Scalable Services for Dynamic Wide-Area Environments

Roie Melamed

Scalable Services for Dynamic Wide-Area Environments

Research Thesis
Submitted in Partial Fulfillment of the
Requirements for the
Degree of Doctor of Philosophy

Roie Melamed

Submitted to the Senate of
the Technion — Israel Institute of Technology
Tamuz, 5766 Haifa July 2006

The research thesis was done under the supervision of Dr. Idit Keidar
in the department of Computer Science.

The generous financial help of the Technion is gratefully acknowledged.

Acknowledgments

First and foremost, I am deeply grateful to my advisor Idit Keidar with who I had the pleasure to work during the last four years. During these years, Idit shared with me her vast knowledge in the area of distributed systems in her unique and graceful way. Idit always knew how to ask the important questions and how to present complicated issues in a few short sentences. During my Ph.D. studies, Idit always had the time to read the endless drafts I sent to her although she always had many other obligations. It is not an exaggeration to say that Idit has been to me an inspiration and an example to how to do things in the right way.

I also thank Ariel Orda with who I had the pleasure to work on the EquiCast protocol. During our joint work, Ariel shared with me his vast knowledge in game theory. Ariel was always willing to read the many drafts I sent to him, and he always had good comments and suggestions for improving our paper.

During my graduate studies, I guided several undergraduate students in projects from which I greatly benefited. I thank Ophir Ovadia for implementing the DSGraph tool that allowed me to evaluate the performance of an Araneola overlay. Ophir also assisted in implementing part of Araneola's code. The Octopus ad-hoc routing protocol was implemented with the help of Yoav Barel, Evgeny (Jenya) Gurevich, Lily Itkin, Yaron (Ronny) Lehman, and Inna Vaisband. I would also like to thank Ilana David, the chief engineer of the software laboratory in the electrical engineering department, for her assistance.

I have remarkably benefited from comments, suggestions, and discussions with Ziv Bar-Yossef, Vadim Drabkin, Maxim Gurevich, Dahlia Malkhi, Amir Ronen, and Igor Yanover.

I am grateful to the Flux research group at the University of Utah, and especially Leigh Stoller and Jay Lepreau, for allowing me to use their network emulation testbed and assisting me with my experiments.

For the last five years, I had the pleasure to share an office with Harel Paz. Harel was and still is a genuine friend, and I will also share an office with Harel at IBM Research Lab in Haifa.

Last but not least, I thank my family Mordechai (Moti), Dina, Ido, and Sharon Melamed who always love and care for me, and for reminding me that there are important things beside my research.

Contents

List of Figures

List of Tables

Abstract

Peer-to-peer (P2P) systems are systems that rely primarily on the computing power and bandwidth of the participating nodes (peers) rather than on a central infrastructure. Such systems are scalable, robust, and can be easily deployed. Hence, P2P computing is a promising architecture for deploying distributed services over the Internet, as well as in mobile ad-hoc networks (MANETs). However, such an architecture also raises many research problems and challenges such as achieving scalability while incurring small load on each node, coping efficiency with failures and dynamic user behavior, and achieving fairness in a network with selfish users. In this dissertation, we review these challenges, and present four P2P studies that address them in different settings.

In Chapter 4, we introduce Araneola, a scalable reliable *application-level multicast (ALM)* system for highly dynamic wide-area environments. Araneola supports multi-point to multi-point reliable communication in a fully distributed manner while incurring constant load on each node. For a tunable parameter $k \geq 3$, Araneola constructs and dynamically maintains an *overlay network* structure in which each node's degree is either k or $k+1$, and roughly 90% of the nodes have degree k. Empirical evaluation shows that Araneola's basic overlay achieves three important mathematical properties of k-regular random graphs (i.e., random graphs in which each node has exactly k neighbors) with N nodes: (i) its diameter grows logarithmically with N; (ii) it is generally k-connected; and (iii) it remains highly connected following random removal of linear-size subsets of edges or nodes. The overlay is constructed and maintained at a low cost: each join, leave, or failure is handled locally, and entails the sending of only about $3k$ messages in total, independent of N. Moreover, this cost decreases as the churn rate increases. Thorough evaluation of Araneola running up to $10,000$ nodes on up to 125 machines, in both LAN and WAN, shows that Araneola successfully addresses the following challenges: (i) providing high reliability despite considerable message loss and failure rates while incurring *constant* load on each node; (ii) incorporating joining nodes and removing leaving (or failing) ones with a low *constant* overhead; and (iii) providing an undisrupted service to nodes that are up despite node joins and leaves.

In Chapter 5, we present EquiCast, a wide-area P2P multicast protocol for large groups of selfish nodes. We tackle the problem of "freeloaders", i.e., users who consume resources without contributing anything in return. We take a game theoretic perspective by modeling the system as a

non-cooperative game. We define a special set of *protocol-obedient strategies (POSs)*. Generally speaking, a strategy out of this set allows a node to determine how many connections to maintain and how many packets to send on each connection though it does not allow users to hack the protocol's code or assume that others do so. We prove that if all nodes choose POSs, then each node receives all the multicast packets. Moreover, in this case, no node can unilaterally reduce its cost by changing its strategy to a non-POS. In addition, we prove that EquiCast incurs low constant load on each node. We note that EquiCast is the *first* P2P multicast protocol that is *formally proven* to enforce cooperation in *selfish environments*.

Next, we consider P2P communication in failure-prone MANETs: in Chapter 6, we introduce Octopus, a fault-tolerant and efficient routing protocol for MANETs. Fault-tolerance is achieved by employing redundancy, i.e., storing the location of each node at many other nodes, and by keeping frequently refreshed soft state. At the same time, Octopus achieves a low location update overhead by employing a novel aggregation technique, whereby a single packet updates the location of many nodes at many other nodes. Octopus is highly scalable: for a fixed node density, the number of location update packets sent does not grow with the network size. And when the density increases, the overhead drops. Thorough empirical evaluation using the ns2 simulator with up to 675 mobile nodes shows that Octopus achieves excellent fault-tolerance at a modest overhead: when all nodes intermittently disconnect and reconnect, Octopus achieves the same high reliability as when all nodes are constantly up.

Finally, in Chapter 7, we define metrics for evaluating unstructured overlay networks for P2P lookup systems. These metrics capture the search dependability and efficiency, and the granularity at which one can control the tradeoff between the two, as well as fairness. According to these metrics, we evaluate different graphs and overlays, including a Gnutella graph, a power law random graph, normal random graphs, a 3-regular random graph, and a 3-Araneola overlay. Our study shows that, according to our metrics, a 3-Araneola overlay achieves the best results, and hence it is an excellent solution for a flooding-based P2P lookup system.

Notations and Abbreviations

aas	-	asymptotically almost surely
ALM	-	Application Level Multicast
c	-	connectivity
cg	-	coverage granularity
DEC	-	Dynamic EquiCast
fe	-	flooding efficiency
GLS	-	The Grid Location Service protocol
IID	-	Independent and Identically Distributed
ISP	-	Internet Service Provider
IP	-	Internet Protocol
LAN	-	Local Area Network
lb	-	load balancing
MAC	-	Media Access Control
MANET	-	Mobile Ad-hoc Network
P2P	-	Peer-to-Peer
POS	-	Protocol-Obedient Strategy
RTT	-	Round Trip Time
TTL	-	Time To Leave
UDP	-	User Datagram Protocol
WAN	-	Wide Area Network

Chapter 1

Introduction and Goals

A *peer-to-peer (P2P)* system is a system that relies primarily on the computing power and bandwidth of the participating nodes (peers) rather than on a central infrastructure. P2P systems are used extensively over the Internet e.g., for file sharing [2, 3, 36] and content distribution [23, 29, 37]. Moreover, P2P protocols are employed for communication in wireless networks; such P2P wireless networks are called *mobile ad-hoc networks (MANETs)*. P2P systems are attractive for several reasons. First, unlike centralized systems, P2P systems can be easily deployed and can grow quickly, since such systems usually do not require any special administrative or financial arrangements [16, 28, 67, 116]. Second, P2P systems achieve high scalability, since they make use of the bandwidth, computation, and storage resources of the participating nodes [16]. And third, P2P systems are typically robust, due to their decentralized and distributed nature [16, 34, 67]. In this dissertation, we focus on *pure* P2P systems, in which all the nodes have symmetric roles, and no infrastructure or "super-peers" exist. We examine such systems over the Internet as well as in MANETs.

In a P2P system, the nodes are typically organized into an *overlay network*, which is a virtual network containing a subset of the connections of some underlying network, e.g., the Internet. Each node typically communicates only with its overlay neighbors. In a structured overlay network, nodes join the overlay according to a specific protocol, whereas in an unstructured one, nodes join the overlay according to some loose constraints. Thanks to the lack of structure requirements, node join and leave events in unstructured overlays can be fast and incur small constant load (independent of the overlay size). Hence, unstructured overlays are suitable for dynamic networks [34].

While P2P computing is a promising architecture for deploying distributed services, it also raises many research problems and challenges [16]. One of the major challenges is achieving scalability, i.e, supporting many users, while incurring small *constant* load on each node, regardless of the number of nodes in the system. However, in most of the current P2P systems, e.g., Chord [115],

Pastry [108], Tapestry [124], and CAN [104], the per-node overhead does increase with the number of nodes.

Another challenge in P2P computing is achieving reliability and efficiency in dynamic failure-prone networks like MANETs and the Internet. In such networks, nodes frequently fail [22], and the message loss rate may be high [98]. In addition, studies have shown that, typically, users frequently join and leave multicast and lookup sessions (such behavior is called *churn*) [11, 12, 13, 109]. Similarly, in MANETs, it is common for mobile wireless nodes to intermittently disconnect from the network, e.g., due to signal blockage. Hence, in both environments, P2P systems need to cope efficiently with high failure and churn rates.

Finally, most of the currently deployed P2P systems do not motivate nodes to cooperate, e.g., contribute upload bandwidth, computation power, or disk space for some other users. Hence, such systems suffer from the problem of "freeloaders", i.e., users who consume resources without contributing anything in return [9, 59]. For example, in the Gnutella P2P file-sharing application [3], nearly 70% of the users share no files [9]. Therefore, current P2P systems, e.g., BitTorrent [37], Avalanche [49], and Gnutella, typically rely on user altruism [9, 59]. For example, in BitTorrent and Avalanche, a node is expected to upload data blocks to other nodes for no return whenever it has available bandwidth [37, 49], and in Gnutella, nearly 50% of all responses are returned by the top 1% of sharing hosts.

Nowadays, user altruism is common since most users are connected to the Internet using static machines via ISPs with a flat pricing model, and hence sending a packet does not incur a cost on its sender. However, these paradigms are changing. First, the increasing access to digital content is expected to drive ISPs to implement a tiered pricing scheme, where high end pricing plans shall allow unlimited downloads and uploads, while lower tier pricing plans shall limit traffic bandwidth [107]. Second, wireless hotspots are proliferating in recent years, and users are increasingly connecting to the Internet and downloading content to mobile devices such as laptops and cell phones. In such networks, pricing is typically based on connection time or transmission volume. Moreover, battery power is a critical resource for mobile devices. Hence, user altruism can hardly be expected in future networks. Therefore, it is important to design P2P systems that work well even when all users are selfish.

In this dissertation, we present four P2P studies addressing the above challenges in different settings. Araneola (see Section 1.1 and Chapter 4) is a scalable reliable application-level multicast system for highly dynamic wide-area environments. EquiCast (see Section 1.2 and Chapter 5) is a wide-area P2P multicast protocol for large groups of selfish nodes. Octopus (see Section 1.3 and Chapter 6) is a fault-tolerant routing protocol for MANETs. And finally, in Chapter 7 (see also Section 1.4), we define metrics for evaluating unstructured overlays for P2P lookup systems, and evaluate different graphs and overlays according to these metrics.

1.1 Araneola: A Scalable Reliable Multicast System for Dynamic Environments

In Chapter 4, we introduce Araneola, a scalable reliable application-level multicast (ALM) system for highly dynamic wide-area environments. Araneola supports multi-point to multi-point reliable communication in a fully distributed manner while incurring constant load on each node. Araneola's overlay approximates a *k-regular random graph*[1] with N nodes. For $k \geq 3$, such a graph is almost always a good expander [45], which implies that (i) its diameter grows logarithmically with N [122]; and (ii) it remains connected after random failures of a linear subset of its nodes and/or edges [50]. In addition, such a graph is generally k-connected, i.e., there are k *disjoint* paths between every two nodes in the graph[2]. For a tunable parameter $k \geq 3$, Araneola constructs and dynamically maintains a basic overlay structure in which each node's degree is either k or $k+1$, and roughly 90% of the nodes have degree k. Empirically, we show that Araneola's overlay achieves the desired properties of k regular random graphs, namely logarithmic diameter, k-connectivity, and high robustness. In particular, we show that Araneola's overlay has a similar diameter and is as robust as graphs generated using a known centralized construction of k-regular random graphs. At the same time, Araneola's overlay construction algorithm is fully distributed and efficient, as each join or leave (or failure) incurs sending roughly about $3k$ messages in a k-degree overlay, regardless of the number of nodes. Remarkably, in dynamic settings, the cost of handling a single join or leave operation *decreases* as the churn rate increases. This is in contrast to virtually all existing structured P2P overlays, with which the overhead for handling joins grows at least logarithmically with the number of nodes.

The low degree of Araneola's basic overlay structure allows for allocating plenty of additional bandwidth for specific application needs. In Section 4.5, we give an example for such a need — communicating with nearby nodes; we enhance the basic overlay with additional links chosen according to geographic proximity and available bandwidth. We show that this approach reduces the number of physical hops messages traverse without hurting the overlay's robustness as compared to completely random Araneola overlays with the same average degree.

Given Araneola's overlay, we sketch out several message dissemination techniques that can be implemented on top of this overlay. We present a full implementation and evaluation of a gossip-based multicast scheme with up to 10,000 nodes. We show that compared to a standard (non-overlay-based) gossip-based multicast protocol, gossiping over Araneola achieves substantial improvements in load, reliability, and latency.

[1] A k-regular random graph with N nodes is a graph chosen uniformly at random from the set of k-regular graphs with N nodes

[2] The probability that a k-regular random graph is not k-connected is $O(N^{2-k})$.

In summary, Chapter 4 makes the following contributions:

- It presents the first efficient distributed algorithm for constructing and maintaining a graph structure that resembles a k-regular random graph and achieves its good properties in dynamic settings.
- It introduces an algorithm that constructs and maintains a richly-connected low degree overlay in which each join or leave operation incurs a constant overhead.
- It describes an overlay-based ALM system that provides an undisrupted multicast service in highly dynamic settings while incurring constant load on each node.
- It features a complete implementation and a thorough evaluation of Araneola running up to $10,000$ nodes on up to 125 machines, in both LAN and WAN, including extensive evaluation of the impact of churn on an ALM system.
- Finally, it constructs an overlay that designates ample bandwidth for each node to communicate with nodes chosen according to application needs, e.g., proximate nodes.

The results of Chapter 4 appear in [93].

1.2 EquiCast: Scalable Multicast with Selfish Users

In Chapter 5, we consider the problem of providing a multicast service in a network with selfish users like the Internet. In such a network, each user tries to minimize its selfish cost, and hence it may not follow a protocol's code. Therefore, in the absence of incentives for cooperation, many users in such a network are "freeloaders", i.e., they consume resources without contributing anything in return.

In order to address this problem, we have designed EquiCast, a wide-area P2P multicast protocol for large groups of selfish nodes. EquiCast tackles the problem of "freeloaders" taking a game theoretic perspective by modeling the system as a *non-cooperative game*. In such a game, nodes are selfish but *rational*, i.e., each user chooses its own *strategy* regarding its level of cooperation so as to minimize its own cost [46]. More specifically, the goal of each node is to receive all the multicast packets while minimizing its sending rate.

We define a special set of *protocol-obedient strategies (POSs)*. Generally speaking, a strategy out of this set allows a node to determine how many connections to maintain and how many packets to send on each connection though it does not allow users to hack the protocol's code or assume that others do so. We believe that it is reasonable to assume that most nodes will run a protocol-obedient strategy (POS), since users usually do not have the technical knowledge required in order

to modify an application code. We prove that if all nodes choose POSs, then each node receives all the multicast packets. Moreover, in this case, no node can unilaterally reduce its cost by changing its strategy to a non-POS. That is, unilateral hacking of the protocol's code cannot reduce a node's cost.

Finally, we prove that EquiCast incurs a low constant load on each node. We note that EquiCast is the *first* P2P multicast protocol that is *formally proven* to enforce cooperation in *selfish environments*.

The results of Chapter 5 appear in [74].

1.3 Octopus: A Fault-Tolerant and Efficient Ad-hoc Routing Protocol

In Chapter 6, we consider the problem of fault-tolerant routing in mobile ad-hoc networks. Such networks consist of mobile wireless nodes that communicate with each other in the absence of infrastructure. In a mobile ad-hoc network, if several of the nodes have an Internet access, then such a network is a wireless extension of the Internet. As opposed to the Internet, however, in a mobile ad-hoc network routing is performed by the end nodes themselves. In addition, in such a network, nodes often intermittently disconnect from the network, e.g., due to signal blockage [20, 84]. Hence, routing in such a network is a challenging task.

We focus on *position-based routing protocols*, in which each node can determine its physical location. Such protocols scale better than non-position-based ones [91]. Typically, the location of each node is stored at some other nodes, which act as *location servers* for that node [56, 91]. When a node wishes to send packets to another node, it first issues a *location query* in order to discover the target's location, and then *forwards* packets to this location.

We present Octopus, a simple and efficient position-based routing protocol that employs synchronized aggregation in order to achieve high fault-tolerance without incurring a high load. Octopus divides the network area into horizontal and vertical strips, and stores the location of each node at all the nodes residing in its horizontal and vertical strips. This approach naturally supports synchronized aggregation: all the nodes in the same strip can learn each other's locations through the propagation of exactly two location update packets along the strip. Note that this location update technique does not require nodes to synchronize their clocks: by knowing its immediate neighbors' locations, a node can determine whether it needs to initiate a strip update. Since synchronized aggregation dramatically reduces the location update overhead, Octopus can update all the location servers at the same high frequency, at a low cost.

On the one hand, Octopus enforces higher redundancy and more freshness of location informa-

tion than previously suggested position-based protocols [63, 83], and hence achieves much better fault-tolerance. On the other hand, by aggregating node locations and synchronizing their propagation, Octopus incurs lower overhead than these protocols in typical scenarios. Moreover, Octopus is highly scalable: for a fixed node density, the number of location update packets sent does not grow with the network size, and when the density increases, the overhead drops.

Octopus has a third important advantage over most previous position-based routing protocols, e.g., [63, 83]: in Octopus, the area in which nodes reside does not need to be pre-known or fixed; it can change at run time. This feature is crucial for rescue missions and battle field environments, in which the borders of the network are not known in advance and are constantly changing. Finally, the redundancy of location information in Octopus has a fourth advantage: nodes use information they have about strip neighbors in order to improve the forwarding reliability. Hence, we eliminate the need to maintain designated information for improving the forwarding reliability.

Finally, we present a thorough empirical evaluation of Octopus using the ns2 simulator with up to 675 mobile nodes. This evaluation shows that Octopus achieves excellent fault-tolerance at a modest overhead: when all nodes intermittently disconnect and reconnect, Octopus achieves the same high reliability as when all nodes are constantly up.

The results of Chapter 6 appear in [94].

1.4 Evaluating Unstructured P2P Lookup Overlays

Unstructured overlay networks incur small constant overhead per single join or leave operation. Hence, they are suitable for dynamic failure-prone environments like the Internet [34]. In addition, lookup systems based on unstructured overlay networks can easily support keyword searches [34]. Therefore, virtually all the currently deployed P2P lookup systems are unstructured ones.

In Chapter 7, we define metrics for evaluating unstructured overlay networks for P2P lookup systems. The metrics we define capture the search dependability and efficiency, and the granularity at which one can control the tradeoff between the two, as well as fairness. According to these metrics, we evaluate different graphs and overlay networks, including a Gnutella graph, a power law random graph, normal random graphs, a 3-regular random graph, and a 3-Araneola overlay. Our study shows that, according to our metrics, a 3-Araneola overlay achieves the best results, and hence it is an excellent solution for flooding-based P2P lookup system.

The results of Chapter 7 appear in [73].

Chapter 2

Related Work

In Section 2.1, we review ALM Systems and fault-tolerant overlay networks and graphs. In Section 2.2, we review P2P multicast protocols for environments with selfish users and incentive-based P2P systems. In Section 2.3, we discuss different ad-hoc routing approaches, and we review leading ad-hoc routing protocols for MANETs. Finally, in Section 2.4, we review P2P lookup systems.

2.1 ALM Systems and Fault-Tolerant Overlay Networks and Graphs

2.1.1 ALM Systems

In recent years, two leading approaches for supporting scalable ALM in dynamic failure-prone networks have emerged: gossip-based (or epidemic) multicast protocols, e.g., [24, 39, 41, 52, 72, 75, 76], and dynamic overlay networks, e.g., [17, 31, 32, 33, 53, 61, 68, 99, 105, 114, 125].

Gossip-based protocols

With gossip-based protocols, each node periodically chooses other random nodes to propagate the information to. Gossip-based protocols usually do not use any infrastructure. Such protocols are highly robust in the presence of failures, and their reliability degrades gracefully as failures amount [41, 85]. They can achieve an average latency of $O(\log N)$ rounds [24, 41]. Moreover, they achieve good reliability (close to 100%) even in dynamic failure-prone settings. However, these protocols also have shortcomings. First, they generally require each node to send each message $O(\log N)$ times [76, 85], which induces a high load. Second, their reliability guarantees are probabilistic, and they generally provide less than 100% reliability even in static failure-free settings [41, 85]. In Chapter 4, we show that gossiping over Araneola eliminates these shortcomings: it has each node send information only k or k+1 times, and guarantees 100% reliability. We

compare the performance of gossiping over Araneola with that of a standard gossip protocol in Section 4.7.1 below, and show that Araneola achieves higher reliability than the gossip protocol while incurring less overhead.

Tree/Mesh-based systems

Most overlay-based ALM systems are tree-based, e.g., [31, 33, 61, 68, 110]. With such systems, no duplicate messages are sent. If the tree topology is mostly stable, and loss-rates are low, then such systems can achieve great performance. However, in the presence of churn, the tree structure frequently becomes partitioned, causing a significant portion of the multicast messages to be lost. Therefore, in order to achieve reliability, such protocols need to detect message loss and recover from it. This can cause recovered messages to be significantly delayed; can induce substantial overhead, especially if failures are frequent; and can inhibit scalability. A second problem with tree-based multicast is uneven load distribution: as recently argued in [30], inner nodes in the tree carry almost all the burden for the multicast, whereas leaf nodes do not share the load.

Pbcast [24] combines best-effort tree-based dissemination, e.g., using IP Multicast, with gossip-based recovery. This approach has the advantages of tree-based ALMs, including fast dissemination and no duplicates in failure-free cases, as well as the robustness of gossip-based protocols. It is therefore effective if a stable tree-based multicast service is available. However, it is also hampered by the difficulty of maintaining stable trees in the presence of high churn.

Mesh-based overlay systems can achieve load balancing and robustness to failures and message loss by including multiple paths between every pair of nodes. SplitStream [30] constructs and maintains a forest of multicast trees, and evenly distributes the forwarding load among all participating nodes. In Bullet [81], nodes self-organize into an overlay mesh, and data packets are distributed to strategic points in the overlay. Nodes are then responsible for locating and retrieving the data packets. Whereas Araneola focuses on providing undisrupted service to nodes that are up despite high churn rates and considerable message loss and failure rates, SplitStream and Bullet are designed for content streaming. Therefore, neither SplitStream nor Bullet were evaluated under the high churn rates we evaluate Araneola. These two systems can induce high overhead: in SplitStream, each join event can incur sending $(k+1)\log(N)$ messages where k is the number of trees in the forest, and in Bullet the average per-node control overhead is approximately 30 Kbps. Moreover, in Bullet, roughly 10% of received data packets are duplicates. In contrast, in Araneola each join or leave operation incurs sending roughly $3k$ messages, and no duplicates are sent. Moreover, in order to achieve high reliability under high churn rates, these two systems need to either use forward error correction techniques, which incur additional overhead, or to employ heavy buffering, which incurs high delay and requires additional disk space. Finally, these two systems

are intended for single-source multimedia transfer and do not support multi-point to multi-point reliable communication as Araneola does.

In the Yoid project [44], nodes auto-configure into two topologies: a shared tree topology for efficient multicast, and a mesh topology for distributing membership information and application content when the tree topology is partitioned. This solution has several limitations. First, the tree configuration is fragile and the discovery of tree partitions may be slow. Second, Yoid trees can be lop-sided, with longer-than-necessary diameters, thus causing high message latency. Finally, membership information is flooded to all the nodes in the system, and hence Yoid is only appropriate for small multicast groups.

Snoeren et al. [112] construct an infrastructure of servers, and each node is connected to k servers from which it receives duplicate packet streams. While this approach achieves high reliability, it also incurs substantial overhead: each packet is sent to each node by k different servers. In contrast, in the absence of packet loss, each Araneola node receives each packet from exactly one node. In addition, Snoeren et al.'s solution is based on server infrastructure, which is not required by Araneola.

ODRI [86] is a dynamic overlay based on de Bruijn graphs that preserves the properties of these graphs namely an average constant in and out degree at each node, a diameter that grows logarithmically with the number of nodes, and good resilience to node and link failures. Whereas in a k-Araneola overlay each node is connected to either k or $k+1$ nodes, in ODRI, each node has k incoming and between k and $O(k \ln N)$ outgoing links. Hence, Araneola achieves better load balancing than ODRI. In addition, the join overhead in ODRI is logarithmic in the number of nodes, whereas in Araneola the join cost is constant.

PRM (Probabilistic Resilient Multicast) [18] is a multicast data recovery scheme based on randomized data forwarding. This recovery scheme incorporated into the NICE protocol [17] achieves reliability of roughly 97% in settings with message loss up to 5% and with up to 5 topology changes per second [18]. In contrast, Araneola achieves full reliability under substantially higher message loss and churn rates.

Finally, we are unaware of a previous P2P multicast system that provides full reliability of message delivery in highly dynamic failure-prone environments. In addition, none of the aforementioned multicast systems was evaluated under the high churn and failure rates that we evaluate Araneola under.

2.1.2 Overlay Structures

Lin et al. [85] construct a static k-Harary graph [60]; such a graph has a logarithmic diameter, a degree of k, and a connectivity level of k, and is therefore an attractive structure for supporting

reliable multicast. Lin et al. study the tradeoffs between a gossip protocol and flooding messages on a static overlay structured like a 4-Harary graph in small fixed networks. Their measurements show that at moderate failure rates, flooding a small overlay achieves the same reliability with a substantially smaller overhead than a gossip protocol. As the failure rate increases, however, the overlay can become partitioned, and the gossip protocol exhibits a much more graceful degradation. This motivates a solution like Araneola, based on a dynamic overlay that detects failures and continuously heals itself.

Recently, several dynamic P2P overlays with logarithmic diameters and bounded node degrees have been suggested, e.g., emulating the Butterfly [89], de Bruijn graphs [70], Small Worlds graphs [79], or random expander graphs with degrees≥ 8 [82]. However, none of these systems can guarantee, with high probability, a lower cost than $O(\log N)$ messages and time for handling joins, since a joining node must search and locate its (random or hashed) joining location prior to joining the system. Chawathe et al. [34] have argued that this logarithmic cost inhibits the scalability of such systems assuming the churn rates measured in Gnutella and Napster [109]. Moreover, the algorithm in [89] is complicated, and the overlay in [82] does not support many concurrent leave operations as Araneola does.

Several overlay structures, e.g., [90, 110], reduce message delivery latency and communication costs by incorporating links between nearby nodes in addition to the random links required for achieving a good overlay. Other overlays, e.g., Pastry [108] and Tapestry [124], achieve local routing by selecting nearby nodes among a large collection of random ones. Land's [8] lookup algorithm achieves a worst case stretch bound of $1 + \epsilon$ by adding local links that increase node degrees by a constant expected factor.

Although adding links to proximate nodes has many benefits, we believe that proximity requirements vary among applications. We therefore advocate a separation of concerns between such specific application needs and generic requirements of wide-area applications. The basic overlay of Araneola addresses the generic needs while incurring a low load, and thus leaves ample bandwidth for the application to address additional needs such as proximity, bandwidth heterogeneity, and so forth. We illustrate this approach in Section 4.5 by extending Araneola's basic overlay with links chosen according to network proximity in order to reduce the latency of message delivery and communication costs. The resulting extended overlay achieves a smaller average degree than [90, 110] and better load balancing than [110].

2.1.3 Centralized Constructions of k-Regular Random Graphs

Araneola builds an overlay structure that approximates a k-regular random graph using a distributed protocol in dynamic environments. Previous algorithms for generating k-regular random

graphs were centralized and static. For example, Bollobas [27] and Bender and Canfield [21] give a centralized construction of a k-regular random graph on N vertices, which works roughly as follows: it duplicates each vertex k times and creates a uniform random perfect matching[1] on these Nk copies of vertices. The resulting graph contains an edge between two vertices, i and j, if the matching contains an edge between copies of i and j. The resulting graph may not be simple, i.e., it may contain self-loops and/or parallel edges. It has been shown [122] that the probability of such a graph being simple is $\exp(-k^2/4)$, and the expected time to obtain a simple k-regular random graph with this algorithm is $O(Nke^{k^2/4})$. McKay and Wormald [92] improve this expected time to $O(N^2k^4)$ using a simple algorithm, and to $O(Nk^3)$ using a complicated and hard to implement algorithm.

Steger and Wormald [113] propose a faster algorithm based on Bollobas's [27] and Bender and Canfield's [21] constructions. This algorithm creates a perfect matching that does not contain self-loops and parallel edges, and hence the resulting graph is always simple. The running time of this algorithm is $O(Nk^2)$. Steger and Wormald prove that if $k = o(N^{1/28})$ then the distribution of the generated graphs is asymptotically uniform[2]. Recently, Kim and Vu [77] have proven that the distribution of graphs generated using this algorithm is asymptotically uniform with k up to $N^{1/3}$.

Araneola is the first distributed and efficient approximation of a k-regular random graph that we are aware of. As opposed to the centralized constructions mentioned above, in which each addition or removal of a single vertex or edge from the graph requires the reconstruction of the graph from scratch, Araneola incrementally incorporates joining nodes and removes leaving ones from the graph, while sending only about $3k$ messages for each such change. In Section 4.4.5, we show that the overlays generated by Araneola have the same diameter and are as robust as the graphs generated using the centralized construction of [77, 113].

2.2 P2P Multicast Protocols for Environments with Selfish Users and Incentive-Based P2P Systems

EquiCast is the first P2P multicast protocol that is formally proven to enforce cooperation in environments with selfish users. We are familiar with only two previous P2P multicast protocols for environments with selfish users [58, 96]. Ngan et al. [96] propose an incentive-based multicast protocol based on detection of selfish nodes and periodic reconstruction of multicast trees that exclude previously misbehaving nodes. However, this protocol induces high overhead. For example, with 500 nodes, the trees' reconstruction requires each node to send 256 control messages

[1] A matching on a graph G is a set of vertex-disjoint edges of G. A perfect matching is a matching that covers all vertices.

[2] The distribution of the generated graphs approaches a uniform distribution as $N \to \infty$.

every two minutes; and when the group size is 2000 nodes, each node sends nearly 400 control messages every two minutes, in addition to data messages. Habib and Chuang [58] propose an incentive-based protocol for media streaming, in which cooperative nodes receive high quality of service whereas "freeloaders" receive low quality streaming. While this protocol rewards cooperation to some extent, it does not solve the problem of "freeloaders". These two solutions, however, consider a different model, in which only a fraction of the nodes are selfish. Moreover, neither is formally proven to enforce cooperation.

Several previous distributed Internet services such as content distribution [37, 49], storage [38], and lookup [34] reward cooperation to some extent by incentivizing cooperative behavior. The BitTorrent [37] and Avalanche [49] content distribution systems support the tit-for-tat strategy, in which a user preferentially uploads blocks of information to users from which it is also downloading blocks. But these systems rely on user altruism, and hence they do not purport to work in a selfish environment where all users are rational and selfish, and every packet incurs a cost on its sender. In the SAMSARA storage system [38], each node is required to contribute as much disk space to the system as it is using, and in the GIA lookup system [34] the quality of service experienced by a node is proportional to its contribution to the system. None of the aforementioned services, however, models the system as a non-cooperative game or formally proves cooperation as we do.

In P2P protocols based on a centralized *reputation system*, e.g., eMule [2] and [25], each node sends to and requests from the system reports about the level of cooperation of other nodes. Hence, a node is motivated to collaborate with other nodes. However, this approach achieves limited scalability [25], since the reputation system continuously communicates with all the nodes.

The BAR-B backup service [10] can tolerate both Byzantine nodes and an unbounded number of selfish nodes by using asynchronous replicated state machine. The replicated state machine approach, however, can support only a limited number of nodes. In addition, this service relies on public key cryptography, which further limits the scalability of this service.

Finally, *cost-sharing* multicast solutions e.g., [42], consider a different model, in which multicast is provided over a dedicated infrastructure, and the infrastructure cost is shared among all nodes. Such an approach, however, is not applicable to P2P systems.

2.3 Ad-hoc Routing Protocols for MANETs

Existing ad-hoc routing approaches can be roughly divided into two categories: *topology-based* and *position-based* [91]. Topology-based protocols do not assume that each node can determine its position. Such protocols usually employ global flooding to distribute either topology information (e.g., DSDV [100]) or queries (e.g., AODV [101], DSR [69], TORA [97], and ZRP [57]), and

hence suffer from limited scalability [83, 91].

By assuming that each node can determine its location, position-based protocols achieve better efficiency and scalability than topology-based ones [91]. Position-based protocols can be classified according to how many nodes act as location servers and how many locations each of them holds [91]. In the *all-for-all* approach used by DREAM [19], every node acts as a location server for all nodes. This approach is fault-tolerant, and is practical in small networks. However, it has been argued that the overhead of this approach is prohibitive in large networks, since location updates are flooded [55, 83].

In the *some-for-some* [56] and *some-for-all* approaches [56, 120], some dedicated nodes act as location servers for some or all other nodes. These approaches are appropriate for failure-free networks, or for settings in which there are reliable servers. However, such approaches are problematic in failure-prone networks, since they are vulnerable to the movement or failure of a single dedicated location server (as explained in [83]).

Octopus employs the *all-for-some* approach, in which each node acts as a location server for some other nodes. Li et al. [83] have shown that this approach can achieve a good tradeoff between reliability and load, and can scale well up to at least 600 nodes. All-for-some-based protocols include GLS [83], GRSS [63], Homezone [48, 117], and [118]. Of these, GLS and GRSS are the only ones that were extensively evaluated in simulations with mobile nodes. Moreover, only GLS was evaluated in settings in which nodes intermittently disconnect from the network, and this study was only conducted in a small network.

Stojmenovic et al. [118] suggest a routing scheme in which each node periodically propagates its position in the north and south directions, and location queries are sent in the east and west directions. Similar approaches were also suggested for efficient content location [119], match-making in sensor networks [14], and as a general scheme for implementing ad-hoc services [95]. However, unlike Octopus, none of these previous works aggregate updates, and they thus miss Octopus's important performance advantage; individually updating so many nodes is bound to induce a prohibitively high overhead [7, 88]. Moreover, of these works, only [119] was evaluated with mobile nodes, and none was evaluated in fault-prone settings. Another difference between Octopus and [118] is that Octopus employs more redundancy by storing node locations at both their horizontal and vertical strips. This additional redundancy yields a quadratic decrease in the probability for query failures. Finally, [118] does not make additional use of the stored location information in order to improve the reliability of forwarding. In fact, we are not aware of any previous ad-hoc routing protocol that exploits location information for more effective forwarding.

The most thoroughly studied position-based protocol thus far, GLS [83], partitions the world into a hierarchy of grids with squares of doubling edge sizes. In each level of the hierarchy, the location of each node is stored at three location servers, for a total of $O(\log N)$ location servers

under uniformity and fixed density assumptions. Under the same assumptions, Octopus stores the location of each node at $O(\sqrt{N})$ location servers (see Section 6.4). In contrast to Octopus, in GLS remote location servers are updated less frequently than close ones. Thanks to the use of more location servers and fresher information, Octopus achieves much higher fault-tolerance than GLS. Thanks to aggregation, Octopus achieves this while incurring lower overhead. Moreover, Octopus is a simpler protocol than GLS.

Although Octopus requires more memory than GLS for storing location information, Octopus's memory requirements are quite reasonable: in our largest experiment, with 675 nodes, location information consumes less than 1KB of memory at each node. Note that in wireless networks, reducing the number of transmissions is most crucial, and 1KB of memory overhead is a small price to pay for the significant reduction in message overhead that Octopus achieves.

In almost all the previous location-based routing protocols, each location update packet includes the location of a single node and updates a single location server. The only exception we are familiar with is GRSS [63]. However, in contrast to Octopus, in GRSS location updates are not synchronized, i.e., several nodes in the same region can initiate a location update simultaneously, thus causing many duplicate packets to be sent. Consequently, as shown in [63], GRSS often fails to achieve lower overhead than GLS. Moreover, as opposed to Octopus, in which each location update packet contains identities of $O(\sqrt{N})$ nodes (assuming the system model described in Section 6.2), in GRSS, a location update packet can contain $O(N)$ node identities. In order to reduce the packet size, GRSS uses Bloom filters. However, this technique may lead to incorrect routing due to false positives [63].

In LAR [80], each node knows only the locations of its immediate neighbors. This approach is efficient when the number of location queries is low. However, when location queries are frequent, this approach is not practical, as location queries may be globally flooded [83].

Finally, some ad-hoc protocols, e.g., Span [35] and GAF [123], reduce energy consumption by allowing nodes to sleep for extensive periods, leaving a minimal set of nodes awake to perform routing. Such an approach employs no redundancy, and hence is inherently not fault-tolerant.

2.4 P2P Lookup Systems

Structured lookup systems, e.g., Chord [116] and Pastry [108], can achieve perfect search reliability, and incur the sending of only $O(\log N)$ messages per search operation. However, such systems incur high joining overhead of $O(M \log(N))$ messages, where M is the number of objects held by the joining node. Assuming $N = 30,000$ and $M = 90$ as in Gnutella [9], a single join operation incurs a prohibitive overhead of more than $1,300$ messages. In addition, structured lookup systems do not support keyword searches, which are highly popular.

Partially structured lookup systems, e.g., KaZaA, usually rely on some infrastructure, e.g., "super-peers". Therefore, such systems can achieve higher scalability compared to pure unstructured lookup systems. However, the infrastructure can be expensive to construct and maintain. Moreover, "super-peers" have high bandwidth consumption. In addition, infrastructure-based systems are much more vulnerable to malicious attacks than pure P2P systems. Moreover, in this chapter, we show that the major problems of pure P2P unstructured lookup systems, e.g., low search efficiency resulting in a high search overhead, which lead to abandoning the pure P2P model for "super-peers", can be eliminated with the use of a good overlay.

Unstructured lookup systems such as Gnutella can scale up to tens of thousands of users, without relying on any infrastructure [34]. In such systems, the search may fail. However, queries usually succeed in locating files due to natural file redundancy [34], that is, popular files are held by many nodes. Search algorithms typically used in unstructured lookup systems are based on flooding and/or random walks [87]. In a random walk, a query is forwarded to a randomly chosen neighbor at each step, until the object is found. While this search technique can incur smaller overhead than flooding, it also dramatically increases the search latency. In addition, in typical dynamic wide-area environments, a random walk usually fails to achieve a similar search reliability to that achieved by flooding. Therefore, most currently deployed P2P lookup systems employ flooding as their search algorithm. In this chapter, we focus on improving the flooding efficiency in unstructured lookup systems.

Lv et al. [87], propose a search algorithm based on multiple random walks, which resolves queries for popular objects almost as quickly as flooding, while reducing the network traffic. However, this search technique is not feasible for low-replicated objects or for failure-prone settings. In addition, Lv et al. evaluate the efficiency of flooding over several graph structures. Their results show that flooding over a normal random graph achieves the best efficiency among the tested graphs. Lv et al., however, do not examine low-degree balanced graphs such as a 3-regular random graph. In this chapter, we show that flooding over such a graph (or an approximation of such a graph) achieves much higher efficiency than flooding over normal random graphs. Moreover, we show that a limited flooding over a 3-regular random graph achieves similar efficiency to that achieved by random walks, while achieving higher reliability and incurring lower latency.

Chapter 3

Methodology

Most of the dissertation (Chapters 4, 6, and 7), focuses on systems, which are mainly evaluated empirically. We evaluate the properties of the different systems using extensive measurements in large-scale environments, such as a large cluster, an emulated environment, or a network simulator. We focus on failure-prone dynamic settings with node and link failures as well as join and leave events.

Chapter 5 includes a theoretical study. The protocol is evaluated using formal proofs. Specifically, the system is modeled as a non-cooperative game, and game theoretic techniques are employed.

We now review the different methodologies we use in each of our studies.

3.1 Methodology Used in Chapter 4, Araneola: A Scalable Reliable Multicast System for Dynamic Environments

The study in Chapter 4 is an empirical one, and is based on evaluation of Araneola in a LAN, as well as over the Internet. We also run Araneola on top of a WAN emulation.

Implementation language and transport protocol. We have implemented Araneola in Java using UDP/IP. No retransmissions are sent, and therefore we do not increase the network load at times of congestion, i.e., when there is high message loss. We use the standard UDP protocol without over-saturating the network.

Evaluation settings. We evaluate Araneola in three environments: (i) in a single LAN in Netbed [121], running up to 10,000 nodes on up to 125 machines; (ii) over the Internet using PlanetLab nodes [102], running 500 nodes on 20 machines; and (iii) WAN emulations over a single LAN in Netbed, running up to 8,000 nodes on up to 100 machines. Our emulated network is based on measurements

of upload bandwidth of P2P clients [109] and measurements of loss rates and RTTs (round trip time) of Internet links [64].

Measurements of fault-tolerance. We study the fault-tolerance and robustness of the Araneola overlay by considering two kinds of failures: communication link failures and node failures. We do so using an offline analysis of overlay snapshots obtained at the end of static experiments with 1000 and 2000 nodes. To study communication failures, we remove random subsets of edges from the overlay graph and analyze the resulting graphs. This allows us to predict Araneola's reliability and latency in the presence of message loss. Similarly, we study Araneola's resistance to node failures by removing random subsets of nodes. In this analysis, no dynamic repairs are made, i.e., after the initial construction of the overlay, no links are added as a result of a node or link failure. Such repairs would have further increased the measured fault-tolerance. As in most previous studies, e.g., [41, 85, 116], we model node and edge failures as *independent and identically distributed (IID)*.

Dynamic settings. Our model for dynamic evaluation is based on studies of user behavior in multicast groups on the MBone [11, 12, 13], and in file sharing applications [109]. These studies model the join and leave rates of most of the nodes using an exponential distribution. Moreover, both studies observe that a small portion of the nodes have substantially longer life times than others. Motivated by these studies, we designate a small subset (roughly 7%) of the nodes as *perseverant*. Perseverant nodes are created at the beginning of the experiment and remain active throughout the experiment. Subsequently, every minute, 50 additional (non-perseverant) nodes are awaken, until all nodes (1000 or 2000) are up. Each non-perseverant awaken node joins the multicast group (becomes *active*) with probability 0.5. Otherwise, the node remains *inactive*. This gradual joining is modeled after the Berkeley session in [12]. Throughout the experiment, each non-perseverant node once a minute flips a coin with probability λ in order to decide whether to change its state from active to inactive and vice versa. We experiment with values of λ ranging from 0.01 (yielding a mean life time of 100 minutes) to 0.15 (giving a mean life time of 6.7 minutes). As a baseline, we also experiment with $\lambda = 0$, in which case nodes do not change their states. There are roughly $\frac{N}{2}$ nodes alive at the end of each experiment with N nodes, regardless of λ, since the join rate is equal to the leave rate.

3.2 Methodology Used in Chapter 5, EquiCast: Scalable Multicast with Selfish Users

Chapter 5 employs game theoretic analysis. We model the system as a non-cooperative game, in which the players are N nodes. Each node chooses a strategy that dictates how it plays the game. We define a special set of *protocol-obedient strategies (POSs)*. Generally speaking, a strategy out of this set must run the protocol as is and can only determine how many connections to maintain and how many packets to send on each connection. Each node is selfish and rational, i.e., it chooses a strategy that minimizes its individual cost, according to some cost function.

3.3 Methodology Used in Chapter 6, Octopus: A Fault-Tolerant and Efficient Ad-hoc Routing Protocol

The evaluation in Chapter 6 uses the ns2 network simulator with CMU's wireless extensions. We run up to 675 mobile nodes. Each node uses the IEEE 802.11 radio and MAC model provided by the CMU extensions, which simulates packet loss in typical MANETs. Each node has a radio range of 250 meters and a throughput of $1\frac{Mb}{sec}$.

The nodes are initially placed uniformly at random in a square universe. In most of our simulations, there are 75 nodes per square kilometer. (Li et al. [83] have experimentally shown that such a node density is required in order to achieve high forwarding reliability.) Each node moves using the random waypoint model used in [83]: it chooses a random destination and moves toward it with a constant speed chosen uniformly between zero and $10\frac{m}{sec}$. When a node reaches its destination, it chooses a new destination and immediately begins moving toward it at the same speed.

For each set of parameters, we run five 300 seconds long simulations, and in each simulation, each node initiates an average of one location query a minute to random destinations, starting 30 seconds into the simulation, and ending at 270 seconds. Our values taken over the five simulations. We also compute the 95% confidence intervals, and show that they are very tight, i.e., the results of the five simulations are very close to each other. This consistency is due to the large number of events in each simulation.

In order to evaluate the reliability of Octopus's forwarding sub-protocol, we run simulations in which data traffic is sent. Our simulation scenario follows the one in [83]. Each node's radio bandwidth is $2\frac{Mb}{sec}$. In each simulation, data traffic is generated by a number of constant bit rate connections equal to half the number of nodes; no node is a source in more than one connection; no node is a destination in more than three connections. Each source sends four 128-byte data packets per-second for 20 seconds. Each simulation lasts 300 seconds, and data packets are sent at random times between 30 and 270 seconds into the simulation.

Finally, in order to evaluate the fault-tolerance of Octopus, we run simulations with connect and disconnect events. We introduce *unstable* nodes, which alternate between being connected and disconnected [83]. Each time an unstable node awakens, it remains connected for a time interval chosen uniformly at random in the range $[0, 120]$ seconds. And when it disconnects, it remains disconnected for a time interval chosen uniformly at random in the range $[0, 60]$ seconds. Thus, at any given time, an average of $\frac{2}{3}$ of the unstable nodes are connected. We experiment with a varying percentage p of *unstable* nodes. The remaining nodes are connected throughout the simulation. We experiment in a fairly large grid of $2.3km$ by $2.3km$. In order to isolate the effect of node disconnections without impacting the density, we fix the average number of connected nodes at a given time at 400. That is, we run $\frac{400}{1-p+\frac{2}{3}p}$ nodes (e.g., 480 nodes when $p = 0.5$).

3.4 Methodology used in Chapter 7, Evaluating Unstructured P2P Lookup Overlays

In Chapter 7, we define metrics for evaluating unstructured overlay networks for P2P lookup systems. We measure each of these metrics (using a serial program) on six undirected graph topologies. In each graph topology, there are 10,000 nodes, in order to allow for a fair comparison.

Chapter 4

Araneola: A Scalable Reliable Multicast System for Dynamic Environments

4.1 Introduction

Our goal in this chapter is to provide a scalable multi-point to multi-point reliable multicast service for large groups in wide-area networks. Examples to applications that require such a service include publish-subscribe applications [23, 29], distributed parallel processing [62], and collaboration applications such as shared document editing [47], whiteboards [66], chat, distributed interactive simulations [65], and multi-player games [5] [103]. Traditionally, IP multicast [40] has been advocated as a solution to scalable multicast. However, due to scalability, reliability, security, and congestion and flow control problems, nowadays IP multicast is mostly unavailable over the Internet [54, 81]. In recent years, Application Level Multicast (ALM) systems have emerged as a promising alternative to IP multicast for scalable wide-area multicast [18, 24, 30, 31, 33, 41, 44, 54, 61, 68, 76, 81, 85, 86, 110, 112]. In such systems, the multicast is supported at the application level, and hence these systems can be deployed over any network without requiring router support.

A protocol deployed in wide-area networks must be able to withstand frequent node failures as well as non-negligible message loss rates [98]. Moreover, studies have shown that, typically, users frequently join and leave multicast sessions [12]; such behavior is called *churn*. A major design goal for our work is therefore coping efficiently with churn. Specifically, we address the following challenges: (i) providing high reliability despite considerable message loss and failure rates while incurring constant load on each node; (ii) incorporating joining nodes and removing leaving (or failing) ones with a low *constant* overhead; and (iii) providing an undisrupted service to nodes that are up despite high churn rates.

We present Araneola, a scalable reliable ALM system for dynamic wide-area environments. Araneola does not rely on any infrastructure such as dedicated servers nor requires router support. Reliability is achieved by constructing a richly-connected overlay and disseminating pertinent in-

formation on multiple *disjoint* paths in this overlay. The number of paths in the overlay can be tuned according to the expected failure and loss rates. Araneola is designed to incur small constant load on each node. To this end, it builds a basic overlay in which each node's degree is bounded by a small constant. Then, this basic overlay can be extended with additional links according to specific application needs, e.g., network proximity. This approach has three advantages: (i) all nodes, including low bandwidth ones, are capable of participating in the basic overlay; (ii) the load on all nodes is similar, so no user is required to contribute more bandwidth than its fair share for the basic overlay; and (iii) nodes have ample remaining bandwidth for connecting to additional nodes according to application needs.

Our search for a robust constant degree overlay leads us to consider k-regular random graphs. A *k-regular random graph* with N nodes is a graph chosen uniformly at random from the set of k-regular graphs with N nodes. In contrast to a normal random graph [27], where node degrees vary, in a k-regular graph, each node's degree is exactly k. For $k \geq 3$, a k-regular random graph is almost always a good expander [45], which implies that (i) its diameter grows logarithmically with N [122]; and (ii) it remains connected after random failures of a linear subset of its nodes and/or edges [50]. In addition, such a graph is generally k-connected, i.e., there are k *disjoint* paths between every two nodes in the graph[1]. In contrast, in order for a normal random graph to be even connected (with high probability), its average degree must be at least logarithmic in N [27]. Note that the diameter, which increases logarithmically, is the only feature of a k-regular random graph that depends on the system size. All the remaining characteristics of k-regular random graphs (connectivity, degree, robustness to random edge and node removals) are independent of N.

We present for the first time an algorithm for constructing and maintaining an overlay that resembles a k-regular random graph in a distributed and efficient manner in dynamic settings. For a given parameter $k \geq 3$, Araneola's basic overlay converges to a graph in which each node has a degree of either k or $k+1$, and no two neighboring nodes have a degree of $k+1$. Empirically, we show that Araneola's overlay achieves the desired properties of k regular random graphs, namely logarithmic diameter, k-connectivity, and high robustness. In particular, we show that Araneola's overlay has a similar diameter and is as robust as graphs generated using a known centralized construction of k-regular random graphs. At the same time, Araneola's overlay construction algorithm is fully distributed and efficient, as each join or leave (or failure) incurs sending roughly about $3k$ messages in a k-degree overlay, regardless of N. Remarkably, in dynamic settings, the cost of handling a single join or leave operation *decreases* as the churn rate increases. This is in contrast to virtually all existing structured P2P overlays, with which the overhead for handling joins grows at least logarithmically with the number of nodes.

[1]The probability that a k-regular random graph is not k-connected is $O(N^{2-k})$.

The low maintenance cost is achieved due to the facts that: (i) each join, leave, or failure is handled locally; and (ii) the selection of random neighbors uses partial membership views maintained by a distributed low cost membership service similar to the ones in [41, 110]. The overhead of the membership service is independent of the number of nodes and of the churn rate.

Many wide-area applications need low-diameter robust overlays. Beyond these general needs, different applications have additional specific needs such as communication with near-by nodes, proximity to content, and exploitation of bandwidth heterogeneity. We believe in separation of concerns between generic requirements on one hand, and needs that vary from application to application on the other. Araneola addresses the former using a low degree overlay, and thus leaves ample bandwidth for applications to address the latter. We illustrate this approach by extending Araneola's basic overlay with links added according to geographical proximity. We show that with this approach, the links in Araneola's overlay traverse substantially fewer physical hops on average. Moreover, we show that if each node in the basic overlay is connected to as little as three or four random nodes, extending the basic overlay with links chosen according to geographical proximity creates an overlay that is as robust to random failures as a basic Araneola overlay with the same average degree, in which all the links are random.

Given Araneola's overlay, it is possible to multicast (broadcast) messages by simply flooding the overlay [85]. This approach yields low latency but also incurs fairly high overhead, as several duplicates of each message are sent to each node. This can be effective if data messages (i.e., payload messages sent by the application) are small, or if bandwidth is abundant, but otherwise, it is wasteful. Alternatively, message identifiers can be flooded instead of the data messages, and each node can request each message that it is missing from exactly one neighbor. This ensures that, in the absence of message loss, each data message is sent exactly $N-1$ times, although many duplicate messages carrying its identifier are sent.

In order to reduce the number of messages sent, one can bundle message identifiers together: each node can locally divide its time into *gossip rounds*, and send one *gossip message* to each of its neighbors in each round, where gossip messages include identifiers of recently received messages. Nodes can then request missing messages from other nodes that have them. This approach is appropriate for software update dissemination, video streaming, and file sharing applications like BitTorrent [37]. Note that with all of the above dissemination techniques, Araneola achieves full reliability of data delivery as long as there are no partitions in the overlay graph. Since, empirically, Araneola's overlay is an expander, Araneola achieves full reliability even under message loss and churn rates that are substantial higher than the ones measured over the Internet.

We empirically evaluate Araneola with the latter (gossip-based) approach, and compare it to a standard gossip-based scalable reliable multicast protocol. Gossiping with Araneola differs from gossip protocols in that with a standard gossip protocol (e.g., [39, 85]), each node chooses different

random nodes to gossip with in each round, whereas in Araneola, each node always gossips with its neighbors in the overlay structure. We show that this difference leads to substantial improvements in load, reliability, and latency.

Contributions. In summary, this chapter makes the following contributions:

- It presents the first efficient distributed algorithm for constructing and maintaining a graph structure that resembles a k-regular random graph and achieves its good properties in dynamic settings.
- It introduces an algorithm that constructs and maintains a richly-connected low degree overlay in which each join or leave operation incurs a constant overhead.
- It describes an overlay-based ALM system to provide an undisrupted multicast service in highly dynamic settings while incurring constant load on each node.
- It features a complete implementation and a thorough evaluation of Araneola running up to $10,000$ nodes on up to 125 machines, in both LAN and WAN, including extensive evaluation of the impact of churn on an ALM system.
- Finally, it constructs an overlay that designates ample bandwidth for each node to communicate with nodes chosen according to application needs, e.g., proximate nodes.

Roadmap. This chapter proceeds as follows: In Section 4.2, we summarize our design goals. Section 4.3 presents the design and pseudo code of Araneola's basic overlay, and Section 4.4 evaluates this overlay. Section 4.5 gives an example how Araneola's basic overlay can be enhanced with additional links chosen according to geographic proximity in order to reduce communication costs, and evaluates this extension over the Internet. Section 4.6 discusses multicasting over Araneola's overlay. Finally, Section 4.7 evaluates a gossip-based multicast implementation.

4.2 Design Goals

The purpose of Araneola is to support scalable reliable multi-point to multi-point communication in dynamic wide-area settings where nodes frequently join and leave (or fail). We have set the following requirements for our service:

- High reliability – 100% reliability as long as the failure and message loss rates do not exceed certain configurable thresholds, and graceful degradation in the face of increasing failure

rates. The reliability should be independent of the number of nodes, i.e., Araneola should withstand a certain failure rate independent of the number of nodes in the system.

- Low latency, increasing at most like $O(\log N)$; the latency should remain low while multiple nodes are joining and leaving (or failing).
- Low constant load on each node, as well as low constant cost for handling joins and failures.
- Quick failure recovery and prompt incorporation of joining nodes.

In addition, Araneola is designed to be suitable for a variety of wide-area applications. We believe that each application has its own considerations for link selection based on application-defined proximity metrics and available bandwidth. Therefore, we construct a generic low-degree overlay that incurs low load on each node, leaving ample bandwidth for each node for communication with additional nodes chosen in an application-specific manner.

Araneola is designed to achieve all the above goals without using any infrastructure, servers, or any elaborate communication mechanism beyond point-to-point UDP communication between pairs of nodes — we assume that every pair of nodes can communicate with each other.

In order to achieve these goals, Araneola strives to build a basic overlay structure with the following characteristics: (i) multiple disjoint paths between every pair of nodes, where the number of paths is a configurable parameter k; (ii) robustness to random removal of a certain percentage of the nodes or edges; (iii) low diameter and average distance, increasing at most like $O(\log N)$; (iv) low bounded degree (3 or more), which leaves plenty of bandwidth for communication with additional nodes according to application needs; and (v) support for local addition and removal of nodes at a constant cost.

As we have seen, k-regular random graphs naturally achieve these goals for $k \geq 3$. Therefore, Araneola strives to construct and maintain an overlay that approximates a k-regular random graph. As noted in Section 2.1.3, creating a perfect k-regular random graph could be difficult and costly. Instead, Araneola is designed to converge to a random graph in which each node has a degree of either k or $k+1$ and no two neighboring nodes have a degree of $k+1$. We show that the desirable graph properties of k-regular random graphs carry over to graphs with this structure.

4.3 Araneola's Overlay

Araneola's protocol has three components: one implements a randomized partial membership service (see Section 4.3.1), the second constructs and maintains the basic overlay (see Section 4.3.2), and the third implements the multicast service (see Section 4.6). All Araneola nodes run these three components. Araneola handles each multicast group independently, i.e., it builds an overlay

structure for each multicast group. Since each group is handled independently, we present the protocol for a single group, and omit the group's name.

4.3.1 The Membership Service

When joining the overlay, a node randomly selects several other nodes to connect to. This requires each node to know some other nodes' identities. To this end, we implement a scalable randomized membership protocol similar to [110], where membership information is gossiped over the overlay's links. Each node maintains a small set of node identities, called a *membership view*, which evolves over time. The size of the membership view S is a predefined parameter. Each node has a log file that contains random node identities received in a previous session. When a new node joins Araneola for the first time it can ask another node for its membership view, and use that as its initial view.

Periodically, each node's membership protocol piggybacks a small amount of membership information on messages sent to the node's neighbors. Specifically, the node sends a certain number of random node identities from its membership view to each neighbor. Upon receiving such membership information from a neighbor, the node adds these node identities to its membership view. Then, if the membership view includes more than S node identities, then random node identities are removed from the membership view until it includes only S node identities.

Whereas in a gossip-based multicast protocol, e.g., Lpbcast [41], each node uses its membership information in every round in order to disseminate multicast data, in Araneola, membership information is used infrequently, only for overlay maintenance. Specifically, a node consults its membership service only when its degree drops below a predefined threshold. Note that, similarly to gossip-based protocols, we could have implemented a gossip-based multicast layer directly on top of the membership service. However, as we explain in Section 2.1.1 and experimentally show in Section 4.7.1, gossiping over Araneola's overlay eliminates the shortcomings of gossip-based protocols, and further improves the scalability of these protocols. Since membership information is used infrequently in Araneola, it can also be disseminated infrequently. Empirically, even under churn rates exceeding those measured over the Internet [109] and the Mbone [12], disseminating membership information once a minute suffices for creating a robust overlay and achieving full reliability of message delivery (see Section 4.7.2). In Section 4.3.3, we calculate the overhead incurred by the membership service. In a typical setting, the per-node membership overhead is 300 bytes per-minute, regardless of the churn rate.

In Section 4.4.4, we evaluate the effect of the membership service on the overlay. We show that the initial distribution of the membership views has a small effect on the quality of the constructed overlay: a k-Araneola overlay is at least $k-1$-connected even when the initial distribution of the

membership views is skewed.

4.3.2 Building and Maintaining the Overlay

The protocol for constructing and maintaining the overlay is composed of three tasks: (i) the *connect task* (see Section 4.3.2) adds new connections when a node's degree is below a configurable parameter called L, which determines the graph's target degree (k); (ii) the *disconnect task* (see Section 4.3.2) tries to reduce a node's degree if it is above L, without causing any node's degree to drop below L; and (iii) the *failure detector* task detects neighboring node failures. This task simply generates an fd_suspect event when messages from a given neighbor fail to arrive for a certain period of time. The failure detector is straightforward and we do not describe it in pseudo-code.

Araneola's data structures are presented in Figure 4.1. The set *neighbors* holds the node's current neighbors in the overlay, with their respective degrees. The degree of a node is the size of its neighbors set, i.e., $|neighbors|$. The set *next_round_connect* contains node identifiers received from redirections of CONNECT requests as explained below. The current time can be read from *clock*. The set *connect_to_node* and the boolean flag *rule2_flag* are used by the reduction task (see Figure 4.3), and are explained below. The parameter L determines the graph's target degree (k), and the parameter H defines the maximum allowed degree for a node. These are configurable parameters: L affects the connectivity and diameter of the overlay, while H affects the overhead of constructing the overlay. A number of timeout values are defined in order to control the frequency at which different events occur.

Data structures:
id – this node's identifier.
neighbors – set of pairs ⟨id,degree⟩, initially ∅.
next_round_connect – set of pairs ⟨id,degree⟩, initially ∅.
clock – the current time.
connect_to_node – set of node identifiers, initially ∅.
rule2_flag – a binary flag, initially 0.
Parameters:
L – target number of neighbors.
H – upper bound on the number of neighbors.
Timeouts: connect_timeout, disconnect_timeout.

Figure 4.1: Araneola's data structures and parameters.

The connect task

When a node's degree is below L, the connect task (see Figure 4.2) periodically attempts to set up as many new connections as it is missing to randomly chosen nodes (lines 1–10). The target nodes are chosen either from the set *next_round_connect* or at random from the local membership view. For each attempted connection, the node sends a CONNECT request (line 9). At bootstrap time, the node issues CONNECT requests to L nodes, and then sleeps for *connect_timeout*. It is expected that during this period enough new connections will be formed, although since some of the chosen nodes may be faulty or overloaded, there may be a need to attempt more connections after the timer expires. The connect task can be awoken by other tasks before the timer expires (line 35).

A node that receives a CONNECT request (line 11) *accepts* it, by calling add_connection, provided that the sum of the sizes of the sets *neighbors* and *connect_to_node* is smaller than H, and otherwise it *redirects* the request, as will be explained shortly. The procedure add_connection adds the sender to neighbors (line 31) and responds with a CONNECT_OK. Upon receiving the CONNECT_OK (line 18), the requester registers the new connection if either its degree is still smaller than H, or the sender of the CONNECT_OK message is in *connect_to_node*. Otherwise, the requester sends a LEAVE message to the sender (line 25). A LEAVE message causes its receiver to remove its connection with the sender (lines 26–27), and wake up the connect task if necessary (lines 34–35).

Redirecting is done by sending a REDIRECT message to the requester, naming the sender's lowest degree neighbor l (line 15). This causes the requester to add l to its *next_round_connect* set (line 17). The next time the requester's connect_task will awaken, it will attempt to connect to l rather than to a random node (line 5). CONNECT and CONNECT_OK messages carry the sender's current degree for initializing the *degree* in the *neighbors* data structure. In addition, every node periodically sends its degree to its neighbors, in order to keep the neighbors data structure up-to-date (this is not shown in the code). A node that voluntarily leaves the system sends a LEAVE message to all its neighbors. An involuntary failure of a neighbor is detected using the failure detector, which generates an fd_suspect event (line 28). When a node detects a neighbor as faulty, it sends that neighbor a LEAVE message and removes the connection by calling remove_connection (line 30).

The disconnect task

With the connect task a node's degree can be as high as H. The disconnect task (see Figure 4.3), which is composed of two rules (*Rule 1* and *Rule 2*), reduces node degrees, so that, eventually, each node's degree is either L or L$+1$, and at most 50% of the nodes have degree L$+1$.

Connect task:
1. **loop forever**
2. $gap \leftarrow \mathrm{L} - |neighbors|$
3. **for** $(i = 0; i < gap; i++)$
4. **if** $|next\_round\_connect| \neq \emptyset$ **then**
5. $n \leftarrow$ element in $next\_round\_connect$
6. remove n from $next\_round\_connect$
7. **else**
8. $n \leftarrow$ random node from membership service
9. send $\langle$CONNECT, $|neighbors|\rangle$ to n
10. sleep (connect_timeout)

Event handlers:
11. **upon** receive $\langle$CONNECT, $d\rangle$ from n **do**
12. **if** $(|neighbors| + |connect\_to\_node| < \mathrm{H})$ **then**
13. add_connection (n, d)
14. **else**
15. send $\langle$REDIRECT, lowest degree neighbor$\rangle$ to n

16. **upon** receive $\langle$REDIRECT, $n'\rangle$ from n **do**
17. $next\_round\_connect \leftarrow next\_round\_connect \bigcup \{n'\}$

18. **upon** receive $\langle$CONNECT_OK, $d\rangle$ from n **do**
19. **if** $(|neighbors| + |connect\_to\_node| < \mathrm{H} \vee n \in connect\_to\_node)$ **then**
20. $neighbors \leftarrow neighbors \bigcup \{n, d\}$
21. **if** $(n \in connect\_to\_node)$ **then**
22. $rule2\_flag \leftarrow false$
23. remove n from $connect\_to\_node$
24. **else**
25. send $\langle$LEAVE$\rangle$ to n

26. **upon** receive $\langle$LEAVE$\rangle$ from n **do**
27. remove_connection(n)

28. **upon** fd_suspect (node_id n) **do**
29. send $\langle$LEAVE$\rangle$ to n
30. remove_connection (n)

Procedures:
Procedure add_connection (node_id n, int d)
31. $neighbors \leftarrow neighbors \bigcup \{n, d\}$
32. send $\langle$CONNECT_OK, $|neighbors|\rangle$ to n

Procedure remove_connection (node n)
33. remove n from *neighbors*
34. **if** $(|neighbors| < \mathrm{L})$ **then**
35. wake up connect task

Figure 4.2: Overlay construction: the connect task.

Rule 1. *Rule 1* removes the connection between a pair of nodes that both have degrees higher than L (Figure 4.3, lines 5–9). Specifically, if a node n's degree is $L+i$, then n attempts to remove i of its neighbors. Neighbors with degrees higher than L are candidates for removal; they are inserted into the set *cands* (line 5). If *cands* contains more than i nodes, the i lowest identifier ones are kept (line 6). If n has a higher id than a node c in *cands*, then n sends a DISCONNECT message to c (line 9). Upon receiving this message (line 17), if c's degree is still higher than L and n is in c's *cands* set, it removes the connection with n, and sends a DISCONNECT_OK message. By checking that n is in c's *cands* set, we ensure that parallel invocations of Rule 1 will not drop c's degree below L. Upon receipt of a DISCONNECT_OK (line 23), n removes the connection with c.

Rule 1 ensures that if from some point onward no nodes join the overlay, then eventually there are no two neighboring nodes that both have degrees higher that L. Every *disconnect_timeout*, each node's *cands* set is set with the i lowest identifier neighbors of the node among the neighbors with degrees higher than L. This ensures that as long as there are two neighboring nodes with degrees>L, each *disconnect_timeout* there are at least two neighboring nodes, a and b, so that $a \in b.cands$ and $b \in a.cands$ (e.g., when a is the lowest-identifier node with degree $> L$, and b is its lowest-identifier neighbor with degree $> L$). Thus, until there are no two neighboring nodes with degrees>L in the overlay, every *disconnect_timeout* at least one link between such two neighboring nodes is removed from the overlay graph, although usually almost all such links will be removed simultaneously. In any case, eventually all such links are removed.

Rule 2. With Rule 1 it is possible for a node to have degree H while all of its neighbors have degree L. This case is solved by *Rule 2*, which is invoked only at a node n when all of n's neighbors' degrees are $\leq$L. With Rule 2, node n chooses its two neighbors with the highest and lowest degrees, h and l, respectively (lines 12–13). If n's degree is at least $l.degree + 2$ and it is not involved in another invocation of Rule 2 (*rule2_flag = false*), then n tries to cause h to shift one of its connections from n to l. But before removing h's connection with n, we ensure that l is willing to accept h's connection. Therefore, n contacts l (rather than h) and asks it to try to connect to h, and to ask h to remove its connection with n. To this end, n sends a $\langle$CONNECT_TO,$h\rangle$ message to l. If upon receiving this message l's degree is still $\leq$L, and l's *rule2_flag* is false (line 26), then l inserts h to *connect_to_node*, and sends it a CHANGE_CONNECTION message. The recipient, h, connects to l by calling add_connection (line 33), provided that its *rule2_flag* is *false*, and sends a DISCONNECT message to n if its degree is higher than L (lines 34–35). Note that h's CONNECT request will be approved by l, since prior to sending the CHANGE_CONNECTION message to h l inserts h to its *connect_to_node* set. This connection with h can increase l's degree, but not to become higher than L+1 since l accepts a CONNECT_TO request only if its degree$\leq$L and its

rule2_flag is *false*. Moreover, note that if l's degree will become higher than L, and n's degree will remain above L, then Rule 1 will eventually reduce l's degree back to L. Finally, each node's *rule2_flag* and l's *connect_to_node* set are set to *false* and $\emptyset$, respectively, after each of them is no longer involved in the current invocation of Rule 2 (Figure 4.2 line 22 and Figure 4.3 lines 22 and 36). rule2_flag ensures that at any moment each node is involved in at most one invocation of Rule 2, and hence no deadlock situations are possible.

Proposition 1. *If there is a time after which no nodes join, leave, fail, or are detected as faulty, then each node's degree is eventually either L or L+1, and at most 50% of the nodes have degree L+1.*

Proof. Rule 1 removes connections between every pair of neighbors with degrees higher than L, without adding new connections. Thus, Rule 1 ensures that eventually, no more than 50% of the nodes have degrees higher than L. Since nodes with degrees lower than H accept new connections, all joining nodes eventually succeed in forming connections with at least L other neighbors. Therefore, the connect task and Rule 1 ensure that eventually, each node's degree is between L and H, and no two neighboring nodes have degree>L. This implies that at least 50% of the nodes have a degree of L.

With Rule 1, it is still possible for a node to have a degree>L+1 when all of n's neighbors have a degree of L. In this case, Rule 2 is invoked at node n, reducing n's degree by one and increasing the degree of n's lowest degree neighbor l by one (in this case l is a random neighbor of n) without changing the rest of n's neighbors' degrees. Now, l's degree equals to L+1 and Rule 1 becomes enabled again, disconnecting the connection between n and l. Thus, after activating Rule 2 and Rule 1 consecutively, n's degree is reduced by 2 while the degrees of the rest of n's neighbors remain L. If n's degree still above L+1, further consecutive activations of the two reduction rules reduce n's degree each time by 2 until its degree becomes either L or L+1. □

Although the worst-case convergence time can be linear in N, in practice, in all of our experiments with up to 10,000 nodes, the overlay converged to a state in which each node's degree is either L or L+1 within less than 10 disconnect timeouts.

The probability for an overlay partition.

The probability that a k-regular random graph is not k-connected is $O(N^{2-k})$ [122]. Empirically, as we show in Section 4.4.5, a k-Araneola overlay achieves a slightly better fault-tolerance to node and link failures than a k-regular random graph. This is since in a k-Araneola overlay the degree of each node is between k and $k+1$ whereas in a k-regular random graph all the nodes have a degree of k. In addition, as we show in Section 4.4.4, the initial distribution of the membership

Disconnect task:

1. **loop forever**
2. sleep (disconnect_timeout)
3. $i \leftarrow |neighbors| -$ L
4. **if** ($i > 0$) **then**
/ Rule 1 */*
5. $cands \leftarrow \{n \in neighbors : n.degree > L\}$
6. **if** ($|cands| > i$) **then** $cands \leftarrow i$ elements of $cands$ with lowest identifiers
7. **foreach** $c \in cands$
8. **if** ($c.id < id$) **then**
9. send $\langle$DISCONNECT$\rangle$ to c
/ Rule 2 */*
10. **if** ($cands = \emptyset \wedge !rule2\_flag$) **then**
11. $rule2\_flag \leftarrow true$
12. $h \leftarrow$ random neighbor among the neighbors with the highest degree
13. $l \leftarrow$ random neighbor among the neighbors with the lowest degree
14. **if** ($|neighbors| \geq l.degree + 2$) **then**
15. $cands \leftarrow cands \bigcup \{h\}$
16. send $\langle$CONNECT_TO,$h\rangle$ to l

Event handlers:

17. **upon** receive $\langle$DISCONNECT$\rangle$ from n **do**
18. **if** ($|neighbors| >$ L $\wedge\ n \in cands$) **then**
19. remove_connection(n)
20. send $\langle$DISCONNECT_OK$\rangle$ to n
21. **if** ($n \in cands$) **then**
22. $rule2\_flag \leftarrow false$

23. **upon** receive $\langle$DISCONNECT_OK$\rangle$ from n **do**
24. remove_connection(n)

25. **upon** receive $\langle$CONNECT_TO, $n'\rangle$ from n **do**
26. **if** ($|neighbors| \leq$ L $\wedge\ !rule2\_flag$) **then**
27. $rule2\_flag \leftarrow true$
28. $connect\_to\_node \leftarrow \{n'\}$
29. send $\langle$CHANGE_CONNECTION,$|neighbors|, n\rangle$ to n'

30. **upon** receive $\langle$CHANGE_CONNECTION,$d, n'\rangle$ from n **do**
31. **if** ($|neighbors| <$ H $\wedge\ !rule2\_flag$) **then**
32. $rule2\_flag \leftarrow true$
33. add_connection (n)
34. **if** ($|neighbors| >$ L) **then**
35. send $\langle$DISCONNECT$\rangle$ to n'
36. $rule2\_flag \leftarrow false$

Figure 4.3: Overlay construction: reducing node degrees.

views has a small effect on the overlay's fault-tolerance. In hundreds of runs, a k-Araneola overlay was always $k-1$ or k connected even when the initial distribution of the membership views was skewed. Moreover, as we show in Section 4.4.2, a 5-Araneola overlay is connected even after a random removal of up to 10% of its edges or after a random removal of up to 15% of its nodes. Therefore, for $k\geq 5$ and $N\geq 1000$, the probability that a k-Araneola overlay becomes partitioned is negligible.

4.3.3 Maintenance Overhead

In Section 4.3.3, we calculate the overhead incurred in a steady state, i.e., in the absence of churn. In Sections 4.3.3 and 4.3.3, we calculate the overhead for the simple case where a single join or leave, respectively, occurs when the system is stable, i.e., each node's degree is either L or L$+1$, and no two neighboring nodes have a degree of L$+1$. In Section 4.4.3, we show that this analysis gives a good estimation for dynamic settings in which the churn rate is low. When the churn rate rises, the overhead decreases because when many join and leave events occur concurrently their costs can be amortized. For example, a join event may increase a node's degree while a leave event is reducing it, eliminating the need for correcting the overlay.

In Sections 4.3.3 and 4.3.3, we denote by p the probability that a node has a degree of L, and the probability that a node has a degree of L$+1$ is $1-p=q$.

Steady state and membership overhead

In a steady state, no control messages are sent. In this case, the overhead is composed out of the membership overhead only. Recall that, every predefined period, each node's membership protocol piggybacks a small number of random node identities on messages sent to the node's neighbors. Specifically, in all of our experiments, every minute, the membership protocol sends 10 random node identities to each of the node's neighbors. We represent each node identity as a 6-cell byte array. Assuming each node has 5 neighbors, the per-node membership overhead is $5{\cdot}10{\cdot}6{=}300$ bytes per-minute. As explained in Section 4.3.1, the per-node membership overhead is fixed, and does not depend in the churn rate.

The overhead for join

We begin by calculating the expected overhead for a single CONNECT request. Assume that node c issues a CONNECT request to node t. We distinguish between three possible cases: (i) t and all of its neighbors have a degree of L; (ii) t has a degree of L and at least one of its neighbors has a degree of L$+1$; or (iii) t has a degree of L$+1$. In the latter case, all of t's neighbors have a

degree of L. The probability for case (i) is p^{L+1}, the probability for case (ii) is $p(1-p^L)$, and the probability for case (iii) is $1-p=q$.

In case (i), c sends one CONNECT message to t and in return, t sends one CONNECT_OK message to c, for a total of two messages. In case (ii), in addition to the CONNECT and CONNECT_OK messages, two additional messages (DISCONNECT and DISCONNECT_OK) are later sent (by Rule 1) in order to reduce the degree of t and one of its neighbors from L+1 to L. Thus, a total of four control messages are sent. In case (iii), after sending the CONNECT and CONNECT_OK messages, t's degree becomes L+2 while the rest of t's neighbors still have degrees of L. In this case, t activates Rule 2. First, t sends to its lowest degree neighbor, l, a CONNECT_TO message with the identity of its highest degree neighbor, h. Then, l sends a CHANGE_CONNECTION message to h with t's identity. In return, h sends a CONNECT message to l and a DISCONNECT message to t. Finally, l sends a CONNECT_OK message to h and t sends a DISCONNECT_OK message to h. Now, the degrees of t and l become L+1 and the degree of h remains L. In the next iteration of the reduce algorithm Rule 1 is applied and either t or l sends a DISCONNECT message to the other and the other replies with a DISCONNECT_OK. The total number of messages sent in case (iii) is thus ten.

The expected number of control messages sent for a single CONNECT request is therefore:

$$2p^{L+1} + 4p(1-p^L) + 10q = 4p + 10q - 2p^{L+1}.$$

Since a joining node sends L CONNECT messages, the expected overhead associated with a single join operation during a stable period is:

$$L(4p + 10q - 2p^{L+1}).$$

The above analysis of the join overhead ignores the possibility for cascading reconnections. In Section 4.4.3, we compare this analyzed join/leave overhead with the measured join/leave overhead, and find that they are very close. That is, cascading reconnections do not have a significant impact on the join overhead.

The overhead for leave

Assume that node l sends a LEAVE message to node t. There are two possible cases: either (i) t has a degree of L; or (ii) t has a degree of L+1. The probability for case (i) is p, and the probability for case (ii) is q. In the first case, l sends a LEAVE message to t. Subsequently, t sends a CONNECT request to a random new node. We showed above that the expected overhead associated with a CONNECT request is $4p + 10q - 2p^{L+1}$. Thus, the expected number of messages sent in the first case is $1 + 4p + 10q - 2p^{L+1}$. In the second case, l sends a LEAVE message to t. However, in this

case, t does not send any messages as its degree becomes L. Thus, the total expected overhead for sending a LEAVE message is:

$$p(1 + 4p + 10q - 2p^{L+1}) + q.$$

The expected number of LEAVE messages a node sends upon leaving the system is:

$$pL + q(L + 1) = L + q.$$

Thus, the expected number of messages sent upon a node leaving the system is:

$$(L + q) * [p(1 + 4p + 10q - 2p^{L+1}) + q].$$

4.4 Evaluation of Araneola's Overlay

We have implemented the code for constructing and maintaining Araneola's overlay in Java using UDP/IP. In our experiments, we set the connect_timeout to 5 seconds and the disconnect_timeout and the connect_to_timeout to 30 seconds. Membership information is gossiped once a minute. At bootstrap, each node's membership view contains ten node identities chosen uniformly at random. In this section, we evaluate Araneola's overlay on a single LAN in Netbed [121]. In the next section, where we extend Araneola to exploit network proximity, we evaluate Araneola's overlay also on a WAN. We begin our study, in Section 4.4.1, by evaluating Araneola's overlay in a static setting; we study the impact of L and H on the overlay as well as the overlay's scalability. In Section 4.4.2 we study the overlay's fault-tolerance. In Section 4.4.3, we measure the join and leave overhead in experiments with high churn. In Section 4.4.4, we evaluate the effect of the membership service on the overlay. Finally, in Section 4.4.5, we compare Araneola's overlay with k-regular random graphs constructed using a centralized algorithm.

4.4.1 Static Evaluation

In our static evaluation, all the nodes are created simultaneously, and remain up throughout the experiment. Each experiment lasts 5 minutes. Empirically, we saw that within this time the overlay converges to a stable state, in which each node's degree is either L or L$+1$ and no two neighboring nodes have a degree of L$+1$. Each experiment (with a given number of nodes and choice of parameter settings) was run at least 4 times, for a total of several dozens.

The impact of L

Araneola's parameter L affects the load imposed on each node. In Section 4.3.3 above we have shown that the join/leave overhead grows roughly linearly with L. Additionally, increasing L increases the multicast overhead, since data or gossip messages are sent on all links.

Nevertheless, increasing L yields a number of benefits. First, it improves the overlay's connectivity and robustness. In Section 4.4.2, we show that when L$= 5$, the overlay generally remains connected after random removal of 15% of its edges or nodes, while when L$= 4$, it remains connected after the removal of only about 10% of the edges or 7% of the nodes. Second, increasing L reduces the overlay's diameter. Note that the connectivity and robustness of a k-regular random graph with a given k is independent of the number of nodes. Therefore, we can set the value of L regardless of the number of nodes in the system. The value of L, however, has a small effect on the overlay's diameter. Below, we examine the relationship of Araneola's overlay's diameter with that expected in a k-regular random graph for different group sizes.

Wormald [122] gives the following formula for the expected diameter of a k-regular random graph: the diameter D *asymptotically almost surely (aas)*[2] satisfies:

$$1 + \lfloor \log_{k-1} N \rfloor + \lfloor \log_{k-1}(\frac{(k-2)}{6k} \log N) \rfloor \leq D \leq 1 + \lceil \log_{k-1}((2+\epsilon)kN \log N) \rceil.$$

To understand the impact of L, we experiment with 8000 nodes (on 100 Netbed machines) for values of L ranging from 3 to 10, and measure the diameter of each overlay. In Table 4.1, we report the highest overlay diameter measured for each value of L, and compare it to the formula above. We see that the highest diameter of Araneola's overlay occurs in the range predicted by the formula.

L	Expected diameter range in L-regular random graphs [122]	Highest measured Araneola diameter
3	13–19	13
4	9–13	9
5	7–11	8
6	6–9	7
7	6–9	6
8	5–8	6
9	5–8	6
10	5–8	6

Table 4.1: The impact of L on Araneola's diameter versus Wormald's formula, 8000 nodes.

Increasing the value of L has a third benefit— it constructs an overlay that more closely approximates a regular graph, in that a higher percentage of the nodes have a degree of L, as shown in Figure 4.4.

In most of the experiments we present below, we set L to 5. We chose this value because it provides a good balance between the desired properties: the load imposed on each node is still

[2]A property holds aas if the probability that it holds approaches 1 as $N \to \infty$.

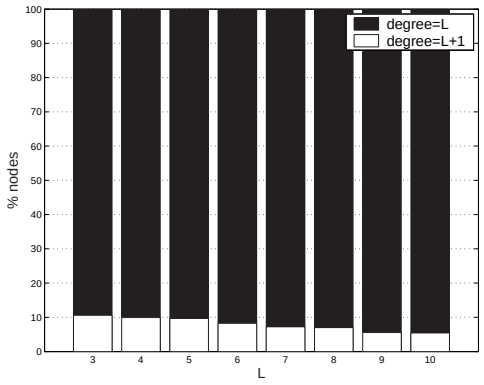


Figure 4.4: Degree distribution, 8000 nodes.

modest, and the overlay's diameter is small. Moreover, as we shall see below, it yields a robust overlay (twice as resilient to node failures as with L$= 4$) and achieves 100% reliability at join and leave rates exceeding those measured on the MBone [12].

The impact of H

Next, we examine the impact of the parameter H. Like L, the choice of H does not depend on the system size. This is since the expected number of control messages received by each node does not vary with the number of nodes. We experiment with $N = 1000$, L$= 5$, and different values of H. We observe that in order to obtain a reasonable overhead, H needs to be at least L$+5 = 10$. When H is lower than 10, we get a high overhead— some nodes send hundreds of CONNECT requests before finding a node with degree lower than H. This occurs since nodes run the reduce task only once in 30 seconds, in the interim, many node's degrees can rise above L, especially in our static experiments where all nodes are created simultaneously. When H is set to L$+5 = 10$, however, this problem is eliminated and the average number of control messages received by each node is between 8 and 9, independently of N. The number of control messages received by each node is normally distributed. We did not observe significant differences among values of H ranging from 10 to 20: for all values of H between 10 and 20, the average number of control messages received was between 8.3 and 8.6. Similar results were obtained for L equal to 4 or 6. We therefore henceforth fix the value of H to be L$+5$.

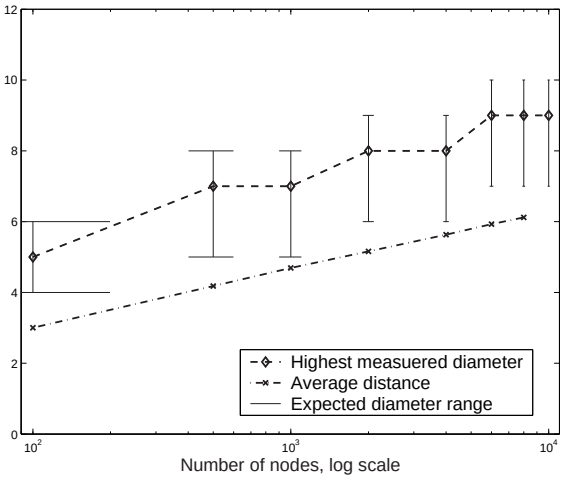
Overlay properties and scalability

In order to understand Araneola's scalability, we vary N, the group size, from 500 nodes (on 10 Netbed machines) to $10,000$ nodes (on 125 Netbed machines). L and H are set to 5 and 10, respectively. At the end of each experiment, we take a snapshot of the overlay structure, and then analyze its properties offline. We measure node degrees as well as the overlay's diameter, average

distance, and connectivity. The results are summarized in Figure 4.5[3]. The first column in the table shows the percentage of nodes whose degree is L (i.e., 5). The remaining nodes' degrees are L$+1$. For all group sizes, over 90% of the nodes have degree L. The percentage of nodes with degree L does not seem related to N.

N	% Nodes degree=5	Measured diameter	Expected diameter	Avg #paths
100	98	5	4–6	5.00
500	91.8	6–7	5–8	5.01
1000	91.4	7	5–8	5.01
2000	92	7–8	6–9	5.01
4000	91.45	8	6–9	5.01
6000	90.42	8–9	7–10	5.01
8000	90.33	9	7–10	5.01
10000	90.36	9	7–10	—

(a)



(b)

Figure 4.5: Scalability of Araneola's overlay.

The second column presents the (smallest and largest) measured diameters for every value of N. The top curve in the graph depicts the highest measured diameter for each value of N, where the x axis is given in logarithmic scale. Note that this value does not necessarily increase when we increase the group size, and hence there are plateaus in this curve. We observe that Araneola's diameter indeed grows logarithmically with N as Wormald's formula predicts; in all of our experiments, Araneola's diameter occurs (again) in the expected range, which is listed in the next column and depicted using range bars in the graph. When flooding multicast messages over the overlay's links, the diameter gives a measure for the *worst case* latency (in the absence of failures and message loss), whereas the average latency depends on the average distance between two nodes in the overlay. This average is presented in the bottom curve in the graph, and we see that it also increases logarithmically with N.

Finally, we measure the overlay's connectivity. In over 90% of our experiments, the overlay is 5-connected, i.e., there are at least 5 disjoint paths between every pair of nodes. In the few cases where the connectivity was less than 5, there were at most 4 nodes with a connectivity of 4, whereas the rest of the nodes had a connectivity of 5. The average number of node-disjoint paths between every pair of nodes is presented in the last column in the table.

[3]We did not analyze the average distance and connectivity for the experiments with $N = 10,000$.

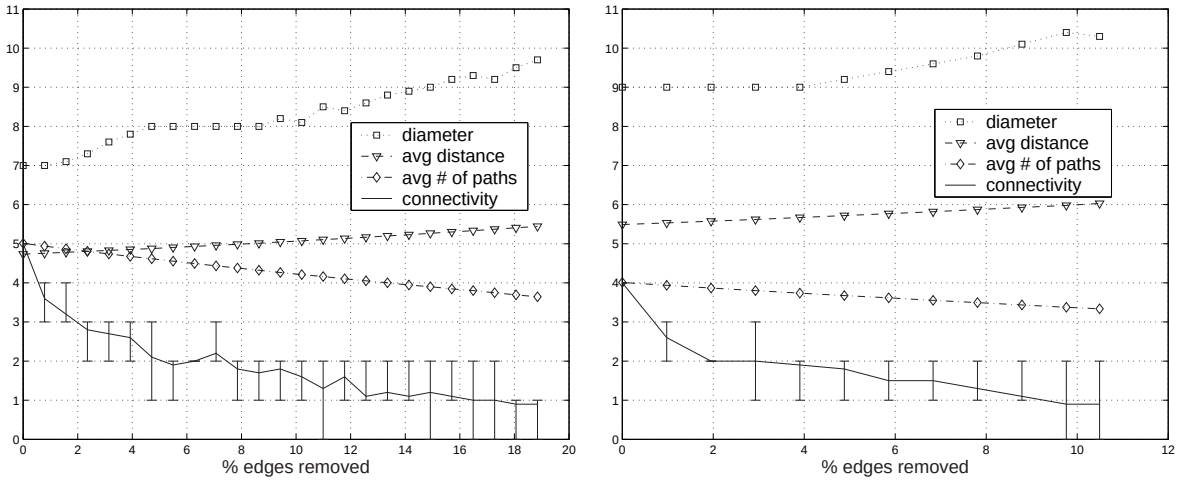
4.4.2 Fault-Tolerance and Graceful Degradation

We now study the fault-tolerance and robustness of the Araneola overlay. We consider two kinds of failures: communication link failures and node failures. We study the overlay's robustness with an offline analysis of the overlay snapshot obtained at the end of static experiments with 1000 and 2000 nodes. To study communication failures, we remove random subsets of edges from the overlay graph and analyze the resulting graphs. This allows us to predict Araneola's reliability and latency in the presence of message loss. Similarly, we study Araneola's resistance to node failures by removing random subsets of nodes. Note that in the analysis in this section no dynamic repairs are done, i.e., after the initial construction of the overlay no links are added as a result of a node or link failure. Such repairs would have further increased the overlay's fault-tolerance.

As in most previous studies, e.g., [41, 85, 116], we model node and edge failures as *independent and identically distributed (IID)*. For node failures, the IID assumption has no significance since the overlay structure is random. Moreover, Bhagwan et al. have found that host failures are indeed independent [22]. For edge failures, the IID assumption fails to capture a situation in which some nodes have poorer links than others. Nevertheless, we show in Section 4.7.3 that even in WAN-like settings where some nodes have only poor links, Araneola exhibits similar performance as when message loss is IID. Designing an overlay that explicitly withstands correlated edge failures can be a consideration for application-specific extensions of Araneola, and it is beyond the scope of this chapter.

Communication failures

We first analyze the impact of edge removals on the overlay with L$= 5$ and $N = 1000$. This overlay has 2547 edges. For each percentage $p \leq 50$ of the edges, we remove 10 different random subsets consisting of p% of the edges from the overlay graph. The overlay becomes partitioned for the first time in one of the ten experiments removing 11% (280) of the edges, and then in one of the experiments removing 15% (380). In both cases, a single node became disconnected from the rest. Figure 4.6(a) shows how the removal of up to 19% of the edges affects the overlay's characteristics. For each p in this range, the overlay is partitioned in at most one out of ten experiments in which p% of the edges are removed. We observe that the average diameter increases from 7 to about 8 when 5–10% of the edges are removed, and to 9 when 15% of the edges are removed. The average distance increases more gradually, suggesting that message loss has a moderate effect on the average latency. The average number of disjoint paths also decreases gradually with the failure rate. The bottom curve illustrates the average connectivity. The bars around each data point show the maximum and minimum connectivity observed in experiments with this p; when the minimum goes does to 0, there was a partition in one of the 10 experiments. We next experiment with L$= 4$



(a) L= 5. (b) L= 4.

Figure 4.6: Resilience of Araneola's overlay to edge removals, 1000 nodes.

and $N = 1000$. The overlay is less robust in this case— it partitions in more than 10% of the cases whenever $p > 11\%$. Figure 4.6(b) shows the overlay's degradation when up to 11% of the edges are removed.

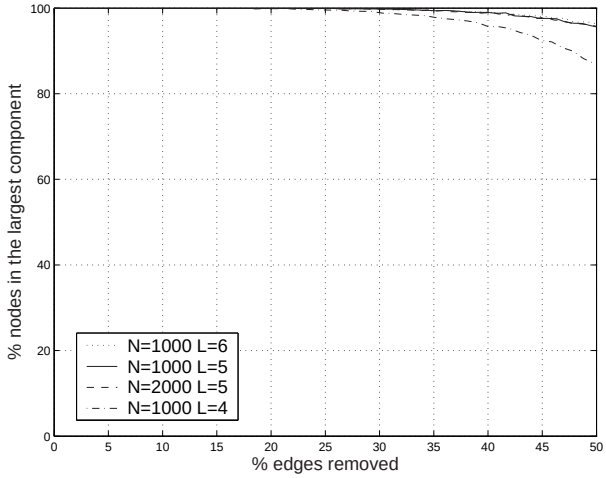
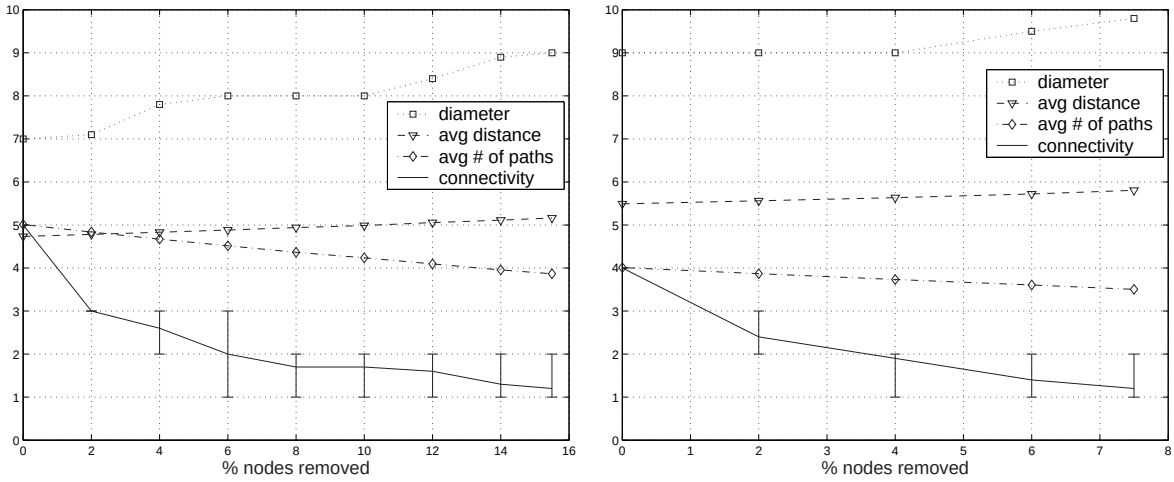


Figure 4.7: Graceful degradation of Araneola's overlay under edge removals, 1000 nodes.

We next examine how many of the nodes are still connected to each other, i.e., what is the size of the largest connected component in the graph. Figure 4.7 depicts the average size of the largest connected component after random edge removals for L= $4, 5, 6$ with $N = 1000$ and for L= 5 with $N = 2000$. We can clearly see that the overlay's resilience to the removal of a given percentage of its edges is *completely independent of* N, as is expected in k-regular random graphs [50]: the

curves for $N = 2000$ and $N = 1000$ (both with L$= 5$) are not distinguishable. As expected, the value of L does impact the overlay's robustness, but the difference between L=5 and L=6 is negligible. Remarkably, for L$= 5$, after the removal of up to 38% of the edges, 99% of the nodes are still connected to each other, and only 1% of the nodes are partitioned from the rest.

Node failures



(a) L$= 5$. (b) L$= 4$.

Figure 4.8: Resilience of Araneola's overlay to node removals, 1000 nodes.

We now turn our attention to node failures. Figure 4.8(a) shows how node removals affect the properties of an overlay with 1000 nodes and L$= 5$ when up to 15% of the nodes are removed. None of the experiments with up to 15% removed nodes resulted in partitions. The overlay becomes partitioned in two of the ten experiments in which 16% (160) of the nodes are removed. This suggests that even if 15% of the nodes running Araneola fail during the brief time interval that it takes to detect and recover from failures (e.g., one minute), Araneola can continue to deliver messages reliably to surviving nodes. As with edge removals, the overlay exhibits graceful degradation: the diameter and average path length increase moderately, while the average number of disjoint paths moderately decreases. When L$= 4$, the overlay is half as robust to node failures as with L$= 5$. It becomes partitioned in two of the ten runs with 8% of the nodes removed. Figure 4.8(b) shows the overlay's degradation when up to 7.5% nodes are removed.

In Figure 4.9, we examine the size of the largest connected component that survives following node failures, for L$= 4, 5, 6$ with $N = 1000$ and for L$= 5$ with $N = 2000$. Again, the overlay's resilience shows exactly the same trend with $N = 1000$ as it does with $N = 2000$. This suggests

that Araneola's resilience to simultaneous failures of a certain percentage of its nodes is also independent of N. When L$=5$, the largest component still includes 99% of the nodes following the failure of up to 38% of the nodes. When L$=4$, 99% of the nodes are still connected following the failure of 28% of the nodes. When 50% of the nodes fail, the largest component with L$=5$ still includes over 95% of the nodes, and with L$=4$, it includes 87%. As with edge removals, increasing L from 5 to 6 achieves only slightly better robustness to node removals when there is an unrealistically high failure percentage.

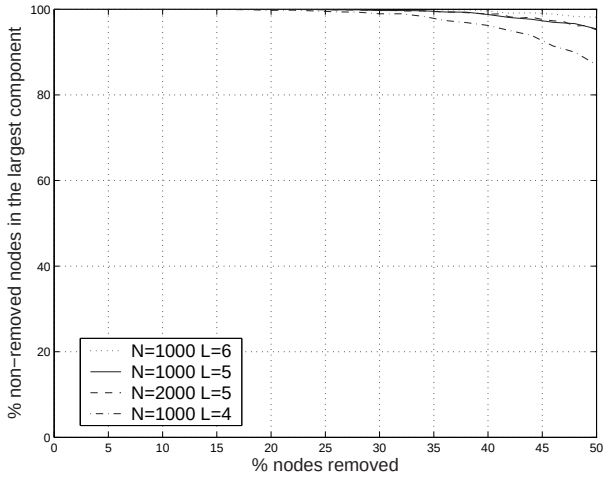


Figure 4.9: Graceful degradation of Araneola's overlay under node removals, 1000 nodes.

Setting L

In Sections 4.4.2 and 4.4.2, we showed for L equal to $4/5/6$ that Araneola's overlay remains connected if the failure-rate does not exceed a certain threshold. Assuming an upper bound on the failure rate, one can choose the minimal value of L that ensures a connected overlay. We note that such a bound is not always known. However, as we show in Sections 4.4.2 and 4.4.2, beyond the failure threshold Araneola's overlay exhibits graceful degradation in the face of increasing failure rates, and therefore inaccurate setting of L has a moderate effect on the overlay's connectivity. Moreover, in Sections 4.4.3 and 4.7.3, we show that with L equal to 5, Araneola's overlay remains connected despite churn rates exceeding the ones measured over the Internet and over the Mbone and in a WAN-like setting, respectively. In this chapter, we do not deal with dynamically adapting L according to changing churn and failure rates. Note that this approach is also used in other studies of scalable multicast, e.g., both the number of trees/stripes (k) in SplitSream [30] and the number of nodes with which each node exchanges digests in Bullet [81] depend in the expected failure rates.

4.4.3 Dynamic Evaluation

Methodology

Our model for this evaluation is based on studies of user behavior in multicast groups on the MBone [12], and in file sharing applications [109]. These studies model the join and leave rates of most of the nodes using an exponential distribution. Moreover, both studies observe that a small portion of the nodes have substantially longer life times than others. However, these studies greatly differ in the mean life times they measure: the mean life time measured on the MBone is generally short, e.g., 7 minutes in a typical multicast session, whereas the average measured life time in a file sharing application is roughly one hour.

Saroiu et al. [109] found that only 20% of the nodes in a P2P lookup system have an uptime of 93% or more. Motivated by this study, we designate a small subset (roughly 7%) of the nodes as *perseverant*. Perseverant nodes are created at the beginning of the experiment and remain active throughout the experiment. Subsequently, every minute, 50 additional (non-perseverant) nodes are awaken, until all nodes (1000 or 2000) are up. Each non-perseverant awaken node joins the multicast group (becomes *active*) with probability 0.5. Otherwise, the node remains *inactive*. This gradual joining is modeled after the Berkeley session in [12]. Throughout the experiment, each non-perseverant node once a minute flips a coin with probability λ in order to decide whether to change its state from active to inactive and vice versa. We experiment with values of λ ranging from 0.01 (yielding a mean life time of 100 minutes) to 0.15 (giving a mean life time of 6.7 minutes). As a baseline, we also experiment with $\lambda = 0$, in which case nodes do not change their states. There are roughly 1000 nodes alive at the end of each experiment with $N = 2000$, (and respectively, 500 when $N = 1000$), regardless of λ, since the join rate is equal to the leave rate. In all the dynamic experiments, we set L to 5 and H to 10.

Join/Leave overhead

We now examine the cost of constructing and maintaining the overlay. This overhead is composed of control messages and membership overhead. The membership protocol piggybacks a small and constant (and hence independent of the churn rate) number of bytes on messages sent to the node's neighbors, as calculated in Section 4.3.3. In this section, we measure the join/leave overhead. The size of control messages is fixed, and consists of less than ten bytes. Therefore, we measure the cost of constructing and maintaining the overlay in terms of the number of control messages. We count the total number of control messages received by all the nodes throughout the experiment, and divide this number by the number of joins and leaves occurring in that experiment. We do not separately measure the overhead for join and leave since we cannot fully distinguish between the two. E.g., when a node receives a CONNECT message, we do not know whether to attribute this

message to a prior LEAVE event that reduced a node's degree, or to a new node trying to join the overlay. There are roughly 1000 more join events than leave events in experiments with $N = 2000$ (respectively 500 in experiments with $N = 1000$). Table 4.2 shows the exact number of join and leave events for experiments with 2000 nodes.

λ	# of join events	# of leave events
0.01	1411	387
0.025	2005	955
0.05	2908	1872
0.075	3825	2768
0.1	4690	3650
0.125	5480	4495
0.15	7965	7029

Table 4.2: The number of join and leave events in experiments with 2000 nodes.

Figure 4.10 shows the overhead measured for different values of λ with $N = 1000$ and $N = 2000$. Remarkably, the overhead *decreases* as the rate of such events increases, the only exception occurring when λ increases from 0 to 0.01. This rise is due to the facts that (i) when $\lambda = 0$, no leave events occur, and (ii) the overhead associated with a leave operation is bigger than the overhead associated with a join operation (see Section 4.3.3). But in general, the overhead associated with a join or leave operation decreases as the churn rate rises because when many join and leave events occur concurrently, their costs can be amortized. E.g., a join event may increase a node's degree while a leave event is reducing it, eliminating the need for correcting the overlay. Furthermore, we observe that the overhead does not increase with N. This is especially impressive given that the overhead for handling joins in structured overlays based on DHTs increases logarithmically with the number of nodes.

Theory versus practice. In the Section 4.3.3, we analyzed the expected number of control messages incurred by a single join or leave operation occurring after the system has stabilized. We now compare this analyzed overhead to the above measured join/leave overhead.

We found out that the expected join and leave overhead during a stable period is: $L(4p+10q-2p^{L+1})$ and $(L+q)*[p(1+4p+10q-2p^{L+1})+q]$, respectively, where p is the percentage of nodes with degree L and $q = 1-p$. Empirically, when the system is stable and L is set to 5, roughly 92% of nodes have a degree of L (see Figure 4.5). Substituting 5 for L, 0.92 for p, and 0.08 for q, we get that the expected join and leave overhead is 16.3 and 20.4 messages, respectively.

When $\lambda = 0$, no leave events occur. The measured average cost per join operation in this case is 15.6, which is close to the expected overhead (16.3). The difference between the expected

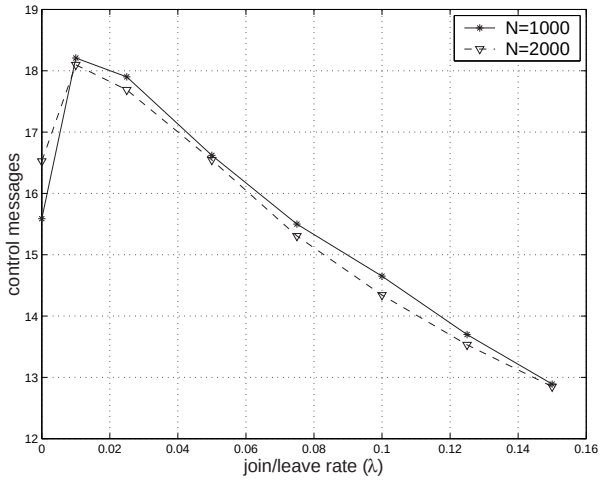


Figure 4.10: Average cost per join/leave with increasing churn rates for different group sizes, L=5.

overhead and the measured one stems from the fact that in a static experiment (when $\lambda = 0$), a node with a degree lower than L may receive a CONNECT request from some other node, reducing the number of CONNECT requests it needs to issue itself.

When $\lambda = 0.01$, the system is similar to a stable system, as the rate of join and leave operations is low. In an experiment with $N = 2000$ and $\lambda = 0.01$ there were 1411 join events and 387 leave events. Thus, the expected overhead for a join/leave operation in this experiment is: $(1411 * 16.3 + 387 * 20.4)/1798 \approx 17.2$. Indeed, the measured overhead in this case, 18.2, is close to the expected one.

4.4.4 The Effect of the Membership Service on the Overlay

In order to evaluate the effect of the membership service on the overlay, we run an experiment with 1000 nodes using the setting of Section 4.4.1 with the exception that in this experiment the initial distribution of the membership views is skewed as follows: at bootstrap, each node's (including a node that is inactive at bootstrap) membership view contains ten node identities chosen uniformly at random out of a set that including only 10% of the nodes, that is, only 10% of the nodes appear in the initial views.

We run this experiment five times in a static setting and five times in a dynamic setting according to the methodology described in Section 4.4.3. We compare these overlays to 1000-node overlays obtained using the setting of Section 4.4.1, where each node's initial view including ten node identities chosen uniformly at random among all 1000 nodes. In all of these experiments, L and H are set to 5 and 10, respectively. We summarize our results in Table 4.3.

As the table shows, in static experiments, the initially skewed distribution of the membership

Initial distribution of membership views	Static/Dynamic	Diameter	Connectivity
skewed	static	7–8	4–5
uniform	static	7	5
skewed	dynamic	7	5
uniform	dynamic	7	5

Table 4.3: The effect of an initially skewed distribution of membership views on the overlay.

views has a small effect on the overlay: whereas all the overlays obtained using an initially uniform distribution are 5-connected and have a diameter of 7, the overlays obtained using a skewed initial distribution have a connectivity of 4 or 5 and a diameter of 7 or 8. In dynamic settings, the initial distribution of membership views has no effect on the properties of the overlay. Below, we explain these results.

In an experiment with an initially skewed distribution, all the nodes' initial L connect requests are sent to 10% of the nodes. However, each of the nodes in this set can maintain only up to H connections. Upon refusing to accept a connection (due to a high degree), the target node n sends its membership view to the node $\hat{n}$ that issued the connect request, and also adds $\hat{n}$ to its membership view. Assume now that another node n' sends a connect request to n. n rejects this request (due to its high degree), and sends its membership view to n'. Now, n' can send a connect request to $\hat{n}$, and $\hat{n}$ will accept this request. Hence, by limiting each node's degree and by sending the membership view upon a rejection of a connect request, our construction protocol overcomes an initially skewed distribution of the membership views.

In a dynamic setting, an initially skewed distribution of the membership views affects only on the first join operation of each node. Since i) empirically, the views' distribution becomes uniform over time; and ii) prior to leaving the overlay each node saves its membership view to a log file, subsequent join operations will create random links. In addition, leave operations lead to the destruction of non-random links, which are replaced by random links created by subsequent join operations. Therefore, in a dynamic setting, join and leave operations "heal" the overlay, and hence the initial distribution of the membership views has no effect on the overlay's properties.

4.4.5 Comparison with k-Regular Random Graphs

We have observed that Araneola's basic overlay achieves the important mathematical properties of k-regular random graphs, namely logarithmic diameter, k-connectivity, and high robustness. In this section, we compare these properties of Araneola overlays to those measured in centrally constructed k-regular random graphs. Specifically, we compare Araneola overlays to L-regular random graphs created by the algorithm of [77, 113] (as described in Section 2.1), for $N = 1000$

and L = 4, 5, and 6. We summarize our results in Table 4.4.

Note that an Araneola overlay contains slightly more edges than the corresponding L-regular random graph, since in Araneola roughly 90% of the nodes have a degree of L, and the rest have a degree of L+1. E.g., when L= 4, 5, and 6, an Araneola overlay with 1000 nodes contains on average 49, 43, and 38 (respectively) more edges than an L-regular random graph with 1000 vertices.

Overlay	Highest diameter	Avg distance	Avg # of disjoint paths
4-regular random graph	11	5.63	4
Araneola, L=4	**11**	**5.49**	**4.01**
5-regular random graph	7	4.71	5
Araneola, L=5	**7**	**4.69**	**5.01**
6-regular random graph	6	4.18	6
Araneola, L=6	**6**	**4.16**	**6.01**

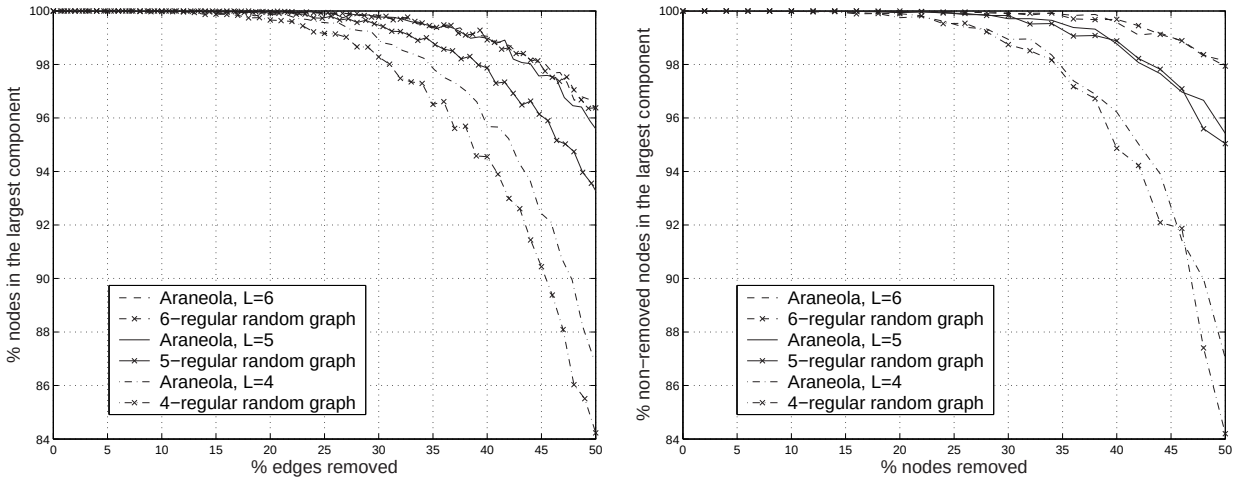
Table 4.4: Araneola versus a centralized construction of L-regular random graphs, 1000 nodes.

The first column in Table 4.4 shows the highest diameter measured for each type of overlay. In all of our experiments, the diameter of the Araneola overlay is identical to the corresponding L-regular random graph. The next column presents the average distance between two nodes in the overlay. In all of our experiments, this distance is slightly smaller in Araneola than in the L-regular graph. The average distance between two nodes in the overlay determines the average latency in which multicast messages are received, and hence this parameter is important. An even more important parameter is the average number of disjoint paths between two nodes in the overlay, presented in the last column of the table. This number determines the robustness of the overlay/graph. In all of our experiments, Araneola contains on average slightly more disjoint paths than the L-regular random graph, again, this is due to the slightly larger number of Araneola edges.

In order to further compare the robustness of Araneola to that of L-regular random graphs, we remove random subsets of edges/nodes from the different Araneola overlays and L-regular random graphs and analyze the resulting graphs. We present our results in Figure 4.11. As the figure shows, in all of our experiments, Araneola achieves the same robustness as the L-regular random graph or slightly better.

4.5 Example of Application-Specific Extension: Exploiting Network Proximity and Bandwidth Heterogeneity

Araneola's basic overlay, like many P2P systems, treats all nodes and all communication links equally: all nodes have almost the same degree, and all links have an equal likelihood of being



(a) Removing edges, largest component.

(b) Removing nodes, largest component.

Figure 4.11: Robustness of Araneola versus centralized construction of L-regular random graphs, 1000 nodes.

used. In reality, however, node capabilities and communication channels are diverse. A wide-area network is typically structured as a collection of LANs, where communication in each LAN is orders of magnitude faster and cheaper than inter-LAN communication.

This section presets an extension to Araneola's basic overlay that exploits network proximity and bandwidth heterogeneity by incorporating additional links between nearby nodes. This extension runs in parallel with and independently of the basic overlay construction and maintenance code presented in Section 4.3. The extension code has two components: (i) a mechanism for locating nearby nodes; and (ii) a connect_nearby task. The first component discovers nearby nodes and stores them in a set named $nearby\_cand$. The second component uses this set.

Generally speaking, Araneola can use a variety of mechanisms for locating nearby nodes. Our implementation does this as follows: at bootstrap time, each node n measures the network-level hop-count distances to the nodes in its local view using the UNIX tracepath utility, and inserts them to the $nearby\_cand$ set in an ascending order of their network-level hop-count distances from n.

The connect_nearby task closely resembles the connect task presented in Section 4.3, except that no reduction rules are applied and no REDIRECT messages are sent. Specifically, there are three control messages: CONNECT_NEARBY, CONNECT_OK_NEARBY, and LEAVE_NEARBY, which correspond to CONNECT, CONNECT_OK, and LEAVE. In addition, both L and H are replaced by the parameter NB, which is the maximum number of nearby neighbors the node is willing to be connected to, and the $neighbors$ set is replaced by the $nearby\_neighbors$ set, which holds the node's current nearby neighbors. Note that every node can set its own NB parameter to

reflect its available bandwidth. Each CONNECT_NEARBY request is issued to the closest node in $nearby\_cand$, rather than to a random node from the local view.

We evaluate this mechanism over the Internet, running 500 nodes over 25 Planet Lab [102] physical machines, with no two machines at the same site. Out of the 25 Planet Lab physical machines, 10 are located in North America, 10 are located in Europe, and 5 are located in Asia. In all the experiments presented in this section, all the nodes are created simultaneously, and remain up throughout the experiment. Although in principal, each node can choose its own NB parameter, in our experiments, we use the same value of NB for all nodes. We denote an experiment in which each node chooses L random neighbors and NB nearby neighbors as $\langle$L,NB$\rangle$.

It is known that in order to achieve the good properties of k-regular graphs, each node should choose at least three random neighbor [122]. Thus, we run experiments in which each node chooses three random neighbors and three nearby neighbors ($\langle$3,3$\rangle$). We contrast these experiments against experiments in which each node chooses six random neighbors ($\langle$6,0$\rangle$), and against experiments in which each node chooses six nearby neighbors ($\langle$0,6$\rangle$). In addition, we run experiments in which the each node's degree is roughly eight ($\langle$3,5$\rangle$, and $\langle$5,3$\rangle$). Note that all the overlays we experiment with have a low degree, of either 6 or 8, compared to those used in previous systems, e.g., in SplitStream [30], Bullet [81], and Saxsons [110], the maximal node's degree is 16, 10, and 16, respectively. For each selection of $\langle$L,NB$\rangle$, we run three experiments. In all our experiments, more than 97% of the nodes end up with NB nearby neighbors, and more than 90% of the nodes have exactly L random neighbors; the overall average node degrees in experiments with $\langle$3,3$\rangle$, $\langle$6,0$\rangle$, and $\langle$0,6$\rangle$ are almost identical as are those of experiments with $\langle$3,5$\rangle$ and $\langle$5,3$\rangle$.

We quantify the effectiveness of our approach by measuring the average number of physical hops that links in the extended overlay traverse. This metric is significant because a smaller hop-count distance implies reduced communication latencies as well as less stress on physical links. The results are summarized in Table 4.5. The first column shows the percentage of links between two nodes running on the same machine. The second column shows the percentage of short links with a hop-count distance of 3. These are Internet2 links between machines deployed at different sites belonging to the same enterprise. Finally, the third column shows the average hop-count in the overlay. Clearly, as NB is increased at the expense of L, there are more local and short links and the average number of physical hops that each link traverses is reduced.

Having verified that this mechanism achieves its goal, we next check its impact on the overlay's robustness. We repeat the experiments of Section 4.4.2, i.e., we remove random subsets of edges and nodes from the overlay graphs and measure the sizes of the largest remaining components. The top two curves in Figure 4.12 and Figure 4.13 are for experiments with $\langle$5,3$\rangle$ and $\langle$3,5$\rangle$. These curves are indistinguishable. Slightly below these are the curves for experiments with $\langle$6,0$\rangle$ and $\langle$3,3$\rangle$, which are also conjoined. The bottom curve in both figures is for experiments with $\langle$0,6$\rangle$.

⟨L,NB⟩	% of links on the same machine	% of short links	Avg hop count
⟨**3,3**⟩	**34.43**	**15.27**	**5.21**
⟨6,0⟩	4.97	6.93	8.69
⟨0,6⟩	74.23	3.4	1.88
⟨**3,5**⟩	**51.18**	**12.25**	**3.82**
⟨5,3⟩	35.6	10.46	5.54

Table 4.5: Hop-count statistics with different selections of ⟨L,NB⟩.

Remarkably, the robustness of an overlay with ⟨5,3⟩ is almost identical to that with ⟨3,5⟩, and the robustness of an overlay with ⟨6,0⟩ is virtually identical to that with ⟨3,3⟩. We believe that this stems from the fact that there is sufficient randomness in the choice of links since: (i) the nodes in $nearb\_cand$ are chosen from the randomized local view; and (ii) each node is connected to at least 3 random neighbors.

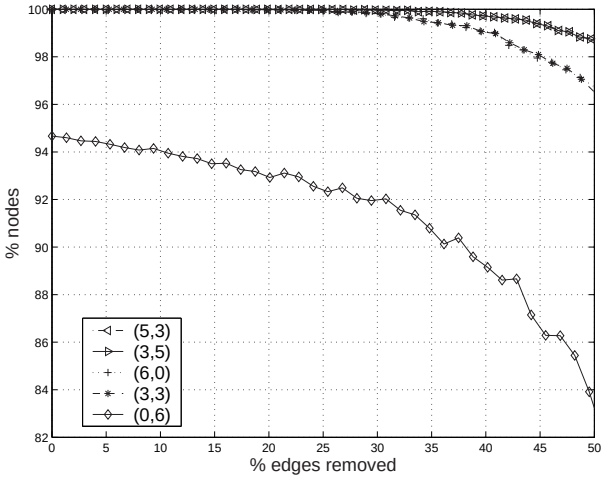


Figure 4.12: Removing edges, largest component, 500 nodes.

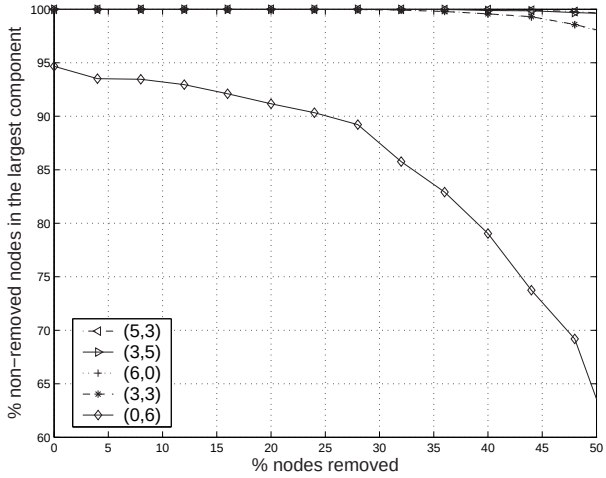


Figure 4.13: Removing nodes, largest component, 500 nodes.

The curves for experiments with ⟨0,6⟩ show why it is important to choose random nodes as neighbors: in all these experiments, the overlay is partitioned even before we remove any edge or node. Moreover, as the percentage of removed edges or nodes increases, the robustness of the overlay deteriorates much quicker than when random edges are used.

We conclude from the experiments in this section that it is preferable for each node to have three random neighbors, and to allocate the rest of its available bandwidth for communication with nearby nodes or other nodes chosen according to application-specific needs.

4.6 Multicasting over Araneola

Given Araneola's overlay, it is possible to disseminate data messages by flooding the overlay, as done, for example, in [85]. Using this approach, messages propagate quickly to all the nodes. The price of using this approach is high bandwidth consumption due to the large number of duplicates sent: Each message is sent at least once on each link in the overlay, i.e., at least $NL/2$ times. Some of the messages may be sent simultaneously by both of the nodes that share the link, but only in case this is not the first time each node receives the message. Thus, when the average degree in the overlay is bounded by $L + 0.5$ (as is guaranteed at static times), a flooded message is sent at most $N(L - \frac{1}{2}) + 1$ times. This approach is appropriate for use in low-degree overlays when bandwidth is not a concern (i.e., there is much more available bandwidth than the application needs), and where low latency and reliability are of essence. For example, it is suitable for instant messaging and chat applications, in which payload messages are typically small.

If the multicast payload consists of large messages, it is possible to flood message identifiers on the overlay in lieu of actual messages, and have nodes request missing messages. Although this approach increases the number of *messages* sent, it can dramatically reduce the bandwidth consumption when payload messages are large. The penalty for using this approach instead of flooding is increasing the message latency by a factor of three. Such a penalty is acceptable for numerous non-real-time applications, for example, file sharing applications like BitTorrent [37], software update dissemination, and video streaming. Many scalable ALM systems are geared towards such applications: virtually all overlay-based and gossip-based ALMs have non-optimal message delays since messages traverse a number of hops that do not necessarily bring the packet closer to its destination. Furthermore, a number of ALMs, like Bullet [81] and Overcast [68], exploit the freedom to delay messages in order to achieve a more bandwidth-efficient system design (using large caches at intermediate nodes in Overcast, and obtaining packets on-demand from nodes that have them in Bullet).

If payload messages are large and are not sent frequently, then flooding message identifiers can be effective. However, if many payload messages are sent, then flooding each identifier in a separate message can induce a high load. In order to overcome such situations, one can bundle a number of messages identifiers together, and periodically send this bundle in a *gossip message*. This is a generalization of the identifier-flooding approach, where the system designer can control the tradeoff between delay and overhead according to specific application needs and traffic characteristics: by sending gossip messages more frequently, one reduces the delay, and by sending them less frequently, one reduces the overhead. For example, by setting gossip rounds to 2 seconds as in [41], in a large group of $10,000$ nodes in an overlay with L$= 5$, we get an average (worst-case, respectively) latency of roughly 7 (18, respectively) seconds. We now describe this gossip-based

approach in more detail.

4.6.1 Gossip-Based Multicast

Each node locally divides its time into *gossip rounds*. A gossip round consists of two phases: in the first phase, each node gossips about recent message identifiers and requests missing messages from its neighbors (the *gossip task*), and in the second, the corresponding missing messages are sent.

The gossip-based implementation is presented in Figure 4.14. *messages* is a FIFO queue of recently received messages. The set *missing_msg* holds identifiers of messages that the node heard of but did not receive. A function *heard_from* maps each identifier in this set to nodes from which it was heard. *recent_mids* holds identifiers of messages received in the latest gossip round along with the identities of the nodes from which they were received.

Every gossip_round_timeout, a node sends a gossip message to each of its neighbors. A gossip message m sent by a node a to its neighbor n is identified by a message identifier, $m.id$, which includes a's identifier (e.g., IP address and port) and a one byte serial number (cyclic counter). The field $m.degree$ holds a's current degree (line 5). The set $m.ids$ includes recent message identifiers that a has received in the last gossip round, and has not heard about from n (line 6). In addition, Araneola piggybacks message requests on gossip messages instead of sending them in separate request messages. Therefore, m includes a set $m.reqs$ of message identifiers that a is requesting from n. These are messages that a is missing, and has heard their identifiers first from n (line 7). Note that a sends a different gossip message to each of its neighbors. After sending the gossip messages, the first element in each *heard_from* list is moved to the end of that list (line 9) in order to vary the node from which the message is requested, and *recent_mids* is reset (line 10) so as not to gossip about the same identifiers again.

When node a receives a gossip message m from neighbor n, for each identifier id in $m.ids$ such that a message with this identifier is not in the *messages* buffer, id is inserted into *missing_msgs* (line 14) and n is appended to *heard_from(id)* (line 15). In addition, a sends to n all the messages requested in $m.reqs$. When a data message arrives, it is enqueued in *messages*, removed from *missing_msgs*, and its identifier and sender are inserted into *recent_mids* (lines 19–21).

Periodically, old messages are purged from *messages* and *missing_msgs*. This garbage collection mechanism is straightforward, and is omitted from the pseudo code.

Load balancing for single-source multicast

Although Araneola is intended for multi-point to multi-point communication, it can also be used for point to multi-point multicast. When the multicast has multiple uniformly distributed sources,

Data structures:
messages – queue of messages tagged with $m.id$, initially empty.
missing_msg – set of messages identifiers, initially $\emptyset$.
heard_from:missing_msgs $\longrightarrow$ list of nodes.
recent_mids – set of pairs $\langle id, from \rangle$, initially $\emptyset$.
Parameters:
Timeout: gossip_round_timeout.

Gossip task:
1. **loop forever**
2. sleep (gossip_round_timeout)
/* *Send gossip messages to neighbors* */
3. **foreach** $n \in neighbors$
4. create new gossip message m, with new $m.id$
5. $m.degree \leftarrow |neighbors|$
6. $m.ids \leftarrow \{i.id : i \in recent\_mids \wedge i.from \neq n\}$
7. $m.reqs \leftarrow \{i \in$ *missing_msgs*: *heard_from(i).first*$= n\}$
8. send $\langle$GOSSIP,$m\rangle$ to n
/* *Update data structures* */
9. move 1st element of each *heard_from(mid)* list to end
10. *recent_mids* $\leftarrow \emptyset$

Event handlers:
11. **upon** receive $\langle$GOSSIP,$m\rangle$ from n **do**
12. $neighbor(n).degree \leftarrow m.degree$
13. **foreach** $id \in m.ids \wedge id \notin messages$
14. *missing_msgs* $\leftarrow$ *missing_msgs* $\cup\{id\}$
15. append n to *heard_from(id)*
/* *Send requested messages to* n */
16. **foreach** $r \in reqs$
17. send $\langle$DATA,message with identifier = $r.id\rangle$ to n

18. **upon** receive $\langle$DATA,$m\rangle$ from n **do**
19. *messages.enqueue(m)*
20. *missing_msgs.remove(m.id)*
21. *recent_mids* $\leftarrow$ *recent_mids* $\cup\{\langle m.id, n\rangle\}$

Figure 4.14: Gossip-based multicast.

the load on Araneola nodes is naturally balanced: each node sends the same number of gossip messages per round, and each nodes handles roughly the same number of message requests on average. However, if the multicast would be initiated at a single source, then the message requests would most often be sent on a spanning tree of the overlay rooted at the source. This can result in a higher load on inner nodes of the spanning tree.

We propose the following simple solution to this difficulty: Let node n be the single-source in a multicast session. Whenever a new data message is created at n, n sends the message to a random set of nodes instead of sending it to its neighbors. A different set of nodes is chosen each time. This simulates a situation in which messages are created by multiple uniformly distributed sources, and the message requests follow many different spanning trees.

The multicast and management overhead

In this section, we assume that each node is connected to L nodes, there is no packet loss, and the load on Araneola nodes is balanced as described in the previous section. We denote the multicast rate as p data packets per *gossip_round_timeout*.

The multicast overhead. In Araneola, as opposed to other multicast protocols, e.g., Lpbcast [41] and Bullet [81], no duplicate data packets are sent. Whereas in Lpbcast, on average, each node receives $\log N$ copies of each data packet, and in Bullet [81], roughly 10% of received data packets are duplicates, in Araneola, each node receives one copy of each data packet. Since the load on Araneola nodes is balanced, on average, each node forwards each data packet to one of its neighbors. Hence, the per-node multicast load is p data packets per *gossip_round_timeout*, which is the multicast rate.

The management overhead. In addition to data packets, every *gossip_round_timeout*, each node sends a gossip message to each of its neighbors. Assuming each node is connected to L nodes, the per-node management overhead is L gossip messages per *gossip_round_timeout*. A gossip message sent from a node n to one of its neighbors $\hat{n}$ contains an one-byte serial number, n's identifier (6-bytes), n's degree (one byte), identifiers of recent messages that n has received in the last gossip round whose source is not $\hat{n}$, and n's message requests from $\hat{n}$. Both a message identifier and a message request are represented by eight bytes. Below, we calculate the average size of a gossip message.

Each node sends each message identifier to $L-1$ nodes, and, in the absence of message loss, requests each message from one of its neighbors. Therefore, on average, each gossip message contains $p\frac{L-1}{L}$ message identifiers and $\frac{p}{L}$ message requests (recall that the multicast rate is p data packets per *gossip_round_timeout*). Hence, the average size of a gossip message is $1 + 6 + 1 +$

$8(p\frac{L-1}{L}+\frac{p}{L}) = 8(1+p)$ bytes. Therefore, the per-node management overhead is $\frac{1}{gossip\_round\_timeout} \cdot 8(1+p) = \frac{8(1+p)}{gossip\_round\_timeout}$ bytes per-second. For example, if the multicast rate is 10 data packets per-second and the *gossip_round_timeout* is 5 seconds, then the per-node management overhead is less than 18 bytes per-second.

4.7 Evaluation of Gossiping over Araneola

We implement the gossip-based multicast module on top of the code for constructing and maintaining the basic Araneola overlay, described in Section 4.3. In order to run large scale simulations, we run most of our experiments on a LAN [121]. In Section 4.7.3, we also run WAN-like simulations of Araneola.

We use the standard UDP protocol as the multicast module's transport protocol. With this approach, no retransmissions are sent, and therefore we do not increase the network load at times of congestion, i.e., when there is high message loss. Even without retransmissions, as we show in this section, Araneola achieves full reliability of data delivery despite high churn and message loss rates by disseminating message identifiers on multiple disjoint paths in Araneola's overlay. We use the standard UDP protocol at the available bandwidth rate of each machine. In particular, we never over saturate the network. Designing a flow control mechanism to adjust this rate is orthogonal to our study. For example, the TFRC transport protocol [43] adjusts its transmission rate on a per-connection basis based on prevailing network conditions [81].

We run multiple Araneola nodes per machine, and therefore need to space the gossip rounds sufficiently so as to allow all the nodes running on the same machine to complete their gossip operation during a round. Thus, we chose a fairly large round duration of 5 seconds. When there is only one node per machine, the round duration can be an order of magnitude smaller. In order to construct and maintain Araneola's overlay we used the code described in Section 4.3.2 with the parameters and timeouts described in Section 4.4. Throughput this section, we measure the rate at which messages propagate on the overlay in terms of an overlay-level hop count— each message is tagged with a counter, and every node that receives the message increases the counter. We use this approach in order to allow a fair comparison between Araneola and a standard gossip-based multicast protocol, in which the latency is measured in terms of gossip rounds (see Section 4.7.1). The actual propagation rate is the propagation rate in terms of an overlay-level hop count multiplied by the round duration.

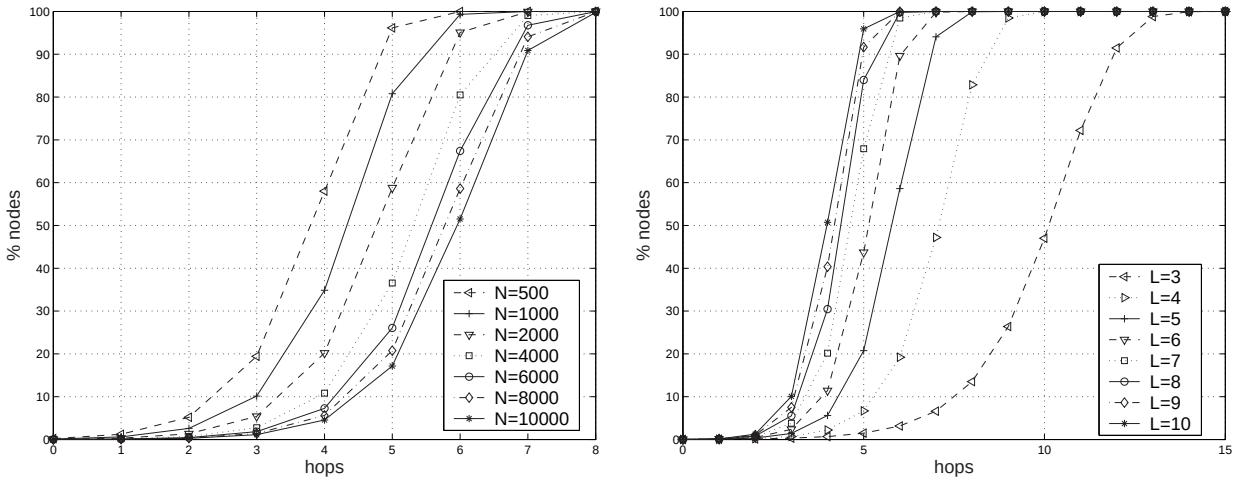
In Section 4.7.1 we evaluate the performance of the multicast layer in static settings, and in Section 4.7.2, we consider high churn. Finally, in Section 4.7.3, we evaluate the performance of the multicast layer in a WAN-like setting.

4.7.1 Static Evaluation

In our static evaluation, all the nodes are created simultaneously, and remain up throughout the experiment. In each round, a single data message is injected into the system (by the application), each time from a different machine. At least 200 data messages are sent in each experiment, and each experiment (with a given number of nodes and choice of parameter settings) is repeated at least twice.

Scalability

We first examine the impact of number of nodes N on the rate at which messages propagate on the overlay. Figure 4.15(a) depicts the message propagation rates measured for values of N ranging from 500 to $10,000$. L and H are set to 5 and 10, respectively. For each number of hops x, the curves depict the average percentage of nodes that receive a message within x hops. As N increases, messages take longer to propagate, but the slow-down is gradual. The average latency in each of our experiments was close to the average distance between two nodes in the overlay presented in Figure 4.5. For each N, an average of over 99% of the nodes receive a message within a number of hops equal to the overlay's diameter. On rare occasions, messages were propagated in more hops than the graph's diameter if they reached their destination on a "longer" path before reaching it on the shortest path between the source and destination.



(a) L$= 5$.

(b) 8000 nodes.

Figure 4.15: Message propagation rates for different degree Araneola overlays.

The impact of L

Araneola's parameter L affects the overlay's diameter, and hence affects the latency of message delivery. We study the impact of this parameter in runs with 8000 nodes (on 100 Netbed machines) and values of L ranging from 3 to 10. In each experiment, H was set to be L$+5$. Increasing the value of L increases the load, since a node with degree k sends k gossip messages in each gossip round, and sends each message identifier $k-1$ times (once to each downstream neighbor). However, such increase also reduces the message latency, as shown in Figure 4.15(b). Each curve in the figure depicts the message propagation rate for a given value of L (in experiments with 8000 nodes). The figure shows that the latency decreases as L increases. When L$=3$, messages reach 99.94% of the nodes within 14 hops, and 100% of the nodes within 15 hops; when L$=4$, messages reach 99.97% of the nodes within 10 hops, and 100% of the nodes within 11; while when L$=5$, messages reach 99.3% of the nodes in 8 hops and all the nodes in 9. The improvement becomes less dramatic as L increases beyond 5.

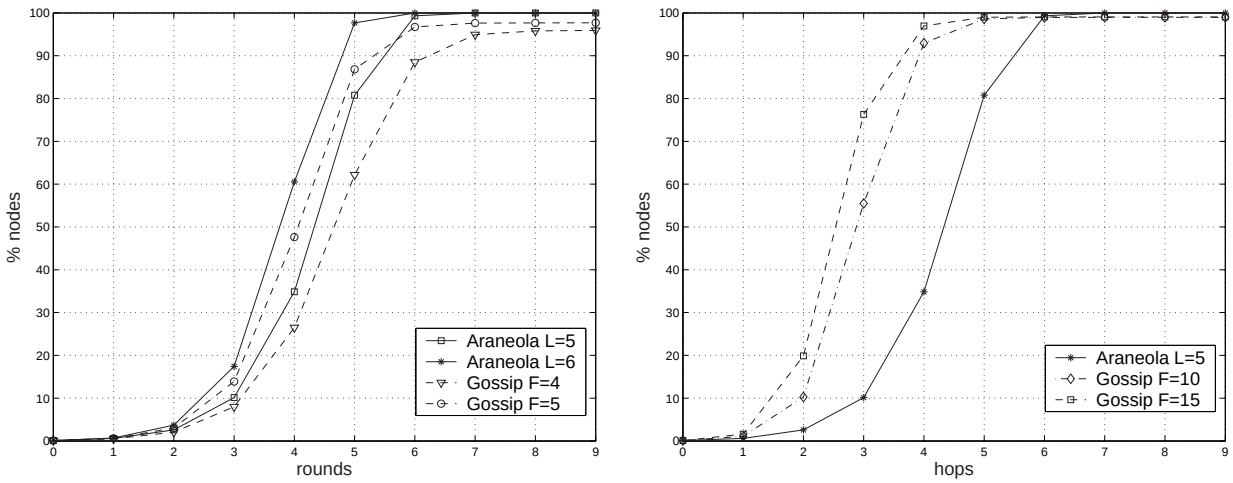
Comparison with gossip protocol

We now compare Araneola to a standard gossip-based multicast protocol (as described in [85]). Similarly to Araneola, such a protocol supports dynamic user behavior: the reliability of a gossip-based protocol gracefully degrades with the churn rate, and each join or leave operation incurs a small overhead. In contrast, as explained in Section 2.1, tree-based overlays like SplitStream [30] and Bullet [81], which are designed for content streaming, do not strive to achieve full reliability under high churn rates and induce higher join/leave overhead than the join/leave overhead incurred by Araneola and a gossip-based multicast protocol.

We have implemented the gossip protocol based on Araneola's gossip-based multicast module. The gossip protocol takes a parameter F, which is its fan-out. Where an Araneola node sends gossip messages to its neighbors, the gossip protocol sends gossip messages to F randomly selected nodes from its membership view. Whereas Araneola sends each message identifier downstream only, the gossip protocol sends all its *recent_mids* to all the chosen targets. Thus, the gossip protocol instantiated with a fan-out of F sends information as many times as Araneola with L$=F+1$. With the gossip protocol, message requests are sent immediately upon receipt of a gossip message, and re-sent periodically in case the requested message is not recovered.

We experiment with 1000 nodes on 20 Netbed machines. In each experiment, 400 messages are sent. Figure 4.16(a) compares the average message propagation rates of Araneola with L$=5$ and 6 to those of the gossip protocol with the corresponding fan-outs – $F=4$ and 5. Evidently, Araneola propagates information much more effectively than the gossip protocol. Initially, the propagation rates are similar, but after about 6 hops, Araneola continues to effectively propagate the message,

while the gossip protocol tapers off. Araneola succeeds in disseminating all the messages to 100% of the nodes in 7 hops with L$= 5$, and in 6 hops with L$= 6$. In contrast, the gossip protocol only reaches 95.91% of the nodes on average with $F = 4$, and 97.69% with $F = 5$. Indeed a fan-out of 5 does not suffice for the gossip protocol with 1000 nodes. According to previous studies [76], a fan-out of 14 is required. This is due to the fact that with the gossip protocol only the out-degree (fan-out) is balanced, while the in-degree (fan-in) may be unbalanced. In contrast, Araneola's in-degrees and out-degrees are balanced as all links in the overlay are bi-directional. As more nodes have a given message, the gossip protocol is more likely than Araneola to "waste" its gossip on nodes that already have the message, and therefore is less effective at spreading the information to additional nodes.



(a) Comprable fan-outs for Araneola and gossip.

(b) Larger fan-outs for gossip.

Figure 4.16: Araneola versus gossip, 1000 nodes.

The second plot in Figure 4.16 shows the propagation rate of gossip with fan-outs of 10, and 15 as compared to Araneola with L$= 5$. The gossip protocol's propagation rate with the large fan-outs is initially much more rapid than that of Araneola with L$= 5$, but after about 6 hops, Araneola already succeeds to reach more nodes than the gossip protocol. While Araneola succeeds in delivering all the messages to all the nodes, the gossip protocol with $F = 15$ fails to do so; it reaches only 99.12% of the nodes on average. With $F = 10$, it reaches 98.97% of the nodes on average.

Our measurements of the gossip protocol are close to those reported in [76] although not identical to them. Whereas [76] reports of 100% reliability with $F = 14$, we measure 99.12% reliability with $F = 15$. We believe that this slight discrepancy stems from differences in the evaluation methodology. First, the evaluation in [76] uses simulations; it sends a single data message at a

time; and it assumes that nodes are never over-loaded and no messages are lost. In contrast, we run multiple nodes on each machine, the nodes communicate over a real network, and multiple data messages diffuse through the system concurrently. Therefore, we do experience some scheduling delays and message loss, although not often. Second, the system of [76] uses servers in order to have the membership views perfectly uniformly distributed. Since our evaluation does not do so, our membership views are less perfectly "random". Since Araneola is evaluated using exactly the same methodology, our comparison is fair.

4.7.2 Dynamic Evaluation

One of Araneola's major design goals is to provide an undisrupted multicast service to nodes that are up despite node and link failures and high churn rates. As long as the overlay contains only one connected component, Araneola's multicast module achieves full reliability of message delivery as it floods message identifiers over the overlay's links. In Section 4.4.2, we studied the fault-tolerance and robustness of the Araneola overlay in static settings and saw that Araneola's overlay remains connected following massive random node and link failures. In this section, we study the performance of the multicast layer under high churn rates using the dynamic simulation scenarios used in Section 4.4.3. An application message was injected into the system by one of the machines in each round. Between 433 and 476 messages were sent in each experiment. The parameters L and H were set to 5 and 10 respectively.

In each dynamic experiment, for each message m, we define *nodes that are up during m's transmission* to be nodes that have joined Araneola's overlay at least 12 rounds before m's transmission, and did not leave at least 12 rounds after the transmission. We chose 12 as a gross over-estimate. In fact, as we show below, nodes can normally begin to receive messages reliably immediately upon requesting to join. However, since we run 50 Araneola nodes on each physical machine, which due to contention at the network interface may cause a joining node's messages to be delayed for several rounds, we have chosen to wait additional rounds before considering the node to be up.

In all of our dynamic experiments, each message was received by 100% of the nodes that were up during its transmission. Moreover, messages were delivered with *the same latency as in static runs*. We illustrate this in Figure 4.17 for $N = 1000$; similar results were obtained with $N = 2000$. The bottom curve depicts the average latency with which messages are delivered for different values of λ. It shows that the latency is unaffected by the join/leave rate. The middle curve shows that, for all values of λ, the average number of hops it takes a given multicast message to reach at least 99% of the live nodes is 6. We did, however, observe a small difference in the message propagation rate: for values of λ up to 0.1, messages reach over 99.9% of the nodes within

6 hops, whereas with $\lambda = 0.125$ and $\lambda = 0.15$, they reach only 99.5% of the nodes within 6 hops. The top curve shows that regardless of λ, it takes 7 hops for messages to reach all the nodes. All the latencies are roughly the same as in static runs with 500 nodes, which is the average number of live nodes in a dynamic experiment with $N = 1000$.

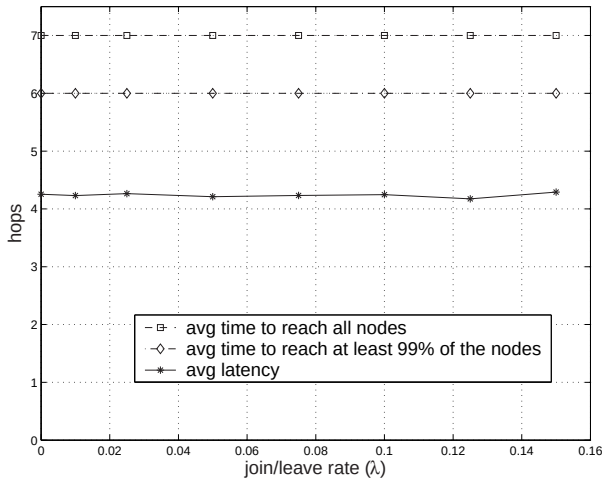


Figure 4.17: Average latencies for different churn rates, 1000 nodes, L$= 5$.

Comparison with gossip protocol. Churn has little effect on the performance of a gossip-based multicast protocol [15]. In this section, we showed that churn has virtually no effect on the performance of Araneola. Moreover, as the simulations in Sections 4.7.1 and 4.7.2 show, under churn, Araneola achieves higher reliability than the reliability achieved by a classic gossip protocol in a static failure-free setting, while incurring smaller delay and overhead.

Fast join. The final aspect of the multicast layer we evaluate is how fast it allows joining nodes to begin to receive messages reliably. In order to avoid scheduling race conditions and contention at the network interface, we ran 100 nodes on a single machine, with L$= 5$ and H$= 10$. Our measurements show that a joining node not only receives all the messages sent after its creation, but actually receives 100% of the messages sent up to 6 rounds before its join. This occurs because it usually takes 5 hops for a message to propagate to all of the new node's neighbors. If any of the new node's neighbors receives the message in 5 hops, then the new node will receive this message in the next round, as the 6th hop. We conclude that Araneola incorporates joining nodes into the multicast group without delay.

4.7.3 WAN Emulation

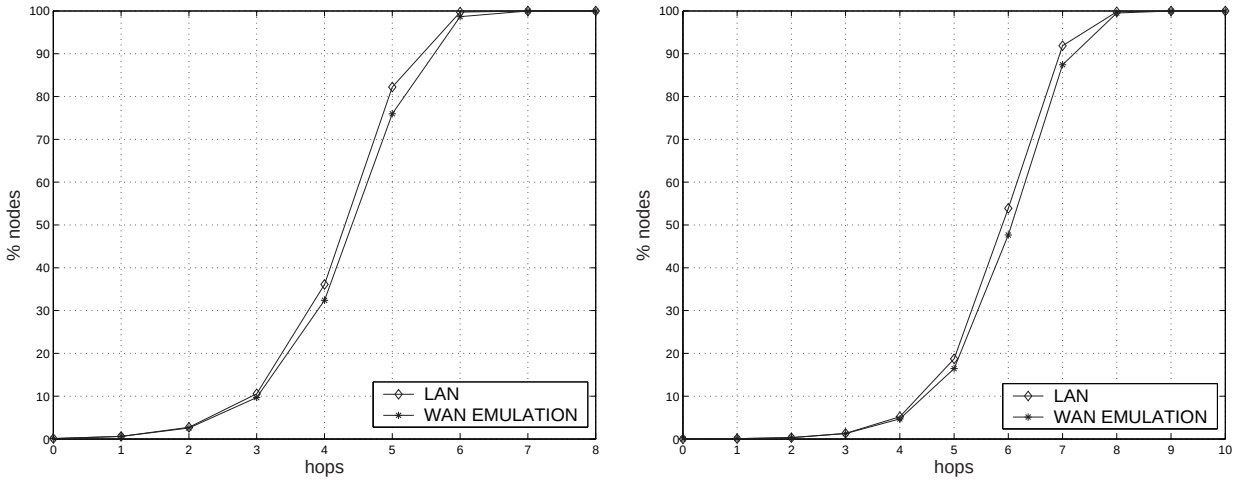
In this section, we report about simulations of Araneola's gossip-based multicast module in a WAN-like setting. We run these simulations in order to evaluate the performance of Araneola in a wide-area setting, in which the message loss rates and delays are much higher than in a LAN setting. Our WAN-like setting is motivated by measurements of upload bandwidth of P2P clients [109] and measurements of loss rates and RTTs (round trip time) of Internet links [64]. For simplicity, in our WAN-like simulation, we use 5 types of links (see Table 4.6). In order to measure the worst case reliability in such a setting, a given node's links are all of the same type; this simulates the node's worst link.

Link Type	Loss Rates	RTTs	% of Nodes
Excellent	$< 0.1\%$	0	0.1%
Good	0.1%–1%	$< 62.5ms$	4.9%
Acceptable	1%–2.5%	$62.5ms$–$125ms$	30%
Poor	2.5%–5%	$125ms$–$250ms$	45%
Very Poor	5%–12%	$250ms$–$500ms$	20%

Table 4.6: Links loss rate and RTT.

We use the setting of Section 4.7.1 for two group size: 1000 and 8000 nodes. L and H are set to 5 and 10, respectively. In Fig. 4.18, we compare the message propagation in WAN-like simulations with the message propagation in LAN simulations. As the figure shows, the differences between a WAN-like setting and a LAN setting are small. Below, we explain these results.

We first note that link latency neither reduces the reliability of message delivery nor increases the latency of message propagation. This is since we measure the message propagation in rounds, and a round duration is an order of magnitude longer than a link latency. The message loss does reduce the reliability. However, as opposed to tree-based multicast systems, in Araneola, as long as the message loss does not exceed a certain threshold (see Section 4.4.2), failures do not reduce Araneola's reliability. This is since there are L disjoint paths between each pair of nodes in the overlay, and hence each message can be retrieved from L different neighbors. In our WAN-like setting, the failure rates do not exceed the above threshold, and therefore Araneola achieves full reliability in this setting. The message loss, however, does slightly increase the latency, since several messages are received not through the shortest path. However, as the figure shows, this increase is small, since the average link loss is small.



(a) 1000 nodes.

(b) 8000 nodes.

Figure 4.18: Message propagation rates for WAN-like and LAN simulations.

Chapter 5

EquiCast: Scalable Multicast with Selfish Users

5.1 Introduction

P2P networks can distribute digital content to a large number of users over the Internet by distributing the load among the peers [37, 107]. However, these networks suffer from the problem of "freeloaders", i.e., users who consume resources without contributing their fair share [25]. In order to discourage "freeloaders", some P2P systems employ incentives to motivate users to cooperate, e.g., contribute upload bandwidth or disk space for some other users. However, while current incentive-based P2P systems reward cooperation to some extent, no existing protocol has been proven to enforce cooperation in selfish environments. Moreover, such systems, e.g., [37, 49], typically rely on user altruism. For example, a node is expected to upload data blocks to other nodes for no return whenever it has available bandwidth [37, 49]. Hence, current incentive-based P2P systems do not solve the problem of "freeloaders" [59], and would not have worked well at all if users would have behaved selfishly, e.g, leaving a content distribution system after they have finished downloading the file [49, 59].

Nowadays, user altruism is common since most users are connected to the Internet using static machines via ISPs with a flat pricing model, and hence sending a packet does not incur a cost on its sender. However, these paradigms are changing. First, the increasing access to digital content is expected to drive ISPs to implement a tiered pricing scheme, where high end pricing plans shall allow unlimited downloads and uploads, while lower tier pricing plans shall limit traffic bandwidth [107]. With such a pricing scheme, users will most likely cease to be altruistic [107]. This may lead to low P2P system availability [59, 71] or even system collapse [25, 96]. Second, wireless hotspots are proliferating in recent years, and users are increasingly connecting to the Internet and downloading content to mobile devices such as laptops and cell phones. In such networks, pricing is typically based on connection time or transmission volume. Moreover, battery

power is a critical resource for mobile devices. Hence, user altruism can hardly be expected in such networks. Therefore, we believe that it is important to design P2P systems that work well even when all users are selfish.

In this chapter, we address this challenge. We introduce *EquiCast*, a wide-area P2P multicast protocol for distributing content to large groups of selfish nodes. We treat the problem of freeloading from a game theoretic perspective, and we model the system as a *non-cooperative game*. In such a game, nodes are selfish but *rational*, i.e., each user chooses its own *strategy* regarding its level of cooperation so as to minimize its own cost [46]. More specifically, the goal of each node is to receive all the multicast packets while minimizing its sending rate. We define a special set of *protocol-obedient strategies (POSs)*. Generally speaking, a strategy out of this set allows a node to determine how many connections to maintain and how many packets to send on each connection though it does not allow to hack the protocol's code or assume that others do so. We believe that it is reasonable to assume that most nodes will run a protocol-obedient strategy (POS), since users usually do not have the technical knowledge required in order to modify an application code. We prove that if all nodes choose POSs, then each node receives all the multicast packets and, moreover, no node can unilaterally reduce its cost by changing its strategy to a non-POS. That is, unilateral hacking of the protocol's code cannot reduce a node's cost. Our formal model and cost function are presented in Section 5.2, and in Section 5.4.2 we formally define the set of POSs.

In EquiCast, a single distribution server *S* (which can be implemented by multiple machines acting as one logical server) organizes the nodes into a static *overlay network*. We divide the time into rounds, and in each round, *S* injects new data packets to a small random subset of the nodes in the overlay. Nodes, then, communicate with their overlay neighbors in order to retrieve missing data packets. *S* also provides a "safety net" for a node whose data receiving rate is lower than the multicast rate, by sending data packets to either the node or its neighbors. This additional overhead incurred on *S* is modest, since most of the nodes are expected to receive most if not all the multicast packets from their overlay neighbors.

EquiCast enforces cooperation through two mechanisms. The first is a *monitoring mechanism*, whereby each node monitors the sending rate of each of its neighbors. Specifically, for each neighbor $\hat{n}$, each node n maintains $\hat{n}$'s *balance*, which is the difference between the number of data packets $\hat{n}$ has sent to n so far and the expected per-link throughput. As long as $\hat{n}$'s balance is greater than or equal to a predefined negative threshold L, $\hat{n}$ is considered to be cooperative, and n continues to send data packets to $\hat{n}$. Otherwise, n terminates its connection with $\hat{n}$. Note, however, that it is always possible for cooperative nodes to have a balance greater than or equal to L with respect to all of their neighbors.

The second mechanism is a per-link *penalty mechanism*, which further motivates nodes to adhere to the expected link throughput. It charges a neighbor with one additional *fine* packet for every

round the neighbor has a negative balance, where a fine packet is a dummy packet that has the same size as a data packet. Fine packets, as opposed to data packets, do not affect the node's balance. Therefore, a node is motivated to achieve a non-negative balance, whenever possible. Note that the multicast rate is tens of data packets per round, and hence the penalty mechanism incurs a modest overhead. In general, in EquiCast, each node is required to have an upload bandwidth that is slightly higher, e.g., by 10%, than the multicast rate. Note that similar requirements are also assumed by multicast systems for cooperative environments [30].

In Section 5.4, we prove that, in environments in which all the nodes are selfish, if all the nodes choose POSs, then EquiCast disseminates all the multicast packets to all the nodes. Additionally, every POS in which a node exclusively cooperates with all its neighbors *strictly dominates* every POS in which it does not. This means that the cost incurred on a "freeloader" node is strictly higher than the cost incurred on a cooperative node, and hence all nodes are expected to follow the protocol out of their own selfish interests. Finally, we prove that if all the nodes choose POSs, then no node can unilaterally reduce its cost by changing its strategy to a non-POS. Moreover, we show that this result holds under some milder conditions, where some of the nodes might adopt non-POSs or even be non-rational. We are unaware of any previous P2P multicast protocol that was formally proven to enforce cooperation in environments in which *all* nodes are selfish.

Finally, for simplicity, throughout most of the chapter we describe only a static version of EquiCast, in which no node joins or leaves the service. In Section 5.5, we sketch out a dynamic version of EquiCast that supports node joins and leaves.

5.2 Model and Problem Statement

We consider a large static collection of N nodes n_1, n_2, ..., n_N. A single distribution server *S* distributes P data packets to the nodes, where P is a random variable distributed exponentially with a large expectation, e.g., larger than $10{,}000$. *S* knows all the node identities, e.g., by each node registering itself at *S*.

5.2.1 Network and Timing Model

Each node can directly communicate with every other node and with *S*. The multicast rate is p data packets per δ time units. Each node has an upload bandwidth of at most $p+kc$ packets per δ time units, where k and c are small constants such that $k\geq 3$, $c\geq 4$, and $p\%k=0$. In addition, we require that $(k^2-k)(c-3)<p$ and $k^2(c-2)-2k<p$, in order to prove that every POS in which a node exclusively cooperates with all its neighbors strictly dominates every POS in which it does not. There is a bound of Δ time units on packet delay, and sending a packet incurs zero delay on

the sender. Local computations also incur zero delay. Finally, for simplicity, we assume no packet loss.

5.2.2 The Game Formulation

We model the system as a *non-cooperative game*, in which the *players* are the N nodes. Each node chooses a *strategy* that dictates how it plays the game. A *strategy profile* is a vector of N strategies, one for each node. A *strategy space* is the set of strategies available to each node. In this chapter, the strategy space consists of the set of all the strategies that send no more than $p+kc$ packets per δ time units.

We define a special set of *protocol-obedient strategies (POSs)*. Generally speaking, a strategy out of this set must run the protocol as is and can only determine how many connections to maintain and how many packets to send on each connection. We defer the precise definition of this set to Section 5.4.2, since it relies on the protocol's code described in Section 5.3.3.

Each node is selfish and *rational*, i.e., n_i chooses a strategy st_i that minimizes its individual cost as defined below. A strategy st *strictly dominates* another strategy st' if choosing st always incurs a lower cost than choosing st', regardless of the strategies chosen by other nodes. A *strongly dominating strategy* strictly dominates all other strategies. Although *S* is not one of the players, we model its random injections of data packets as its strategy st_0, and hence our proof of cooperation is valid regardless of *S*'s random choices. Note that st_0 does not determine the length of the session, i.e., P. Denote by r_i the total number of data packets received by n_i throughout the multicast session, and by s_i the total number of packets sent by n_i throughout the multicast session. Then, the cost function for a node n_i is defined as:

$$f_i(st_0, st_1, ..., st_N) = \begin{cases} \infty & \text{if } r_i < P \\ s_i & \text{if } r_i = P. \end{cases}$$

That is, if n_i receives all the multicast packets, then its cost is the number of packets it has sent during the multicast session. Otherwise, n_i's cost is infinite.

5.2.3 Problem Statement

Our goal is to design a scalable P2P multicast protocol, in which if all nodes choose POSs, then (i) each node receives all the multicast packets; and (ii) no node can reduce its cost by unilaterally changing its strategy to a non-POS. A second goal is efficiency. The maximal total receiving and sending overhead incurred on each node throughput the entire multicast session is $P(1+\frac{ak}{p})$ and $P(1+\frac{ak}{p})+Hk$ packets, respectively, where a is a small constant and H is a non-negative constant determined by each node.

5.3 EquiCast

Section 5.3.1 describes EquiCast's architecture. Section 5.3.2 provides a high-level description of EquiCast's cooperation enforcement scheme, and Section 5.3.3 describes the protocol in detail.

5.3.1 Architecture

S organizes the nodes into a static overlay that satisfies the following properties: (KRRG1) each node in the overlay has exactly k neighbors for some parameter k; (KRRG2) the overlay's diameter is logarithmic in N; and (KRRG3) the expected distance between a given node and a random node in the overlay equals the average distance between a pair of nodes in the overlay. For $k \geq 3$, a *k-regular random graph*[1] satisfies these properties with high probability [45, 78, 122]. *S* constructs the overlay, e.g., using one of the constructions in [122], and sends to each node the identities of its overlay neighbors, henceforth, simply called *neighbors*. Note that since the construction is centralized, no node cooperation is required.

In the next section, we show that, under our model assumptions, for each node, maintaining connections with its k neighbors is a dominating strategy. Hence, connections are expected to persist. However, if a given connection is terminated, e.g., due to a node failure, then a node n can end up with less than k neighbors. In such cases, n contacts *S*, and *S* emulates a selfish rational EquiCast node $\hat{n}$, and a new connection is formed between n and $\hat{n}$. $\hat{n}$'s interface is identical to the interface of each EquiCast node with the following two exceptions: i) n's balance with respect to $\hat{n}$ is initialized to the lowest possible balance, i.e., L; and ii) in each round, n must send a fine packet to $\hat{n}$ regardless of its balance with respect to $\hat{n}$, otherwise $\hat{n}$ terminates its connection with n. Hence, as we show in Section 5.4.2, a node prefers to maintain a connection with a non-emulated node over an emulated one.

5.3.2 Overview

We divide the time into $R = \lceil \frac{P}{p} \rceil$ rounds. Every round, *S* creates p new data packets, and for each node n, *S* sends all the copies of these p packets to n with a probability of $\frac{k}{N}$, so that, on average, each data packet is sent to k nodes.

In each round, every node n gossips with its neighbors about new data packets it has received in the previous round, i.e., for each neighbor, n sends a gossip packet containing the identities of all the data packets it has received in the previous round. After receiving gossip packets from its neighbors, n requests from each of its neighbors data packets that the neighbor has and were not

[1] A *k-regular random graph* with N nodes is a graph chosen uniformly at random from the set of k-regular graphs with N nodes.

previously received by n. If a given packet is available at more than one neighbor, then n randomly picks one of those neighbors to request the packet from. Finally, n sends its neighbors the data packets they requested from it.

We note that since a given packet is sent by *S* to each of the nodes with equal probability and since the expected distance between a random node and a given node equals the average distance between a pair of nodes in the overlay (KRRG3), if all the nodes comply with the protocol, then the average latency with which nodes receive data packets is identical for all nodes, and the expected throughput is $\frac{p}{k}$ data packets per-round on every overlay link. In a previous study [93], we used a similar technique in order to support reliable multicast in cooperative environments. The aim of this study is achieving similar results in a non-cooperative environment.

In order to motivate cooperation, we introduce a *monitoring mechanism*, whereby each node n monitors the sending rate of each of its neighbors. For each neighbor $\hat{n}$, n maintains *n.neighbor-_balance[$\hat{n}$]*, which is the difference between the number of data packets $\hat{n}$ has sent so far and the expected per-link throughput of $\frac{p}{k}$ data packets per-round. Note that, in a given round, $\hat{n}$ may have less than $\frac{p}{k}$ new data packets that have not yet been received at n, whereas in another round it may have more than $\frac{p}{k}$ data packets for n. Therefore, we allow for some slack in the balance. The allowed imbalance is captured by a negative threshold L. As long as $\hat{n}$'s balance with respect to n is greater than or equal to L, $\hat{n}$ is considered to be cooperative by n. But if $\hat{n}$'s balance with respect to n drops below L, then n terminates the connection with $\hat{n}$. Note that, as long as $\hat{n}$'s balance with respect to n is greater than or equal to L, the uploading rate from n to $\hat{n}$ is unaffected by the downloading rate from $\hat{n}$ to n. This independence is required in order to prove cooperation.

In order to further motivate nodes to adhere to the expected throughput, we introduce a per-link *penalty mechanism* that charges a neighbor with one additional *fine* packet for every round the neighbor has a negative balance with respect to the node, where a fine packet contains no useful data but has the same size as a data packet. If the node does not receive a fine packet from a neighbor with a negative balance, then it terminates its connection with that neighbor. Fine packets, as opposed to data packets, do not affect the node's balance. Therefore, a node is motivated to achieve a non-negative balance, where all sent packets contribute to its balance. Moreover, it is beneficial for nodes to have a strictly positive balance whenever possible. This is because there is no guarantee that a given neighbor will request at least $\frac{p}{k}$ packets from the node in forthcoming rounds. If a neighbor requests fewer than $\frac{p}{k}$ packets when the node's balance toward it is zero, then the balance becomes negative, and the node pays the fine. Each node chooses its maximal balance with respect to a given neighbor. This maximal balance is captured by the non-negative threshold H. As long as the node's balance with respect to a given neighbor $\hat{n}$ does not exceed H the node sends all the data packets that $\hat{n}$ requests from it, yet it refrains from sending data packets that would increase its balance with respect to $\hat{n}$ beyond H. Note that a node cannot optimize the value

of H according to the session duration, as P is a random variable distributed exponentially. Note also that the penalty mechanism does not eliminate the need for L, since without this threshold, a selfish node could have sent only fine packets.

Although nodes are motivated to have a non-negative balance, due to randomness, a node n may have an insufficient number of new packets for a given neighbor in order to be able to maintain a balance greater than or equal to L. In order to avoid a disconnection in such a scenario, n can ask *S* to send up to $\frac{p}{k}$ new data packets on behalf of it to a given neighbor $\hat{n}$ in return for sending the same number of fine packets to *S*. $\hat{n}$ counts *S*'s packets towards n's balance only if ignoring these packets would drop n's balance with respect to $\hat{n}$ below L. Hence, n contacts *S* only when its balance with respect to $\hat{n}$ drops below L. In addition, after the end of the multicast session, n can ask *S* to send to it up to $|L|k$ data packets in return for sending the same number of fine packets to *S*.

On the one hand, the allowed imbalance should be large enough to reduce the probability of a cooperative node reaching L, in order to avoid overloading *S*. On the other hand, a high imbalance allows a selfish node to receive many data packets, i.e., $|L|k$, without sending any data packets in return. Hence, there is an inherent tradeoff between the overhead incurred on *S* and the number of data packets a node can receive for free. For example, setting L to -200 is a good tradeoff between the two opposite requirements. On the one hand, if k=3, then a node can get only 600 data packets without contributing anything in return to the system. Since we assume that the multicast session is significantly longer, including at least $10{,}000$ packets, it seems like users will not be satisfied with getting a mere 600 packets and will therefore be motivated to contribute. On the other hand, such a bound is expected to incur a modest overhead on *S*. Note that the value of L is independent of all the other system parameters.

5.3.3 Detailed Description

The source protocol

On each round, *S* creates p new data packets, and for each node n it sends all the copies of these packets to n with a probability of $\frac{k}{N}$. In the rare case in which, at a given round, no node is chosen to receive all the copies of the p new data packets, *S* restarts the round. Note that this does not add to the round duration, since computation time is zero.

Upon receiving a request from a node n to send x data packets to another node $\hat{n}$, *S* verifies that: (i) $x \leq \frac{p}{k}$; (ii) this request is followed by the sending of x fine packets from n; (iii) n and $\hat{n}$ are neighbors; (iv) neither n nor $\hat{n}$ has asked *S* to replace the other node with an emulated node; and (v) n is not pretending to be another node (IP-spoofing). The latter is checked, e.g., by sending a random string to n that n should send back to *S* in one of the fine packets. If n passes the checks,

Data structures:
neighbors – set of the overlay neighboring nodes.
my_balance[k] – outgoing balance, initially $\forall n \in neighbors$, $my\_balance[n] = 0$.
neighbor_balance[k] – incoming balance, initially $\forall n \in neighbors$, $neighbor\_balance[n] = 0$.
H – an upper bound on the balance, chosen by the node.
ids – set of data packet identifiers that the node has not yet received, initially $\emptyset$.
reqs[n] – a set of data packets identifiers to ask from neighbor n, initially $\forall n \in neighbors$, $req[n] = \emptyset$.
Parameters:
L – a lower bound on the balance (a negative number).

Figure 5.1: EquiCast's data structures and parameters.

then *S* sends to $\hat{n}$ copies of x new data packets that it intends to distribute in the next round. If two or more of $\hat{n}$'s neighbors ask *S* to send data packets to $\hat{n}$, then *S* sends to $\hat{n}$ different packets on behalf of each neighbor. We neglect the possibility that in the next round $\hat{n}$ will be chosen by *S* to receive data packets from it, as the probability for this scenario is $\frac{k}{N}$.[2]

After the end of the multicast session, *S* provides a "safety net" for cooperative nodes that did not receive all the P multicast packets. Specifically, upon receiving x fine packets from a node n, *S* sends x data packets to n, for $x \leq |L|k$. In order to avoid server overloading in the end of the multicast session, we use the randomized back-off strategy described in [111].

The node protocol

Figure 5.1 presents the data structures and parameters maintained by an EquiCast node. The set *neighbors* holds the node's neighbors. The array *my_balance* holds the node's balance with respect to each of its neighbors, and the array *neighbor_balance* holds the neighbors' balances with respect to the node. The set ids contains identifiers of data packets that the node heard about (from one or more of its neighbors) but has not yet received. The array *reqs* holds identifiers of data packets that the node asks its neighbors to send to it. The (negative) threshold L determines the minimal allowed balance. Finally, each node chooses its own upper bound H on its balance with respect to a given neighbor, which defines its level of cooperation.

The pseudo-code of the node's protocol is presented is Figure 5.2. It consists of four phases, which are executed sequentially.

In the first phase, which lasts Δ time units, a node sends to its neighbors identifiers of data packets it received in the previous round (lines 1–5).

In the second phase, which also lasts Δ time units, if the node does not receive a gossip packet from some neighbor, then the node replaces its connection with that neighbor with a connection with an emulated node by calling to the procedure replace_neighbor (lines 6–7). Then, the node

[2] In this case, if $\hat{n}$ is chosen by *S* to receive data packets in round t, then *S* can send data packets to $\hat{n}$ in round $t+1$.

upon bootstrap: $neighbors \leftarrow$
identities of nodes received from *S*

Procedure replace_neighbor (node n)
$neighbors \leftarrow neighbors \setminus \{n\}$
contact *S* and ask for an emulated neighbor $\hat{n}$
$neighbors \leftarrow neighbors \bigcup \{\hat{n}\}$

Phase I (gossip)
1. */* Send gossip packets to neighbors */*
2. **foreach** $n \in neighbors$
3. create new gossip packet p with all the data packet identifiers received in the last round
4. send $\langle$GOSSIP,$p\rangle$ to n
5. wait Δ time

Phase II (process gossip, send requests)
6. **foreach** $n \in neighbors$ from which no GOSSIP packet arrived
7. replace_neighbor (n)
8. $ids \leftarrow$ set of identifiers received in gossip packets, whose corresponding data packets were not received yet
9. **foreach** $n \in neighbors$ $reqs[n] \leftarrow \emptyset$
10. **foreach** $id \in ids$
11. $id\_set \leftarrow$ set of neighbors that gossiped about id
12. $ne \leftarrow$ a random neighbor from id_set so that $|reqs[ne]| < \frac{p}{k} + c - 3$
13. **if** there is no such ne **then continue**
14. $reqs[ne] \leftarrow reqs[ne] \cup \{id\}$
15. **foreach** $n \in neighbors$
16. send $\langle$REQUEST,$reqs[n]\rangle$ to n
17. wait Δ time

Phase III (send data)
18. **foreach** $n \in neighbors$ from which no REQUEST packet arrived
19. replace_neighbor (n)
20. */* Send data packets */*
21. send up to x data packets to n according to n's request, where $x = min(H + \frac{p}{k} - my\_balance[n], \frac{p}{k} + c - 3)$
22. $my\_balance[n] \leftarrow my\_balance[n] + x - \frac{p}{k}$
23. **if** $my\_balance[n] < L$ **then**
24. $w \leftarrow min(\frac{p}{k} + c - (x+3), \frac{p}{k})$
25. send *S* w fine packets and ask it to send w data packets to n
26. $my\_balance[n] \leftarrow my\_balance[n] + w$
27. wait $\delta - 3\Delta$ time

Phase IV (update data structures, pay fine)
28. **foreach** neighbor n
29. $d \leftarrow$ number of data packets that I asked n to send me in phase II and were received from n in this round
30. **if** $neighbor\_balance[n] < L + \frac{p}{k}$ and I received in this round m data packets from *S* on behalf of n **then**
31. $d \leftarrow d + m$
32. $neighbor\_balance[n] \leftarrow neighbor\_balance[n] + d - \frac{p}{k}$
33. */* Send a fine packet (if needed) */*
34. **foreach** neighbor n
35. **if** $my\_balance[n] < 0$ **then**
36. send a FINE packet to n
37. wait Δ time
38. **foreach** neighbor n
39. */* Check if neighbor is OK */*
40. **if** $neighbor\_balance[n] < L$ or $neighbor\_balance[n] < 0$ and n did not send me a FINE packet in this round **then**
41. replace_neighbor (n)

Figure 5.2: Code for EquiCast node.

processes gossip packets it has received from its neighbors. For each identifier in ids, the set id_set holds all the neighbors that have the corresponding data packet. One such neighbor n is randomly chosen from this set, and the node asks n to send it the corresponding data packet by appending the identifier to $reqs[n]$.

In the third phase, which lasts $\delta - 3\Delta$ time units, if the node does not receive a request packet from some neighbor, then the node replaces its connection with that neighbor with a connection with an emulated node by calling to the procedure replace_neighbor (lines 18–19). Then, the node sends data packets to each of its neighbors. Note that, according to the model (see Section 5.2), each node has an upload bandwidth of at most $p + kc$ packets per δ time units. Therefore, in the third phase, the node sends up to $x = \frac{p}{k} + c - 3$ data packets to a given neighbor n, as long as its

balance with respect to n does not exceed H (line 21). Additionally, the node increases its balance with respect to n by x. If the node's balance with respect to n is smaller than L, then the node asks *S* to send to n sufficiently many packets so that in the end of the current round the node will have a balance that is equal to or larger than L with respect to n (lines 23–25).

In the fourth phase, which lasts Δ time units, the node updates each neighbor's balance according to the number of data packets it received from the neighbor and from *S* on behalf of the neighbor in the previous phase (lines 28–32). Note that the node does not accept unsolicited data packets from its neighbors. Likewise, the node accepts data packets from *S* on behalf of some neighbor n only if, in the beginning of the fourth phase, n has a balance lower than $L+\frac{p}{k}$ with respect to the node. Then, if the node has a negative balance with respect to n, then it sends one fine packet to n. Finally, if n either has a balance lower than L or did not send the fine packet it was required to, then the node terminates its connection with n.

5.4 Proof of Cooperation

Recall that P, the number of data packets in a session, is a random variable distributed exponentially with a large expectation, at least an order of magnitude larger than $|L|k$. Hence, in every round, *S* is expected to create more than $|L|k$ new data packets in the future. In this section, we neglect the probability that, starting from some round t, *S* will create less than $|L|k$ new data packets, and hence we assume that, in every round, the probability that *S* will create more that $|L|k$ data packets in the future is 1. Moreover, for every constant $const$, $\frac{const}{R}$ is negligible (recall that $R = \frac{P}{p}$ is the total number of rounds in the multicast session), and for simplicity is assumed to be 0.

We say that a node n *maintains* a connection with another node $\hat{n}$ in some phase z of some round t if, in phase z of round t, n runs the protocol's code (described in Figure 5.2) with respect to $\hat{n}$ without changing any of the protocol's parameters except H. Note that $\hat{n}$ can be either a real node or a node emulated by *S*. We say that n maintains a connection with $\hat{n}$ throughout the multicast session if n maintains a connection with $\hat{n}$ in every phase of each of the first R rounds of the multicast session (i.e., in every round in which *S* creates new data packets).

Throughout this section, we use the following notations related to a node n and a given neighbor $\hat{n}$ of n: $b_t(n,\hat{n})$ is n's balance towards $\hat{n}$ after t rounds as stored in $n.my\_balance[\hat{n}]$. $x_t(n,\hat{n})$ is the number of data packets n (or *S* on behalf of n) sends to $\hat{n}$ during round t, and $X_t(n,\hat{n}) = \sum_{i=0}^{i=t} x_t(n,\hat{n})$.

In Section 5.4.1, we prove several basic (technical) properties of the protocol. In Section 5.4.2, we define the set of *protocol-obedient strategies (POSs)*, and we prove that every POS in which a node cooperates with all its neighbors strictly dominates every POS in which it does not. In addition, we prove that if all the nodes choose POSs, then each node receives all the multicast

packets. In Section 5.4.3, we prove that if all the nodes choose POSs, then no node can unilaterally reduce its cost by changing its strategy to a non-POS. In addition, we prove that, in this case, all the nodes receive all the multicast packets. Finally, in Section 5.4.4, we prove that each node chooses a non-negative H parameter.

5.4.1 Basic Properties

Lemma 1. *For every two neighboring nodes n and $\hat{n}$, if, starting from the initialization of the connection between them, both n and $\hat{n}$ maintain the connection between them in every phase of the first t rounds, then $n.my\_balance[\hat{n}] = \hat{n}.neighbor\_balance[n]$ in the end of round t', for every $t' \leq t$.*

Proof. By induction on the round number.
Base: There are two cases. In the first case, both n and $\hat{n}$ are not emulated nodes. In this case, the connection between the two nodes is initialized in the beginning of the multicast session (i.e., in the end of round "0"), and, upon the initialization of the connection, $n.my\_balance[\hat{n}] = \hat{n}.neighbor\_balance[n] = 0$. In the second case, either n or $\hat{n}$ is an emulated node. Without loss of generality, assume that $\hat{n}$ is an emulated node, and that n receives the identity of $\hat{n}$ from *S* during round r, for some $r \geq 0$. In this case, the connection between the two nodes is initialized in round r, and $n.my\_balance[\hat{n}] = \hat{n}.neighbor\_balance[n] = L$ upon the initialization of the connection. We note that, in round r, n and $\hat{n}$ do not send data packets to each other; this is since the connection is initialized after the end of the first (gossip) phase of round r, and hence, in round r, no gossip packets are sent on the connection between these two nodes, and therefore, in this round, no data packets are sent on this connection either. Therefore, in the end of round r, $n.my\_balance[\hat{n}] = \hat{n}.neighbor\_balance[n] = L$.
Step: Assume that, in the end of round t, $n.my\_balance[\hat{n}] = \hat{n}.neighbor\_balance[n]$. We will prove that, in the end of round $t+1$, $n.my\_balance[\hat{n}] = \hat{n}.neighbor\_balance[n]$.

In round $t+1$, both $n.my\_balance[\hat{n}]$ and $\hat{n}.neighbor\_balance[n]$ are reduced by $\frac{p}{k}$ (see Figure 5.2, lines 22 and 32). In addition, in round $t+1$, $n.my\_balance[\hat{n}]$ and $\hat{n}.neighbor\_balance[n]$ are increased upon the sending of data packets from n and from *S* on behalf of n to $\hat{n}$ (see Figure 5.2, lines 22, 26, and 32).

When n sends x new data packets to $\hat{n}$, $n.my\_balance[\hat{n}]$ is increased by x (see Figure 5.2, line 22). Since there is no packet loss, these packets are received at $\hat{n}$. We note that n sends to $\hat{n}$ only data packets that $\hat{n}$ requested from it in phase II of round $t+1$; this is since $\hat{n}$ ignores unsolicited data packets, as it maintains the connection with n (see Figure 5.2, line 29). Therefore, upon receiving the x data packets, $\hat{n}$ increases $\hat{n}.neighbor\_balance[n]$ by x.

If $n.my\_balance[\hat{n}]$ drops below L during phase III of round $t+1$, then n sends w fine packets to

S and it asks *S* to send w data packets on behalf of it to $\hat{n}$ (see Figure 5.2, lines 23–25). Additionally, $n.my\_balance[\hat{n}]$ is increased by w (see Figure 5.2, line 26). We note that n does not request from *S* to send to $\hat{n}$ more than $\frac{p}{k}$ data packets, as *S* ignores such requests (see Section 5.3.3). Hence, upon receiving the request from n, *S* sends w data packets on behalf of n to $\hat{n}$. Since there is no packet loss, these packets are received at $\hat{n}$. According to the induction assumption and since n and $\hat{n}$ maintain the connection between them, $\hat{n}.neighbor\_balance[n] < L + \frac{p}{k}$ in the beginning of phase IV of round $t+1$. Hence, $\hat{n}$ accepts the data packets received from *S*, and it increases $\hat{n}.neighbor\_balance[n]$ by w (see Figure 5.2, lines 30–32). Finally, we note that if n asks *S* to send data packets to $\hat{n}$ when $n.my\_balance[\hat{n}] \geq L$, then $\hat{n}$ ignores these data packets (see Figure 5.2, line 30), since in this case $\hat{n}.neighbor\_balance[n] \geq L + \frac{p}{k}$, and hence n asks *S* to send data packets to $\hat{n}$ only if $n.my\_balance[\hat{n}] < L$ during phase III of a given round. □

Lemma 2. *For every two neighboring nodes n and $\hat{n}$, if, starting from the initialization of the connection between them, both n and $\hat{n}$ maintain the connection between them in every phase of the first t rounds, then this connection is not terminated in the first t rounds.*

Proof. Without loss of generality, we will prove that $\hat{n}$ does not terminate the connection with n (i.e., $\hat{n}$ does not call to the procedure replace_neighbor with n's identity) in round t', for every $t' \leq t$. Since $\hat{n}$ maintains the connection with n, then it terminates the connection with n in round t' only if one (or more) of the following situations occurs: (i) it does not receive a gossip packet from n in phase I of round t'; or if (ii) it does not receive a request packet from n in phase II of round t'; or if (iii) either $\hat{n}$ is an emulated node or, in the end of round t', $\hat{n}.neighbor\_balance[n] < 0$, and $\hat{n}$ does not receive a fine packet from n in round t'; or if (iv) in the end of round t', $\hat{n}.neighbor\_balance[n] < L$.

Since n maintains the connection with n, then (i), (ii), and (iii) do not happen. In addition, we note that n can ensure that, in the end of each round, $n.my\_balance[\hat{n}] \geq L$ by asking *S* to send data packets to $\hat{n}$ when $n.my\_balance[\hat{n}] < L$ (during phase III of a given round). Hence, according to Lemma 1, (iv) does not happen either. □

Lemma 3. *If a node n maintains a connection with another node $\hat{n}$ through the first t rounds of the multicast session, then $X_t(n, \hat{n}) = \frac{tp}{k} + b_t(n, \hat{n})$.*

Proof. By induction.
Base: t = 0. $X_0(n, \hat{n}) = b_0(n, \hat{n}) = 0$. Therefore, $X_0(n, \hat{n}) = \frac{tp}{k} + b_0(n, \hat{n})$.
Step: Assume $X_t(n, \hat{n}) = \frac{tp}{k} + b_t(n, \hat{n})$. We will prove that $X_{t+1}(n, \hat{n}) = \frac{(t+1)p}{k} + b_{t+1}(n, \hat{n})$.

$X_{t+1}(n, \hat{n}) = X_t(n, \hat{n}) + x_{t+1}(n, \hat{n}) = \frac{tp}{k} + b_t(n, \hat{n}) + x_{t+1}(n, \hat{n})$. Since n maintains the connection with $\hat{n}$, we know that $b_{t+1}(n, \hat{n}) = b_t(n, \hat{n}) + x_{t+1}(n, \hat{n}) - \frac{p}{k}$ (see Figure 5.2, lines 22 and 26). Therefore, $b_t(n, \hat{n}) + x_{t+1}(n, \hat{n}) = b_{t+1}(n, \hat{n}) + \frac{p}{k}$. Hence, $X_{t+1}(n, \hat{n}) = \frac{(t+1)p}{k} + b_{t+1}(n, \hat{n})$. □

Lemma 4. *If k neighbors of a node n maintain a connection with it throughout the multicast session, then n receives from its neighbors and from S on behalf of its neighbors at least $P+Lk$ data packets*[3].

Proof. By Lemma 3, for every neighbor $\hat{n}$ of n, $X_R(\hat{n}, n) = \frac{Rp}{k} + b_R(\hat{n}, n)$. Recall that $b_R(n, \hat{n}) \geq L$ and $R = \frac{P}{p}$. Hence, from all its k neighbors, n receives at least $Rp + Lk = P + Lk$ data packets. □

Lemma 5. *The per-round overhead of maintaining a connection over the entire multicast session is at least $\frac{p}{k}+2$ packets and at most $\frac{p}{k}+c$ packets.*

Proof. The overhead incurred on a node n for maintaining a connection with another node $\hat{n}$ consists of: (i) data overhead (X_R), i.e., packets that contribute to n's balance with respect to $\hat{n}$, (ii) gossip/request packets, and (iii) penalty packets.

The maximum data overhead incurred by maintaining the connection with $\hat{n}$ is $\frac{p}{k} + c - 3$ data packets per-round (see Figure 5.2, lines 21–25). By Lemma 3, and since L is fixed, $\frac{X_R(n,\hat{n})}{R} = \frac{p}{k} + \frac{b_R(n,\hat{n})}{R} \geq \frac{p}{k} + \frac{L}{R} = \frac{p}{k}$. The gossip/request overhead is fixed, namely: two packets per-round. The penalty on either a negative balance or on maintaining a connection with an emulated node is one fine packet per round, and zero otherwise. Hence, the minimal and maximal per-round overheads are $\frac{p}{k}+2$ and $\frac{p}{k}+c$ packets, respectively. □

Lemma 6. *If a node n maintains connections with at most k-1 nodes throughout the multicast session and in addition, it communicates with a bounded number of nodes throughout a bounded number of rounds, then $f_n=\infty$.*

Proof. We first note that, during the multicast session, n cannot request from *S* to send it data packets. From at most $k-1$ neighbors with which n maintains connections throughout the multicast session (and from *S* on behalf of these neighbors), n can receive at most $x=(k-1)(\frac{p}{k}+c-3)$ data packets per round. Recall that $(k^2-k)(c-3)<p$. Hence, $x<p$. We note that $x<p$ even if n communicates with an additional bounded number of nodes throughout a bounded number of rounds. This is since n receives a bounded number, denoted as num, of data packets from these nodes, and hence $\frac{xR+num}{R}=x< p$. Finally, if n receives up to $|L|k$ data packets from *S* after the end of the multicast session, then it still cannot receive all the P multicast packets, since $\frac{xR+|L|k}{R}=x< p$. Hence, $f_n=\infty$. □

Lemma 7. *If a node n maintains connections with k nodes that also maintain a connection with it throughout the multicast session, then $f_n<\infty$ and $\frac{f_n}{R}\leq p+kc$.*

[3]Recall that L is negative.

Proof. By Lemma 4, if n maintains connections with k nodes that also maintain connections with it throughout the multicast session, then n receives at least $P+Lk$ data packets from its neighbors and from *S* on behalf of its neighbors. In addition, after the end of the multicast session, n can receive up to $|L|k$ data packets from *S* (in return for sending *S* a fine packet for each data packet), and hence $f_n=s_n<\infty$.

By Lemma 5, maintaining k connections incurs sending at most $p+kc$ packets per-round. In addition, since $\frac{|L|k}{R}=0$ (L and k are fixed), sending at most $|L|k$ fine packets in the end of the multicast session does not increase the per-round overhead. Thus, $\frac{f_n}{R}\leq p+kc$. □

We now discuss the case in which a node n maintains connections with more than k nodes throughout the multicast session. Note that by Lemma 5 and due to bandwidth limitations, n cannot maintain more than $\lfloor\frac{p+kc}{\frac{p}{k}+2}\rfloor$ connections. Below, we prove that maintaining connections with $k+1$ or more nodes incurs a higher cost than maintaining connections with k nodes.

Lemma 8. *Every strategy in which a node n exclusively maintains connections with k nodes (i.e., n communicates only with these k nodes) throughout the multicast session incurs a lower cost than every strategy in which n maintains connections with j nodes throughout the multicast session, where $j>k$.*

Proof. By Lemma 5, if n maintains connections with $k+1$ or more nodes throughout the multicast session, then $\frac{s_n}{R}\geq(k+1)(\frac{p}{k}+2)$, i.e., $\frac{f_n}{R}\geq(k+1)(\frac{p}{k}+2)$. By Lemma 7, if n exclusively maintains connections with k nodes throughout the multicast session, then $\frac{f_n}{R}\leq p+kc$. Recall that $k^2(c-2)-2k<p$. Hence, $p+kc<(k+1)(\frac{p}{k}+2)$. □

5.4.2 The Set of Protocol-Obedient Strategies (POSs)

We now define the set of POSs. Roughly speaking, a node that chooses a POS can choose which connections to maintain among those allowed by the protocol, and it doesn't communicate with anyone with which it does not maintain a connection. In this section, we prove that, if all nodes choose dominating POSs, then each node maintains connections with its initial k neighbors throughout the entire multicast session and it receives all the multicast packets.

Definition 1 (Protocol-obedient strategy (POS)). *A node's strategy is protocol-obedient if:*

POS 1. In the beginning of the multicast session, n chooses some subset of the initial k nodes given to it by S to be connected to.

POS 2. In the beginning of every phase of every round, n chooses for each node that it is connected to whether to disconnect from it or to remain connected to it, and moreover, n chooses whether

to ask S for any number of new emulated nodes to connect to as long as n does not maintain connections with more than k emulated nodes.

POS 3. For each node $\hat{n}$, n communicates with $\hat{n}$ in some phase z of some round t if and only if n is connected to $\hat{n}$ in phase z of round t according to the choices above, and moreover, in this case, n maintains the connection with $\hat{n}$ in phase z of round t.

POS 4. For any node $\hat{n}$, if $\hat{n}$ does not maintain the connection with n in phase z of round t, then n terminates its connection with $\hat{n}$ (according to the protocol) in phase $z+1$ of round t, and it does not further communicate with $\hat{n}$ starting from this phase.

Note that, in particular, following the protocol (see Figure 5.2) is a POS.

We believe that it is reasonable to assume that most users will run POSs, since the typical user usually does not have the technical knowledge to modify an application code. In addition, in many P2P applications, a node communicates with nodes whose identities are received from a centralized server. For example, in BitTorrent, a node locates other nodes by contacting a "tracker", which is a centralized process that keeps track of all nodes interested in a specific file [37, 59]. Moreover, in the next section, we prove that for each node n, if all the nodes that n communicates with choose POSs, then n also chooses a POS. That is, hacking the protocol's code cannot reduce n's cost if neither at least one of n's initial neighbors also hacked the protocol's code nor n succeeds to locate by itself identities of nodes that also hacked the protocol's code.

Definition 2 (k-protocol-obedient strategy (k-POS)). *A POS in which a node maintains exactly k connections throughout the entire multicast session is called a k-POS.*

We note that a k-POS is always feasible, since a node can always maintain connections with k emulated nodes that will also maintain connections with it. The following lemma shows that a k-POS is a dominating POS.

Lemma 9. *A k-POS strictly dominates every POS in which n maintains connections with j nodes, where $j \neq k$.*

Proof. We first note that n can communicate only with either the initial k neighbors given to it by *S* or with up to k emulated nodes; this is since n chooses a POS. We also note that maintaining a connection for a bounded number of rounds cannot reduce n's cost, since from this connection n receives a bounded number of data packets, denoted as num, and $\frac{num}{R}=0$. Hence, the lemma follows from Lemmas 6, 7, and 8. □

We next show that, if all nodes choose POSs, then a node benefits more from connections with its original k neighbors than from connections with emulated ones.

Lemma 10. *Assume that (i) all nodes choose POSs; (ii) a node n maintains a connection with a non-emulated node $\hat{n}$ in some phase z of some round t; and (iii) $\hat{n}$ also maintains a connection with n in phase z of round t. Then, n does not replace its connection with $\hat{n}$ with a connection with an emulated node e.*

Proof. Recall that e's interface is identical to $\hat{n}$'s interface with the following two exceptions: i) n's balance with respect to e is initialized to the lowest possible balance, i.e., L; and ii) in each round, n must send a fine packet to e, regardless of its balance with respect to e, otherwise e terminates its connection with n. Hence, there is no difference between the data receiving rate from $\hat{n}$ and the data receiving rate from e.

The overhead of maintaining a connection with either $\hat{n}$ or e is composed of: (i) data overhead, (ii) gossip/request packets, and (iii) penalty packets. The gossip/request overhead is fixed. The data sending rate to e is larger than or equal to the data sending rate to $\hat{n}$, since n's balance with respect to e is initialized to the lowest possible balance, i.e., L. The penalty overhead incurred by maintaining a connection with e is larger than or equal to the penalty overhead incurred by maintaining a connection with $\hat{n}$, since, at each round, n is required to send a penalty packet to e, regardless of its balance with respect to e. Finally, in order to maintain a connection with e, n needs to send a join message to *S*. Hence, the overhead of maintaining a connection with e is larger than the overhead of maintaining a connection with $\hat{n}$. Therefore, since there is no difference between the data receiving rate from $\hat{n}$ and the data receiving rate from e, n does not replace its connection with $\hat{n}$ with a connection with e. □

Theorem 1. *If all nodes choose strongly dominating strategies out of the set of POSs, then every node n exclusively maintains connections with its initial k neighbors throughout the multicast session, and it receives all the multicast packets.*

Proof. By Lemmas 9 and 10, n's strategy is to exclusively maintain connections with its initial k neighbors throughout the multicast session. By Lemma 2, these connections are maintained. Hence, n exclusively maintains connections with its initial k neighbors throughout the multicast session. Finally, by Lemma 7, n receives all the multicast packets. □

5.4.3 Unilateral Defection from the Protocol

In this section, we prove that if all the nodes, except for one node n, choose a strategy out of the set of possible POSs, then n's cost is minimized by choosing a k-POS. In other words, if all the nodes choose POSs, then no node can reduce its cost by unilaterally changing its strategy to a non-POS. Furthermore, we show that, in this case, all the nodes receive all the multicast packets.

Theorem 2. *If all the nodes, except for one (rational) node n, choose a strategy out of the set of possible POSs, then n also chooses a POS.*

Proof. We shall prove that n complies with statements POS 1– POS 4, which together comprise the definition of a POS (see Section 5.4.2).

We first note that each node, including n, complies with POS 1, since, in the beginning of the multicast session, each node receives from *S* identities of k nodes, and each node can choose whether to connect to each of its initial k neighbors.

Throughout the entire multicast session, n benefits nothing from sending packets to nodes that their identities were not received from *S*, since these nodes send no packets to n; this is since these nodes choose POSs which prohibit communication with nodes whose identities were not received from *S*. In addition, we note that if a connection between n and one of its neighbors $\hat{n}$ is terminated, then $\hat{n}$ refuses to communicate (i.e., to send packets) with n, since $\hat{n}$ chooses a POS which replaces a neighbor that does not maintain a connection with an emulated neighbor. Hence, starting from the second round, n adds connections to emulated nodes only. Finally, we note that n cannot be connected to more than k emulated nodes, since *S* does not allow such a case. Therefore, n complies with POS 2.

We note that n does not communicate with any node $\hat{n}$ is some phase z of some round t if it does not maintain a connection with $\hat{n}$ in this phase; this is since $\hat{n}$ chooses a POS that dictates terminating the connection with n in phase $z+1$ of round t if n does not maintain the connection with $\hat{n}$ in phase z of round t, and therefore n benefits nothing from communication with $\hat{n}$ in phase z of round t if this communication is not according to the protocol. Hence, n complies with POS 3.

Since n is rational, we note that n terminates a connection with a neighbor $\hat{n}$ if $\hat{n}$ does not maintain the connection with n; this is since $\hat{n}$ chooses a POS, and hence either $\hat{n}$ maintains the connection with n or it does not send any packets to n, and hence n benefits nothing from maintaining a connection with $\hat{n}$ if $\hat{n}$ does not maintain a connection with n. Hence, n complies with POS 4. □

Note that Theorem 2 holds even if all the nodes, except for n, are not rational.

Next, we establish that hacking the protocol's code cannot reduce a node's cost if neither at least one of the node's initial neighbors also hacked the protocol's code nor the node's succeeds to locate by itself identities of nodes that also hacked the protocol's code.

Theorem 3. *If all of a node's n's initial k neighbors are rational and choose POSs and n cannot locate an identity of a node that does not choose a POS, then n exclusively maintains connections with its initial k neighbors throughout the multicast session, and it receives all the multicast packets.*

Proof. The theorem follows directly from Theorems 1 and 2. □

5.4.4 Choosing H

Next, we prove that each node chooses a non-negative H parameter.

Lemma 11. *Assume that a node n maintains connections with k nodes throughout the multicast session. Assume also that some neighbor $\hat{n}$ of n requests from n to send to it $q \leq \frac{p}{k}+c-3$ data packets in some round r, and in the beginning of round r, n has a negative balance of b with respect to $\hat{n}$. Then, in round r, n sends $min(|b|, q)$ data packets to $\hat{n}$.*

Proof. We first note that, in the end of each round t, $b_t(n, \hat{n}) \geq L$, since n maintains the connection with $\hat{n}$. Thus, the sending rate to $\hat{n}$ does not affect the data receiving rate from $\hat{n}$, and hence n can minimize its sending rate to $\hat{n}$ in order to minimize its cost.

The per-round overhead incurred by maintaining the connection with $\hat{n}$ consists of: (i) data overhead ($\frac{X_R}{R}$), (ii) gossip/request packets, and (iii) penalty packets. The gossip/request overhead is fixed. Hence, n tries to minimize the data and penalty overheads.

By Lemma 3, $\frac{X_R(n,\hat{n})}{R} = \frac{p}{k} + \frac{b_R(n,\hat{n})}{R}$. The per-round data overhead is bounded from below by $\frac{p}{k} + \frac{L}{R}$. Since L is a constant that does not depend on R, we can neglect $\frac{L}{R}$, i.e., assume it is zero. The per-round penalty overhead is the percentage of rounds in which the balance is negative. Recall that, in each round, the probability that *S* will create more that $|L|k$ data packets in the future is 1. Hence, the overall cost is lower if n maintains a zero balance with respect to $\hat{n}$ in the end of each round when this is possible. Therefore, n sends $min(|b|, q)$ data packets to $\hat{n}$ in round r. □

5.5 Dynamic Setting

We now describe in a nutshell a dynamic version of EquiCast, called *DEC (Dynamic EquiCast)*, in which nodes can join and leave the protocol during its execution. Below, we detail only the differences between the two versions.

Architecture

DEC is deployed on top of a dynamic overlay that supports node joins and leaves. For example, we can use the overlay in [82], which is a dynamically maintained k-regular graph composed of $\frac{k}{2}$ Hamiltonian cycles.

The cost function

DEC's cost function is obtained from EquiCast's cost function by replacing the requirement to receive all the P multicast packets with the requirement to receive $m{\cdot}p$ data packets, where m is the number of rounds during which the node is connected to the overlay.

A join operation

A joining node n sends a *join* message to *S*. Upon receiving this request, *S* incorporates n into the overlay, e.g., by inserting n between $\frac{k}{2}$ pairs of neighboring nodes [82]. For example, assume that nodes n_1 and n_2 are connected to the overlay prior to n's joining, and n becomes n_1's neighbor instead of n_2. We describe how *S* sets n's and n_1's incoming ($neighbor\_balance$) and outgoing ($my\_balance$) balances with respect to each other.

Prior to incorporating n into the overlay, *S* asks both n_1 and n_2 for their incoming and outgoing balances with respect to each other. If these balances do not match, then *S* disconnects both n_1 and n_2 from the overlay by sending an appropriate message to all their neighbors. Hence, since both n_1 and n_2 are rational, they could be expected to correctly report about their incoming and outgoing balances with respect to each other.

Denote n_1's outgoing and incoming balances with respect to n_2 in the end of round t as B_{12} and B_{21}, respectively. We would like to ensure that n_1's cost will not increase due to n's joining. Therefore, in the beginning of round $t{+}1$, both n_1's outgoing balance with respect to n and n's incoming balance with respect to n_1 are set to B_{12}. Additionally, in the beginning of round $t{+}1$, both n_1's incoming balance with respect to n and n's outgoing balance with respect to n_1 are set to $max(B_{21}, 0)$. This is to ensure that n will not pay a fine for n_2's negative outgoing balance with respect to n_1. Finally, if $B_{21}{<}0$, then *S* sends $|B_{21}|$ new data packets to n_1, in order to ensure that it receives at least $m{\cdot}p$ data packets, where m is the number of rounds during which n_1 is connected to the overlay. Similarly, if $B_{12}{>}0$, then *S* sends B_{12} new data packets to n.

A leave operation

A leaving node n sends a *leave* message to *S*. Upon receiving this request, *S* removes n from the overlay, e.g., by connecting each pair of n's neighbors with each other [82]. For example, assume that, prior to n's leave, n was connected to nodes n_1 and n_2, and n_1 and n_2 become neighbors after n's leave. We describe how *S* sets n_1's and n_2's incoming and outgoing balances with respect to each other.

Prior to leaving the overlay, n sends to *S* its incoming and outgoing balances with respect to both n_1 and n_2. Note that n cannot gain anything from reporting about false balances, and hence n could be expected to correctly report about its balances with respect to n_1 and n_2.

Denote n_1's and n_2's outgoing balances with respect to n in the end of round t as B_{1n} and B_{2n}, respectively. We would like to ensure that n_1's and n_2's cost will not increase due to n's leave. Therefore, in the beginning of round $t{+}1$, n_1's and n_2's outgoing balances with respect to each other are set to B_{1n} and B_{2n}, respectively. Additionally, in order to ensure the protocol's correctness, in the beginning of round $t{+}1$, n_1's and n_2's incoming balances with respect to each other are set to B_{2n} and B_{1n}, respectively.

Denote n_1's and n_2's incoming balances with respect to n in the end of round t as B_{n1} and B_{n2}, respectively. If $B_{2n}{>}B_{n1}$, then n_1 may not receive $m{\cdot}p$ data packets, where m is the number of rounds during which n_1 is connected to the overlay. Hence, in such a case, *S* sends $B_{2n}{-}B_{n1}$ new data packets to n_1. Similarly, if $B_{1n}{>}B_{n2}$, then *S* sends $B_{1n}{-}B_{n2}$ new data packets to n_2.

Finally, a node n' that is connected to the overlay for m rounds may receive less than $m{\cdot}p$ data packets if it has negative incoming balances with respect to its neighbors on leave time. Hence, after it leaves the overlay, n' can receive up to $|L|k$ data packets from *S* in return for sending *S* a fine packet for each data packet.

Chapter 6

Octopus: A Fault-Tolerant and Efficient Ad-hoc Routing Protocol

6.1 Introduction

MANETs consist of mobile wireless nodes that communicate with each other without relying on any infrastructure. Therefore, routing in MANETs is performed by the mobile nodes themselves. Such nodes often intermittently disconnect from the network due to signal blockage [20, 84]. Thus, an important challenge that ad-hoc routing protocols should address is coping with such failures (or disconnections) without incurring high overhead. Our goal is to provide *fault-tolerance*, i.e., high routing reliability when many nodes frequently disconnect and reconnect, without sacrificing efficiency in routing in large MANETs consisting of hundreds of mobile nodes.

We consider *position-based routing protocols*, in which each node can determine its physical location. Such protocols scale better than non-position-based ones [91]. Typically, the location of each node is stored at some other nodes, which act as *location servers* for that node [56, 91]. When a node wishes to send packets to another node, it first issues a *location query* in order to discover the target's location, and then *forwards* packets to this location. In position-based protocols, reliability is measured as the success rate of location queries [83].

Position-based protocols differ from each other mainly in how many location servers store each node's location [91]. E.g., in DREAM [19], each node acts as a location server for all nodes, and in LAR [80], each node is a location server for its one-hop neighbors only. It has been argued [83] that neither of these extreme approaches is appropriate for large networks, since they both use flooding to disseminate either position information (DREAM) or location queries (LAR). Li et al. [83] have proposed the Grid Location Service (GLS), which stores each node's location at small number of nodes. They have shown that this approach, called *all-for-some* [91], achieves good tradeoff between reliability and load: each node updates its location at small number of nodes without flooding the network, and location queries incur a reasonable overhead. Li et al. have

further shown that in a small network, GLS tolerates intermittent node disconnections well [83]. However, as we show in Section 6.5.4, in large networks, GLS's fault-tolerance greatly degrades. For example, in a grid of $2.3km$ by $2.3km$, with an average of 400 nodes connected to the network at a given time, when half the nodes intermittently disconnect and reconnect, GLS's query success rate is only 65%; when all the nodes intermittently disconnect and reconnect, it drops to 53%.

There is an inherent tradeoff between fault-tolerance and load in all-for-some protocols, since fault-tolerance is achieved by constantly updating the location of each node at multiple location servers, which in typical all-for-some protocols [83, 63] are far from each other (in order to allow for quick location discovery). Thus, each node updates each of its location servers separately, causing the load to increase with the level of redundancy. Moreover, a location update packet is typically relayed several times before it reaches the appropriate location server, and the average number of relays increases with the network area. In order to reduce the location update overhead, in most all-for-some routing protocols, e.g., [83, 63], remote location servers are updated less frequently than close ones. In Section 6.5.4, we show that in large networks this approach greatly degrades the fault-tolerance as routing often uses stale information.

In order to achieve a better tradeoff between load and fault-tolerance, we introduce a new location update technique called *synchronized aggregation*. In this technique, each location update packet includes the locations of several nodes and updates many location servers. Moreover, updates are synchronized in the sense that only one node initiates the propagation of an aggregate update from a given region, and hence no duplicate updates are sent. It is worth noting that such a synchronized aggregation technique is not feasible in existing all-for-some protocols, e.g, [83, 63], in which the locations of nearby nodes are stored at non-adjacent location servers.

In Section 6.3, we present Octopus, a simple and efficient all-for-some routing protocol that employs synchronized aggregation in order to achieve high fault-tolerance without incurring a high load. Octopus divides the network area into horizontal and vertical strips, and stores the location of each node at all the nodes residing in its horizontal and vertical strips. This approach naturally supports synchronized aggregation: all the nodes in the same strip can learn each other's locations through the propagation of exactly two location update packets along the strip. Note that this location update technique does not require nodes to synchronize their clocks: by knowing its immediate neighbors' locations, a node can determine whether it needs to initiate a strip update. The propagation of a strip update packet does not require synchronization at all. Since synchronized aggregation dramatically reduces the location update overhead, Octopus can update all the location servers at the same high frequency- at a low cost.

On the one hand, Octopus enforces higher redundancy and more freshness of location information than previously suggested all-for-some protocols [63, 83], and hence achieves much better fault-tolerance. On the other hand, by aggregating node locations and synchronizing their propa-

gation, Octopus incurs lower overhead than these protocols in typical scenarios.

Octopus has a third important advantage over most previous all-for-some protocols, e.g., [63, 83]: In Octopus, the area in which nodes reside does not need to be pre-known or fixed; it can change at run time. This feature is crucial for rescue missions and battle field environments, in which the borders of the network are not known in advance and are constantly changing.

Finally, the redundancy of location information in Octopus has a fourth advantage: nodes use information they have about strip neighbors in order to improve the forwarding reliability. Hence, we eliminate the need to maintain designated information (for example, two-hop neighbor lists as in [83]) for improving the forwarding reliability.

In Section 6.4, we analyze Octopus's scalability: we prove that under a fixed node density, the number of location update packets per node per seconds is constant, and the byte complexity grows as $O(\sqrt{N})$ with the number of nodes N. We also analyze the probability for update and query propagation failures in Octopus's horizontal and vertical strips, and show that under reasonable density assumptions, the probability for holes is very small.

In Section 6.5, we evaluate Octopus's performance using extensive ns2 simulations with up to 675 mobile nodes. Our results show that Octopus achieves high routing reliability, low overhead, good scalability, and excellent fault-tolerance. For example, in a grid of $2.3km$ by $2.3km$ with nodes that *all* intermittently disconnect and reconnect, and an average of 400 connected nodes at a given time, Octopus achieves a query success rate of 95%, which is identical to the success rate when all nodes are constantly up. We also compare Octopus to GLS, the position-based protocol that achieved the best reliability-load tradeoff thus far. Our results indicate that in the absence of failures, Octopus achieves slightly better reliability than GLS, at lower overhead (both packets and bytes). In failure-prone settings, Octopus's reliability is greatly superior to that of GLS.

6.2 System Model

The network consists of a collection of mobile nodes moving in a rectangular space. The set of nodes can change over time as nodes connect and disconnect. The coordinates of the space can also change over time. Each node can determine its own position, e.g., using GPS. Each node can broadcast packets to all its neighbors within a certain radius r called the radio range. Packets can be lost due to MAC-level collisions or barriers.

In our simulations, we use the MAC layer provided by the *ns2* simulator, which simulates packet loss in typical MANETs. As in other protocols [63, 83], a certain minimal node density throughout the grid is required in order to ensure reliability. Thus, we assume that the number of nodes grows proportionally with the area of the network. As in [63, 83], we assume that nodes are uniformly distributed in the space.

Octopus divides the space into horizontal and vertical strips. The strip width, w, is constant and known to all nodes. Knowing w, the zero longitude and latitude, and its current location, each node can determine which horizontal and vertical strips it resides in at a given time. For example, in Fig. 6.1, node S resides in the highlighted horizontal and vertical strips, and its radio range neighbors are circled. Each strip has a unique identifier (of type StripID), identifying its location relative to the global zero coordinates.

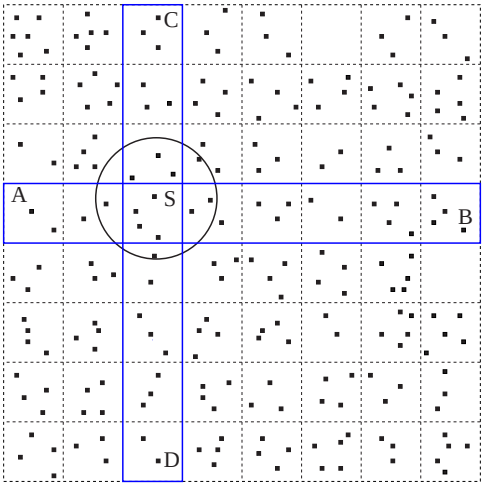


Figure 6.1: Node S's neighbors and strips. $A, B, C,$ and D are end nodes in the highlighted strips.

6.3 Octopus

Octopus is composed of three sub-protocols: *location update*, *location discovery*, and *forwarding*. The location update protocol maintains each node's location at its designated location servers, as well as at its radio range neighbors. When a node wishes to send packets to another node, it first issues a *location query* to the location discovery protocol in order to discover the target's location, and then uses the forwarding protocol to forward packets to this location. Sections 6.3.1, 6.3.2, and 6.3.3 present Octopus's location update, location discovery, and forwarding sub-protocols, respectively. The types and data structures used in the three sub-protocols are presented in Fig. 6.2.

In all three sub-protocols, we use limited retransmissions in order to partially overcome packet loss: Whenever a node A sends a packet to a node B, and B is expected to send a packet in return (e.g., to propagate/forward the packet further or respond to a location query), node A waits to hear the appropriate packet from B. If A does not hear B's packet within a *retransmissions_timeout*, then

Types:
NodeID – a node identifier.
StripID – a strip identifier.
Direction – in $\{$*north*$= 0$, *south*$= 1$, *west*$= 2$, *east*$= 3\}$
Node – $\langle$NodeID id, Real x, Real y, Time $time$, StripID hid,
StripID vid, Real p_x, Real p_y, Time $p\_time\rangle$
Data structures
Node $this$ – this node.
Set of Node $neighbors, strip[4], recent\_locations$.

Figure 6.2: Octopus's types and data structures.

A chooses another node C, distinct from B, and re-sends the packet to C. Up to two retransmission attempts are made per packet.

6.3.1 Location Update

Octopus synchronizes location updates by having them initiated only by each strip's *end nodes*. A north (south) end node is a node that has no neighbors in direction north (respectively, south) in its vertical strip, and a west (east) end node is a one that has no neighbors to the west (respectively, east) in its horizontal strip. For example, in Fig. 6.1, A, B, C, and D are end nodes in S's strips. Periodically, an end node initiates a strip update packet, which propagates along the strip towards the end node at the other side of the strip.

The location update protocol maintains two data structures at each node: *neighbors* – radio range neighbors, and *strip[i]* for $i \in \{$*north*, *south*, *west*, *east*$\}$ – nodes residing in direction *i* in the node's strip. Each element in these sets is of type Node. As shown in Fig. 6.2, this type is a tuple including the following fields: id – the node's identifier, x, y – the node's last reported coordinates, $time$ – the time of the last received coordinates report, hid, vid – the node's horizontal and vertical StripIDs, p_x, p_y – the node's previous coordinates, and p_time – the time of the previous received coordinates report.

The *neighbors* set is updated upon receiving a short HELLO packet from another node. This packet is broadcast by every node every $hello\_timeout$ seconds, and it contains the broadcasting node's identity and physical coordinates. If a node does not hear from some neighbor n for $2hello\_timeout$ seconds, it removes n from *neighbors*.

The pseudo-code for maintaining *strip[*]* is presented in Fig. 6.3. The locations of all the nodes in a given strip are propagated through the strip via the periodic diffusion of STRIP_UPDATE packets initiated by the end nodes of the strip every *strip_update_timeout*. An end node broadcasting a STRIP_UPDATE packet to direction d includes in the packet all its *neighbors* that are in the same

strip. A STRIP_UPDATE packet also includes the strip identifier, the packet direction, and a target node, which will forward this packet further. The target is chosen to be the farthest node in the propagation direction.

```
1.  loop forever
2.      foreach Direction d do
3.          if (I have no neighbors in direction d) then
4.              strip[d] ← ∅
5.              StripID sid ← get_strip_id (d)
6.              propagate_packet(sid, opposite direction to d)
7.      sleep (strip_update_timeout)

Event handler:
8.  upon receive ⟨STRIP_UPDATE, sid, d, set, next⟩ do
9.      if (sid = this.vid ∨ this.hid) then
10.         strip[opposite direction to d] ← set
11.     /* If I am the packet target */
12.     if (this = next) then
13.         propagate_packet (sid, d)

Procedures:
14. set of Node get_nodes_in_strip (sid)
15.     return {this} ∪ {n ∈ neighbors | n.hid = sid ∨ n.vid = sid}

16. StripID get_strip_id (d)
17.     if d ∈ {north, south} then
18.         return this.vid
19.     return this.hid

20. void propagate_packet (sid, d)
21.     set of Node set ← strip[opposite direction to d]
                          ∪ get_nodes_in_strip(sid)
22.     Node next ← farthest node in direction d in set
23.     /* If propagation is not complete */
24.     if (this ≠ next) then
25.         bcast ⟨STRIP_UPDATE, sid, d, set, next⟩
```

Figure 6.3: The strip update protocol.

Upon receiving a STRIP_UPDATE packet, a node updates the appropriate entry in *strip[*]*. If the node is designated as the packet target and is not the strip's end-node, then it appends to the packet all its *neighbors* that reside in the packet's strip, chooses a new target, and broadcasts the packet. The propagation of a STRIP_UPDATE packet completes when it reaches an end node, i.e., when the farthest node in direction d is the current node (*this = next*). For example, in Fig. 6.1, a STRIP_UPDATE packet with direction south begins at node C and propagates to the south-most node of the strip, D.

Forwarding holes

We define a *forwarding hole* to be a situation in which a node X cannot forward a STRIP_UPDATE packet to direction d in a strip s although there is another node in s that is in direction d of X. For example, in Fig. 6.1, there is a forwarding hole south of node B. In a typical scenario, the probability for a forwarding hole is small (less than 0.02, see Section 6.4.2). Moreover, as we describe in Section 6.3.2, storing each node's location at both the horizontal and vertical strips quadratically decreases the probability for query failures due to forwarding holes.

Although the probability for a routing failure due to forwarding holes is small, we have implemented a simple bypass mechanism in order to overcome such failures: in this mechanism, a node that cannot forward a STRIP_UPDATE packet to direction d in a strip s forwards the packet to a node that is in direction d of it and resides in an adjacent strip to s. Empirically, the additional reliability achieved by this bypass mechanism is negligible (less than 2%), since Octopus already achieves high reliability without it. Therefore, for simplicity reasons, we present and evaluate Octopus without the bypass mechanism.

Correctness

We now identify circumstances under which Octopus's location update protocol achieves 100% reliability, i.e., correctly stores node locations at all of their designated location servers. We note, however, that in the presence of failures, movements, packet loss, and uneven node distribution, these ideal circumstances are not always achieved. Nevertheless, in Section 6.5, we show that in typical scenarios with frequent failures and movements, Octopus's reliability is close to 95%.

Lemma 12. *In a run in which there are no node movements or failures and no packet loss, if the strip width $w \leq \frac{\sqrt{3}}{2}r$ and the bound on packet delay is less than $hello\_timeout$, then in every segment of a strip in which there are no forwarding holes, every node eventually knows the identities and locations of all the nodes that reside in this segment.*

Proof. We first note that all the nodes' neighbors' sets are accurate, i.e., include exactly all the nodes within their radio range, since there is no packet loss, the bound on packet delay is less than $hello\_timeout$, and a node is removed from the current node's *neighbors* set only if the current node does not hear from this node for $2hello\_timeout$ seconds. Therefore, after at most $2hello\_timeout$ seconds, only an end node initiates a propagation of a STRIP_UPDATE packet. Note also that in a segment of a strip with no holes, a propagation of a STRIP_UPDATE packet from one end node is guaranteed to eventually reach the other end node of the segment, since there is no packet loss or failures.

Consider a segment of strip s with no holes. Assume that the segment's end node A sends a STRIP_UPDATE packet $m1$ to node B, and then B sends a STRIP_UPDATE packet $m2$ to node

C. Without loss of generality, assume that s is a horizontal strip. Consider a node N in s whose x coordinate is between A's and B's, at distance Δx from A's x coordinate. If $\Delta x \leq \frac{r}{2}$, then N is in A's radio range, and hence it receives $m1$. Since $w \leq \frac{\sqrt{3}}{2}r$ and A's *neighbors* set is accurate, $m1$ contains all the nodes in s within $\frac{r}{2}$ meters of A in the direction of $m1$, as all these nodes are within A's radio range (see Fig. 6.4). Therefore, after receiving $m1$, N knows the identities and locations of all the nodes between it and A. If $\Delta x > \frac{r}{2}$, then N receives $m2$ as it is in B's radio range (see Fig. 6.4). According to the protocol, since A's and B's *neighbors* sets are accurate, $m2$ contains all the nodes in s that are within A's and B's radio ranges. Thus, in both cases, after the broadcast of $m2$, N knows the identities and locations of all the nodes in s whose x coordinates are between N's and A's. Note that, since there are no movements or failures, and since only end nodes initiate updates, parallel propagations of different STRIP_UPDATE packets do no violate the protocol's correctness, as such packets contain the same information.

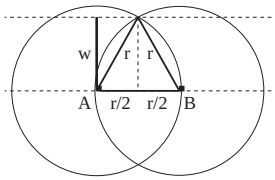


Figure 6.4: A strip of width $w = \frac{\sqrt{3}r}{2}$.

By induction, we get that after propagating a STRIP_UPDATE packet from A to Z, the end node at the other end of the segment, each node knows the identities and locations of all the nodes in the segment between it and A. Likewise, after propagating a STRIP_UPDATE packet from Z to A, each node knows the identities and locations of all the nodes in s between it and Z. □

Although Lemma 12 requires $w \leq \frac{\sqrt{3}r}{2}$ to ensure that nodes are not missed by a STRIP_UPDATE propagation, the simulations in Section 6.5.1 show that increasing w from $\frac{\sqrt{3}r}{2}$ to r does not hurt the reliability, since increasing w also reduces the probability for forwarding holes (see Section 6.4.2), and hence may increase the reliability.

6.3.2 Location Discovery

The location discovery protocol uses the information stored in *strip[*]* and *neighbors*, as well as the set *recent_locations*, which is a cache of recently discovered target locations. The cache entries expire after *strip_update* seconds. The location discovery protocol is presented in Fig. 6.5.

The interface to the location discovery protocol consists of the function *locate*, which upon success results in addition of its target to recent_locations. It first searches the target in one of

```
locate (Node_ID tid)
1.    Node target ← search_locally (tid)
2.    if (target = null) then
3.        search_location (this, tid, north)
4.        search_location (this, tid, south)
5.        sleep (discovery_timeout)
6.        if (target ∉ recent_locations) then
7.            search_location (this, tid, west)
8.            search_location (this, tid, east)

Event handlers:
9.  upon receive ⟨QUERY, src, t_id, d, next⟩ do
10.   if (next = this) then
11.       Node target ← search_locally (t_id)
12.       if (target = null) then
13.           search_location (src, t_id, d)
14.       else /* target found - send reply */
15.           Direction d' ← opposite direction to d
16.           send_reply (src, target, d')

17. upon receive ⟨REPLY, src, target, d, next⟩ do
18.   recent_locations ← recent_locations ∪ {target}
19.   if (next = this) then
20.       send_reply (src, target, d)

Macro:
21. strip_neighbors[d] ≜ (neighbors ∩ strip[d]) ∪ {this}

Procedures:
22. Node search_locally (target_id)
23.   if (∃n s.t. n ∈ neighbors ∪ strip[*] ∪ recent_locations
24.       ∧n.id = target_id) then
25.       return n
26.   return null

27. search_location (src, t_id, d)
28.   Node next ← farthest node in strip_neighbors[d] in the
29.                same square as this or in an adjacent square
30.   if (next ≠ this) then
31.       bcast ⟨QUERY, src, t_id, d, next⟩

32. send_reply (src, target, d)
33.   Node next ← closest node to src in strip_neighbors[d]
34.   if (next ≠ this) then
35.       bcast ⟨REPLY, src, target, d, next⟩
```

Figure 6.5: The location discovery protocol.

the locally maintained sets (*strip[*], neighbors*, and *recent_locations*). If the target's location is not found in these sets, the protocol broadcasts two QUERY packets to the node's north-most and south-most neighbors in its square or in adjacent squares in its vertical strip. The recipient of a QUERY packet continues the search in the same manner, forwarding the packet in the same direction if needed. Once a QUERY packet reaches a node that knows the target, it broadcasts a REPLY packet with its information about the target towards the source. Every node that receives a REPLY packet adds the located target to its *recent_locations*. In rare cases in which no REPLY packet is received within *discovery_timeout* seconds, the search is repeated in the same manner in a west-east directions.

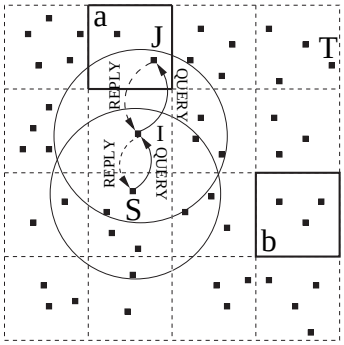


Figure 6.6: Successful query location.

Fig. 6.6 depicts how node S discovers node T's location. S broadcasts QUERY packets to the north and south. The next hop of the north-going packet is I. I fails to discover T's location locally, and forwards the packet to its north-most neighbor J. T is in J's *strip[east]*. Thus, J broadcasts a REPLY packet containing T's location towards S. This packet reaches I, which in return broadcasts the packet to S.

Correctness

As in the previous section, we identify circumstances under which Octopus's location discovery service achieves 100% reliability.

Lemma 13. *Consider a run with no node movements, node disconnections, or packet loss, and assume that* $w \leq \frac{\sqrt{3}}{2}r$ *and the bound on packet delay is less than* $hello\_timeout$*. Consider a location query with nodes* S *and* T *as the query's source and target, respectively. Let square* a *(*b*) be the intersection between* S*'s vertical (horizontal, respectively) strip and* T*'s horizontal (vertical, respectively) strip (see Fig. 6.6). If there are no forwarding holes between* S *and* a *and between* T *and* a*, or there are no holes between* S *and* b *and between* T *and* b*, then* S*'s* recent_locations *eventually includes* T*'s location.*

```
1.  send (m, T)
2.      spawn thread to run locate(T)
3.      wait until exists n ∈ recent_locations s.t. n.id = T
4.      forward (m, n)

5.  forward (Packet p, Node target)
6.      update_coordinates (target)
7.      Node next ← closest node to target in neighbors ∪ {this}
8.      if (next = this) then
9.          target' ← closest node to target in strip[*]
10.         update_coordinates (target')
11.         next ← closest node to target' in neighbors
12.     bcast ⟨FORWARD, p, target, next⟩

Event handler:
13. upon receive ⟨FORWARD, p, target, next⟩ do
14.     if (target = this) then
15.         deliver p
16.     else if (next = this) then
17.             forward (p, target)

Procedure:
update_coordinates (t)
18.     Update t.x, t.y, t.time according to the current time and t's
19.     direction of movement obtained from t's last two reported
20.     coordinations. Store old values in t.p_x, t.p_y, t.p_time.
```

Figure 6.7: The forwarding protocol.

Proof. Without loss of generality, assume that there are no forwarding holes between S and a and between T and a. Since QUERY packets never skip over squares (see *search_location* in Fig. 6.5) and there is no packet loss, a QUERY packet propagating along the strip reaches to some node N that resides in a. By Lemma 12, N knows T's location. Since N does not move or fail, it initiates a REPLY packet. Since there are no holes or packet loss, this packet propagates back to S, and S includes T in its *recent_locations* set. □

6.3.3 Data Forwarding

Fig. 6.7 describes the process of sending a data packet m from the current node S to a target node T. First, S calls to the function $locate$ (see Fig 6.5) in a separate thread. When S's $recent\_locations$ set contains T's location, S forwards the data packet to T using the interface $forward$ of the forwarding protocol.

Octopus employs geographic forwarding [91] in order to forward data packets to their destinations. The basic version of geographic forwarding works as follows: each node has knowledge

of its one-hop neighbors and their locations. Each intermediate node forwards a data packet to its neighbor that is geographically closest to the packet's destination. This protocol is efficient, but it may fail if an intermediate node is a *local maximum*, i.e, it is closer to the destination than all of its neighbors.

In case of a forwarding failure, Octopus chooses an alternative target, $target'$, which is the closest node to the packet destination from the sets ***strip[*]*** and forwards the packet to its neighbor that is geographically closest to $target'$. We illustrate this recovery technique in Fig. 6.8, where node S needs to forward a data packet to node T. S is closer to T than all of its radio range neighbors. S chooses node E (the closest node to T from S's ***strip[*]***) as an alternative target, and forwards the packet to A (S's closest neighbor to E). Note that the packet's ultimate destination remains unchanged, and subsequent forwarding steps follow the basic geographic forwarding if possible. In Section 6.5, we show that this recovery technique is very effective, achieving the same reliability as two-hop geographic forwarding as used in [83] .

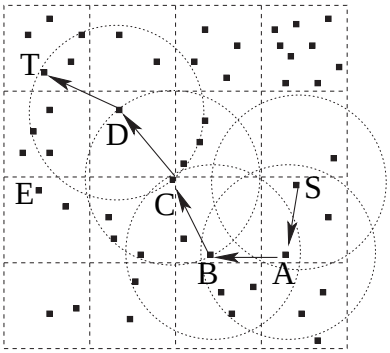


Figure 6.8: Octopus's forwarding protocol.

Since nodes continue to move while packets are en route to them, it is important to constantly *re-estimate* the target's location. In each forwarding step, the forwarding node forwards the data packet to the target's estimated location. This location is calculated according to the target's last two reported coordinates, which are included in the Node data structure sent in REPLY and FORWARD packets.

6.4 Analysis

In Section 6.4.1, we analyze Octopus's scalability, and in Section 6.4.2 we analyze the probability for forwarding holes.

6.4.1 Scalability

The following lemma shows that the message complexity of Octopus's location update protocol is constant with respect to the network size.

Lemma 14. *Assuming a fixed node density ρ, the per node per second packet complexity of the location update protocol does not grow with the network size.*

Proof. We first observe that the average distance that a STRIP_UPDATE packet traverses each time it is forwarded to a node that it is not at the end of a strip is independent of the network size: this distance depends only on the radio range, the node density, and the strip width. Asymptotically, when the grid is large, most of the nodes are not close to the ends of the grid. Hence, we neglect the effect of the location of the forwarding node on the average propagation distance. Denote the average propagation distance by δ.

Second, we observe that the probability for a forwarding hole at any particular point in the strip is independent of the network size. Therefore, the average percentage of the strip in which there are no forwarding holes is constant with respect to the network size. Denote this portion by α.

In a single iteration of the strip update protocol, the propagation of STRIP_UPDATE packet(s) along a strip with an edge length of e requires an average of $\frac{\alpha e}{\delta}$ transmissions in each direction. Denote $\sigma = 1/$*strip_update_timeout*. Then on average, $\frac{2\alpha e\sigma}{\delta}$ STRIP_UPDATE packets per strip are sent in a second. In order to obtain the average per node message complexity, we divide this number by the expected number of nodes in a strip, which is ρew, and multiply it by 2 since STRIP_UPDATE packets are propagated in both horizontal and vertical strips. Therefore, on average, each node broadcasts $\frac{4\alpha e\sigma}{\delta\rho ew} = \frac{4\alpha\sigma}{\delta\rho w}$ STRIP_UPDATE packets per second, which is independent of the network size.

In addition to STRIP_UPDATE packets, the location update protocol also sends HELLO packets. Since each node broadcasts HELLO packets at a fixed frequency, the total per node per second message complexity incurred by the location update protocol is constant with respect to the network size. □

The next lemma shows that the byte complexity of Octopus's location update protocol with N nodes is $O(\sqrt{N})$.

Lemma 15. *Assuming a fixed node density, the per node per second byte complexity incurred by the location update protocol with N nodes is $O(\sqrt{N})$.*

Proof. Recall that in our model, we assume that N nodes are uniformly distributed in the network area. Therefore, assuming a fixed node density, when we increase N, the network edge size, e, increases by $O(\sqrt{N})$, and therefore, the number of nodes in each strip increases like $O(\sqrt{N})$.

Thus, the number of bytes in STRIP_UPDATE packets increases like $O(\sqrt{N})$. The size of a HELLO packet is constant.

From Lemma 14, we get that the number of packets sent per node does not increase with N, and therefore the overall per node byte complexity of the location update protocol is $O(\sqrt{N})$. □

6.4.2 Update/Query Propagation Reliability

Forwarding holes in strips may hamper Octopus's reliability, as they may prevent location updates from propagating in the entire strip. We now analyze the probability for forwarding holes. We show that under reasonable density assumptions, this probability is very small, which explains why Octopus achieves excellent reliability in the simulations below.

A forwarding hole occurs when a node has no radio range neighbors in the strip in the direction the packet is going, i.e., when there are no nodes in the intersection between the forwarding node's radio range and the strip in the packet's direction. For example, in Fig. 6.9, a hole in N's east direction occurs if there are no nodes in the area denoted by A. The size of this area depends on w, r, and the node's location relative to the strip boundaries. Without loss of generality, let us examine a horizontal strip. Consider a node whose y coordinate is at distance d from the south boundary of the strip. Using the equation for the area of a circular segment [1], we compute A as follows:

$$\begin{aligned} A_s(d) &= r^2 \cos^{-1}\left(\frac{d}{r}\right) - d\sqrt{r^2 - d^2} \\ A\ (d) &= \frac{\Pi r^2 - (A_s(d) + A_s(w-d))}{2} \end{aligned}$$

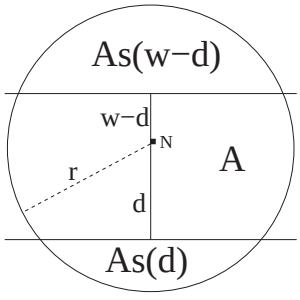


Figure 6.9: Node N has a forwarding hole in direction east if area A is uninhabited.

For an asymptotic analysis, we use a Poisson node distribution. Since the expected number of nodes in an area of size A is ρA, we get that the probability of no nodes residing in A is:

$$Pr_d = e^{-\rho A(d)}$$

Since this probability varies with d, in order to compute the average probability for a forwarding hole we need to average Pr_d for d's in $[0, w]$. We observe that Pr_d monotonically decreases when d grows from 0 to $w/2$ (as the area gets larger), and then symmetrically increases as d grows from $w/2$ to w. The highest probability occurs when $d = 0$ or $d = w$. We compute a coarse lower bound of the probability for holes by considering two cases: first, when d is between $w/4$ and $3w/4$, and second when d is not in the middle half of the strip. We bound the probability for the first case by looking at its minimum point, where $d = w/4$, and we bound the second case by looking at its minimum point, where $d = 0$. We get the following:

$$Pr[\text{hole}] < \frac{1}{2}Pr_{w/4} + \frac{1}{2}Pr_0$$

When we instantiate the formula above with $\rho = 75$, $w = r = 0.25$ (used in most of our simulations), we get that $Pr[\text{hole}] < 0.02$. This explains why Octopus achieves high query success rate in typical scenarios. With a strip width of $0.2 = \frac{4r}{5} < \frac{\sqrt{3}r}{2}$, which ensures that location updates and queries are received at all the nodes residing in segments of the strip they propagate through, we get that $Pr[\text{hole}] \approx 0.0327$. Hence, we see that two opposing tendencies affect the protocol's reliability: increasing w beyond $\frac{\sqrt{3}r}{2}$ reduces the probability for a forwarding hole, and hence increases the reliability, but it also increases the probability that a location update or query will not be received by all the nodes residing in segments of the strip it propagates through. Our simulations in Section 6.5.1 show that these two strip widths achieve virtually the same reliability.

6.5 Evaluation

We now evaluate Octopus using simulations. Octopus is implemented in *ns2* [6] with CMU's wireless extensions. Each node uses the IEEE 802.11 radio and MAC model provided by the CMU extensions, with a radio range r of 250 meters and a throughput of $1\frac{Mb}{sec}$. The nodes are initially placed uniformly at random in a square universe. In most of our simulations, there are 75 nodes per square kilometer. (Li et al. [83] have experimentally shown that such a node density is required in order to achieve high forwarding reliability.) Each node moves using the random waypoint model used in [83]: it chooses a random destination and moves toward it with a constant speed chosen uniformly between zero and $10\frac{m}{sec}$. When a node reaches its destination, it chooses a new destination and immediately begins moving toward it at the same speed. For each set of parameters, we run five 300 seconds long simulations, and in each simulation, each node initiates an average of one location query a minute to random destinations, starting 30 seconds into the simulation, and ending at 270 seconds. In all of our experiments, the results of all the five simulations were very close to each other. This consistency is due to the large number of events in each simulation.

In Section 6.5.1, we discuss our choice of the protocol's parameters. In Section 6.5.2, we examine Octopus's scalability as the number of nodes and network area increase. In Section 6.5.3 we evaluate the reliability of Octopus's forwarding sub-protocol and compare it with two-hop geographic forwarding. In Section 6.5.4, we study Octopus's fault-tolerance. Finally, in Section 6.5.5, we compare Octopus's reliability, overhead, and fault-tolerance to those of GLS.

6.5.1 The Choice of Parameters

In the simulations reported below, each node broadcasts a HELLO packet every 2 seconds, as was done in GLS [83]. We chose this frequency in order to allow a fair comparison between the two protocols. Nevertheless, we also ran experiments with a $hello\_timeout$ of up to five seconds, and the results were virtually identical. This occurs due to the nature of movement in the random way point model, which allows a node to predict a neighbor's location in the near future from the neighbor's last two reported coordinations.

We set the *strip_update_timeout* to 10 seconds. Empirically, reducing this value, e.g., to 5 seconds, results in a negligible increase in the protocol's reliability. On the other hand, increasing this timeout to 20 seconds, decreases the reliability by $5\%-10\%$.

The *retransmissions_timeout* and *discovery_timeout* were set to 2 seconds each, as in other protocols, e.g., LAR [80]. This timeout value was chosen since, in all our failure-free experiments, more than 95% of the successful queries are received at the source within two seconds from the time they are issued. We allow up to two retransmissions per packet. Empirically, we observed that increasing the number of retransmissions beyond two has a negligible effect of the protocol's reliability.

Finally, we examine the effect of the strip width on the protocol's reliability and overhead. In Section 6.3.1, we proved that when $w \leq \frac{\sqrt{3}}{2}r$, location updates are guaranteed to cover all the nodes residing in segments of the strip they propagate through. Increasing w beyond this threshold may cause some nodes to be missed by location updates passing next to them. Nevertheless, increasing w does not necessarily hamper Octopus's reliability. This is so because it reduces the probability for forwarding holes, as it increases the area of the intersection between nodes' radio ranges and their strips (see Section 6.4.2), and thus reduces the probability that no nodes reside in this area. When $r = 250m$, $\frac{\sqrt{3}}{2}r = 216m$. We experiment with strip widths of 200 and 250 meters. In order to ensure a fair comparison, we examine grid edge lengths that are divisible by both 250 and 200. Fig. 6.10 shows the query success rate as a function of the number of nodes and the grid's edge length for OCTOPUS-250 (where $w = 250$) and OCTOPUS-200 (where $w = 200$). The 95% confidence intervals for the results presented in this figure are very tight: up to $\pm 0.8\%$ of the average value. We see that the query success rate is very similar for both strip widths. We conclude

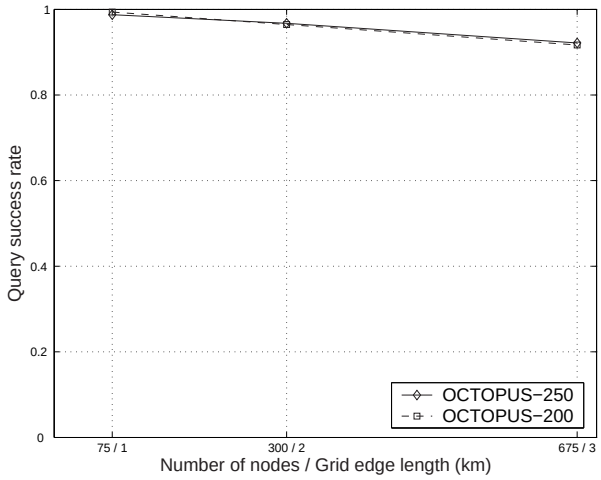
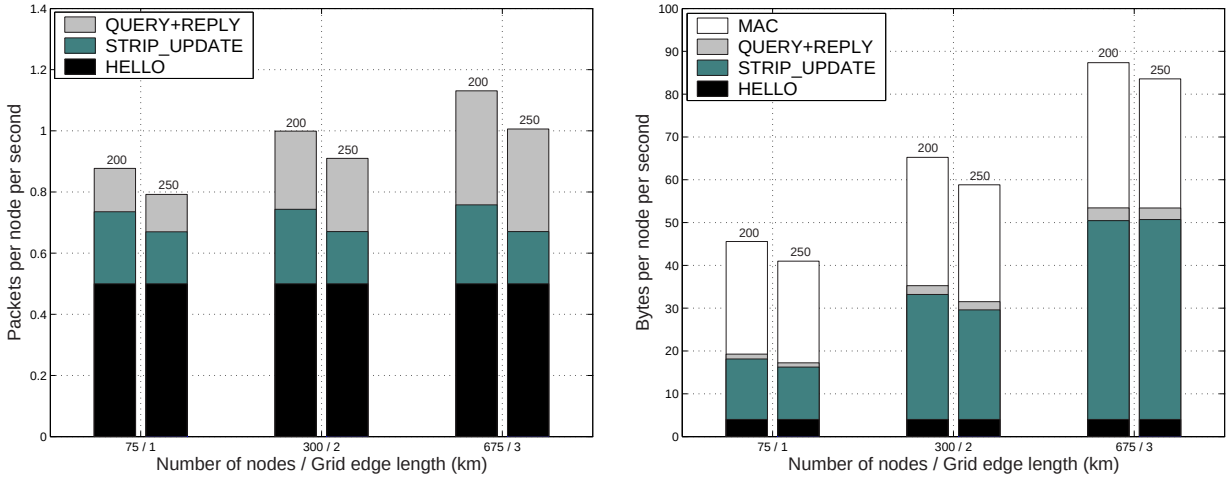


Figure 6.10: Octopus's query success rates for different strip widths.

that under a density of 75 nodes per square kilometer, setting $w = r$ does not reduce the reliability compared to choosing $w \leq \frac{\sqrt{3}}{2}r$.



(a) packet overhead (b) byte overhead

Figure 6.11: Octopus's overhead for different strip widths.

At the same time, increasing w reduces the number of STRIP_UPDATE packets sent, since there are fewer strips. Although the size of each STRIP_UPDATE packet increases as there are more nodes in each strip, the total number of node locations sent in all STRIP_UPDATE packets does not change. Since each transmitted packet also includes a MAC header, sending the same information in fewer packets reduces the total number of bytes sent by the protocol. Indeed, Fig. 6.11(a) and Fig. 6.11(b) show that increasing the strip width from $200m$ to $250m$ reduces

the per node packet and byte complexities of Octopus. Fig. 6.11(a) shows for each setting the average number of packets of each type and Fig. 6.11(b) shows the average number of protocol bytes in each packet type as well as (in white) the average number of bytes in MAC headers. The 95% confidence intervals for the results presented in Fig. 6.11(a) and Fig. 6.11(b) are up to ± 0.01 packets and ± 0.1 bytes of the average value, respectively, indicating that the results are accurate. Henceforth, we fix the strip width at $250m$.

6.5.2 Scalability

We now examine Octopus's scalability. We first examine the impact of increasing the network size while maintaining a fixed node density, and then focus on the effect of increasing the node density.

Increasing the network size

As the network area increases, the probability for forwarding holes in the update/query path increases, and therefore, the reliability inevitably degrades. We observe that regardless of strip width or density, this degradation is very gradual (see Fig. 6.10).

Figure 6.11 examines the increase of Octopus's overhead as the network size and the number of nodes grow. Fig. 6.11(a) shows that the number of location update packets sent by Octopus is constant, matching the analysis in Section 6.4.1. The overall packet overhead gradually increases with the network size and the number of nodes. The moderate increase in the per query overhead stems from the increased failure probability of the first discovery attempt (in the north-south directions), which leads to more cases in which locations are also searched in the east-west directions. Nevertheless, this increase is gradual, because the failure probability is low even in large grids. We note that similar phenomena occur in other all-for-some protocols [83, 63, 48, 117], where the probability for query failures also increases with the network area. This, in turn, increases the overhead due to query retries or trying alternative location servers.

Fig. 6.11(b) examines the increase in Octopus's byte overhead as the network size and the number of nodes grow. We note that the byte (and packet) overhead incurred by broadcasting HELLO packets is constant with respect to the networks size. Although most of the broadcasted packets are of type HELLO, their byte overhead is small, since these packets are very small. As expected, the number of bytes in STRIP_UPDATE packets increases with the network size (see Section 6.4.1). As explained above, the number of QUERY and REPLY packets also increases with the network size (see Fig.6.11(a)), and hence the number of bytes in these two types of packets also increases with the network size. However, this increase is negligible, as these packets are very small.

The effect of node density

We now examine what happens when the node density increases from 75 to 100 nodes per square kilometer. Fig. 6.12 shows that the query success rate remains similar. This occurs because of two opposing tendencies: On the one hand, increasing the density reduces the probability for forwarding holes, and thus improves reliability. On the other hand, as the node density increases, the probability for MAC-level collisions increases, and therefore, more packets are lost, which reduces the reliability. The 95% confidence intervals for the results presented in Fig. 6.12 are up to $\pm 1\%$ of the average value.

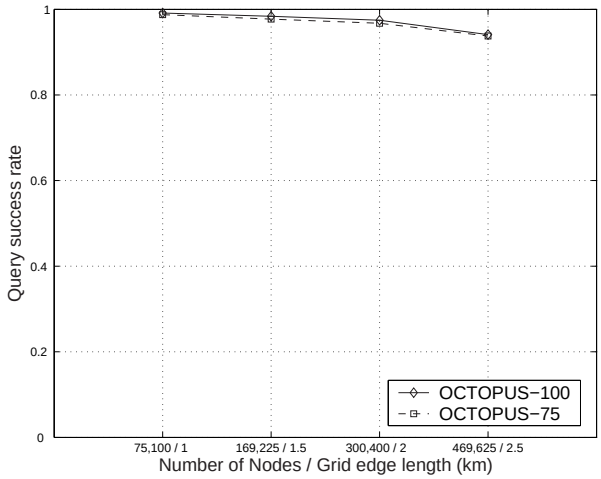
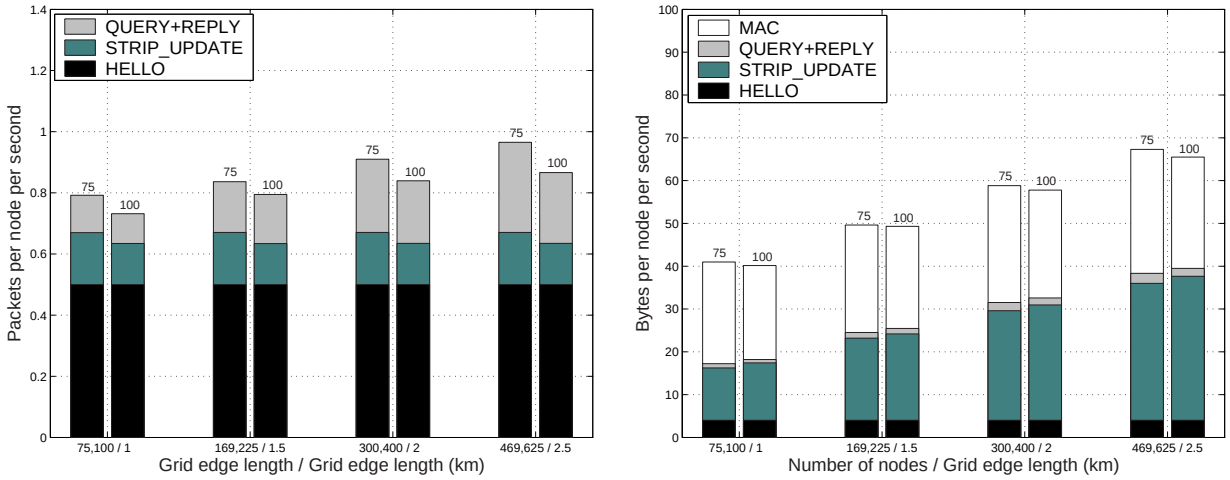


Figure 6.12: Octopus's query success rates for different node densities.

In Fig. 6.13(a) and Fig. 6.13(b), we see that increasing the density reduces Octopus's per node message and byte complexity. The message complexity is reduced since the number of STRIP_UPDATE packets sent in each strip does not grow, while these packets are divided among more nodes. Although the number of node locations sent in each STRIP_UPDATE increases, sending fewer packets per node reduces the MAC overhead, and the overall per node byte complexity is therefore also reduced. The 95% confidence intervals for the results presented in Fig. 6.13(a) and Fig. 6.13(b) are up to ± 0.01 packets and ± 0.1 bytes of the average value, respectively.

6.5.3 Data Forwarding

In order to evaluate the reliability of Octopus's forwarding sub-protocol, we run simulations in which data traffic is sent. Our simulation scenario follows the one in [83]. Each node's radio bandwidth is $2\frac{Mb}{sec}$. In each simulation, data traffic is generated by a number of constant bit rate connections equal to half the number of nodes; no node is a source in more than one connection;



(a) packet overhead (b) byte overhead

Figure 6.13: Octopus's overhead for different node densities.

no node is a destination in more than three connections. Each source sends four 128-byte data packets each second for 20 seconds. Each simulation lasts for 300 seconds, and data packets are sent at random times between 30 and 270 seconds into the simulation. All other parameters are as in the simulations described above. We vary the number of nodes and the grid's edge length, while maintaining a node density of roughly 75 nodes per square kilometer.

We compare the reliability of Octopus's forwarding sub-protocol with that of two-hop geographic forwarding, which is employed, e.g., by GLS. For both protocols, target locations are discovered using Octopus's location discovery sub-protocol. Fig. 6.14 shows that the forwarding reliability of the two protocols is virtually identical. The 95% confidence intervals for the results presented in this figure are up to $\pm 1\%$. We conclude that the high redundancy of Octopus's location information is an adequate substitute for storing dedicated information for increasing forwarding reliability. Note that the additional overhead for maintaining the two-hop neighbor lists needed for two-hop forwarding is substantial, and it grows with the node density.

6.5.4 Fault-Tolerance

Octopus's main design goal was to provide high fault-tolerance in the presence of intermittently disconnecting nodes. We now examine whether this design goal is met. To this end, we introduce *unstable* nodes, which alternate between being connected and disconnected [83]. Each time an unstable node awakens, it remains connected for a time interval chosen uniformly at random in the range $[0, 120]$ seconds. And when it disconnects, it remains disconnected for a time interval chosen uniformly at random in the range $[0, 60]$ seconds. Thus, at any given time, an average of $\frac{2}{3}$ of the

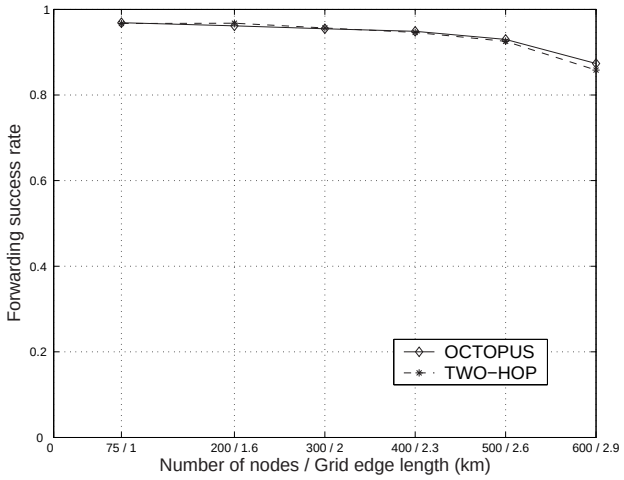


Figure 6.14: Octopus's data forwarding reliability.

unstable nodes are connected. We experiment with a varying percentage p of *unstable* nodes. The remaining nodes are connected throughout the simulation. We experiment in a fairly large grid of $2.3km$ by $2.3km$. In order to isolate the effect of node disconnections without impacting the density, we fix the average number of connected nodes at a given time at 400. That is, we run $\frac{400}{1-p+\frac{2}{3}p}$ nodes (e.g., 480 nodes when $p = 0.5$). Note that although the average density of live nodes at any given time is not reduced, it is still challenging to achieve high reliability, since part of the global state is lost with each node disconnect, whereas new nodes connect without any location information. Therefore, protocols that employ low redundancy, e.g., GLS, fail to achieve high routing reliability in the face of disconnects (see Fig 6.19).

Clearly, location queries for nodes that are disconnected during the location query or shortly beforehand or afterwards are bound to fail. Likewise, nodes that disconnect shortly after issuing a location query will inevitably not receive the query response. We therefore only take into account queries whose target is connected during the interval $[t - 10, t + 10]$ seconds, where t is the query issue time, and whose query source is connected during the interval $[t, t + 10]$ (the same approach was taken in [83]). Note that we only require the source and query target to remain connected– all other nodes, including the target's location servers and the nodes along the search path, can disconnect at any time. A successful query location is followed by the transmission of one 128-byte data packet from the source to the target.

Fig. 6.15 shows the query success rate and the overall data forwarding reliability as a function of the percentage of unstable nodes. The 95% confidence intervals for the results presented in this figure are up to $\pm 1.4\%$. We see that Octopus achieves perfect fault-tolerance: its query and

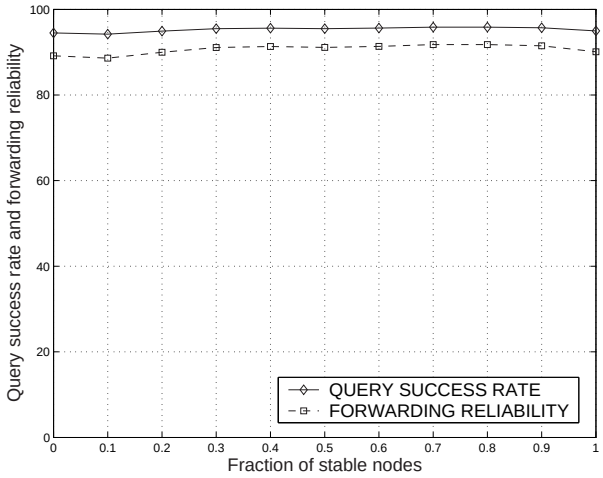


Figure 6.15: Octopus's fault-tolerance: query success rate and data forwarding reliability are virtually unaffected by the percentage of the unstable nodes.

forwarding success rates do not degrade at all as we increase the percentage of unstable nodes. This impressive fault-tolerance is achieved thanks to the high level of redundancy in Octopus, and the freshness of the redundant information: Consider a source S issuing a query for a target T. The query succeeds when it reaches a location server in the intersection of S and T's strips. There are at least two such squares (one in S's horizontal strip, and one in its vertical strip). Every 10 seconds, T's location is stored at all the nodes residing in these two squares (since *strip_update_timeout* is 10 seconds). Assuming there are no forwarding holes, as long as one of the nodes in these squares remains connected during the 10 seconds interval, the query should be successful. When the node density is 75, the average population of these two squares is 9.375 nodes. Even when all the nodes in the network are unstable, the probability of all these nodes failing within 10 seconds is negligible. Note also that the probability for holes does not increase when nodes are unstable, since the average node density is fixed. Therefore, Octopus's forwarding reliability does not degrade as we increase the percentage of unstable nodes. This is due to the fact that forwarding failures mainly occur due to holes. In addition, forwarding failures due to node disconnections are usually overcome using retransmissions to alternative nodes.

6.5.5 Comparison with GLS

We now compare the reliability, overhead, and fault-tolerance of Octopus to those of GLS. We use the ns2 implementation of GLS from MIT [4]. In these experiments, we use the grid sizes and densities from GLS's original evaluation [83], with one exception: in the smallest grid ($1km$ by $1km$) we place 75 nodes instead of 100 in order to maintain a similar node density of roughly

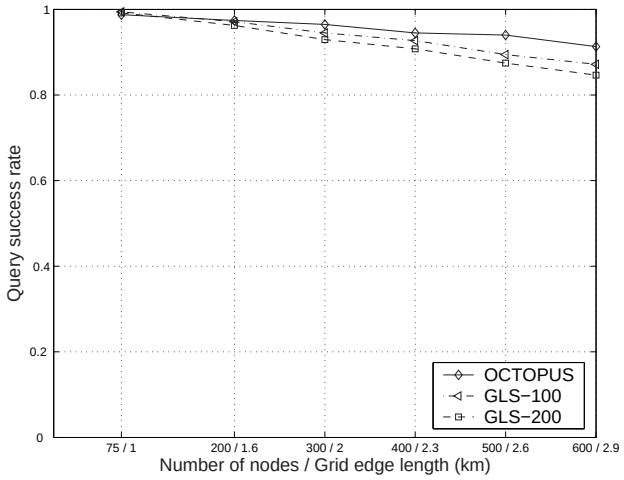
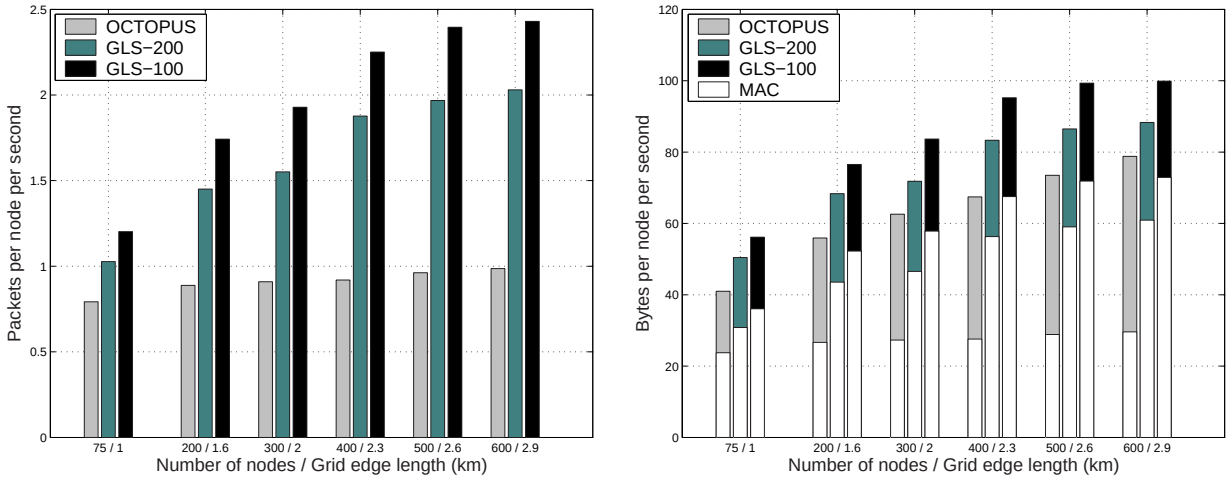


Figure 6.16: Octopus versus GLS: query success rates.

75 nodes per square kilometer in all grid sizes. Note that these scenarios are not optimized for Octopus, since most of the grid edge sizes are not multiples of Octopus's strip width ($250m$).

Fig. 6.16 shows the query success rates of Octopus and GLS. The 95% confidence intervals for the results presented in this figure are up to $\pm 0.8\%$. GLS-100 and GLS-200 are GLS simulations with a location update threshold of $100m$ and $200m$, respectively. In GLS-d, a node updates its order-i location servers after each movement of $2^{i-2}d$ meters. We see that with either threshold, Octopus achieves similar reliability to GLS in a small network, and better reliability than GLS in medium and large networks. Octopus's advantage is most notable in the largest grid, where Octopus's reliability is roughly 4% and 7% higher than GLS-100's and GLS-200's, respectively. The reliability gap between Octopus and GLS increases with the grid size because of the lower freshness of location information stored at GLS's remote location servers. Whereas in Octopus, a node updates all its location servers at the same high frequency (every 10 seconds), in GLS, the average frequency at which a node updates its location servers grows with the grid size. For example, in the $2.9km$ by $2.9km$ grid, a GLS-100 node updates its order-4 location servers only after moving 400 meters, and its order-5 location servers after a movement of 800 meters. Thus, a node moving at the average speed ($5\frac{m}{sec}$) updates its order-4 (order-5) location servers only every 80 (respectively, 160) seconds.

Fig. 6.17 compares Octopus's overhead to that of GLS. The 95% confidence intervals for the results presented in Fig. 6.17(a) and Fig. 6.17(b) are up to ± 0.01 packets and 0.1 bytes, respectively. We observe that thanks to aggregation, Octopus sends a smaller number of packets than GLS. Moreover, as the network size grows, GLS's packet overhead increases drastically, while Octopus's packet overhead increases very moderately. This occurs since, as opposed to Octopus,



(a) packet overhead (b) byte overhead

Figure 6.17: Octopus versus GLS: overhead.

GLS does not employ aggregation, and hence the number of location servers each node needs to update grows with the network size. In addition, the average distance between a node and its location servers also grows with the network size. Although Octopus's location update packets are larger than GLS's, by sending fewer packets, Octopus reduces the number of bytes sent in MAC-level headers. Therefore, overall, Octopus's byte complexity is smaller than GLS's (see Fig. 6.17(b)). Although GLS's overhead appears to grow more moderately in large networks, this is simply because its reliability drops more sharply in such settings: e.g., in a $2.9km$ by $2.9km$ grid, GLS's reliability drops to only 85%, and therefore many location update and query packets do not reach their destinations, and are hence relayed less times than needed.

Next, we consider simulations with data traffic. In Section 6.5.3, we showed that the reliability of Octopus's forwarding sub-protocol is similar to the reliability achieved by the two-hop geographic forwarding protocol employed by GLS. We now compare their overhead. We measure the total (data and protocol) packet overhead incurred by both protocols in the simulation scenario of Section 6.5.3. Fig. 6.18 shows the average per node per second number of packets sent by Octopus and the more efficient version of GLS, GLS-200. The 95% confidence intervals for the results presented this figure are up to ± 0.01 packets. We do not measure the byte overhead, because it is dominated by the data traffic. As the figure shows, Octopus sends fewer packets than GLS. In addition, Octopus's overhead grows more moderately with the network size than GLS's overhead.

Finally, Octopus's greatest advantage over GLS is its fault-tolerance. In Fig. 6.19, we contrast Octopus's fault-tolerance against that of the more reliable version of GLS, GLS-100. The 95% confidence intervals for the results presented in both of these figures are up to $\pm 1.4\%$. As explained in Section 6.5.4, we experiment with an average of 400 connected nodes at a time, on a $2.3km$ by

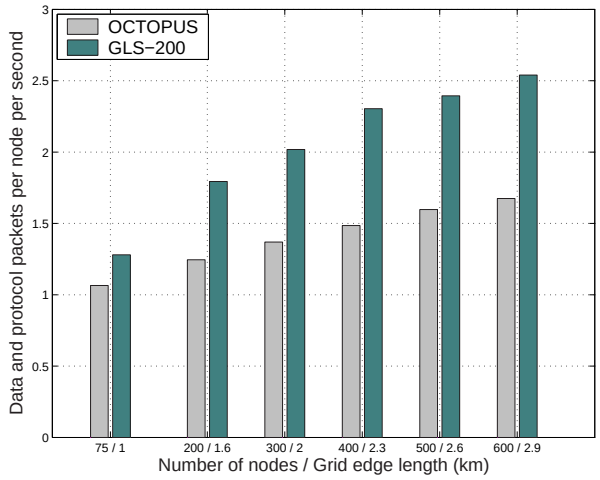


Figure 6.18: Octopus versus GLS: data and protocol packets sent.

$2.3km$ grid. Whereas Octopus's reliability does not degrade when the percentage of unstable nodes increases, GLS's reliability greatly degrades with the number of unstable nodes: when 50% of the nodes are unstable, GLS's query success rate goes down to less than 65%, and when all the nodes are unstable, it drops to less than 53%. GLS is less fault-tolerant than Octopus for two reasons: first, GLS employs less redundancy, and second, in GLS, it takes reconnecting nodes a long time to update their remote location servers.

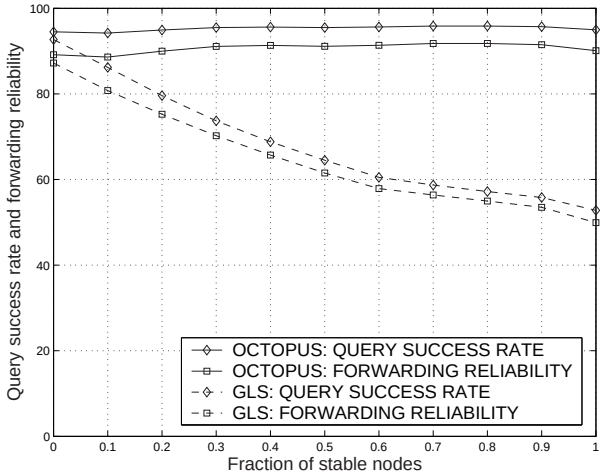


Figure 6.19: Octopus versus GLS: fault-tolerance.

Chapter 7

Evaluating Unstructured P2P Lookup Overlays

7.1 Introduction

In unstructured P2P lookup systems, peers self organize into unstructured overlay networks. Examples to such systems include eMule, Freenet, and Gnutella. Unstructured lookup systems incur small constant overhead per single join or leave operation, and can easily support keyword searches. Chawathe et al. [34] have argued that these two features of unstructured lookup systems are highly important, as users frequently join and leave lookup sessions, and keyword searches are more popular than exact-match queries. Indeed, most of the currently deployed lookup systems are unstructured ones.

In unstructured lookup systems, the search is not structural and may fail. However, queries usually succeed in locating files due to natural file redundancy [34], that is, popular files are held by many nodes. Most unstructured P2P lookup systems and some partially-structured ones employ flooding in order to locate a searched object, at least among a subset of the nodes, e.g., super-peers in KaZaA. Due to the natural file redundancy, it is usually enough to limitedly flood the network in order to locate a searched object [87]. The main reason for using flooding is due to the high search reliability achieved by it. Nevertheless, as with all other search techniques, the dependability of flood-based search depends on the robustness of the overlay: in a highly connected overlay, flooding achieves high reliability, even in dynamic failure-prone environments, whereas in a disconnected overlay, it may fail to locate an object that is stored in the system. Flooding also incurs low latency, and can locate many copies of a searched object. However, flooding is also inefficient, as it creates a high number of duplicate search messages, i.e., multiple copies of a query may be sent to a given node by its multiple neighbors. Another problem with flooding is the difficulty to choose the appropriate TTL (Time-To-Live), which controls the flooding propagation. A high TTL achieves high search reliability but also incurs high overhead. The flooding effectiveness versus

the overhead it incurs mainly depends on characteristics of the overlay. These characteristics also determine how the flooding overhead is distributed among the different nodes, and the overlay's dependability. In this chapter, we define metrics capturing the above important overlay features and evaluate a number of overlays according to these metrics.

Our first metric, *c*, is the overlay's connectivity, i.e., the minimal number of disjoint paths between a pair of nodes in the overlay. This metric measures the overlay's fault-tolerance in the presence of node failures and disconnections, and hence captures the search dependability.

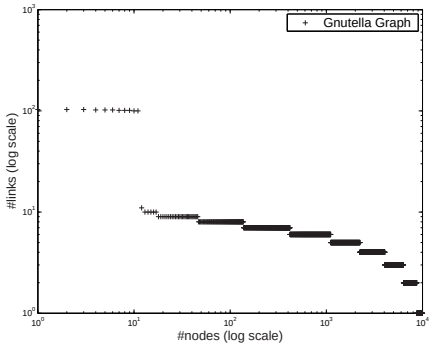
The second metric, *fe (flooding efficiency)*, evaluates the flooding coverage versus the overhead it incurs. Assume a query q is propagated from a random node with a TTL of i. Then, *fe(i)* is defined as $\frac{N_i}{M_i}$, where N_i is the expected number of nodes that receives q and M_i is the expected number of copies of q that are sent. A high *fe* value implies a small number of duplicates, and hence high efficiency.

The third metric, *cg (coverage granularity)*, measures the difference in the coverage when increasing the TTL by one. A small *cg* allows one to build an adaptive dependable lookup system that adjusts to varying failure rates, where faults include node and link failures. For example, if *cg* is small, increasing the TTL by one upon multiple query failures will increase the search reliability at the expense of a slightly higher overhead. Likewise, reducing the TTL by one upon succeeding to locate many copies of searched objects will result in overhead reduction while achieving similar search reliability. For a given TTL i, we define *dcg(i)* as $\frac{N_{i+1}}{N_i}$, where N_i is as defined above.

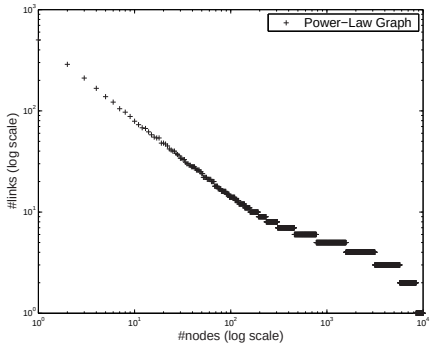
Our final metric, *lb (load balancing)*, evaluates how the flooding overhead is distributed among the nodes. Assume a query is initiated from a random node with a certain TTL. In a random overlay, the probability that a random node is requested to forward this query to its neighbors is proportional to the node's degree. Therefore, it is desirable that overlays would be degree-balanced, in order to incur similar overhead on all nodes. This is becoming more important now, as many ISPs have started to limit the maximal bandwidth consumption of every user. For random overlays, we define *lb* as $\frac{d_{max}}{d_{min}}$, where d_{max} (d_{min}) is the maximal (minimal, respectively) node degree.

We evaluate different graphs and overlays according to the above four metrics. We start by evaluating a Gnutella graph, which is a typical file sharing application graph. We proceed by applying our metrics on several synthetic graph structures, including a power law graph, normal random graphs, and a 3-regular random graph (a *k-regular random graph* with N nodes is a graph chosen uniformly at random from the set of k-regular graphs with N nodes). Finally, we evaluate an Araneola's overlay [93], which is a distributed approximation of a k-regular random graph. Our results show that a 3-regular random graph and a 3-Araneola overlay achieve the best (virtually identical) results.

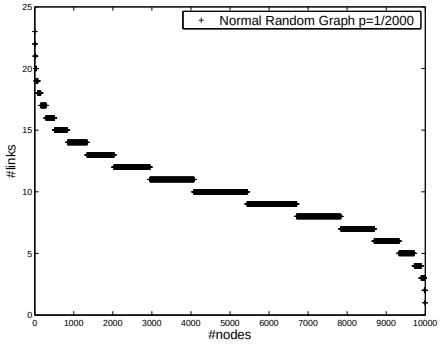
In addition, we examine the join overhead in each of the graphs mentioned above. We observe that a Gnutella graph and an Araneola overlay incur the lowest construction and maintenance



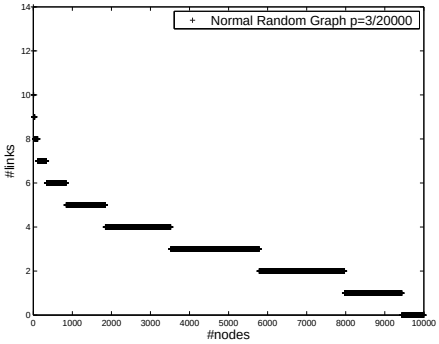
(a) A Gnutella-like graph.



(b) A power-law random graph.



(c) A normal random graph, $p = \frac{3}{20000}$.



(d) A normal random graph, $p = \frac{1}{2000}$.

Figure 7.1: Distribution of node degrees in four graphs. Note that we use log scale for the power-law random and Gnutella graphs, while for the normal random graphs we use a linear scale.

overhead: in these two graphs structures, each join (or leave) operation is handled locally and entails the sending of a small constant number of messages. In normal random graph constructions, a join or leave operation is also handled locally, though such operation incurs sending $O(\log N)$ messages, where N is the number of nodes in the system. In a power law graph, some nodes have a high degree, proportional to N, and hence joining/leaving of such nodes inevitably entails high overhead. In contrast to the above four graph structures, there are no known distributed constructions of k-regular random graphs. Therefore, with this graph structure, a single join or leave operation requires reconstructing the graph anew, and hence leads to an overhead of $\Omega(N)$ messages.

This chapter proceeds as follows: In Section 7.2, we describe in detail the tested graphs, and in Section 7.3 we evaluate these graphs according to our metrics. Finally, in Section 7.4, we analyze the join cost in each of the graphs.

7.2 The Evaluated Overlays

In our study, we use six undirected graph topologies. In all of the graphs, there are $10,000$ nodes. We start with a Gnutella-like graph. This graph was constructed using a node degree distribution of a real Gnutella graph taken from [106]. In order to allow a fair comparison among all the six topologies, we extrapolated the data from [106] in order to create a $10,000$ node graph. We kept an average of 3.4 links per-node as in [106], and a node degree distribution similar to the one in [106]. In such a graph, there is a small number of highly-connected nodes, with 100 or more links, and the majority of the nodes have a degree between 3 and 10. Similar characteristics also occur in other P2P file sharing applications [34, 51]. Fig 7.1(a) shows the node degree distribution of the Gnutella-like graph. We compare this graph with a power-law random graph. In this graph, the ith node chooses $\frac{w}{i^{\alpha}}$ other nodes as its neighbors, where $w = 500$, $\alpha = 0.8$, and $1 \leq i \leq 10,000$. We use this setting in order to achieve an average node degree of 3.4 links per-node, as in the Gnutella-like graph. Fig 7.1(b) shows the node degree distribution of this graph.

Next, we use two normal random graphs, one with $p = \frac{3}{20,000}$ and the second with $p = \frac{1}{2000}$, in which a node creates a connection with a given other node with a probability of $\frac{3}{20,000}$ and $\frac{1}{2,000}$, (respectively). The resulting average node degrees are 3 and 10, (respectively). We use the first normal random graph in order to allow a fair comparison with the previous two graphs. However, since such a graph is not connected (a normal random graph is connected if and only if $p = O(\log N)$ [26]), we also use the second connected normal random graph. Fig 7.1(c) and Fig 7.1(d) show the node degree distributions of these two graphs.

Next, we use a 3-regular random graph, in which each node is connected to three other random nodes. Finally, we use a 3-Araneola overlay [93], in which roughly 90% of the nodes have a degree of 3, while the rest have a degree of 4, leading to an average node degree of roughly 3.1.

7.3 The Metrics

7.3.1 Connectivity

Table 7.1 presents the connectivity of the different graphs. A k-regular random graph and a k-Araneola graph are almost always k connected [93, 122]. Therefore, such graphs achieve high dependability even with high failure-rates, (includes node and link failures). A normal random graph is connected with high probability if p is at least logarithmic in the number of nodes [26]. Therefore, the first normal random graph is disconnected (connectivity 0). The second one has a connectivity of 1. The power-law random graph and the Gnutella-like graph have a connectivity of 1, as several nodes in these graphs have a degree of 1. Such nodes are very likely to be disconnected from the overlay graph. For a given number of links, we observe that a 3-regular random graph

Graph	Connectivity
3-regular random graph	3
3-Araneola overlay	3
Normal random graph($p = \frac{3}{20{,}000}$)	0
Normal random graph($p = \frac{1}{2{,}000}$)	1
Gnutella-like graph	1
Power-law random graph	1

Table 7.1: Connectivity: A 3-regular random graph and a 3-Araneola overlay has a connectivity of 3. The rest of the graphs have a connectivity of 1 or 0.

and a 3-Araneola overlay achieve much higher connectivity than a Gnutella graph, a power-law random graph, and normal random graphs, due to their regular structure. In fact, a 3-regular random graph and a 3-Araneola overlay, in which the average node degree is roughly 3, achieve higher connectivity than a normal random graph with an average node degree of 10.

7.3.2 Flooding Efficiency

We now evaluate the flooding efficiency in all the graphs except the normal random graph with $p = \frac{3}{20{,}000}$, as this graph is not connected. For each graph, we run the flooding protocol $10,000$ times, one time from each node, and we calculated the average flooding efficiency. We report about our results in Fig. 7.2, and Fig. 7.3 shows the coverage achieved with each TTL.

In a power-law random graph and in a Gnutella-like graph, starting from a TTL of 4, the flooding efficiency, i.e., the coverage divided by the overhead, is poor. This is due to the presence of high-degree nodes in both of the graphs, which create and receive many duplicate search messages. A similar phenomenon occurs in the normal random graph with $p = \frac{1}{2{,}000}$, as the degrees in such a graph range from 1 to 23. In contrast, in low degree balanced graphs such as a 3-Araneola overlay and a 3-regular random graph, the flooding efficiency is very high. For small TTLs (≤ 8), the flooding efficiency of the 3-regular random graph and the 3-Araneola overlay is very close to one. Hence, for such TTLs, flooding is as efficient as random walks. Fig. 7.3 shows that with a TTL of $7/8/9$, flooding over a 3-Araneola overlay and a 3-regular random graph reaches, on average, to $485/989/1957$ (roughly 4.85%/9.9%/20%) and $376/739/1424$ (roughly 3.8%/7.4%/14%) nodes, respectively. Therefore, with a 3-Araneola overlay and a 3-regular random graph, it is possible to reach any desired portion of the nodes efficiently; this is thanks to their good coverage granularity, as discussed in the next section.

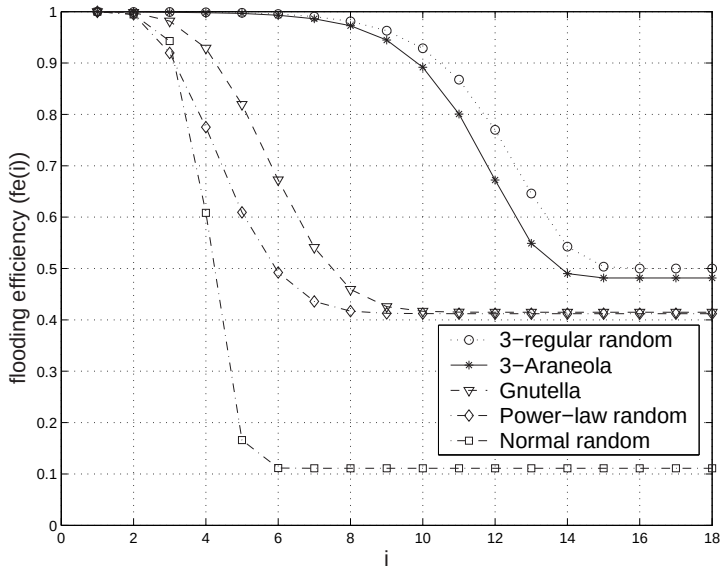


Figure 7.2: Flooding efficiency: for effective TTLs, a 3-Araneola overlay and a 3-regular random graph achieve a near to perfect search efficiency. Other graphs achieve much lower search efficiency.

7.3.3 The Coverage Granularity

Recall that *cg(i)* is defined as $\frac{N_{i+1}}{N_i}$, where N_i is the expected number of nodes that receive a query that originates from a random node with a TTL of i. Fig. 7.4 shows *cg(i)* for the five graphs evaluated in the previous section. As the figure shows, a 3-Araneola overlay and a 3-regular random graph have a low (virtually identical) *cg(i)* value for all TTLs. In addition, in these two graphs, *cg(i)* is very similar for all the TTLs. This is due to the fact that k-regular random graphs are good expanders. Therefore, in these two graphs, one can adapt the search dependability and overhead according to the failure rate. In contrast, in the rest of the graphs, *cg(i)* is very high for small (effective) TTLs and low for high (ineffective) TTLs. In addition, in these graphs, the low coverage granularity is achieved only when the flooding efficiency is poor (see Section 7.3.2).

7.3.4 Load Balancing

It is desirable that the flooding overhead would be distributed equally among all nodes. Recall that for a random overlay, we define the load balancing (*lb*) as $\frac{d_{max}}{d_{min}}$, where d_{max} (d_{min}) is the maximal (minimal, respectively) node degree. In the normal random graph with $p = \frac{3}{20{,}000}$, we ignore nodes with degree 0, as they are not connected to the overlay. Table 7.2 shows the *lb* value of the different graphs. The 3-regular random graph achieves perfect load balancing, i.e., 1. Next, the 3-Araneola overlay achieves excellent load-balancing: $\frac{4}{3}$. The two normal random graphs have *lb* values of

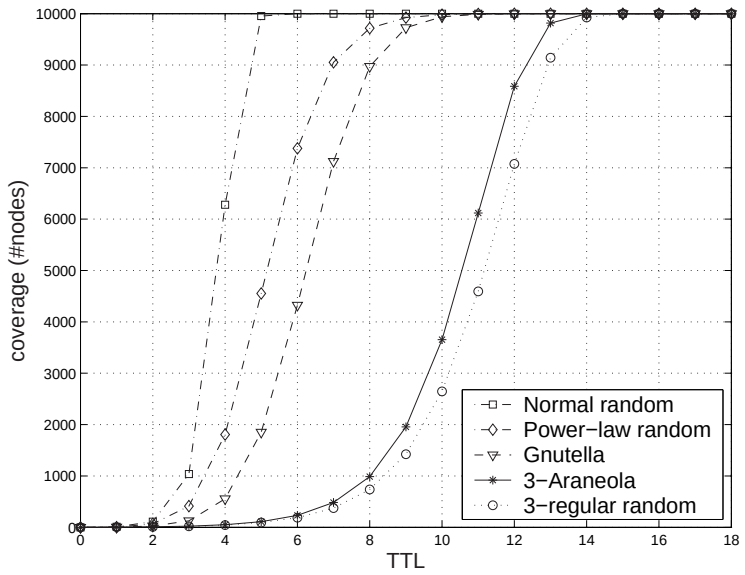


Figure 7.3: Coverage versus TTL.

Graph	$lb = \frac{d_{max}}{d_{min}}$
3-regular random graph	1
3-Araneola overlay	$4/3$
Normal random graph($p = \frac{3}{20{,}000}$)	$14/1$
Normal random graph($p = \frac{1}{2{,}000}$)	$23/1$
Gnutella-like graph	$103/1$
Power-law random graph	$502/1$

Table 7.2: Load balancing: a 3-regular random graph achieves perfect load balancing of 1. A 3-Araneola overlay achieves a good load balancing of $\frac{4}{3}$. The rest of the graphs achieves poor load balancing.

$\frac{14}{1}$ and $\frac{23}{1}$. In such graphs, assuming queries are distributed uniformly, the overhead incurred on a highly-connected node may be $O(\log N)$ times the overhead incurred on a low-connected node, as in such graphs a connected node's degree is between 1 and $O(\log N)$. In the Gnutella-like graph and the power-law random graph the load balance is even worse, as the overhead incurred on a highly-connected node can be two orders of magnitude greater than the overhead incurred on a low-connected node.

7.4 The Join Cost

The results of Section 7.3 have shown that the 3-regular random graph and the 3-Araneola overlay are the best overlays among the tested graphs. We now examine the cost/feasibility of distributed

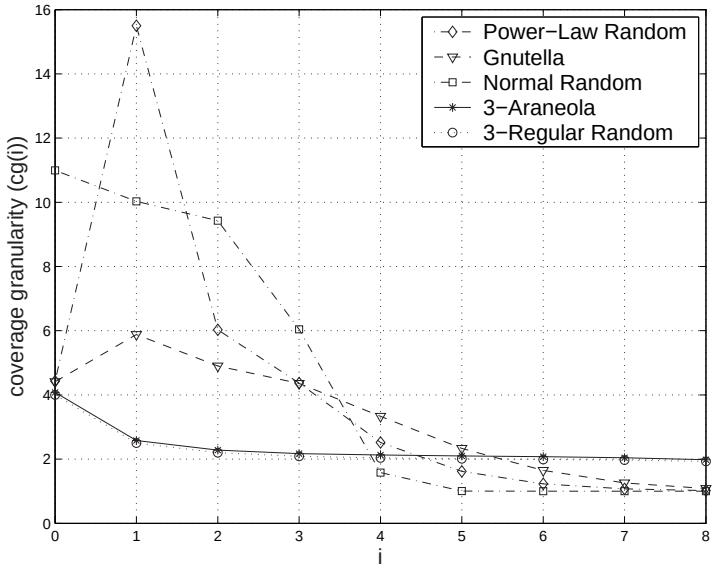


Figure 7.4: Coverage granularity: a 3-Araneola overlay and a 3-regular random graph achieve a good *cg* value for all TTLs. In the rest of the graphs, *cg(i)* is very high for small (effective) TTLs and low for high (ineffective) TTLs, in which the flooding efficiency is poor.

constructions of the tested graphs. Specifically, we examine the join overhead in each of the graphs. We evaluate this overhead in two ways. We first assume the existence of a membership service that maintains at each node a small number of random node identities. Examples to such scalable membership services can be found in [41, 93]. Next, we evaluate the join overhead without relying on the existence of a membership service. In this case, we assume that a joining node knows the identity of some other node that is currently in the system. We assume, however, in this case that a random walk of $O(\log N)$ steps from a given node reaches a random node. Law et al. [82] have shown that this assumption is true for expander graphs. Note that a scalable membership service amortizes the logarithmic cost of knowing a random node by aggregating membership information, and hence it is more efficient than a random walk for retrieving random node identities.

Table 7.3 shows the join cost for each graph in both cases. In a 3-Araneola overlay, a join operation requires sending $3k = 9$ messages, assuming the existence of a membership service. In the absence of such a service, connecting to a random node requires sending $O(3+\log N) = O(\log N)$ messages. In a Gnutella graph, the overheads are similar to the overheads above. However, for a high degree node, i.e., one that has 100 or more links, the leave overhead is very high, as such a node is connected to many other nodes.

In a normal random graph and in a power-law random graph, given a membership service, the join cost is the node's degree. In a connected normal random graph this degree is logarithmic in the number of nodes in the system, and in a power-law random graph a node's degree can be $O(N)$.

Graph	The join cost (with membership service)
3-Araneola overlay	9
Gnutella-like graph	constant
Normal random graph	$O(\log N)$
Power-law random graph	$O(N)$
3-regular random graph	$\Omega(N)$

Graph	The join cost (without membership service)
3-Araneola overlay	$O(\log N)$
Gnutella-like graph	$O(\log N)$
Normal random graph	$O(\log^2 N)$
Power-law random graph	$\Omega(N)$
3-regular random graph	$\Omega(N)$

Table 7.3: The join cost: A 3-Araneola overlay achieves the lowest join cost.

Therefore, assuming the existence of a membership service, the joining overhead in a (connected) normal random graph and in a power-law random graph is $O(\log N)$ and $O(N)$, respectively. In the absence of a membership service, these overheads need to be multiple by $O(\log N)$, the overhead for retrieving a random node. Finally, in a k-regular random graph, since no distributed constructions of such a graph are known, a join operation requires the reconstruction of the entire graph, leading to a prohibitive overhead of $\Omega(N)$ messages.

Chapter 8

Discussion, Results, and Conclusions

P2P systems achieve high scalability and robustness, and can be easily deployed. Hence, P2P computing is a promising architecture for deploying distributed services. However, in order to realize their full potential, P2P systems need to cope better with real-world problems like failures, dynamic behavior, and selfishness. This dissertation has addressed these problems in two different network settings: over the Internet and in MANETs. Although there are substantial differences between these settings, we have seen that similar considerations and challenges arise in both. Below, we review the challenges in P2P computing we have studied in this dissertation, and our solutions to these challenges.

Fault-tolerance. Conventional wisdom suggests that failures are overcome using redundancy. The challenge is doing so without creating unreasonably high load. We have presented techniques for providing redundancy at a lower cost than previous work, thanks to the use of an optimal overlay structure (in Araneola, as shown in Chapter 4) and aggregation (in Octopus, as shown in Chapter 6).

Dynamic behavior. Dynamic behavior further emphasizes the need for allowing fast low-overhead incorporation into the system. Araneola quickly incorporates joining nodes and removes leaving (or failing) ones thanks to the use of an unstructured overlay network: in Araneola, a joining node not only receives all the messages sent after its creation, but actually receives 100% of the messages sent up to 6 rounds before its join. In addition, each join, leave, or failure is handled locally, and entails the sending of only about $3k$ messages in total, independent of the number of nodes. Octopus achieves perfect fault-tolerance to node connections and disconnections thanks to employing high level of redundancy, as well as the freshness of the redundant information. This fault-tolerance is achieved without incurring high overhead thanks to aggregating node locations and synchronizing their propagation.

Selfishness. P2P networks suffer from the problem of "freeloaders", i.e., users who consume resources without contributing anything in return. EquiCast enforces cooperation through two mechanisms. The first is a *monitoring mechanism*, whereby each node monitors the sending rate of each of its neighbors. The second mechanism is a per-link *penalty mechanism*, which further motivates nodes to adhere to the expected link throughput.

We now review the main results in each chapter.

8.1 Results of Chapter 4, Araneola: A Scalable Reliable Multicast System for Dynamic Environments

In Chapter 4, we have presented Araneola, a scalable reliable multi-point to multi-point application-level multicast system for dynamic environments. We have evaluated Araneola over both a LAN and a WAN, and have shown that Araneola is scalable. The only aspect of Araneola that varies with the number of nodes is message latency, which increases logarithmically with the group size, whereas Araneola's load, reliability, resilience to message loss, resilience to simultaneous node failures, and overhead for handling join and leave events are all independent of the group size. Araneola can deliver messages with high reliability and predictable latency in the presence of sizable message loss rates, simultaneous failures of a certain percentage of the nodes, and high churn. The failure rates that Araneola can withstand depend on a tunable parameter, L. As the failure rate increases beyond its expectation, Araneola's reliability degrades gracefully. We have also shown how to extend Araneola to exploit available bandwidth for communication with nearby nodes. Such an approach substantially reduces the communication costs and message latency without hurting the overlay's robustness to random failures.

Recall that we have set the following design goals for Araneola:

- High reliability – 100% reliability as long as the failure and message loss rates do not exceed certain configurable thresholds, and graceful degradation in the face of increasing failure rates. The reliability should be independent of the number of nodes, i.e., Araneola should withstand a certain failure rate independently of the number of nodes in the system.

- Low latency, increasing at most like $O(\log N)$; the latency should remain low while multiple nodes are joining and leaving (or failing).

- Low constant load on each node, as well as low constant cost for handling joins and failures.

- Quick failure recovery and prompt incorporation of joining nodes.

We now show that all these design goals are met.

8.1.1 High Reliability and Fault-Tolerance

Araneola achieves high reliability and fault-tolerance by constructing a richly-connected overlay and disseminating pertinent information on multiple *disjoint* paths in this overlay. In Section 4.4.2, we studied the fault-tolerance and robustness of the Araneola overlay by removing random subsets of edges and nodes from the overlay graph and analyzing the resulting graphs. This allows us to predict Araneola's reliability and latency in the presence of message loss (in case of edge removals) and node failures (in case of node removals).

Our analysis has shown that Araneola achieves high fault-tolerance to node and link failures (see Figures 4.6(a), 4.7, 4.8(a), and 4.9). This fault-tolerance is *independent* of the number of nodes. Araneola's overlay becomes partitioned only if at least 11% of the nodes or the edges are randomly removed from the overlay graph. Moreover, remarkably, for $\mathrm{L}= 5$, after a random removal of roughly 40% of the edges or the nodes, 99% of the remaining nodes are still connected to each other, and only 1% of the remaining nodes are partitioned from the rest. Finally, the overlay exhibits graceful degradation: as the failure rate increases, the diameter and average path length increase moderately, while the average number of disjoint paths moderately decreases.

8.1.2 Low Latency with High Churn

In Figure 4.15(a), we have shown that, in static setting, the message latency grow logarithmically with the number of nodes. Moreover, the message latency does not increase with the churn rate. As Figure 4.17 shows, in dynamic settings, each multicast message was received by 100% of the nodes that were up during its transmission, and messages were delivered with *the same latency as in static runs*. That is, Araneola provides an undisrupted service to nodes that are up despite high churn rates exceeding the ones measured over the Internet and over the Mbone.

8.1.3 Low Constant Load on Each Node, as Well as Low Constant Cost for Handling Joins and Failures

Each Araneola's node communicates only with either L or $\mathrm{L}+1$ nodes (its overlay neighbors). Hence, Araneola incurs a *constant* load on each node, regardless of the number of nodes. Araneola also incorporates joining nodes and removes leaving (or failing) ones with a low *constant* overhead thanks to the use of an unstructured overlay network. In Section 4.3.3, we calculated the join and leave overheads for the simple case where a single join or leave, respectively, occurs when the system is stable, i.e., each node's degree is either L or $\mathrm{L}+1$, and no two neighboring nodes have a degree of $\mathrm{L}+1$. We have shown that these overheads are small and independent of the number of nodes. We have also verified our analysis through dynamic experiments, in which nodes join

and leave the overlay. Our empirical results, which are close to our analysis, have shown that each join or leave operation incurs the sending of only about $3k$ messages in total, independent of the number of nodes.

8.1.4 Quick Failure Recovery and Prompt Incorporation of Joining Nodes

Araneola quickly incorporates joining nodes and removing leaving (failing) ones thanks to the use of an unstructured overlay network, in which nodes join the overlay according to some loose constraints. In Section 4.7.2, we evaluated how fast Araneola allows joining nodes to begin receiving messages reliably. Our measurements have shown that a joining node not only receives all the messages sent after its creation, but actually receives 100% of the messages sent up to 6 rounds before its join.

8.2 Results of Chapter 5, EquiCast: Scalable Multicast with Selfish Users

In Chapter 5, we have introduced EquiCast, a P2P multicast protocol for selfish environments. We treated the problem of freeloading from a game theoretic perspective, and modeled the system as a *non-cooperative game*. In such a game, nodes are selfish but *rational*, i.e., each user chooses its own *strategy* regarding its level of cooperation so as to minimize its own cost [46]. More specifically, the goal of each node is to receive all the multicast packets while minimizing its sending rate.

We defined a special set of *protocol-obedient strategies (POSs)*. Generally speaking, a strategy out of this set allows a node to determine how many connections to maintain and how many packets to send on each connection though it does not allow users to hack the protocol's code or assume that others do so. In Theorem 1, we have proved that, in EquiCast, if all nodes choose strongly dominating strategies out of the set of POSs, then every node exclusively maintains connections with its initial k neighbors throughout the multicast session, and it receives all the multicast packets. In this case, EquiCast incurs a constant load on each node, and hence it can support large groups of users.

In Theorem 2, we have proved that if all the nodes, except for one (rational) node n, choose a strategy out of the set of possible POSs, then n also chooses a POS.

In Theorem 3, we have proved that if all of a node's n's initial k neighbors are rational and choose POSs and n cannot locate an identity of a node that does not choose a POS, then n exclusively maintains connections with its initial k neighbors throughout the multicast session, and it receives all the multicast packets. That is, unilateral hacking of the protocol's code cannot reduce

a node's cost.

Finally, we have described a dynamic version of EquiCast, which supports node joins and leaves. We are unaware of any previous P2P multicast protocol that has been shown to enforce cooperation in environments in which all the nodes are selfish.

8.3 Results of Chapter 6, Octopus: A Fault-Tolerant and Efficient Ad-hoc Routing Protocol

In Chapter 6, we have presented Octopus, a simple, fault-tolerant, and efficient routing protocol for large MANETs, which supports movement of the area in which nodes are located. We have proven Octopus's scalability: in Octopus, as opposed to other ad-hoc routing protocols, e.g., [63, 83], the number of location update packets does not increase with the network size. The number of bytes in such packets grows like $O(\sqrt{N})$ with the number of nodes N (and the network size). Empirically, this constitutes a smaller increase in the overhead than exhibited by previous protocols, e.g., [63, 83].

We have conducted thorough empirical evaluation of Octopus using the ns2 simulator with up to 675 mobile nodes. Our extensive simulations have shown Octopus to be scalable, efficient, and have illustrated Octopus's perfect fault-tolerance: in a large grid with hundreds of nodes that intermittently disconnect and reconnect, Octopus achieves the same high reliability as when all nodes are constantly up. This impressive fault-tolerance is achieved thanks to the high level of redundancy in Octopus, and the freshness of the redundant information. At the same time, Octopus incurs less overhead than previous efficient position-based routing protocols. This is achieved thanks to the use of synchronized aggregation. While we employed aggregation only in the context of location discovery, we believe that similar aggregation can be used to improve the fault-tolerance of various additional protocols and to reduce their overhead, e.g., by aggregating queries or information about various searchable resources in resource location services [7].

We have also introduced a recovery technique that overcomes forwarding failures by using information stored at the location servers. We have shown that the basic geographic forwarding protocol combined with this recovery technique achieves similar reliability to two-hop geographic forwarding, while incurring substantially less overhead.

8.4 Results of Chapter 7, Evaluating Unstructured P2P Lookup Overlays

In Chapter 7, we have defined metrics for evaluating unstructured overlays for P2P lookup systems. These metrics capture the search dependability and efficiency, the granularity at which one can control the tradeoff between the two, and also the fairness. According to these metrics, we have evaluated different graphs and overlays, including a Gnutella graph, a power law random graph, normal random graphs, a 3-regular random graph, and a 3-Araneola overlay. Our results have shown that a 3-regular random graph and a 3-Araneola overlay achieve the best results in term of all four metrics. Moreover, using such overlays eliminates the main drawback due to which unstructured overlays were abandoned, namely the search inefficiency. In fact, with such overlays, one can reach up to 20% of the nodes with almost perfect search efficiency.

As opposed to a 3-regular random graph, a 3-Araneola overlay supports dynamic user behavior. In such an overlay, each single join or leave operation is handled locally, and incurs the sending of only 9 messages on average (or $O(\log N)$ messages in the absence of a membership service). Therefore, we conclude that a 3-Araneola overlay is an excellent solution for a flooding-based P2P lookup system.

Bibliography

[1] `http://mathworld.wolfram.com/CircularSegment.html`.

[2] EMULE-PROJECT.NET. eMule site. http://www.emule-project.net/.

[3] Gnutella. http://gnutella.wego.com.

[4] Grid modules for ns2. `http://www.pdos.lcs.mit.edu/grid`

[5] Microsoft Combat Flight Simulator 3. http://www.microsoft.com/games/combatfs3/.

[6] The network simulator - ns-2. www.isi.edu/nsnam/ns/.

[7] I. Abraham, D. Dolev, and D. Malkhi. Lls: a locality aware location service for mobile ad hoc networks. In *DIALM-POMC '04: Proceedings of the 2004 joint workshop on Foundations of mobile computing*, pages 75–84, New York, NY, USA, 2004. ACM Press.

[8] I. Abraham, D. Malkhi, and O. Dobzinski. Land: stretch (1 + epsilon) locality-aware networks for dhts. In *SODA '04: Proceedings of the fifteenth annual ACM-SIAM symposium on Discrete algorithms*, pages 550–559, Philadelphia, PA, USA, 2004. Society for Industrial and Applied Mathematics.

[9] E. Adar and B. A. Huberman. Free riding on gnutella. *First Monday*, Sept. 2000.

[10] A. S. Aiyer, L. Alvisi, A. Clement, M. Dahlin, J.-P. Martin, and C. Porth. Bar fault tolerance for cooperative services. In *SOSP '05: Proceedings of the twentieth ACM symposium on Operating systems principles*, pages 45–58, New York, NY, USA, 2005. ACM Press.

[11] K. C. Almeroth and M. H. Ammar. Characterization of mbone session dynamics: Developing and applying a measurement tool. Technical Report GIT-CC-95/22, Georgia Institute of Technology, June 1995.

[12] K. C. Almeroth and M. H. Ammar. Collecting and modeling the join/leave behavior of multicast group members in the MBone. In *HPDC*, pages 209–216, 1996.

[13] K. C. Almeroth and M. H. Ammar. Multicast group behaviour in the Internet's Multicast Backbone (MBone). *IEEE Communication Magazine*, June 1997.

[14] I. Aydin and C. C. Shen. Facilitating match-making service in ad hoc and sensor networks using pseudo quorum. In *Proceedings of 11th International Conference on Computer Communications and Networks (ICCCN 2002)*, pages 4–9, 2002.

[15] G. Badishi, I. Keidar, and A. Sasson. Exposing and eliminating vulnerabilities to denial of service attacks in secure gossip-based multicast. *IEEE Transactions on Dependable and Secure Computing (TDSC)*, 3:1, March 2006.

[16] H. Balakrishnan, M. F. Kaashoek, D. Karger, R. Morris, and I. Stoica. Looking up data in p2p systems. *Communications of the ACM*, 46(2):43–48, February 2003.

[17] S. Banerjee, B. Bhattacharjee, and C. Kommareddy. Scalable application layer multicast. *SIGCOMM Comput. Commun. Rev.*, 32(4):205–217, 2002.

[18] S. Banerjee, S. Lee, B. Bhattacharjee, and A. Srinivasan. Resilient multicast using overlays. *SIGMETRICS Perform. Eval. Rev.*, 31(1):102–113, 2003.

[19] S. Basagni, I. Chlamtac, V. R. Syrotiuk, and B. A. Woodward. A distance routing effect algorithm for mobility (DREAM). In *ACM/IEEE MobiCom*, pages 76–84, 1998.

[20] C. Basile, M.-O. Killijian, and D. Powell. A survey of dependability issues in mobile wireless networks. Technical report, LAAS CNRS, France, February 2003.

[21] E. Bender and R. Canfield. The asymptotic number of labeled graphs with given degree sequences, *J. Combinatorial Theory Ser.* A 24 (1978), no. 3, 296–307.

[22] R. Bhagwan, S. Savage, and G. Voelker. Understanding availability. In *2nd International Workshop on Peer-to-Peer Systems (IPTPS '03)*, pages 256–267, 2003.

[23] K. P. Birman. The process group approach to reliable distributed computing. *Communications of the ACM*, 36(12):36–53, December 1993.

[24] K. P. Birman, M. Hayden, O. Ozkasap, Z. Xiao, M. Budiu, and Y. Minsky. Bimodal multicast. *ACM Trans. Comput. Syst.*, 17(2):41–88, 1999.

[25] A. Blanc, Y.-K. Liu, and A. Vahdat. Designing incentives for peer-to-peer routing. In *Proceedings of the IEEE Infocom Conference*, pages 374–385, 2005.

[26] B. Bollobas. Random Graphs, Cambridge University Press.

[27] B. Bollobas. A probabilistic proofof an asymptotic formula for the number of labelled regular graphs, *European J. Combin.* 1 (1980), no. 4, 311316.

[28] c Net News. Napster among fastest-growing net technologies. October 2003. http://news.com.com/2100-1023-246648.html.

[29] A. Carzaniga, D. S. Rosenblum, and A. L. Wolf. Design and evaluation of a wide-area event notification service. *ACM Transactions on Computer Systems*, 19(3):332–383, 2001.

[30] M. Castro, P. Druschel, A.-M. Kermarrec, A. Nandi, A. Rowstron, and A. Singh. Splitstream: High-bandwidth multicast in a cooperative environment. In *ACM SIGOPS Symposium on Operating Systems Principles (SOSP)*, pages 298–313, October 2003.

[31] M. Castro, P. Druschel, A.-M. Kermarrec, and A. Rowstron. SCRIBE: a large-scale and decentralized application-level multicast infrastructure. *IEEE J. Selected Areas in Comm. (JSAC)*, 20(8):1489–1499, 2002.

[32] M. Castro, M. B. Jones, A.-M. Kermarrec, A. Rowstron, M. Theimer, H. Wang, and A. Wolman. An evaluation of scalable application-level multicast built using peer-to-peer overlays. In *IEEE INFOCOM*, pages 1510–1520, April 2003.

[33] Y. Chawathe. Scattercast: An adaptable broadcast distribution framework. *Special issue of the ACM Multimedia Systems Journal on Multimedia Distribution*, 9(1):104–118, 2003.

[34] Y. Chawathe, S. Ratnasamy, L. Breslau, N. Lanham, and S. Shenker. Making gnutella-like p2p systems scalable. In *ACM SIGCOMM*, pages 407–418, August 2003.

[35] B. Chen, K. Jamieson, H. Balakrishnan, and R. Morris. Span: An energy-efficient coordination algorithm for topology maintenance in ad hoc wireless networks. In *Proceedings of the 7th ACM International Conference on Mobile Computing and Networking*, pages 85–96, Rome, Italy, July 2001.

[36] I. Clarke, O. Sandberg, B. Wiley, and T. W. Hong. Freenet: A distributed anonymous information storage and retrieval system. *Lecture Notes in Computer Science*, 2009:46+, 2001.

[37] B. Cohen. Incentives build robustness in BitTorrent. In *1st Workshop on the Economics of Peer-to-Peer Systems*, pages 251–260, 2003.

[38] L. Cox and B. Noble. Samsara: Honor among thieves in peer-to-peer storage. In *ACM SIGOPS Symposium on Operating Systems Principles (SOSP)*, pages 120–132, 2003.

[39] A. Demers, D. Greene, C. Hauser, W. Irish, J. Larson, Shenker, Stuygis, D. Swinehart, and D. Terry. Epidemic algorithms for replicated database maintenance. In *6th ACM Symposium on Principles of Distributed Computing (PODC)*, pages 1–12, 1987.

[40] D. S. E. Multicast routing in a datagram internetwork. PhD thesis, Stanford University, December 1991.

[41] P. T. Eugster, R. Guerraoui, S. B. Handurukande, P. Kouznetsov, and A.-M. Kermarrec. Lightweight probabilistic broadcast. *ACM Trans. Comput. Syst.*, 21(4):341–374, 2003.

[42] J. Feigenbaum, C. H. Papadimitriou, and S. Shenker. Sharing the cost of multicast transmissions. *Journal of Computer and System Sciences*, 63(1):21–41, 2001.

[43] S. Floyd, M. Handley, J. Padhye, and J. Widmer. Equation-based congestion control for unicast applications. In *SIGCOMM 2000*, pages 43–56, Stockholm, Sweden, August 2000.

[44] P. Francis. Yoid: Extending the internet multicast architecture. Technical Report, ICSI Center for Internet Research, 2000.

[45] J. Friedman. On the second eigenvalue and random walks in random d-regular graphs. Combinatorica, vol. 11, pp. 331-362, 1991.

[46] D. Fudenberg and J. Tirole. *Game Theory*. The MIT Press, 1991.

[47] J. Gemmell, J. Leibeherr, and D. Bassett. In search of an api for scalable reliable multicast. TR MSR-TR-97-17, Jun 1997.

[48] S. Giordano and M. Hamdi. Mobility management: The virtual home region. Technical Report SSC/1999/037, EPFL, Lausanne, Switzerland, 1999.

[49] C. Gkantsidis and P. R. Rodriguez. Network coding for large scale content distribution. In *Proceedings of the IEEE Infocom Conference*, pages 2235–2245, 2005.

[50] A. Goerdt. The giant component threshold for random regular graphs with edge faults. *Theoretical Comput. Sci.*, 259(1-2):307–321, 2001.

[51] K. Gummadi, R. Dunn, S. Saroiu, S. Gribble, H. Levy, and J. Zahorjan. Measurement, modeling, and analysis of a peer-to-peer file-sharing workload. In *SOSP*, pages 314–329, 2003.

[52] I. Gupta, A.-M. Kermarrec, and A. J. Ganesh. Efficient epidemic-style protocols for reliable and scalable multicast. In *21st IEEE International Symposium on Reliable Distributed Systems (SRDS)*, pages 180–189, October 2002.

[53] Y. h. Chu, S. G. Rao, S. Seshan, and H. Zhang. Enabling conferencing applications on the internet using an overlay multicast architecture. In *ACM SIGCOMM*, pages 55–67, August 2001.

[54] Y. h. Chu, S. G. Rao, S. Seshan, and H. Zhang. A case for end system multicast. *IEEE Journal on Selected Areas in Communication (JSAC), Special Issue on Networking Support for Multicast*, 20(8), 2002.

[55] Z. J. Haas, J. Y. Halpern, and L. Li. Gossip-based ad hoc routing. In *IEEE INFOCOM 2002*, pages 1707–1716, 2002.

[56] Z. J. Haas and B. Liang. Ad hoc mobility management with uniform quorum systems. IEEE/ACM Trans. on Networking, vol. 7, no. 2, pp. 228–240, Apr 1999.

[57] Z. J. Haas and M. R. Pearlman. The performance of query control schemes for the zone routing protocol. In *SIGCOMM*, pages 167–177, 1998.

[58] A. Habib and J. Chuang. Incentive mechanism for peer-to-peer media streaming. In *International Workshop on Quality of Service (IWQoS '04)*, pages 171–180, 2004.

[59] D. Hales and S. Patarin. How to cheat bittorrent and why nobody does. TR UBLCS-2005-12, Department of Computer Science University of Bologna, May 2005.

[60] F. Harary. The maximum connectivity of a graph. *The National Academy of Science*, 48:1142–1146, 1962.

[61] D. A. Helder and S. Jamin. End-host multicast communication using switchtrees protocols. In *Global and Peer-to-Peer Computing on Large Scale Distributed Systems*, 2002.

[62] M. Hofmann, T. Braun, and G. Carle. Multicast communication in large scale networks. In *Proceedings of Third IEEE Workshop on High Performance Communication Subsystems (HPCS), Mystic, Connecticut*, pages 147–150, Aug. 1995.

[63] P. H. Hsiao. Geographical region summary service for geographical routing. Mobile Computing and Communications Review, vol. 5, no. 4, 2001.

[64] ICFA-SCIC Monitoring WG. January 2003 Report of the ICFA-SCIC Monitoring Working Group. http://www.slac.stanford.edu/xorg/icfa/icfa-net-paper-dec02/.

[65] Instiute for Simulation and Training. Standard for distributed interactive simulation - application protocols. TR IST-CR-94-50, University of Central Florida, Orlando, 1994.

[66] V. Jacobson and S. McCanne. Using the LBL network whiteboard. Lawrence Berkeley Laboratory, University of California, Berkeley, 1994.

[67] K. Jain, L. Lovasz, and P. A. Chou. Building scalable and robust peer-to-peer overlay networks for broadcasting using network coding. In *podc*, pages 51–59, July 2005.

[68] J. Jannotti, D. K. Gifford, K. L. Johnson, M. F. Kaashoek, and H. W. O. Jr. Overcast: reliable multicasting with an overlay network. In *Symp. Operating Systems Design and Implementation (OSDI)*, pages 197–212, 2000.

[69] D. Johnson. Routing in ad hoc networks of mobile hosts. In *Workshop on Mobile Computing Systems and Applications*, pages 158–163, Santa Cruz, CA, U.S., 1994.

[70] F. Kaashoek and D. Karger. Koorde: A simple degree-optimal hash table. In *2nd Intl. Workshop on Peer-to-Peer Systems (IPTPS)*, pages 98–107, 2003.

[71] T. Karagiannis, P. Rodriguez, and D. Papagiannaki. Should isps fear peer-assisted content distribution? In *ACM USENIX IMC*, 2005.

[72] R. Karp, C. Schindelhauer, S. Shenker, and B. Vöcking. Randomized rumor spreading. In *IEEE Symposium on Foundations of Computer Science*, pages 565–574, 2000.

[73] I. Keidar and R. Melamed. Evaluating Unstructured Peer-to-Peer Lookup Overlays. In *ACM Symposium on Applied Computing (SAC 2006), Dependable and Adaptive Distributed Systems (DADS) Track*, pages 675–679, 2006.

[74] I. Keidar, R. Melamed, and A. Orda. EquiCast: Scalable Multicast with Selfish Users. In *25th ACM Symposium on Principles of Distributed Computing (PODC)*, 2006.

[75] D. Kempe, J. Kleinberg, and A. Demers. Spatial gossip and resource location protocols. In *33rd ACM Symp. on Theory of Computing (STOC)*, pages 163–172, 2001.

[76] A.-M. Kermarrec, L. Massouli, and A. J. Ganesh. Probabilistic reliable dissemination in large-scale systems. *IEEE Transactions on Parallel and Distributed Systems*, 14(3):248–258, March 2003.

[77] J. H. Kim and V. H. Vu. Generating random regular graphs. In *Proceedings of the thirty-fifth ACM symposium on Theory of computing*, pages 213–222. ACM Press, 2003.

[78] M. Kim and M. Medard. Robustness in large-scale random networks. In *Proceedings of the IEEE Infocom Conference*, 2004.

[79] J. Kleinberg. The small-world phenomenon: An algorithmic perspective. In *32nd ACM Symp. on Theory of Computing (STOC)*, pages 163–170, 2000.

[80] Y.-B. Ko and N. H. Vaidya. Location-aided routing (LAR) in mobile ad hoc networks. In *Mobile Computing and Networking*, pages 66–75, 1998.

[81] D. Kostic, A. Rodriguez, J. Albrecht, and A. Vahdat. Bullet: High bandwidth data dissemination using an overlay mesh. In *ACM SIGOPS Symposium on Operating Systems Principles (SOSP)*, pages 282–297, October 2003.

[82] C. Law and K. Siu. Distributed construction of random expander networks. In *IEEE Infocom*, pages 2133–2143, 2003.

[83] J. Li, J. Jannotti, D. De Couto, D. Karger, and R. Morris. A scalable location service for geographic ad-hoc routing. In *Proceedings of the 6th ACM International Conference on Mobile Computing and Networking (MobiCom '00)*, pages 120–130, Aug. 2000.

[84] Q. Li and D. Rus. Communication in disconnected ad hoc networks using message relay. *Parallel Distrib. Comput.*, 63:75–86, 2003.

[85] M. J. Lin, K. Marzullo, and S. Masini. Gossip versus deterministically constrained flooding on small networks. In *14th International Symposium on DIStributed Computing (DISC)*, pages 253–267, 2000.

[86] D. Loguinov, A. Kumar, V. Rai, and S. Ganesh. Graph-theoretic analysis of structured peer-to-peer systems: Routing distances and fault resilience. In *In Proceedings of the ACM SIGCOMM '03 Conference*, pages 395–406, 2003.

[87] Q. Lv, P. Cao, E. Cohen, K. Li, and S. Shenker. Search and replication in unstructured peer-to-peer networks. In *Proceedings of the 16th international conference on Supercomputing*, pages 84–95. ACM Press, 2002.

[88] D. Malkhi. Locality-aware network solutions (a survey). TR 2004-6, School of Computer Science and Engineering, The Hebrew University, 2004.

[89] D. Malkhi, M. Naor, and D. Ratajczak. Viceroy: A scalable and dynamic emulation of the butterfly. In *ACM Symposium on Principles of Distributed Computing (PODC)*, pages 183–192, July 2002.

[90] L. Massoulie, A.-M. Kermarrec, and A. J. Ganesh. Network awareness and failure resilience in self-organising overlay networks. In *22nd IEEE International Symposium on Reliable Distributed Systems (SRDS)*, pages 47–55, October 2003.

[91] M. Mauve, J. Widmer, and H. Hartenstein. A survey on position-based routing in mobile ad hoc networks. *IEEE Network Magazine*, 15(6):30–39, November 2001.

[92] B. D. McKay and N. C. Wormald. Uniform generation of random regular graphs of moderate degree. *Journal of Algorithms*, 11:52–67, 1990.

[93] R. Melamed and I. Keidar. Araneola: A Scalable Reliable Multicast System for Dynamic Environments. In *3rd IEEE International Symposium on Network Computing and Applications (IEEE NCA)*, pages 5–14, 2004.

[94] R. Melamed, I. Keidar, and Y. Barel. Octopus: A Fault-Tolerant and Efficient Ad-hoc Routing Protocol. In *24th IEEE International Symposium on Reliable Distributed Systems (SRDS)*, pages 39–49, 2005.

[95] B. Nath and D. Niculescu. Routing on a curve. In *HotNets-I, Princeton, NJ*, pages 155–160, 2002.

[96] T.-W. J. Ngan, D. S. Wallach, and P. Druschel. Incentives-compatible peer-to-peer multicast. In *2nd Workshop on the Economics of Peer-to-Peer Systems*, 2004.

[97] V. D. Park and M. S. Corson. A highly adaptive distributed routing algorithm for mobile wireless networks. In *INFOCOM (3)*, pages 1405–1413, 1997.

[98] V. Paxson. End-to-end Internet packet dynamics. In *ACM SIGCOMM*, pages 277–292, September 1997.

[99] D. Pendarakis, S. Shi, D. Verma, and M. Waldvogel. ALMI: An application level multicast infrastructure. In *Proceedings of the 3rd USENIX Symposium on Internet Technologies and Systems (USITS)*, pages 49–60, 2001.

[100] C. Perkins and P. Bhagwat. Highly dynamic destination-sequenced distance-vector routing (DSDV) for mobile computers. In *ACM SIGCOMM'94 Conference on Communications Architectures, Protocols and Applications*, pages 234–244, 1994.

[101] C. Perkins, E. Royer, and S. R. Das. Ad hoc on demand distance vector (aodv) routing. internet draft (work in progress), internet engineering task force, october 1999. `www.ietf.org/internet-drafts/ draft-ietf-manet-aodv-04.txt`.

[102] L. Peterson, T. Anderson, D. Culler, and T. Roscoe. A blueprint for introducing disruptive technology into the internet. In Proceedings of ACM HotNets-I, October 2002.

[103] B. Quinn and K. Almeroth. IP Multicast Applications: Challenges and Solutions. RFC 3170, September 2001. Network Working Group.

[104] S. Ratnasamy, P. Francis, M. Handley, R. Karp, and S. Shenker. A scalable content addressable network. In *Proceedings of ACM SIGCOMM 2001*, pages 161–172, 2001.

[105] S. Ratnasamy, M. Handley, R. Karp, and S. Shenker. Application-level multicast using content-addressable networks. In *3rd International Workshop on Networked Group Communication (NGC)*, pages 14–29, November 2001.

[106] M. Ripeanu, I. Foster, and A. Iamnitchi. Mapping the gnutella network: Properties of large-scale peer-to-peer systems and implications for system design. *IEEE Internet Computing Journal*, 6(1), 2002.

[107] P. Rodriguez, S.-M. Tan, and C. Gkantsidis. On the feasibility of commercial, legal p2p content distribution. In *ACM/SIGCOMM CCR*, pages 75–78, 2006.

[108] A. Rowstron and P. Druschel. Pastry: Scalable, decentralized object location, and routing for large-scale peer-to-peer systems. *Lecture Notes in Computer Science*, 2218:329–350, 2001.

[109] S. Saroiu, K. Gummadi, and S. Gribble. A measurement study of peer-to-peer file sharing systems. In *Multimedia Computing and Networking*, pages 156–170, January 2002.

[110] K. Shen. Structure management for scalable overlay service construction. In *the First USENIX/ACM Symposium on Networked Systems Design and Implementation (NSDI'04), San Francisco CA*, pages 281–294, March 2004.

[111] R. Sherwood, R. Braud, and B. Bhattacharjee. Slurpie: A cooperative bulk data transfer protocol. In *Proceedings of IEEE INFOCOM*, pages 941–951, 2004.

[112] A. C. Snoeren, K. Conley, and D. K. Gifford. Mesh based content routing using XML. In *18th ACM Symposium on Operating Systems Principles (SOSP '01)*, pages 160–173, 2001.

[113] A. Steger and N. Wormald. Generating random regular graphs quickly. *Combinatorics, Probab. and Comput*, 8:377–396, 1999.

[114] I. Stoica, D. Adkins, S. Zhuang, S. Shenker, and S. Surana. Internet indirection infrastructure. In *SIGCOMM*, pages 19–23, August 2002.

[115] I. Stoica, R. Morris, D. Karger, F. Kaashoek, and H. Balakrishnan. Chord: A scalable Peer-To-Peer lookup service for internet applications. In *Proceedings of the 2001 ACM SIGCOMM Conference*, pages 149–160, 2001.

[116] I. Stoica, R. Morris, D. Liben-Nowell, D. Karger, M. F. Kaashoek, F. Dabek, and H. Balakrishnan. Chord: A scalable peer-to-peer lookup service for internet applications. *IEEE Transactions on Networking*, 11, February 2003.

[117] I. Stojmenovic. Home agent based location update and destination search schemes in ad hoc wireless networks. Technical Report TR-99-10, Computer science, SITE, University of Ottawa, 1999.

[118] I. Stojmenovic and P. Pena. A scalable quorum based location update scheme for routing in ad hoc wireless networks. TR 99-09, SITE, University of Ottawa, 1999.

[119] J. Tchakarov and N. Vaidya. Efficient Content Location in Wireless Ad Hoc Networks. In *Proceedings of IEEE International Conference on Mobile Data Management (MDM)*, page 74, January 2004.

[120] P. Tsuchiya. The Landmark Hierarchy : A New Hierarchy for Routing in Very Large Networks. In *ACM Sigcomm*, pages 35–42, 1998.

[121] B. White, J. Lepreau, L. Stoller, R. Ricci, S. Guruprasad, M. Newbold, M. Hibler, C. Barb, and A. Joglekar. An integrated experimental environment for distributed systems and networks. In *Symp. Operating Systems Design and Implementation (OSDI)*, pages 255–270, Boston, MA, Dec. 2002.

[122] N. Wormald. Models of random regular graphs. *Surveys in Combinatorics*, 276:239–298, 1999.

[123] Y. Xu, J. S. Heidemann, and D. Estrin. Geography-informed energy conservation for ad hoc routing. In *Mobile Computing and Networking*, pages 70–84, 2001.

[124] B. Y. Zhao, L. Huang, J. Stribling, S. C. Rhea, A. D. Joseph, and J. D. Kubiatowicz. Tapestry: A global-scale overlay for rapid service deployment. *IEEE Journal on Selected Areas in Communications*, 22(1):41–53, 2003. Special Issue on Service Overlay Networks.

[125] S. Zhuang, B. Zhao, A. Joseph, R. Katz, and J. Kubiatowicz. Bayeux: An architecture for scalable and fault tolerant wide-area data dissemination. In *11th International Workshop Network and Operating System Support for Digital Audio and Video (NOSSDAV)*, pages 11–20, June 2001.