbert van Renesse, and Gilles Trédan. How robust are gossip-based communication protocols? *Operating Systems Review*, 41(5):14–18, 2007.

[29] Ashok Anand, Archit Gupta, Aditya Akella, Srinivasan Seshan, and Scott Shenker. Packet caches on routers: the implications of universal redundant traffic elimination. In Victor Bahl, David Wetherall, Stefan Savage, and Ion Stoica, editors, *SIGCOMM '08*, pages 219–230. ACM, 2008.

[30] Ashok Anand, Chitra Muthukrishnan, Aditya Akella, and Ramachandran Ramjee. Redundancy in network traffic: findings and implications. In *SIGMETRICS '09: Proceedings of the eleventh international joint conference on Measurement and modeling of computer systems*, pages 37–48, New York, NY, USA, 2009. ACM.

[31] Villu Arak. Skype blogs: What happened on August 16? `http://share.skype.com/sites/en/2007/08/what_happened_on_august_16.html` (accessed in July 2009), August 2007.

[32] Lars Backstrom, Daniel P. Huttenlocher, Jon M. Kleinberg, and Xiangyang Lan. Group formation in large social networks: membership, growth, and evolution. In Tina Eliassi-Rad, Lyle H. Ungar, Mark Craven, and Dimitrios Gunopulos, editors, *KDD*, pages 44–54. ACM, 2006.

[33] Mahesh Balakrishnan. *Reliable Communication for Datacenters*. PhD thesis, Cornell University, Ithaca, NY, January 2009.

[34] Mahesh Balakrishnan, Kenneth P. Birman, Amar Phanishayee, and Stefan Pleisch. Ricochet: Lateral error correction for time-critical multicast. In *NSDI*. USENIX, 2007.

[35] T. Ballardie and J. Crowcroft. Multicast-specific security threats and counter-measures. In *SNDSS'95: Proceedings of the Symposium on Network and Distributed System Security*, page 2, Washington, DC, USA, 1995. IEEE Computer Society.

[36] A. L. Barabasi and R. Albert. Emergence of scaling in random networks. *Science*, 286(5439):509–512, October 1999.

[37] S. Bhattacharyya, C. Diot, and L. Giuliano. An Overview of Source-Specific Multicast (SSM). Technical report, RFC 3569, July 2003.

[38] Ken Birman, Anne-Marie Kermarrec, Krzysztof Ostrowski, Martin Bertier, Danny Dolev, and Robbert Van Renesse. Exploiting gossip for self-management in scalable event notification systems. Distributed Event Processing Systems and Architecture Workshop (DEPSA), 2007.

[39] Kenneth P. Birman, Mark Hayden, Oznur Ozkasap, Zhen Xiao, Mihai Budiu, and Yaron Minsky. Bimodal Multicast. *ACM Transactions on Computer Systems (TOCS)*, 17(2):41–88, 1999.

[40] H.B. Burton. Space/Time trade-offs in hash coding with allowable errors. *CACM*, 13(7):422–426, 1970.

[41] R. Canetti, J. Garay, G. Itkis, D. Micciancio, M. Naor, and B. Pinkas. Multicast security: a taxonomy and some efficient constructions. In *INFOCOM 1999*, volume 2, 1999.

[42] Boaz Carmeli, Gidon Gershinsky, Avi Harpaz, Nir Naaman, Haim Nelken, Julian Satran, and P. Vortman. High throughput reliable message dissemination. In Hisham Haddad, Andrea Omicini, Roger L. Wainwright, and Lorie M. Liebrock, editors, *SAC*, pages 322–327. ACM, 2004.

[43] Fay Chang, Jeffrey Dean, Sanjay Ghemawat, Wilson C. Hsieh, Deborah A. Wallach, Mike Burrows, Tushar Chandra, Andrew Fikes, and Robert E. Gruber. Bigtable: a distributed storage system for structured data. In *OSDI '06: Proceedings of the 7th symposium on Operating Systems Design and Implementation*, pages 205–218, Berkeley, CA, USA, 2006. USENIX Association.

[44] Aaron Clauset, Cosma R. Shalizi, and M. E. J. Newman. Power-law distributions in empirical data. `http://arxiv.org/abs/0706.1062v1` (accessed in July 2009), June 2007.

[45] Brian F. Cooper, Raghu Ramakrishnan, Utkarsh Srivastava, Adam Silberstein, Philip Bohannon, Hans-Arno Jacobsen, Nick Puz, Daniel Weaver, and Ramana Yerneni. PNUTS: Yahoo!'s hosted data serving platform. *Proc. VLDB Endow.*, 1(2):1277–1288, 2008.

[46] David J. Crandall, Dan Cosley, Daniel P. Huttenlocher, Jon M. Kleinberg, and Siddharth Suri. Feedback effects between similarity and social influence in online communities. In Ying Li, Bing Liu, and Sunita Sarawagi, editors, *KDD*, pages 160–168. ACM, 2008.

[47] Giuseppe Decandia, Deniz Hastorun, Madan Jampani, Gunavardhan

Kakulapati, Avinash Lakshman, Alex Pilchin, Swaminathan Sivasubramanian, Peter Vosshall, and Werner Vogels. Dynamo: Amazon's highly available key-value store. In *SOSP*, pages 205–220, New York, NY, USA, 2007. ACM Press.

[48] S. Deering. Host Extensions for IP Multicasting. *RFC 1112*, August 1989.

[49] Alan J. Demers, Daniel H. Greene, Carl Hauser, Wes Irish, John Larson, Scott Shenker, Howard E. Sturgis, Daniel C. Swinehart, and Douglas B. Terry. Epidemic algorithms for replicated database maintenance. In *PODC*, pages 1–12, 1987.

[50] Patrick Th. Eugster, Pascal A. Felber, Rachid Guerraoui, and Anne-Marie Kermarrec. The many faces of publish/subscribe. *ACM Comput. Surv.*, 35(2):114–131, 2003.

[51] Michalis Faloutsos, Petros Faloutsos, and Christos Faloutsos. On power-law relationships of the internet topology. In *SIGCOMM*, pages 251–262, 1999.

[52] Robert W. Floyd. Algorithm 97: Shortest path. *Commun. ACM*, 5(6):345, 1962.

[53] Sally Floyd, Van Jacobson, Ching-Gung Liu, Steven McCanne, and Lixia Zhang. A Reliable Multicast Framework for Light-weight Sessions and Application Level Framing. *IEEE/ACM Transactions on Networking*, 5(6):784–803, 1997.

[54] J. Gemmel, T. Montgomery, T. Speakman, N. Bhaskar, and J. Crowcroft. The PGM Reliable Multicast Protocol. *IEEE Network*, 17(1):16–22, 2003.

[55] Sanjay Ghemawat, Howard Gobioff, and Shun-Tak Leung. The Google file system. *SIGOPS Oper. Syst. Rev.*, 37(5):29–43, 2003.

[56] Sarunas Girdzijauskas, Gregory Chockler, Roie Melamed, and Yoav Tock. Gravity: An interest-aware publish/subscribe system based on structured overlays. In *LADIS '08: Proceedings of the 2nd Large-Scale Distributed Systems and Middleware Workshop*, 2008.

[57] Mark Jelasity, Rachid Guerraoui, Anne-Marie Kermarrec, and Maarten van Steen. The peer sampling service: Experimental evaluation of unstructured

gossip-based implementations. In *Middleware*, Toronto, Canada, October 2004.

[58] P. Judge and M. Ammar. Gothic: a group access control architecture for secure multicast and anycast. In *INFOCOM 2002*, volume 3, 2002.

[59] R. Karp, C. Schindelhauer, S. Shenker, and B. Vocking. Randomized rumor spreading. In *FOCS*, pages 565–574, 2000.

[60] David Kempe, Jon M. Kleinberg, and Alan J. Demers. Spatial gossip and resource location protocols. In *STOC*, pages 163–172, 2001.

[61] Jon M. Kleinberg, Ravi Kumar, Prabhakar Raghavan, Sridhar Rajagopalan, and Andrew S. Tomkins. The web as a graph: Measurements, models, and methods. 1999.

[62] Tom Leighton. The challenges of delivering content on the Internet. In *PODS '01: Proceedings of the twentieth ACM SIGMOD-SIGACT-SIGART symposium on Principles of database systems*, page 246, New York, NY, USA, 2001. ACM.

[63] Jure Leskovec, Lada A. Adamic, and Bernardo A. Huberman. The dynamics of viral marketing. In *EC '06: Proceedings of the 7th ACM conference on Electronic commerce*, pages 228–237, New York, NY, USA, 2006. ACM.

[64] John D. C. Little. A proof for the queuing formula: $L = \lambda W$. *Operations Research*, 9(3):383–387, 1961.

[65] Hongzhou Liu, Venugopalan Ramasubramanian, and Emin Gün Sirer. Client behavior and feed characteristics of RSS, a publish-subscribe system for web micronews. In *IMC '05: Proceedings of the 5th ACM SIGCOMM conference on Internet Measurement*, pages 3–3, Berkeley, CA, USA, 2005. USENIX Association.

[66] M. Mitzenmacher. A brief history of generative models for power law and lognormal distributions. *Internet Mathematics*, 1(2):226–251, 2004.

[67] MJ Moyer, JR Rao, and P. Rohatgi. A survey of security issues in multicast communications. *IEEE Network*, 13(6):12–23, 1999.

[68] David Newman. Multicast performance differentiates across switches. `http://www.networkworld.com/reviews/2008/`

032408-switch-test-performance.html (accessed in June 2008), 2008.

[69] M. E. J. Newman. Assortative mixing in networks. *Physical Review Letters*, 89:208701, 2002.

[70] M. E. J. Newman. Mixing patterns in networks. *Physical Review E*, 67:026126, 2003.

[71] M. E. J. Newman. The structure and function of complex networks. *SIAM Review*, 45:167, 2003.

[72] M. E. J. Newman. Power laws, Pareto distributions and Zipf's law. *Contemporary Physics*, 46:323, 2005.

[73] Krzysztof Ostrowski, Ken Birman, Danny Dolev, and Jong Hoon Ahnn. Programming with live distributed objects. In Jan Vitek, editor, *ECOOP*, volume 5142 of *Lecture Notes in Computer Science*, pages 463–489. Springer, 2008.

[74] John Risson and Tim Moors. Survey of research towards robust peer-to-peer networks: Search methods. *Computer Networks*, 50(17):3485–3521, 2006.

[75] Robert A. Van Valzah, Todd L. Montgomery and Eric Bowden. Topics in High-Performance Messaging. http://www.29west.com/docs/THPM/thpm.html (accessed in July 2009).

[76] L. Rodrigues, U. De Lisboa, S. Handurukande, J. Pereira, J. Pereira U. do Minho, R. Guerraoui, and A.-M. Kermarrec. Adaptive gossip-based broadcast. In *DSN*, pages 47–56, 2003.

[77] P. Rosenzweig, M. Kadansky, and S. Hanna. The Java Reliable Multicast Service: A Reliable Multicast Library. *Sun Labs*, 1997.

[78] Neil T. Spring and David Wetherall. A protocol-independent technique for eliminating redundant network traffic. In *SIGCOMM*, pages 87–95, 2000.

[79] Allen Stern. Update from Amazon regarding friday's S3 downtime. http://www.centernetworks.com/amazon-s3-downtime-update (accessed in August 2009), February 2008.

[80] W.R. Stevens, B. Fenner, and A.M. Rudoff. *UNIX Network Programming: The Sockets Networking API*. Addison-Wesley, 2004.

[81] Yoav Tock, Nir Naaman, Avi Harpaz, and Gidon Gershinsky. Hierarchical clustering of message flows in a multicast data dissemination system. In *IASTED PDCS*, 2005.

[82] Robbert Van Renesse, Kenneth P. Birman, and Werner Vogels. Astrolabe: A robust and scalable technology for distributed system monitoring, management, and data mining. *ACM Trans. Comput. Syst.*, 21(2):164–206, May 2003.

[83] Robbert van Renesse, Yaron Minsky, and Mark Hayden. A gossip-based failure detection service. In *Middleware 1998*, pages 55–70, England, September 1998.

[84] Robbert Van Renesse, Yaron Minsky, and Mark Hayden. A gossip-style failure detection service. Technical Report TR98-1687, August, 1998.

[85] Ymir Vigfusson, Hussam Abu-Libdeh, Mahesh Balakrishnan, Ken Birman, and Yoav Tock. Dr. Multicast: Rx for Datacenter Communication Scalability. In *7th ACM Symposion on Hot Topic in Networks (HotNets VII)*, October 2008.

[86] Ymir Vigfusson, Hussam Abu-Libdeh, Mahesh Balakrishnan, Ken Birman, and Yoav Tock. Dr. Multicast: Rx for Datacenter Communication Scalability. In *LADIS '08: Proceedings of the 2nd Large-Scale Distributed Systems and Middleware Workshop*, September 2008.

[87] Ymir Vigfusson, Ken Birman, Qi Huang, and Deepak Nataraj. GO: Platform support for gossip applications. In *P2P'09: Proceedings of the Ninth International Conference on Peer-to-Peer Computing, Seattle, Washington*, August 2009.

[88] Ymir Vigfusson, Ken Birman, Qi Huang, and Deepak Nataraj. Optimizing information flow in the gossip objects platform. In *LADIS '09: Proceedings of the 3rd Large-Scale Distributed Systems and Middleware Workshop*, October 2009.

[89] Thorsten von Eicken, Anindya Basu, Vineet Buch, and Werner Vogels. U-net: A user-level network interface for parallel and distributed computing. In *SOSP*, pages 40–53, 1995.

[90] Tina Wong and R. Katz. An analysis of multicast forwarding state scalability. In *ICNP 2000*.

[91] Tina Wong, Randy H. Katz, and Steven McCanne. An evaluation on using preference clustering in large-scale multicast applications. In *INFOCOM 2000*.