A Few Randomized Protocols in Peer-to-Peer Networks

By:

Richard TANAKA

A thesis submitted to the School of Graduate and Postdoctoral Studies in partial fulfillment of the requirements for the degree of

Master of Science in Computer Science

FACULTY OF BUSINESS AND INFORMATION TECHNOLOGY

ONTARIO TECH UNIVERSITY

OSHAWA, ONTARIO, CANADA

August, 2019

THESIS EXAMINATION INFORMATION

Submitted by: **Richard Tanaka**

Master of Science in **Computer Science**

Thesis title:
A Few Randomized Protocols in Peer-to-Peer Networks

An oral defense of this thesis took place on August 14, 2019 in front of the following examining committee:

Examining Committee:

Chair of Examining Committee	Patrick Hung
Research Supervisor	Ying Zhu
Examining Committee Member	Jing Ren
Thesis Examiner	Ying Wang, Ontario Tech University

The above committee determined that the thesis is acceptable in form and content and that a satisfactory knowledge of the field covered by the thesis was demonstrated by the candidate during an oral examination. A signed copy of the Certificate of Approval is available from the School of Graduate and Postdoctoral Studies.

ONTARIO TECH UNIVERSITY

Abstract

Faculty of Business and Information Technology

Master of Science in Computer Science

A Few Randomized Protocols in Peer-to-Peer Networks

by Richard TANAKA

This thesis studies a few randomized algorithms in application-layer peer-to-peer networks. The significant gain in scalability and resilience that peer-to-peer networks provide has made them widely used and adopted in many real-world distributed systems and applications. The unique properties of peer-to-peer networks make them particularly suitable for randomized algorithms such as random walks and gossip algorithms. We study these by developing implementations based on the Docker virtual container technology. We can thus analyze their behaviour and performance in realistic settings. We further consider the problem of identifying high-risk bottleneck links in the network with the objective of improving network reliability. We propose a randomized algorithm to solve this problem and evaluate its performance by simulations.

Declaration of Authorship

I hereby declare that this thesis consists of original work of which I have authored. This is a true copy of the thesis, including any required final revisions, as accepted by my examiners.

I authorize the University of Ontario Institute of Technology to lend this thesis to other institutions or individuals for the purpose of scholarly research.

I further authorize University of Ontario Institute of Technology to reproduce this thesis by photocopying or by other means, in total or in part, at the request of other institutions or individuals for the purpose of scholarly research. I understand that my thesis will be made electronically available to the public.

Signed:

_____

Date:

_____

Richard TANAKA

STATEMENT OF CONTRIBUTIONS

I hereby certify that I am the sole author of this thesis and that no part of this thesis has been published or submitted for publication. I have used standard referencing practices to acknowledge ideas, research techniques, or other materials that belong to others. Furthermore, I hereby certify that I am the sole source of the creative works and/or inventive knowledge described in this thesis.

Acknowledgements

To Professor Ying Zhu, my research supervisor, for all the work, patience, and advice in getting me through this process.

To my parents, for all your support during this endeavour.

Contents

List of Figures

Chapter 1

Introduction

1.1 Motivation

Peer-to-peer networks are an extremely important and widely-used type of distributed system. Many distributed applications are built on top of an infrastructure of peer-to-peer networks. A peer-to-peer network is an application-layer network of end systems interconnected through TCP or UDP connections over the underlying IP network. The nodes in the network are end systems and the links between pairs of nodes are application-layer connections (each of which consists of a path in the underlying IP network). The networks are usually dynamic and autonomous, with nodes joining and leaving freely. When a new node joins, it needs to establish connections to a number of existing nodes in the network, and these are called its neighbours. Even though it is theoretically possible, due to concerns of efficiency and scalability, each node is not connected to every other node in the network, i.e., every node has all other nodes as neighbours with the network topology forming a complete graph. Perhaps the most salient property of peer-to-peer networks is that every node is completely equal to every other node, with all the nodes operating in the same way and executing the same protocols, hence the name. This is critically important to achieve the level of scalability that peer-to-peer networks are well-known for, because every node is carefully and deliberately designed to contribute to the overall services offered by the network and this

means the large the number of nodes, the more resources are available for the equally large number of resource consumers (which the nodes are as well). A crucial feature is that the network would operate correctly without the aid of any kind of centrally administered servers. The highly scalable nature of peer-to-peer networks, especially compared with the traditional client-server paradigm in distributed applications, makes them particularly attractive and useful, and therefore they are widely used in many diverse distributed systems.

Peer-to-peer networks first emerged to support very large scale file sharing systems [11], then their scalability motivated research efforts towards peer-to-peer middleware platforms which essentially provide the fundamental service of routing requests to specific nodes in the network holding the required data resources. A few well-known examples of such platforms are Pastry, Tapestry and Chord [37, 42, 39]. They have been fully developed and deployed in applications, demonstrating their strengths in the ability to service very large number of clients and to be self-organizing without incurring overhead that is significant relative to the number of nodes in the system. The type of services provided by these platforms is generic and not application-specific, and can be used by distributed systems in a wide variety of contexts.

Randomized algorithms and protocols play an important role in the operations of peer-to-peer networks. Random walks — a type of randomized algorithm where messages are sent to neighbours that are randomly selected at each step — have been used for the core functionality of searching and topology maintenance [32, 31]. The peer nature also motivates the design of distributed algorithms in which nodes only send and receive messages with their (few) neighbours in every iteration of the algorithm, with the objective of cooperating and coordinating together to implement certain computation or service. To increase the efficiency of the computation and communication,

a class of randomized algorithms called gossip algorithms have been extensively studied and proposed for this purpose [25, 27, 26].

We study random walks and gossip algorithms for distributed computation of global average. In order to fully investigate these important distributed randomized protocols, we implement them in realistic peer-to-peer network topologies and evaluate their performance. Without a large number of end hosts at our disposal, we are able to achieve the closest approximation of implementing virtual containers (using the Docker technology) to serve as nodes and construct networks of them conforming to graph topologies that have the same properties as real-world networks. The strength of our approach is that the implementation of these node processes can be directly deployed on a real network, because the virtual container technology is the same as real end systems with an underlying network stack and operating system.

We further consider the problem of maintaining membership lists in peer-to-peer networks, which is an essential core functionality of network maintenance. We implement two different data structures for storing and exchanging membership lists, using linked list and randomized hash index, and compare their performance.

The dynamic nature of peer-to-peer networks combined with their decentralized property lead to the possibility that node or link failures go undetected (for a while at least) which sometimes cause network partitioning that therefore also go undetected. An undetected network partition proves to be a huge problem for the network, leading to incorrect and inconsistent views of the network on different nodes which may have serious consequences later on. We observe that sometimes the network topology evolves to a state where there are a few links which are special in two ways: they see a high volume of message traffic and there are only a few of them connecting two clusters of nodes in the network so that if they become disconnected, then the network

will either partition or become highly congested in another link that now has to take the extra load. So it is very important to be able to identify these high-risk bottleneck links. Once they are identified and monitored, it is easier and faster to detect network partitions. We study the problem of identifying these high-risk links. We propose a method that involves gossip algorithm and randomized shortest-path computation, to solve this problem. We evaluate our method using simulations and show its effectiveness in identifying the high-risk links.

1.2 Contributions

We have developed implementations of random walks and gossip algorithms in peer-to-peer networks. Our implementations are based on virtual container technology which makes them real testbeds and directly deployable on real end systems. This allows us to investigate these randomized algorithms in the most realistic way possible. Their behaviour and performance are exactly as they would be in a real peer-to-peer network. We have also implemented a bloom filter hash index for storing and exchanging membership lists, which drastically reduces the storage requirement and query time for node membership.

We also studied the important of problem of identifying high-risk bottleneck links that should be monitored in the case of failure which would result in network partitioning. We proposed a randomized decentralized method for identifying such links and implemented simulations to evaluate our method and show its effectiveness.

1.3 Thesis Organization

We begin by presenting the details of and the technologies used in our implementations in Chapter 2. Then our investigation and implementation of random walks is presented in Chapter 3. Chapter 5 contains the study and implementation of gossip algorithms for distributed computation of global averages. The comparison of data structures for membership lists is given in Chapter 4. The problem of identifying high-risk links and the proposed method for solving it are presented in Chapter 6.

Chapter 2

Implementation of distributed protocols in peer-to-peer networks

This chapter describes the tools and technology we used for the implementation of distributed protocols in peer-to-peer networks which will be presented in detail in the remainder of this thesis.

We have implemented three different distributed protocols: random walks, gossip algorithm for estimation of the average, and the dissemination of membership lists. These algorithms all share the properties of being distributed and randomized, which makes it difficult and sometimes infeasible to theoretically analyze them to observe their performance and behaviour. To study and better understand them, we generated random network topologies that follow the same characteristics observed in naturally arising real-world peer-to-peer networks, and we implemented these protocols to execute in these networks. We implement them in the recently invented Go programming language to take advantage of its built-in concurrent programming constructs. The protocols are first implemented as part of simulations. Then we take it one step further, and implement them using the recently emerged Docker container virtualization. This Docker implementation goes beyond simulations, making our implementation completely real and immediately deployable in real networks. In the following, we will present the technologies used in our implementations.

2.1 Go programming language

Go is a programming language created by Google, and is currently an open source project (`https://golang.org`). The motivation for its conception was to provide a language that provides first-class support for currency and parallelism in computers with multicore CPUs; has tractable resource management in large-scale software systems including garbage collection and provides safe automatic memory management. It was envisioned that this new language would ease the development process for network dependent systems such as data centers, web programming models, and distributed systems in general.

Key design principles of Go include:

- Ease of learning for new developers through clear and concise syntax and semantics
- Build time reduction by efficient package dependency management
- Meeting modern computing demands (built-in concurrency, networking, and ease of web application development)

The syntax of Go is inherited from C, enhanced advantageously by adoptions from dynamic typed languages for variable declaration and initialization. Go, like the popular programming language Python, is statically typed and is compiled to native code for execution which allows it to have the performance and safety of languages such as C++ and Java. The language supports pointers, but unlike in C/C++, pointer arithmetic is not permitted. Go automatically manages memory allocation and garbage collection at runtime.

Go is not typically referred to as an object-oriented programming language, however, it does have object-like primitives that are available as the type `structs` and invocable methods that can be defined for them. However,

there is no inheritance and this reduces the overhead required to declare relationships between types. Instead, types satisfy interfaces by implementing a subset of their methods. Unlike object-oriented inheritance which allows only one parent, types can satisfy multiple interfaces at once.

2.1.1 Networking for distributed computing

With Go's support for distributed computing, applications can be implemented with just the core language. For example, the "net" package is included in the standard library and it provides access to low-level networking primitives such as TCP/IP, UDP, Unix domain sockets, and domain name resolution. Many clients will only utilize the basic communication interfaces, such as Listener, Dial, and Conn. Package "net/http" provides client and server interfaces for communication at the application layer, and is designed for simplicity in assigning HTTP routes. Using Go greatly reduces implementation efforts in managing connection threads for incoming requests.

2.1.2 Concurrency

The key feature of Go that makes it suitable for distributed systems modelling is its core support for concurrency. Concurrency is identified as the composition of multiple independently executing processes. Although parallelism may be a related concept, the two are not the same. Parallelism, in contrast, is a simultaneous execution of processes that distribute a large computation into sub units. In a single processor, a program can be concurrent, however it cannot be parallel. In a distributed system, nodes may be handling multiple communication channels during execution, which makes concurrency a fundamental requirement in the design of such a system. In many concurrent programming environments implementing correct synchronization to access shared data can be challenging. Go's approach is passing the

reference of the shared data around, instead of all threads having the reference and coordinating the read and writes. By passing the reference, only one thread will have access to the data at any given time therefore eliminating data-races between threads that use channels. Go's authors have summarized this design paradigm to a slogan, "do not communicate by sharing memory; instead, share memory by communicating". It should be noted that this paradigm is used alongside mutex locks and not replace it. In a trivial reference counter, a mutex around the counter would be the clear solution. Go's concurrency model originates from Hoare's *Communicating Sequential Processes* [24]. The two fundamental constructs for concurrency are goroutines and channels, which are described in the following.

2.1.3 Goroutines

Goroutines are described as lightweight threads that run concurrently with other goroutines and asynchronously with the calling code. An independently executing function may return control to the caller before completion, these functions are known as coroutines [29]. Essentially the concept of goroutines is to multiplex coroutines on to a smaller number of OS threads which results in a N:1 mapping of coroutines and threads. When a system blocking call is made by a coroutine, the rest of the coroutines on the blocked OS thread are automatically moved to another, active, OS thread by Go's runtime which manages the scheduling. By migrating, the rest of the routines will not be blocked. Typically, the number of OS threads is set to the number of cores on the system to maximize CPU utilization. Goroutines have a significantly reduced memory footprint compared to OS threads. When a goroutine is created, Go's runtime allocates a few kilobytes to a resizable and bounded stack, which then grows or shrinks as per the execution demands. With this architecture, goroutines are very lightweight and a Go program can

run hundreds of thousands of goroutines whereas traditional thread-based concurrency would run out of resources. To create a goroutine, the syntax begins with the "go" statement followed by a function call. It is common practice to call anonymous functions, known as a function literal, as goroutines as they will not be called anywhere else in the program. Function literals are closures, which allows them to refer to variables in the surrounding function.

Unlike regular function calls, goroutines are asynchronous in the sense that the main program execution does not wait for the invoked goroutine to reach completion. An invoked goroutine will independently complete the function call and exit. Input parameters are evaluated as regular functions however return values are discarded. Because of this independent execution, the main program may complete without knowing the state of the invoked routine. In order to allow communication and synchronization between goroutines, Go uses channels.

Channels are the mechanism for inter-process communication between concurrently executing goroutines. They are explicitly typed and bidirectional, however they can be used unidirectionally by using a send-only or receive-only directive. Channels are unbuffered by default, which provides synchronization of states between goroutines on opposite ends of the channel. In an unbuffered channel, senders block until a receiver has retrieved the value from the channel before sending additional data. Receivers always block until there is data to be received, regardless of whether the channel is buffered or unbuffered. A buffered channel removes the synchrony as senders can now send data asynchronously until the capacity of the channel has been reached.

2.2 Docker container virtualization

Docker is the industry standard in container technology. Docker containers resemble virtual machines in providing portability of applications regardless of the host machine operating system. However, a container is not a virtual machine, it is much more lightweight. A container is a standardized unit of software that includes everything that is needed to run an application on any host OS: code, any dependencies, and any system tools and libraries and settings required. Any host OS that runs Docker engine will be able to run any Docker containers. Containers do not contain an OS (as in the case of virtual machines), instead, they all share the underlying machine's OS system kernel — this makes them much more lightweight than virtual machines, while still retaining the property of portability.

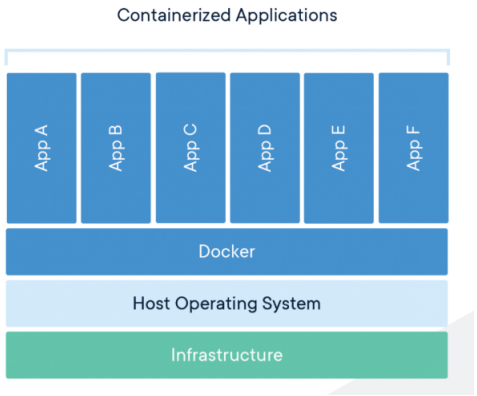

FIGURE 2.1: https://www.docker.com/resources/what-container

There are *images* and *containers*. An image is an application including all the executable binaries and source code and libraries. A container is an *instance* of the image running as a process. You can have many containers from the same image.

After a Docker network has been created and containers have been connected to it, it's important not to rely on IP addresses for inter-container communication. Because IP addresses are assigned dynamically and therefore, as

the containers start and stop, they will change. For containers to find each other, Docker uses DNS name resolution, the default hostname for a container is its container name.

2.3 Implementation program design

The programs we implemented for running experiments and simulations are written in the Go programming language. The concurrency and Go channel features allow for easier implementation of multiple networking tasks which may be occurring at the same time. The Go channels enable coordination between the Go routines in instances where any linear processing is required. To mimic the nature of a set of independent devices, the core of the simulation is independently running node programs. These nodes communicate between each other only through the network stack using the Go language's net library. Each of these nodes represent one vertex in the graph. The only other program being run is a control program used to manage the simulations and gather data back from the graph nodes in the topology.

2.3.1 Node program

Structure

Node programs take in several command line parameters upon launch including the ID, IP address, port to listen on, and cluster (used only in certain experiments). The main program function contains the scope for functions that will be launched as go routines. They are within this scope so those functions will be able to access shared data variables which are controlled by mutual exclusion functions to prevent race conditions or overwriting conditions. Upon starting the main function, the data variables are initialized and a go routine that listens on the assigned port is launched using the net

library. Once an inbound connection is received, a separate listener handler is launched in a go routine to handle all communication during the life of this connection. The listen handler parses the messages and the functions that execute the simulation aspects of each node are contained here. The connections being used are TCP to ensure that all messages are received. The message sizes and overhead from the connection-oriented communication for these experiments are minimal in relation to the available bandwidth. A resource constrained assumption may warrant a non-connection-oriented protocol. Later experiments around the maintenance of membership lists in a dynamic environment also start up some additional independent functions in go routines as some timers are used, or responses are required from external nodes.

Access safe data

Data for each node is local to the main function so the various go routines can all access and update this information. These variables, however, are not equipped to handle concurrent access so they have been wrapped in a few mutual exclusion functions. For the purposes of these experiments, there are no operations which we would consider data intensive that would introduce bottlenecks or impact performance. The data being sent in messages are text based. Messages are terminated by a newline character and connection listeners take in data until the newline terminator is received before attempting to parse. Messages are delimited by the colon character, and the first section always identifies the type/purpose of the message. The remaining sections of the delimited are purpose specific.

2.3.2 Control program

To facilitate all aspects of setting up, initiating, controlling, and terminating the experiments, a separate control program is used. This program has been designed with functions for running each of the simulation experiments. Most simulations incorporate similar core functions. The program takes command line parameters which provide what simulation is being run and the corresponding variables that might be changing in different setups. For example, in the random-walk there are only two parameters, the function and the number of times to run the trials. After initializing variables and go channels, the control program reads the topology from a text file. This information is used to run the node program once for each vertex in the given graph. Each is provided with information such as its ID, IP address, and port to listen on. The control program then establishes a connection with each of the nodes. This connection is not used for any simulation messages as part of the topology. The connection is used to conduct further setup and statistics reporting to the control node depending on the simulation. In the random-walk example, the control node at this point tells each node which other nodes they need to connect to in order to build the specified network topology. Following the random-walk example, the control node will now pick one node at random and instruct it to start the random-walk simulation. When the final node is reached, it uses its connection to the control node to report the final number of hops. The control node then messages each node to reset and repeats this process the specified number of times. The results are output to the console. As with the node program, the control program uses multiple go routines to handle concurrent processing of functions such as each of the listener handles to for each open connection. It also utilizes mutual exclusion protected data variables to prevent race conditions or overwrites since the data types are not currency safe.

2.3.3 Docker implementation

In the case of the docker versions of experiments, there is an additional control program used to complete functions such as launching containers, copying code files (for node and control), compiling the code and finally running each node/control with the appropriate parameters. Since communications between instances of the node programs is conducted exclusively through the network stack, there is no programming difference between running on a single operating system and multiple docker containers. Depending on the operating system, however, there may be a difference when running the node instances and networking through the loopback address. That is, its possible some operating systems may not truly go through all the stages of the network stack once it realizes the message is bound for itself. Using the docker containers forces a true network usage as though there are multiple devices and communications. For the purposes of running these experiments, it was more convenient to copy the code and compile instead of rebuilding a new container image every code change.

2.3.4 Graph topology generation

A separate program is used to generate the starting topologies required. The program takes in the required parameters including topology type, number of nodes and minimum number of connections per node. The finished topology is output to a file which is later used by the control program to setup the simulation. Details of the construction of the different types of topologies used can be found in the corresponding experiments later in this document.

Chapter 3

Random walks in peer-to-peer networks

3.1 Background on random walks

Random walks provide an accurate and faithful way to describe and model the dynamics of many naturally arising systems and networks. Examples include the dispersion of information in social networks [21, 23], the behaviour of stochastic search algorithms [28, 13], and both searching and construction of unstructured peer-to-peer networks [32, 22]. Random walks have many applications in the design and analysis of randomized algorithms in network graphs that model various phenomena.

To define a random walk, we first define an undirected graph with n vertices and m edges, denoted $G = (V, E), |V| = n, |E| = m$, where V is the set of vertices and E is the set of edges. Note that throughout this thesis, we will be using the terms *vertices* and *nodes* interchangeably, as well as *edges* and *links*. A random walk is a stochastic process in which at each step, a vertex u is visited and the next vertex to visit is randomly selected from the set of neighbours of u, i.e., those vertices that are connected to u by an edge in G. Each neighbour of u has an equal probability to be selected as the next one to be visited. We say that the next vertex is uniformly selected at random.

The choice of neighbour is independent of previous choices. This procedure is repeated at every step of the random walk.

The two most salient characteristics of random walks are the uniformly random selection of the next step throughout the walk, and requiring *only* knowledge of immediate neighbours by the vertices and *not* assuming any knowledge of the global topology of the graph. These two characteristics make random walks applicable in many dynamic systems involving network topologies. For example, in a social network where the vertices are people and the edges are connections between people who directly communicate with each other, the process of a piece of information being diffused in this social network can be modeled as some combination of random walks where the information is "walking" from vertex (person) to vertex (person) along edges (person-to-person communication) with an element of randomness that arises out of natural online interpersonal communications. They also allow us to analyze and investigate the underlying structure of networks whose global topology information is not available and only local connections are known. This is one of the most fundamental properties of peer-to-peer networks, therefore it is natural to examine random walks in peer-to-peer networks for analysis and design of distributed algorithms and protocols.

3.2 The Markov chain model

In the theoretical study of random walks, a useful abstraction that is often used is the Markov chain. A Markov chain is a stochastic process defined as follows. There is a set of n states S and a probability transition matrix P which is $n \times n$, with P_{ij} being the probability of transitioning to state j when the current state is i. For all $i, j \in S$, $0 \leq P_{ij} \leq 1$ and $\sum_{i,j} P_{ij} = 1$. Note that

a Markov chain has the *memoryless* property, that is, every state transition is completely independent of all previous state transitions and states.

A random walk can be abstracted to a Markov chain in two possible ways. In the first way, we model the vertices of the graph as the states, and each state transition is a step in the random walk of going from one vertex to one of its neighbours. The transition matrix P can be constructed from the information of the degree d_i of each vertex i: $P_{ij} = 1/d_i$. The second way is to replace each undirected edge with two directed edges and model each directed edge as a state, and the state transition is a traversal of the directed edge during each step of the walk. So the states are (i, j) where i, j are vertices and connected by a directed edge from i to j. The transition matrix P is constructed as $P_{(i,j),(j,k)} = 1/d_j$.

An interesting phenomenon in Markov chains is that there may exist a steady-state behaviour of the Markov chain. Let π be a probability distribution on the states, i.e., π is a row vector whose i-th element π_i is the probability of being in state i. The distribution π called a *stationary distribution* of a Markov chain with probability transition matrix P if $\pi = \pi P$. If the Markov chain finds itself in the stationary distribution at some step t, then it remains in the same stationary distribution at step $t + 1$, and so on and so forth. This is the steady-state of the Markov chain.

A Markov chain is said to be *finite* if it has a finite number of states. It is said to be *irreducible* if the (undirected) graph that induces it is a connected graph. It is also said to be *aperiodic* if its states are aperiodic. In a Markov chain induced by random walks on an undirected graph G, states are aperiodic if the greatest common divisor (gcd) of the length of all closed walks in G is 1 (a closed walk is one where it ends on the same vertex it started on). Generally, in a connected undirected graph G that is not bipartite, there exist closed walks of length 2 as well as closed walks of odd length (i.e., odd cycles), therefore the gcd of all the closed walks is 1. This means that the

Markov chain induced by G is aperiodic.

The most basic theorem on Markov chains (*Fundamental Theorem of Markov Chains*) states that one of the properties for any irreducible, finite and aperiodic Markov chain is that there exists a *unique* stationary distribution π such that, for $1 \leq i \leq n, \pi_i > 0$.

3.3 Cover times of random walks in peer-to-peer networks

In the application of random walks in peer-to-peer networks, a relevant and interesting concept is the *cover time* C_G of the connected undirected graph $G = (V, E)$ for the peer-to-peer network; it is defined as $C_G = \max_{u \in V} C_u$, where C_u is the expected number of steps taken to visit every node in G in the random walk that begins at node u. The theoretical study of the cover time of connected graphs has been extensive.

The classic result from [1] uses the existence of a unique stationary distribution for any irreducible, finite and aperiodic Markov chain, to prove an upper bound for the cover time of any connected graph: $C_G \leq 2m(n-1)$, where $m = |E|, n = |V|$. Tighter upper and lower bounds are derived in [16, 15] for any connected graph. In [9], random regular graphs are studied and the asymptotic limit of cover time in those graphs are shown. These work did not study cover times of random walks in the types of scale-free network graphs that are characteristic of real-world peer-to-peer networks, e.g., [3, 41, 34]. By and large, the body of theoretical results on random walks is focused on regular graphs and conventional random graphs.

However, the real-world peer-to-peer networks have graph topologies that are scale-free and random. It is simple to define scale-free, random graphs by considering the construction of one. Add a new vertex to the

graph at each step, by connecting k edges from the new vertex to existing vertices in the graph and these vertices are selected at random with probability proportional to their current degree. Intuitively, only a few vertices will emerge with the highest degrees and the degrees of other vertices will decrease exponentially.

More recently, there have been a few studies [10, 35] of random walks in such scale-free, random graphs. In [35], an equation is derived for the cover time of random walk starting at any vertex, but the equation involved the calculation of not only the stationary distribution but also an infinite series (summing up the differences between probabilities of closed walks of all lengths and the stationary distribution probabilities). The authors also formulated an interesting quantity called the *random walk centrality* (RWC) of a vertex, which essentially captures how easily information can diffuse through the graph (through a random walk) to reach this vertex. For both cover time and RWC, the paper calculated numerical approximations.

It is shown in [10] that the cover time of a random walk on a scale-free random graph with n vertices and m edges is asymptotic to $2m \log n/(m-n)$. This is asymptotic behaviour of the cover time, and since the work is entirely theoretical, it is unclear how close and accurate this number is in realistic peer-to-peer network topologies.

We are therefore motivated to investigate this in a practical implementation. Our objective is to generate realistic scale-free peer-to-peer network topologies and implement random walks on them to evaluate how the cover times behave in these networks as the number of nodes and degree constraints change.

3.4 Implementation and evaluation

Topology generation

The graphs used for the random walk simulations followed two main kinds of topology generation – uniform-random and power-law. Two parameters were used in the generation program – the number of nodes and the minimum number of connections k, for each node.

Uniform-random

In uniform-random graphs, each node has no preferential attachment weighting to any other node, when selecting a node to connect with an edge the entire population of nodes has an equal probability to be selected.

Pseudo-code for uniform-random graph generation is as follows.

- The n count population of nodes is created
- Iterating through all the nodes 1 through n
- If the total number of connections for this node is less than the minimum number of nodes, then node is selected at random from the entire population, with equal weighting to every node.
- If the selected node is not the selecting node itself, and the selected node is not already connected to the selecting node, then add this connection.
- Upon completion of the process, every node in the n population will have a minimum number of k connections.

It should be noted that this process does not preclude the possibility of a graph with disjoint sections.

Power-law

The generation of the base graph for the power-law topology is based on Barabasi's work [3], which can be a model for many real-world networks

such as the world wide web where there a small number of nodes possessing many connections.

Pseudo-code for power-law base graph generation is as follows.

- Start with a graph of two nodes connected to each other
- Iterating through nodes 3 through n:
  - Select a random node in the existing graph, where the weighting of each is increased by the degree of existing connections.

The probability of any node being selected is its degree of connections divided by the sum of the total degree of connection in the existing graph. For example, if the existing graph consists of five nodes, where one node has 3 connections, one node has 2 connections, and the remaining three have only 1 connection, then the probability of the most connected node is 3/[3+2+1+1+1]=3/8. In this way, the more connected a node is, the more likely it is to become increasingly connected.

It should be noted that this method of graph construction, where each node is added to the existing graph ensures that there are no disjoint sections. For power-law graphs requiring additional connections of more than 1 connection, a second round of connections is made in the same manner as the uniform-random generation. Each node is iterated through, and if the total connections is less than the requested k connections, then additional nodes to connect to are randomly selected (that are not of existing connected nodes, nor the node itself). The purpose of adding these uniform-random connections was to shorten the potential Markov chains which can become very long in a large power-law topology.

Random-walk simulation setup

Five main topologies were used in the simulation testing:

1. Power-law with 1000 nodes and 1 connection minimum per node
2. Power-law with 1000 nodes and 2 connections minimum per node
3. Power-law with 1000 nodes and 3 connections minimum per node
4. Uniform-random with 1000 nodes and 2 connections minimum per node
5. Uniform-random with 1000 nodes and 3 connections minimum per node

Uniform-random graphs with only one connection minimum for each node frequently tend to yield disjoint graphs, thus the choice was made to exclude this model.

3.4.1 Simulation method

Each node in the simulation keeps track of whether it has already been visited in the random walk and the connections it has received messages from during the walk. The remaining parameters being tracked are stored in the transmitted message and updated at each node.

The message format used was as follows:

`RandomWalk:TotalHops:ExpectedUnique:UniqueCount:NodeList :PreviousNodeID.`

The fields are delimited by the colon symbol. The first section identifies the type of message, in this case a random-walk. The second field counts the total hops made so far in this random walk. The third field is the total number of nodes in the simulation. The fourth field is the total number of unique nodes visited so far. The fifth field contains a string tracking which unique nodes have been visited and the order in which this occurred (comma delimited). The sixth field contains which node was the last to be visited and sent the message to the current node.

The simulations were run with a randomly selected node each time to start the random walk, which is messaged by the control node. This node

is considered visited and the flag set to true. The message parameters are initialized with this node as the only one visited, and where the expected unique count is given by the control node in the initialization message. This node's ID is stored as PreviousNodeID field and a random connection is selected for the message to be sent to.

Upon receiving the RandomWalk message, each node checks to see if it has been visited before. If it has not, then it sets its visited flag to true, increments the UniqueCount and appends its ID to the NodeList. If the ExpectedUnique count has been reached by UniqueCount, then the random-walk is complete, and the control node is messaged with the final random-walk statistics. If not, the TotalHops field is incremented and continuation of the random-walk continues with the selected of a random connection to send the updated message to.

This simulation was run on each topology 100 times and the average number of hops for each topology was recorded.

Random-walk simulation results and discussion

The averaged results of the number of hops required for a complete random-walk of the complete graph is given in Figure 3.1. The results of the simulation show that the lower the degree of connectedness, the increased number of hops was required for a random-walk to fully traverse the graphs illustrated in Figure 3.2 and Figure 3.3. The most pronounced difference is between the power-law base topology with minimum one connection, and the addition of a second minimum connection.

The most likely reason for these patterns is the more connections are very likely to decrease the longest distance between any two nodes in the graph. In the power-law base topology, the sparse connections mean there are large sequences of nodes where the traversal may take many hops to eventually cross (essentially "getting stuck" going back and forth). The probability of

moving from one end of a Markov chain to the other becomes significantly smaller the longer the chain is. The introduction of additional connections is likely to decrease the potential length of any paths which mimic Markov chains where the random walk would need to traverse a long sequence of nodes in one direction.

	Average number of hops
Power-law min 1 connection	58858.18
Power-law min 2 connections	19035.61
Power-law min 3 connections	13657.87

	Average number of hops
Random min 2 connections	24835.47
Random min 3 connections	13322.81

FIGURE 3.1: cover times for random walks

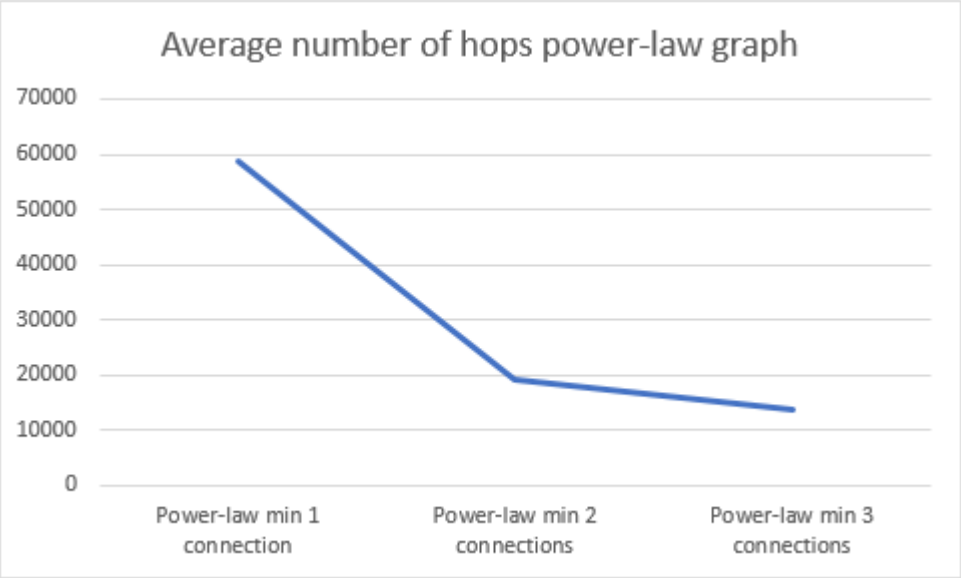

FIGURE 3.2: cover times for random walks

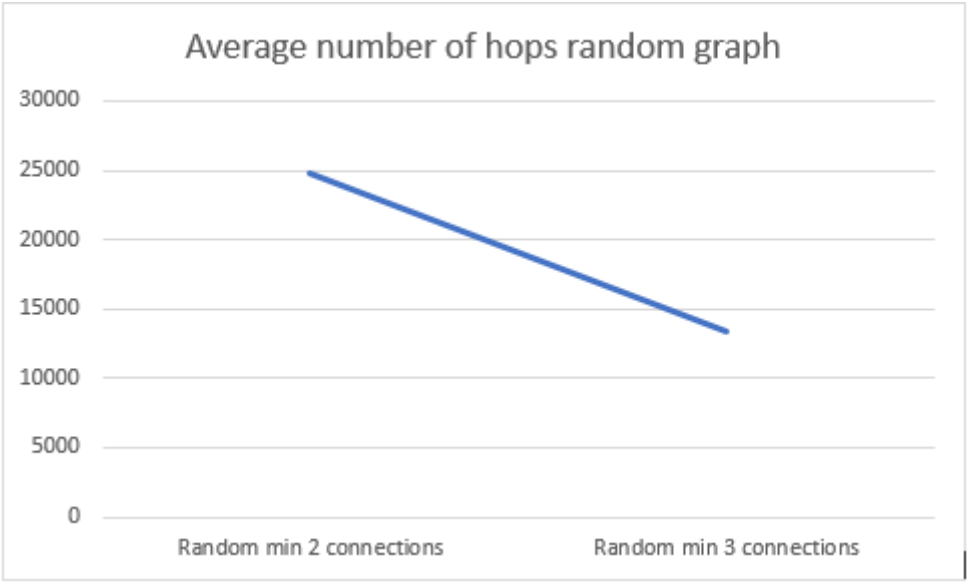

FIGURE 3.3: cover times for random walks

Chapter 4

Maintaining membership list in peer-to-peer networks

4.1 Motivation and objective

Peer-to-peer networks that provide the infrastructure for gossip-based algorithms are desirable for their scalability and performance, but they also have certain requirements on the types of information the nodes in the network must be able to obtain and maintain. One crucial piece of information that must be disseminated among the nodes in the network is the *membership list*. The peer nodes must know about the other peer nodes in the network, in order for any reasonable gossip-based algorithms and protocols to work and be effective. For one thing, some of the gossip-based protocols require that every node must be able to randomly select a number of other nodes from the membership list, and this node then proceeds to *gossip* to this set of nodes. There are various other specific reasons that would require the obtaining and maintaining of such a membership list at each node. An important one is the monitoring of the network by all the nodes to detect events of network partitions and disconnects. Moreover, it is well-known that in any distributed systems, the concept of group membership is essential and critical to the correct and efficient operations of the distributed system.

There have been a number of works on providing robust and scalable protocols for maintaining group membership in peer-to-peer networks, e.g., [20, 44, 40]. These works have focused on increasing efficiency and performance by having the nodes maintain randomized partial membership lists and by proposing distributed protocols that allow the nodes to speed up the dissemination of membership information across the entire network graph.

In contrast, we focus on the aspect of the data structure used for storing and exchanging membership lists among the nodes in the network. We rely on a gossip-based protocol to distribute complete membership lists so that every node in the network is, ideally speaking, in possession of the information of all the nodes currently connected in the same network. We believe that this has tremendous benefits in the maintenance and operations of peer-to-peer networks. One scenario where this would be helpful and advantageous is when the network becomes partitioned, new nodes joining the network in one disconnected component would not be known by nodes in the other disconnected component — the nodes there would be exchanging among themselves membership lists that are not updated and not accurate. Once the network partition is repaired and the whole network is connected again, these nodes that have stale and inaccurate membership lists should have a way to obtain the accurate and up-to-date membership lists. And if all the nodes are periodically cooperating together to disseminate entire membership lists, it is only a matter of time that these ignorant nodes are able to update their knowledge of the membership list.

However, this means that entire membership lists must be exchanged frequently between nodes all over the network. This places a burden on network resources in terms of message overhead. One way to improve message overhead is to try to decrease the size of each of these many messages. Our objective is to use a highly space efficient way to store the entire network

membership list and thereby minimize the message sizes of the gossip protocol that diffuses these membership lists to all the nodes in the network.

We implemented two different ways of storing and communicating membership lists: linked lists, and a bit vector-based hash index called bloom filters; and we show the difference in the message overhead when we use these two different data structures.

4.2 Bloom filters

We briefly present the background on bloom filters. Given a set S of N elements, and an integer x, the problem is to determine whether x is in S.

One can think of creating a bit vector of length n to represent the range of integers $[0, \ldots, n-1]$, where n is larger than the largest element in S. In the bit vector, bit i is set to 1 if $i \in S$ and 0 if $i \notin S$. To check whether x is in S is a simple look-up of bit x in the bit vector. This works fine if the range of integers dictated by the elements in S is not too large and the set S is not too sparse over this range. Otherwise, the resulting bit vector is too wasteful in space and look-up time.

An alternative is to use a hash function h to map the elements of S to a fixed range of integers $[0, \ldots, m-1]$. Now m (or size of the bit vector) does not depend on elements of S, moreover, the elements of S are not restricted to integers and may be of any type. The collision probability of the hash function $h(\cdot)$ is directly correlated to the false positive rate. To maintain a low false positive rate as N increases, the range of $h(\cdot)$ must increase and therefore m or the bit vector size must increase, thus resulting in higher space and time cost.

Bloom filter offers a better way to control the size of the bit vector while keeping the false positive rate low, by using a number of hash functions simultaneously.

A bloom filter uses k hash functions, $h_i, i = 1, \ldots, k$, each with the same range $[0, \ldots, m-1]$. To add an element x to S, set all k bits of $h_i(x), i = 1, \ldots, k$ to 1 in the bit vector. Given x, if not all these k bits are 1, then it is known with certainty that $x \notin S$; otherwise, it is with high probability that $x \in S$ (if $x \notin S$, then this is a false positive). With a pre-specified set size N and false positive rate, one can find the required size of bit vector m. Even for a low false positive rate of 0.1, m is only approximately $10N$ bits. This is considered constant space because m does not depend on the data size of each element in S.

The bloom filter data structure is space-efficient which simply represents a set of data for membership queries [4], and has been widely used in various applications such as overlay collaboration [8] and network intrusion detection [17, 43, 38][38]. Several variants [14, 33, 30, 5, 36] have been proposed for performance speedup and space efficiency.

4.3 Using linked lists and bloom filters for membership lists

To test and illustrate the differences of using a linear or a constant time data structure, the membership list was implemented twice, one using a linked list and another using Bloom filters.

Linked list

Data structures such as linked lists that are deterministic have the advantage of being able to provide their entire data store accurately. For instance, if I wanted to know whether node x was a member of a group, a search of the dataset would always return an accurate result. The main disadvantage is the poor scaling during operations that could require traversal of the entire

dataset, such as searching or removal, or returning the entire membership list.

In the membership list implementation, our linked list is a simple unidirectional list. A pointer to the head and tail of the list is maintained. New nodes are inserted to the tail of the list (although nodes could have been added to the head, with no tail pointer maintained). Since the insertion to the desired node is direct, this operation has runtime of $O(1)$.

The operation to search for members or to remove members (if they exist) requires linear traversal of the dataset. In the worst case the entire dataset would need to be searched if the target was the last member or not a member at all. If we assume the target is in the list then the average case would require n/2 links to be searched, however, this is not a realistic scenario, leaving us with a space between n/2 and n runtime. The general magnitude of runtime for these operations is therefore O(n). In smaller graphs the computational difference is small, however, as repeated tests on a larger graph population illustrate the differences can be significant enough to be measured.

Bloom filter

The use of a hash function to map keys such as IDs onto a bitfield has the advantage of being a constant runtime operation $O(1)$. The disadvantage is that the operation is probabilistic and cannot tell with full accuracy if the member is present. The larger the bitfield size allocated, the lower the chance of collisions and false positives. For this simulation, opensource code by Andreas Briese [7] was used to implement the bloom filter. The bloom filter size was created using a constructor designed for an upper-bound population of $2^{16}(65,536)$ members with error collision rate of less than 1%. Most simulations were under this population upper bound, so the collision rate would have been less than 1%.

The bloom filters work by inputting our node ID into the hash function and a few bitfield locations are returned (depending on the size of the filter, seven in our case). The add function hashes the input and then flags the designated bitfield locations as true. The search function again hashes the input to test ID and then checks if those resulting locations are already all marked as true. If not all the locations are marked as true, then the bloom filter can say with certainty that the entry is not in the list. If they are all present, then the bloom filter can say with over 99% probability (100% - the error margin) that the inputted ID is part of the list.

Since the bloom filter bitfield can have overlapping component locations, removing a node from the member list cannot be done by hashing the ID and setting the locations to false. For instance, say we have two node IDs that have already been entered into the membership bloom filter, where node ID X might hash to locations 5, 9, 32 and node ID Y might hash to locations 9, 40, 62. There is an overlapping location at 9, so removing either node by setting all locations to false would cause false negative searches. To account for removed nodes, we instead maintain a second bloom filter that records entries for removed nodes. Searching to see whether a node is a member then becomes two constant time operations. First, we check to see if it is in the member list. If not found, then we can say with certainty that the node is (and never was) a member. If an entry is found, then the node is (or was) probably a member, and we must now also search the second bloom filter for removed nodes. If the removed bloom filter returns negative, then we know with certainty the node was not removed, and due to being on the member bloom filter that the node is most likely a member. If the removed filter returns positive, then we can conclude that the node was most likely one time a member and has since been removed.

While the constant time operation of bloom filters is a big positive, there

are disadvantages. One is the lack of ability to readily produce a list of members in either the member or removed list. To derive this information, a range of possible keys would need to be tracked and each tested one by one. Another disadvantage is the inability to use the bloom filter for a task that requires extended use or re-use of keys. In the first scenario, both the member and removal bloom filters will eventually saturate once the expected population created at the beginning is reached. While a larger bitfield could be created initially, this still presents a non-scaling solution whose resizing would require an intensive search operation of the entire possible key set (insert all current keys into larger, new bloom filter). The second scenario of key reuse cannot be accommodated due to the one directional operation of the bloom filters and potential overlapping space of hashed keys.

4.4 Simulation and evaluation

Unlike in the random-walk simulation where a fixed graph topology was used, for the membership lists we simulate a system of dynamic churn, where nodes are added to and removed from the system at random. We begin with a fully connected graph, however, due to the random nature of removals it becomes possible that clusters of nodes will become disjoint at times during the simulation.

The simulation is provided with a lower and upper bound range from which random time intervals are used to add and remove nodes from the system. For the purposes of the discussed results, nodes were added every 300 to 600 milliseconds, and removed every 600 to 1200 milliseconds. Each of the newly added nodes randomly selects between 2 to 4 random nodes with which to connect to in the graph.

4.4.1 Maintaining membership lists

Two mechanisms are implemented to maintain an accurate membership list in the dynamic environment of nodes joining and leaving the system. The first is to have all nodes broadcast to their connections if they are being removed or are newly added. If this information is new to the recipient nodes, they will further propagate the message to their connections. The new information requirement prevents loops and endless message repeats while allowing for fast dissemination of the updated membership changes.

This incremental membership tracking works effectively most of the time, however, in the case of partitioning that may periodically occur due to the random removal churn sometimes a complete survey of the connected population may be required. For example, if a fully connected graph was split in two, each side of the graph would not be able to determine if its removal would result in a partition and thus be unable to determine what large portion of nodes were being disconnected. Following this line, if the two partitions were to be rejoined by a new additional node, each half would need to be updated with the reintegration of the population in each segment, as well as the other incremental updates that have occurred since the time of split. The solution chosen is instead to have periodic survey probes initiated by random nodes to wholly update the membership information.

To implement this procedure, each node maintains a countdown timer of randomly chosen time between 4.5 to 400 seconds. Upon expiration of this timer the node first checks to see if another survey probe is already ongoing. If one is, the timer is reset to a new random time duration within the specified range. If not, the node will initiate a probe by flooding out to all its connections a message probe which contains the message header “Probe”, the ID of originating node, the IP address and port of the originating node, as well as a timestamp of the initiation time.

When a node receives a probe message, it first checks to see if it has already received another probe message that has not yet been resolved with a final membership list. In the event of two simultaneous ongoing probes, the last timestamp parameter is used to resolve which one is older and that one is given precedence. If the stored probe timestamp is older, then this new probe will be discarded. If the newly received probe is older (or no other probe is ongoing), then the node will store the new survey probe information, propagate the message to all its connections, and finally open a temporary direct connection to the probe origination node with a "ProbeReply" header, the origination probe ID, this node's ID, IP address and listening port. The temporary connection is then closed.

During this time, the survey probe origination node maintains a timeout timer in a go routine. It is initially set to timeout after 150 milliseconds. Every time a new reply is received, this timer is reset to twice the time between the start of the probe and the received reply time. In this way, the timer scales to accommodate larger graphs which may be waiting on distant node replies. To maintain consistency, nodes will not remove themselves if a probe is ongoing.

4.4.2 List search efficiency testing and results

To contrast the efficiency of the two data types, the simulation was run until the connected graph population reached 100, 550, and 2000 nodes for both the single linked list and bloom filter data structures. A single node was then made to make 25,000 consecutive queries on whether its membership list contained a random node ID. The large number of queries is to provide a meaningful and measurably significant time scale for comparison. The random selection field consisted of half valid members and half IDs that are not part of the population. As previously discussed, the injection of queries for

non-members is expected to have no impact on the hash-based search, but due to the linear nature of the linked list will require full traversal of the data structure. The results are displayed in Table 4.1.

Data structure	Number of nodes	Total time to resolve 25k searches
Hash	100	4 ms
Hash	550	4 ms
Hash	2000	4.2 ms
Single linked	100	14 ms
Single linked	550	78 ms
Single linked	2000	264.7 ms

FIGURE 4.1: Comparison of linked list and bloom filter hash index in search time for membership in membership lists

The results show the hash-based bloom filter completes the queries in approximately 4 milliseconds each time, whether it was dealing with a population membership list of 100 nodes or 2000. The single linked list by comparison starts off taking around 14 milliseconds for 100 nodes and linearly grows to 265 milliseconds when the list is 2000 nodes long. Based on the constant time operation of the hash function and the linear nature of the linked list these results are as expected.

Valid rate	Time to resolve 25k searches	Avg num links searched	Percentage of links searched
1.00	76.8 ms	510	50.0%
0.50	114.3 ms	765	74.9%
0.10	142.6 ms	967	94.7%
0.01	146.8 ms	1013	99.1%

FIGURE 4.2: Effect of not-found percentage on search performance in linked list implemented membership lists

Further examining the linear nature of the linked list, Table 4.2 illustrates how in a 1020 node list, increasing the percentage of not found queries increases the total search time due to the total number of links required to be traversed on average. In the case of all found queries (valid rate 100%), the expected on average half number of nodes is needed to be searched. As the

valid rate decreases, the pool of queries for not-found targets increases requiring full traversal of the list, and pushing the percentage of links searched to nearly 100% in the case of only 1% valid member searches.

Chapter 5

Implementation of gossip algorithms in peer-to-peer networks

We are interested in gossip-based algorithms [25, 6, 12, 2] in our study of peer-to-peer networks because they are useful in disseminating some sort of aggregate information to all the nodes in the network while remaining distributed in terms of execution and light-weight in terms of message overhead. Gossip-based algorithms are also a type of randomized algorithm that are also closely related to random walks. They are especially effective and efficient in distributed networks where the nodes do not have any global knowledge of the network but only has information about their local connections. By executing these gossip algorithms, the nodes can cooperate to achieve the objective of collecting and spreading various kinds of aggregate information or statistics over the entire network.

There are many applications that would require the nodes to be apprised of some global information. The scalability and effectiveness of peer-to-peer networks are rooted in their distributed and decentralized nature, and they would not be practical or scalable at all if they were assumed to maintain global state of the entire network — that would only true in a centralized

system.

For example, suppose every node in the network maintains a numerical value and the objective is to have every node eventually obtain the average of all the values held by the nodes. A naive centralized solution would be to assign a single node to obtain values from all the nodes and calculate this average. However in this case, there exists a single point of failure; if this centralized node fails, another node cannot even simply take over and pick up where its predecessor left off since the successor would have no information about the values already sent to the predecessor, the successor would have to start from scratch by contacting all the nodes again. This can potentially be a very expensive and slow process. In case of failure of a single node, the nodes in the network cannot even obtain an approximate of the average value.

It is also important to observe that peer-to-peer networks lack infrastructure in the conventional sense, and are subject to dynamics in the ad-hoc join and departure of nodes, as well as to constraints of resources precisely due to its lack of persistent infrastructure and its dynamic nature. These unique characteristics of peer-to-peer networks dictate that the algorithms or protocols that execute in them must be light-weight, distributed, resilient against network dynamics, and do not incur large overheads.

The problems inherent in the centralized approach and the requirements on algorithms that are suitable for peer-to-peer networks are addressed by the randomized gossip-based protocols.

Gossip algorithms utilize a style of communication among the nodes similar to that of the spreading of *gossip* or rumour, hence its name. Gossip or rumour spreading is characterized by unreliable and asynchronous exchange of information between entities. The minimalism and wide applicability of these algorithms have made them important and popular in the new types of networks that have emerged, e.g., peer-to-peer, social and sensor networks.

In general, the class of gossip algorithms all must possess the following properties:

1. In the algorithm, each node must only use information it has received in messages from its direct neighbours in the network graph.

2. In every local step of the execution of the algorithm, its computational complexity must not exceed roughly the product of the node degree and the log of the total number of nodes.

3. The storage required at each node should also not exceed roughly log of the number of nodes.

4. No synchronization is needed between a node and its neighbours, meaning they don't need to take strictly ordered turns to send messages.

5. The result disseminated by the algorithm is robust to its random nature, i.e., roughly the same result is achieved regardless of differences introduced by randomness in different executions of the algorithm.

5.1 Implementation of a gossip algorithm to compute the average

Assuming that every node in the network holds a numerical value to begin with, we implement a gossip protocol for each node to calculate a local estimate of the *average* of the values held by all the nodes. We describe the gossip protocol we implemented in the following. The implementation is based on gossip protocols from [25].

Every node is seeded with a random floating point value to start with, say in the range from 0 to 1000. Messages are sent in coordinated rounds where an aggregate calculation is iteratively done and progresses/converges to the average (arithmetic mean). We implement two variants of the algorithm: (1)

In each gossip round, a single message is sent to one randomly select neighbour. (2) In every round, a message is sent to every connected neighbour of the node. We also designate one of the nodes as the control node. The control node simply keeps track of rounds in the gossip protocol, and does not participate in the exchanges of messages or the estimation of the average. In the case of the failure of the control node, another node can easily be selected to take on this role without any disruption or slowdown of the progress of the algorithm.

To calculate an estimate of the average, if it's the first variant, i.e., only sending to one neighbour, the node retains half the stored value, and a 0.5 "weight". Each node will send half its value and the 0.5 weight to a random neighbour. The control node coordinates the rounds so there are no race conditions. At the end of a round, each node adds up all the received "values" and "weights". It then calculates a new value calculated as the sum of all the received values divided by the sum of all the weights. If the node had received no other inputs, then dividing the fraction it "saved" by the same weight yields back the original amount.

In the second variant, i.e., sending the (value, weight) pair to every neighbour, the weight and fraction of value sent is 1/(total connections + 1). The plus one is for "saving to itself". Diffusing the value ensures that nodes with a higher number of connections do not get greater influence diffusing out.

5.2 Simulation and Evaluation

To test the efficiency of both gossip algorithm variations, simulations were run on topologies generated using the power-law and uniform random graph methods described previously in chapter 3. Four were based on the power-law topology. One was the base power-law and three additional had parameters of minimum two, three, and four connections per node, with the

additional connections being random. There were also two uniform-random topologies used, one with a minimum of two connections per node and another with a minimum of three connections. All topologies were fully connected graphs and were not subjected to node additions or removals.

To begin, all nodes are seeded with a random value from 0 to 1000. The actual arithmetic mean is stored by the control node. Each node is notified that a gossip round has begun and depending on the gossip algorithm a portion of the stored value is sent to either one random connected node (single push) or a smaller fractional weight to all connected nodes (all connection push). Nodes are notified of the end of a round, and a new average is calculated based on the stored and received weights of values. This process is repeated until the target number of rounds is completed. At the end of the process all nodes report their local calculated average to the control node where the average standard deviation and percentage from the arithmetic mean is calculated and recorded. Each gossip algorithm was run five times on each topology. The results are shown in graphs Figure 5.1 and Figure 5.2.

As with the random walk, topologies with more connectedness converged to the actual average faster and was the case for both gossip algorithms. The long, sparse chains of nodes centered by a few super nodes in the power-law topology is clearly the slowest to converge, and for the other graphs, the more connections each node has the faster it converges. Our inference is that the additional connections in some topologies reduce the potential maximum degree of separation of any nodes and therefore increase the probability that convergence will occur faster. For instance, if one segment of the graph had a cluster of distant values, the low number of connections to that cluster would be a barrier to diffusion.

Examining the two gossip algorithm variations, gossiping to all connections also provided a much greater convergence rate due to forced diffusion. By forcing out an average to all connections each round, the rate of diffusion

is increased by a factor of the number of connections a node has, compared to the single gossip message version which can take many rounds to achieve the same amount of diffusion. Additionally, if only sending to one node per round there is a chance that the diffusion is not equal and, in some areas, could repeatedly send to the same nodes slowing effective spreading. The vertical scale for standard deviation percentage in both graphs is the same distance and illustrates the much faster convergence rates on all graph topologies.

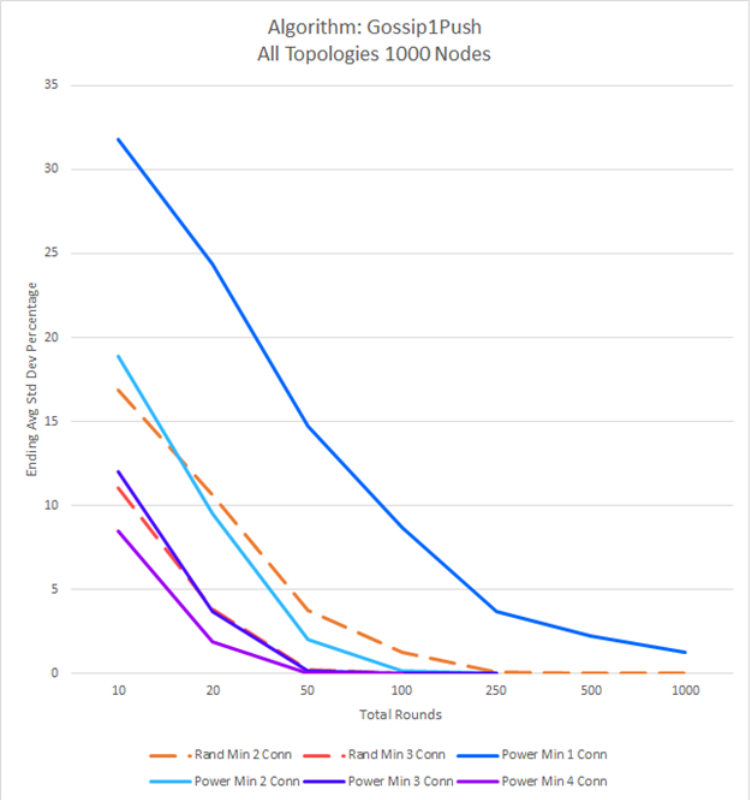

FIGURE 5.1: Convergence of gossip algorithm variant 1 (gossip to one neighbour at a time)

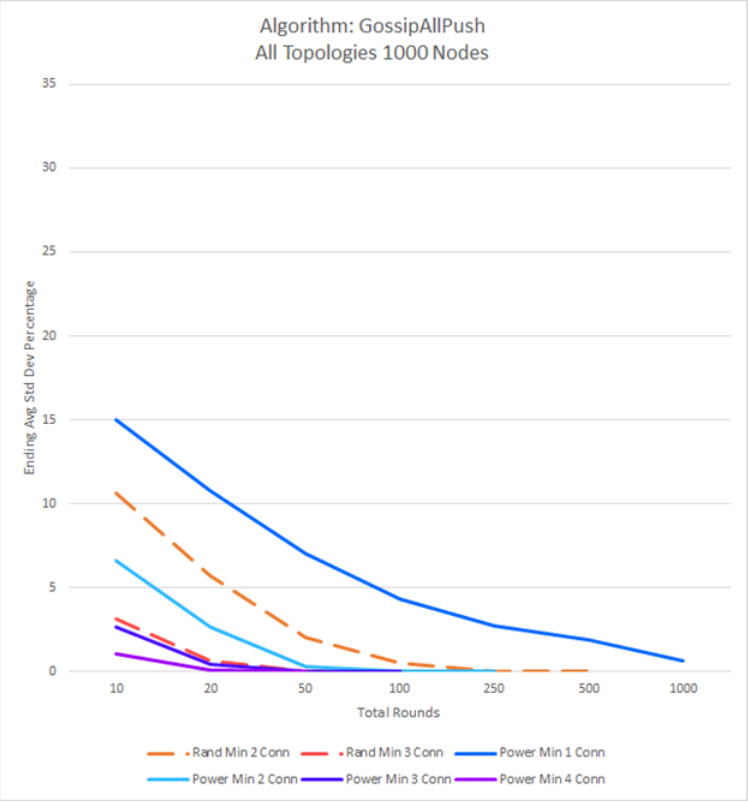

FIGURE 5.2: Convergence of gossip algorithm variant 2 (gossip to all neighbours at a time)

Chapter 6

Identifying high-risk edges

As with the rest of this thesis, we consider a peer-to-peer network that is a scale-free random graph, which has been observed to realistically model most real-world evolving networks despite the diversity in the different network types. To illustrate the motivation for detection of high-risk edges in such networks, we consider the following scenario. Suppose that in a peer-to-peer network, there are two components of nodes and within each component the nodes are well-connected, but between these two components there exist only one or two edges connecting the components. These edges connecting the two components are bottleneck edges or *high-risk* edges; if they were disconnected then the entire network would become disconnected and partitioned. An example of a high-risk edge is shown in Figure. 6. Here, we have a network where node u and v both have high degrees and are connected to two non-intersecting connected subgraphs, therefore the edge e connecting u and v is a high-risk edge.

A second example is shown in Figure. 6, where there are two edges, $e1$ and $e2$, that provide the connection between the cluster of nodes on the left with the cluster on the right. It can be observed that these two edges actually constitute a *minimum cut* of this undirected graph, with the exception of cuts of the graph that partition a single node from all the other nodes. So one idea is to use graph algorithms that find min-cut to find these edges. However, there are two challenges that must be addressed if this were the route

to be taken. The first challenge is that min-cut algorithms usually find the sum total of weights on the edges that form the minimum cut and not which edges they are, and in this small example, that would be 2 and would include those cuts that isolate single nodes. It is not at all obvious how one might go about adapting the traditional min-cut algorithms for the purpose of finding edges such as $e1$ and $e2$ in our example. Another related issue is the problem of making the min-cut algorithm distributed, since the conventional graph algorithms are almost always centralized and have the basic assumptions of global information of the graph topology and central control of all the nodes in the graph. This is not a trivial problem.

The second challenge is that in cases like in our example, it is arguable whether it is important to determine both edges, that is, it may not be all that important to identify the edge $e2$ and that it is sufficient to identify the edge $e1$ as a high-risk edge. The reason is that if $e2$ were to fail, the network would still function perfectly well mainly due to the fact that the two nodes incident on $e2$ do not have high degrees and hence are not central to the network and are not high-traffic nodes. They would not have seen much message forwarded to and from each other anyway, so the edge between them becoming disconnected will not inconvenience significantly the message transiting throughout the network. The small number of messages that would have traversed the edge $e2$ would simply be re-routed through the already high-traffic and central nodes u and v. On the other hand, however, if the edge $e1$ were to fail while $e2$ remains in working order, the network would still suffer in a substantial way even though it would not be partitioned. The large amount of traffic that normally would go through u, $e1$ and v would now have to be re-routed through $e2$ which are incident on two nodes which have low degrees, which would lead to bottleneck situations forming around these two nodes and this would in turn lead to serious deterioration of network performance. So it may not be worth the effort or overhead to try to

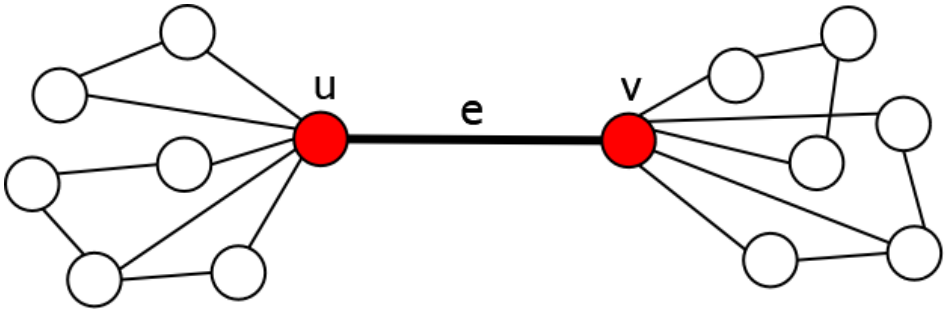

FIGURE 6.1: Example of a high-risk edge

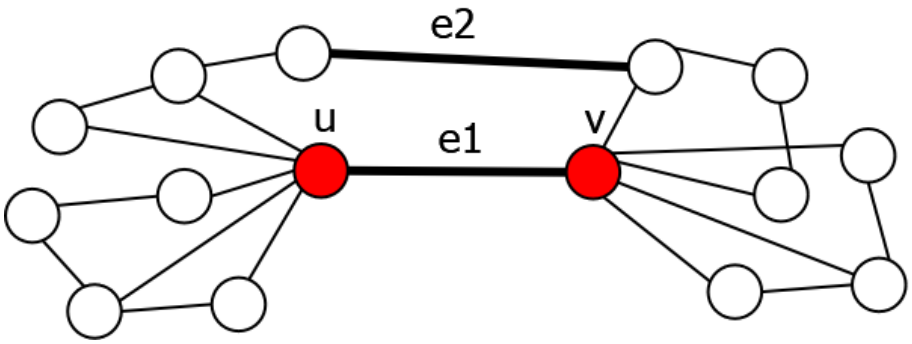

FIGURE 6.2: Example of two edges that form a minimum cut

find all the edges that constitute a meaningful minimum cut, at any rate, it is not worthwhile to find the edges that are incident on low-degree nodes regardless of their positions in the topology.

The intuitive observation here seems to be that we should consider as high-risk edges only those edges that belong to a minimum cut and are incident on high-degree nodes.

It is of interest to detect such bottleneck or high-risk edges so they can be monitored closely and in the case of their disconnection, the network can recover more quickly and seamlessly — this greatly increases the reliability and resilience of the network.

The characteristics of a high-risk edge e are that it is incident to two nodes u and v where u and v are each densely connected to a set of nodes and these two sets of nodes that do not intersect. Since we are dealing with network topologies that follow a power-law distribution in node degrees, it is

certainly the case that there are a small number of nodes with high node degrees while the remaining nodes have exponentially lower node degrees. In such a topology, if two high-degree nodes are connected by an edge and they are respectively connected to components of the network which do not intersect, then the edge connecting these two high-degree nodes is a high-risk edge because if it were disconnected, then the network is partitioned. Ideally, the network should be aware of the existence of this kind of high-risk edges and allocate more resources to monitoring them. If a regular edge is disconnected, it does not usually lead to the network becoming partitioned, so it doesn't have much effect on the network as a whole. The incident nodes will eventually become aware of the disconnection between them and find ways to repair the connection, but even if they do not re-establish connection, it does not have dire effects on the whole network. In contrast, if a high-risk edge fails, then the whole network is affected seriously and globally. But if high-risk edges are identified and constantly monitored, then their failure could be detected in a timely fashion and the network partition could be repaired by re-connecting the network.

6.1 Using random walks and gossip algorithm to detect high-risk edges

The first approach we undertook to detect high-risk edges was to use random walks. The idea was that because a high-risk edge is connected to high-degree nodes and are the only connection between two components of the network, random walks in the network may end up traversing the high-risk edge more frequently than other edges. We can try to identify the high-risk edge in two phases. The first phase is to execute a random walk in the network. The weights of all edges in the network are initialized to zero. As the

probe message from the random walk visits every edge, the weight of that edge is incremented.

Assuming that the high-risk edge will record a higher weight than other edges, the second phase should pick out those edges that have the highest weights which are significantly higher than the weights of most edges. This is made difficult by not having global knowledge of the network. We cannot assume that a list of all the edge weights will be available in one place and the maximum pick off the top of that list. The whole concept of peer-to-peer networks is that there is no such global coordination to be counted on. So we must do this in a distributed way. One way to accomplish this is to use gossip algorithm to estimate the average of the edge weights, as discussed previously in the gossip chapter. Each node will begin by sending to its neighbours the weights of its incident edges, and at each round, the local estimate of the average is updated according to the values received from neighbours. Eventually each node will arrive at an estimated average of the edge weights. An estimated average value is not sufficient for each node to use to compare with its own local incident edges weights, for determining if the local values are significantly higher than the average. We also need an estimated standard deviation. One heuristic way to obtain that is to execute the above gossip-based aggregation of the average a number of times and take the resulting average as a single sample in a distribution (which can reasonably established based on properties of the specific type of topology). With a set of samples for the average, an estimate of the standard deviation can be calculated. Now each node can compare its own local incident edge weights against a threshold calculated based on the average and standard deviation. For instance, an edge weight that is more than two standard deviations away from the average could be deemed to be a high-risk edge.

After conducting some experiments, we discovered that **random walks do not result in the weight of the high-risk edge being consistently and**

noticeably higher than other edges.

6.2 Using randomized shortest paths to detect high-risk edges

The second method we adopted is to randomly select a pair of nodes in the network, u and v, find the *shortest path* from u to v, and send a message along this path. Every time a message traverses an edge in the network as part of some shortest path, the edge weight is incremented. All edge weights are initialized to zero. This process is repeated a number of times; it can be terminated when the weights of bottleneck high-risk edges are significantly and sufficiently higher than the rest of the edges such that these high-risk edges can be detected.

In order to see the reason of adopting this method, we note that bottleneck high-risk edges are characterized by their importance and centrality in the network, and the fact that under normal operations of the peer-to-peer network, these edges are expected to be traversed by higher volume of messages than the rest of the edges. As discussed in the previous section, a high-risk edge has two main observable properties: it is incident to a pair of high-degree nodes, and it is a bottleneck between two components of the network — i.e., all or most of the traffic between the two components in the network have to transmit over this edge.

Assuming more-or-less normal conditions in the network, that is, every pair of nodes are roughly equally likely to communicate with each other. This is a big assumption, however, given no other prior information about the operations of the network, this assumption is the most reasonable one to make. Following the same logic, if there were a bottleneck high-risk edge (a couple of examples are depicted in Figure 6, 6), every pair of nodes (u, v) where

u is in one component of the network while v is in the other component, will be equally likely to exchange message with each other. And every time messages are exchanged between any such pair (u, v), they will have to traverse a bottleneck high-risk edge, yet all the other edges in the path between u and v are, generally speaking, different given different choices of the pair (u, v). So the intuition behind the method in this section is that if we keep randomly picking pairs of nodes (u, v), the path between them will contain the bottleneck edge with much higher probability than any other edge.

We conducted simulations of this method in a randomly generated network graph that has two components with bottleneck edges between them. After a relatively small number of such repeated path traversing, we find that the bottleneck edges are easily distinguishable from the remaining edges in having not only the highest weights but significantly higher than the rest. This is an effective way to detect these bottleneck edges.

6.2.1 Analysis of method

Now we will present an analysis of the method we have described and show that it indeed results in a bottleneck high-risk edge having significantly higher edge weight than all the other edges, and therefore it is an effective tool in identifying and detecting high-risk edges.

We consider the scenario where a pair of random nodes are selected and a shortest path between them is traversed, and calculate the probability of a bottleneck edge being part of that path and also the probability of a random non-bottleneck edge in the network being part of that path. When an edge is part of the path, its weight is incremented. Let $\hat{e}$ denote a bottleneck edge and e denote a non-bottleneck edge.

The probability that the weight of $\hat{e}$ is incremented is the probability that $\hat{e}$

is part of the path connecting u and v which means that they belong to different components. Since the two nodes are selected randomly, this is either u is in component 1 and given that u is in component 1, v is in component 2, or vice versa. Suppose the two components have n_1 and n_2 nodes, respectively, and $n_1 + n_2 = n$ is the total number of nodes.

$$P(\hat{e} \text{ is incremented in weight}) = \frac{n_1}{n}\frac{n_2}{n} + \frac{n_2}{n}\frac{n_1}{n} = \frac{2n_1n_2}{n^2}$$

Now consider a non-bottleneck edge e, the probability that it is part of the shortest path can be estimated as the average shortest path length in the network divided by the total number of edges. From previous work [34, 19], we can find estimates on the average shortest path length in a scale-free random graph:

$$l = \frac{\ln(n/z_1)}{\ln(z_2/z_1)} + 1,$$

where z_1 is the average number of nearest neighbours and z_2 is the average number of order-two neightbours (those that are two hops away). Hence,

$$P(e \text{ is incremented in weight}) = \frac{l}{z_1 * n/2}$$

The denominator is the total number of edges calculated from the average degree of each node multiplied by the total number of nodes divided by 2.

Each time a shortest path between two randomly selected nodes is traversed and whether each edge weight is incremented or not can be considered as a Bernoulli trial. Suppose the path traversals are repeated m times. Then the *expected* weight of a bottleneck edge and a non-bottleneck edge are:

$$E[\text{weight of } \hat{e}] = m \cdot \frac{2n_1n_2}{n^2}$$

and

$$E[\text{weight of } e] = m \cdot \frac{l}{z_1 * n/2}$$

For example, assume $n = 200, n_1 = 100, n_2 = 100, z_1 = 3, z_2 = 9$, and the shortest path traversal is repeated m times. The expected weight of a bottleneck edge is calculated to be $m/2$, whereas the expected weight of a non-bottleneck edge is $0.016 \cdot m$ in contrast.

6.3 Evaluation

We evaluate our randomized shortest-path method by simulating randomly generated network topologies that are constructed carefully to be comprised of two connected components with one or two bottleneck edges connecting these two components. We then implement our method and examine the resulting edge weights of all the edges in the network, and see if the edge weights of the bottleneck edges are sufficiently higher than weights of all the other edges, to make detection of them feasible. We use the programming language Python for our simulations in this section.

The first step is to construct two random network graphs using power-law node degrees and the preferential attachment procedure, exactly as we have done in previous chapters. These two graphs serve as the two components in our final network topology. A node of high degree is randomly selected from each of these two graphs, an edge is then added to connect these two nodes — this is a bottleneck high-risk edge. We also repeat this procedure to add a second bottleneck edge between the two components. One such simulated random network topology with two bottleneck edges is shown in Figure. 6.3.

We then execute K iterations of random shortest-path selection and message traversal along the selected shortest-path. At the end of K iterations,

every edge in the network graph has an associated edge weight. To visualize these edge weights, we use a Python library to visually display the network with each edge having a shade of gray that is proportional to its edge weight, so higher edge weights show up as darker edges. This is depicted in Figure 6.3, in a random topology of 200 nodes. It can be clearly seen that the two bottleneck high-risk edges have significantly darker colours than the rest of the edges, indicating that they have much higher weights than the other edges, thus allowing them to be feasibly identified and detected.

We also ran this simulation with networks of 400 and 1000 nodes, whose edge weights after executing the shortest paths are shown in Figure 6.4 and Figure 6.5, respectively. Similar results are observed in these larger size topologies. For these larger topologies, the number of randomly selected shortest paths to be computed also needs to increase; it is highly encouraging that the number of shortest paths required to distinguish high-risk edges from the rest tends to increase only linearly with the number of nodes. For the 200-node topologies, the number of shortest paths required is 20, while for 400-node and 1000-node topologies, the numbers of shortest paths needed are 40 and 100, respectively.

We also generate a histogram of the edge weights of all the edges in the network, in Figure 6.6. It can be seen here that vast majority of edges have a weight of 1, with a small number of them (below 100) having a weight of 2, and only two edges have weights higher than 2 — these two being the bottleneck high-risk edges, as seen in the visualization of the network.

The histograms for the larger topologies of 400 and 1000 nodes are shown in Figure 6.7 and Figure 6.8, respectively. Again, similar results are seen in all the different sizes of network topologies.

In our simulations, we have tried with different values of K, the number of iterations of random shortest path selection and message traversal. For each of these values, we would visualize the edge weights and plot the histogram,

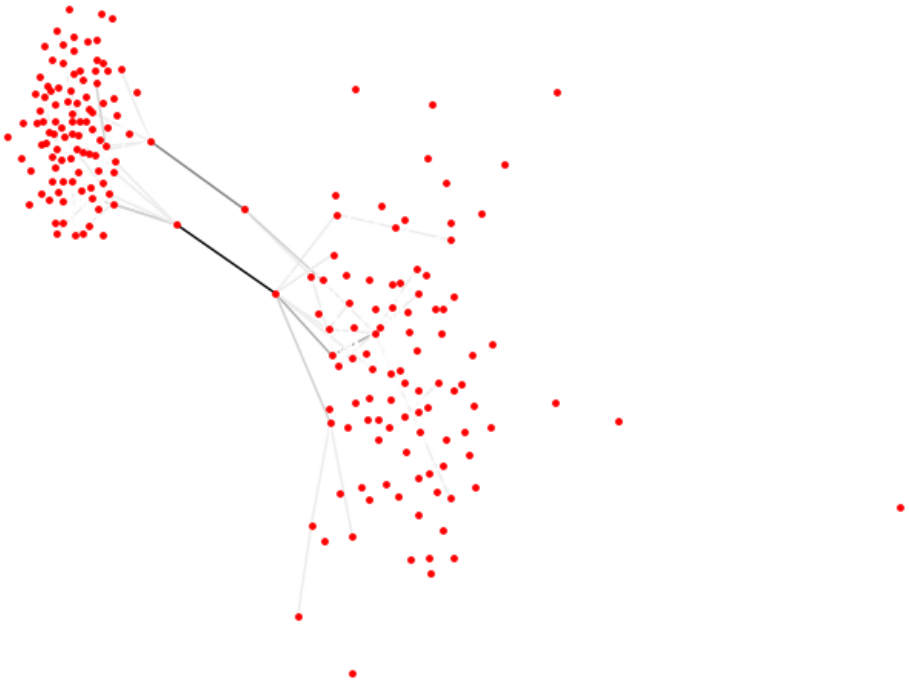

FIGURE 6.3: A random network topology of 200 nodes with two bottleneck high-risk edges

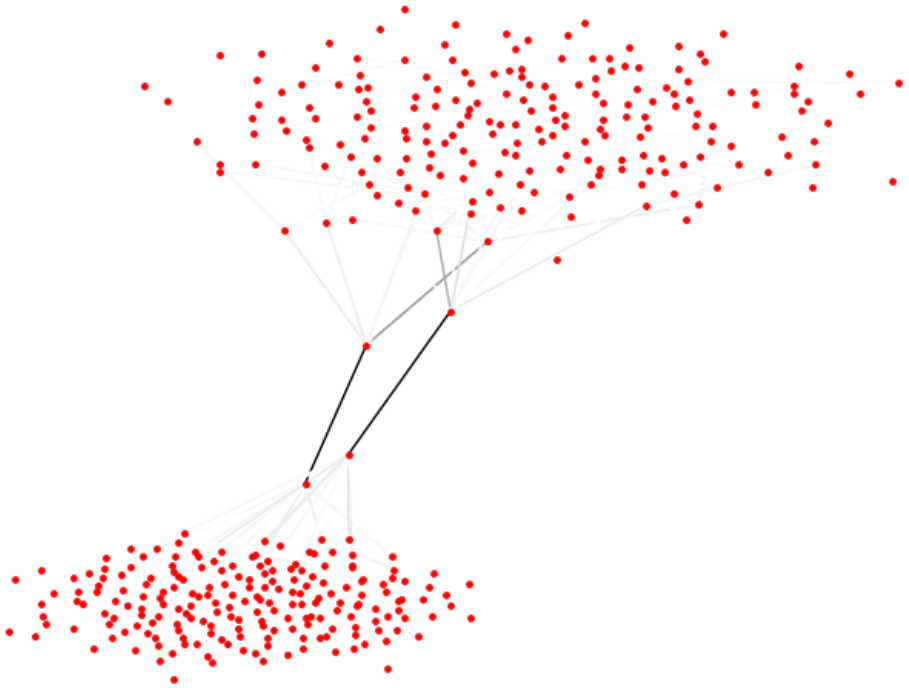

FIGURE 6.4: A random network topology of 400 nodes with two bottleneck high-risk edges

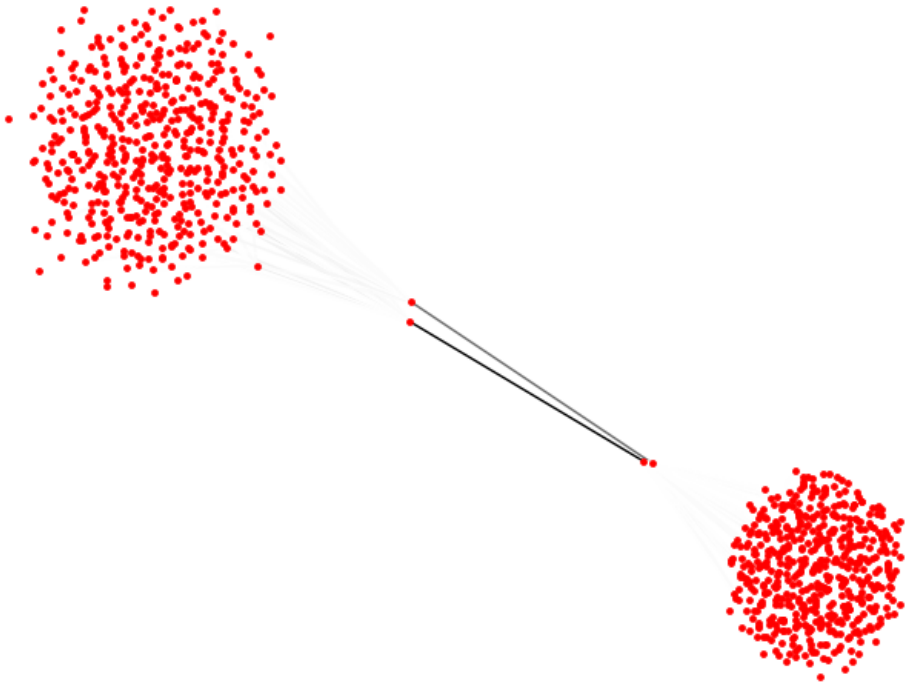

FIGURE 6.5: A random network topology of 1000 nodes with two bottleneck high-risk edges

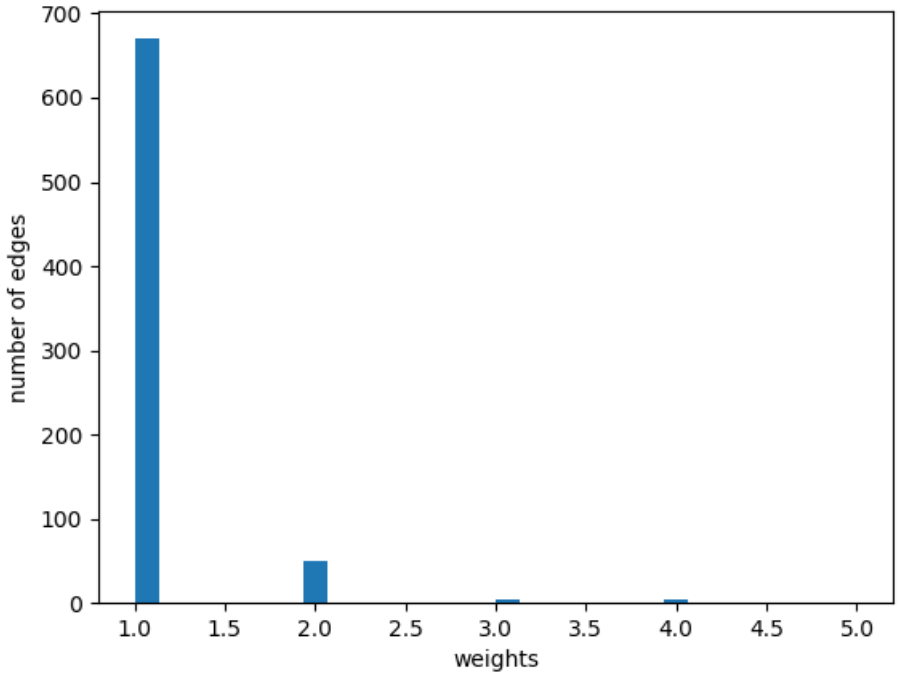

FIGURE 6.6: Histogram of edge weights in the above random network topology of 200 nodes with two bottleneck high-risk edges

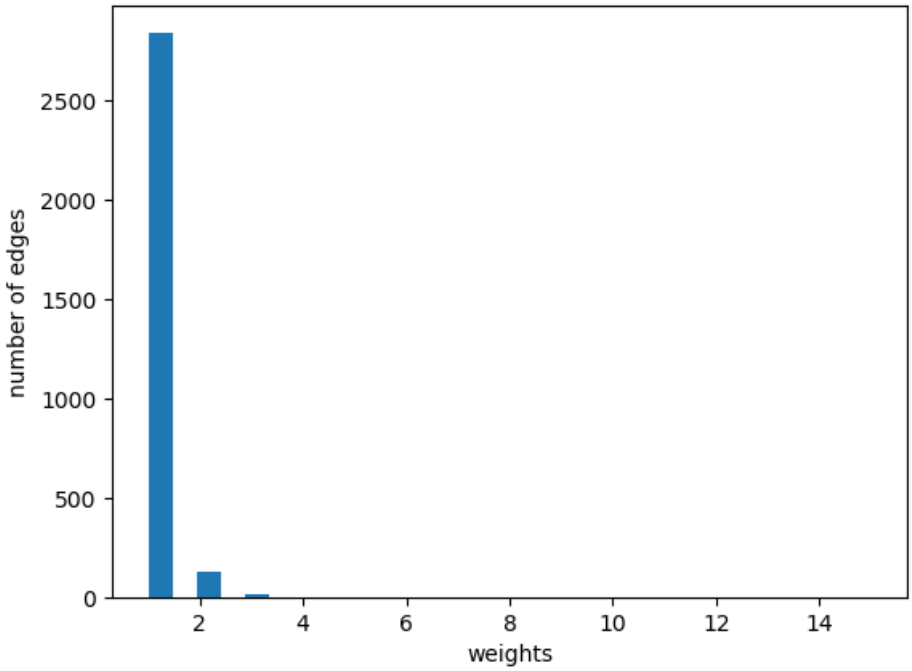

FIGURE 6.7: Histogram of edge weights in the above random network topology of 400 nodes with two bottleneck high-risk edges

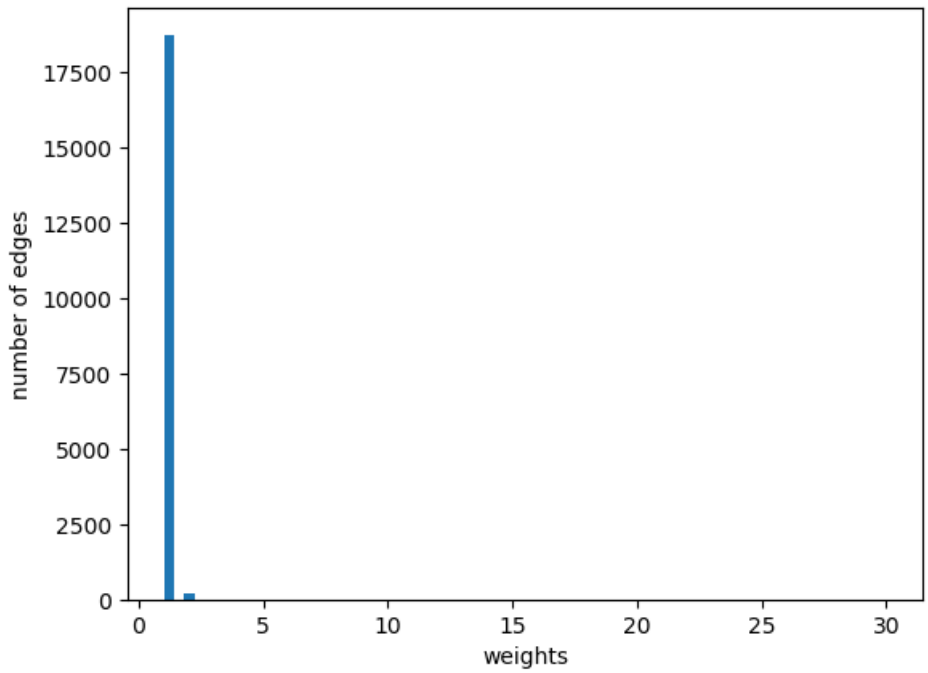

FIGURE 6.8: Histogram of edge weights in the above random network topology of 1000 nodes with two bottleneck high-risk edges

to see if the bottleneck edges will be highlighted and distinguished. We have empirically observed that this K does not need to be large, it is sufficient to have values of as low as 20 to see the desired distinctions of the bottleneck edges. It seems encouraging that we do not need to execute a large number of shortest-path computations. This makes our method efficient and feasible, because the computational complexity of finding shortest path is quite reasonable, it is in the order of $O(m + n \log n)$, where n is the total number of nodes and m is the total number of edges, [18]. But this is only one iteration, so if the number of iterations required for distinguishing bottleneck edges is high, then the overhead becomes prohibitive and infeasible. Therefore, it is encouraging that from our simulations, we have found that the number of iterations required is relatively low, which means our method is feasible and efficient.

6.4 Gossip protocol to aggregate mean and standard deviation for high-risk edge detection

In the previous sections, we have proposed — and verified its effectiveness in both analysis and simulations — an efficient method for every edge to acquire a weight that is proportional to its centrality in the network, i.e., the amount of message traffic that traverse the edge under normal network behaviour and operations. Our objective of finding the edges with the highest weights would be simple if all the edge weights in the entire network were *global* information — either every node has this global information or there exists a central node which collects this global information. However, this would be defeat the purpose of having a decentralized peer-to-peer system.

Without this assumption of global knowledge of the edge weights, we need to devise a way to somehow allow the nodes to use local knowledge

and distributed protocols to arrive at a conclusion on whether their local edge weights are the highest ones in the network. We can use the gossip algorithm we implemented in Chapter 5 for each node to estimate the *average* of all the edge weights in the network. This brings us a little closer to our goal, in the sense that each local edge weight can then be compared to this global average and see if it's higher than the average. However, this does not allow the decision to be made on whether it's *significantly higher than the average* which is what we want. We need to be more precise about this concept of significantly higher, and we also need to find a way for nodes/edges to locally determine this.

Consider the set of all edge weights in the network, just like the node degrees, they most likely follow a certain distribution, with a mean (average) μ and a variance σ^2. Instead of the variance, the standard deviation σ is often used to measure the how far away the values in the distribution stray from the mean. We can use the mean μ and standard deviation σ to precisely define "significantly higher" than the mean: An edge weight w is considered significantly higher than the mean if $w > \mu + k \cdot \sigma$, where k is an integer (>2) heuristic parameter that we can try to determine.

One problem that arises is how do the nodes learn the variance or standard deviation in a distributed manner, or more realistically, how should they estimate it. We note that for a set of n values $\{x_i\}_{i=1}^{i=n}$, the variance is defined as the expected value or mean of the sum of squares of $x_i - \mu$, where μ is the mean of the set of values:

$$\sigma^2 = E[(X - \mu)^2] = \frac{1}{n}\sum_{i=1}^{n}(x_i - \mu)^2$$

and the standard deviation σ is just the square root of the variance. So we can take the cue from using gossip algorithms to aggregate the mean or average of a set of values being held locally at nodes and distributed throughout the

network. Here we can use the same idea, because all we need is to estimate the average of a set of values, only each value is not x_i but $(x_i - \mu)^2$ and μ can already be estimated by using gossip algorithm.

The process must have two phases. In the first phase, each node holds one or more edge weights x_i, and gossip is used for each node to estimate the mean μ of all the edge weights. In the second phase, every node computes $(x_i - \mu)^2$ and the gossip algorithm is executed once again for the nodes to estimate the mean of these values, which is precisely the variance; an estimate of the standard deviation σ can be simply calculated by square rooting the estimate of the variance. Once every node has estimates of the mean μ and the standard deviation σ, these values can be used to determine if a given edge incident to the node is a high-risk bottleneck edge, by checking if the edge weight is greater than $\mu + k \cdot \sigma$, i.e., more than k standard deviations higher than the average. In our simulations, we have found that values of k that are effective in determining the high-risk edges are around 6 or 7.

Pareto distribution of edge weights

Another interesting observation we have made in our simulations is about the *distribution of edge weights* after they have been accumulated from the randomly selected shortest paths. For instance, examine the *cumulative distribution function (CDF)* of the edge weights after 20 shortest paths have been executed in a topology of 200 nodes, in one of our simulations, presented in Figure 6.9.

The shape of the CDF seems to suggest that the distribution of edge weights may follow the Pareto distribution. Pareto distribution is another term for power-law, it is used in the fields of statistics and probability. The cumulative distribution function (CDF) of a Pareto random variable X is

$$CDF(x) = 1 - \left(\frac{x}{x_m}\right)^{-\alpha},$$

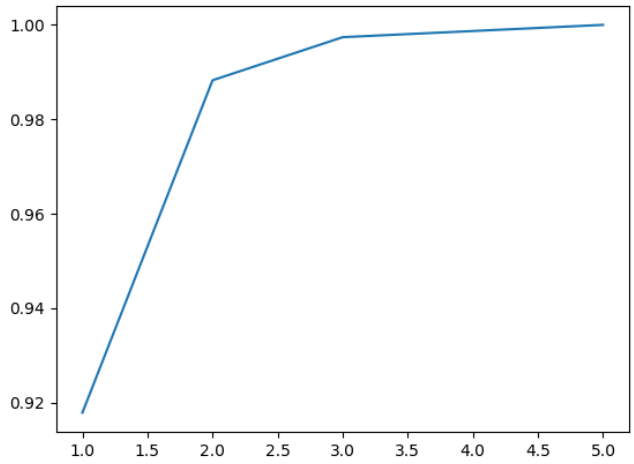

FIGURE 6.9: CDF of edge weights after shortest paths computation in a 200-node topology, with a couple of high-risk edges

where x_m is the minimum value that X can take, and both x_m and the parameter α determine the location and shape of a particular Pareto distribution.

In order to verify whether our suspicion is true or not, we can use a commonly used method for determining if a given sample of data values follow a Pareto distribution. To see how this works, consider the complementary CDF (CCDF) of the Pareto distribution:

$$y = 1 - CDF(x) = \left(\frac{x_m}{x}\right)^{\alpha} \log y \approx -\alpha(\log x - \log x_m)$$

So by plotting $\log y$ against $\log x$, we would get a straight line with the slope of $-\alpha$ and y-intercept of $\alpha \log x_m$.

We take the CCDF of our edge weights and generate a log-log plot of it, shown in Figure 6.10. It appears to be a straight line, which confirms our conjecture that the edge weights follow a Pareto distribution. This is a very interesting observation, because we know that the random network topology that was generated follows a Pareto (or power-law) distribution in the *node degrees*, and now we have demonstrated that, if we were to have a couple of high-risk bottleneck edges in this topology, and monitor normal traffic (in

this case, we mean every pair of nodes have equal probability in exchanging messages with each other) in this network, we would find that the traffic traversing the edges would also follow a Pareto distribution.

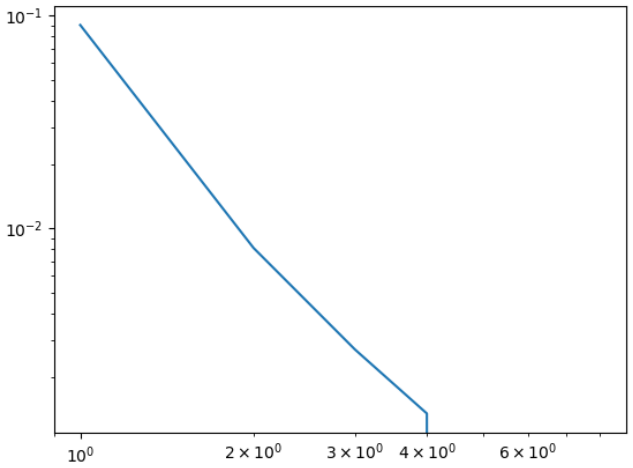

FIGURE 6.10: Log-log plot of the complementary CDF (CCDF) of dge weights after shortest paths computation in a 200-node topology, with a couple of high-risk edges

Chapter 7

Conclusions and future work

We have developed implementations of random walks, randomized gossip algorithms, and the maintenance operation of disseminating membership lists in peer-to-peer networks. These randomized algorithms are extremely important in distributed systems and applications, and our implementation allows us to study them in the most realistic context. We based our implementations on the Docker virtual container technology which closely approximates real end systems in having an underlying operating system and network stack. We conducted experiments using the implementations and evaluated the behaviour and performance of these randomized algorithms. The peer-to-peer network topologies we constructed were also faithful to the topology of real-world networks. We also studied the problem of identifying high-risk bottleneck links whose failure would lead to the disastrous state of network partition. We proposed a randomized method that involves gossip algorithms and randomized shorte-path computation, and verified its effectiveness in our simulations.

The implementations we have developed using Docker and the Go language are advantageous in that they are instantly deployable in real systems. There are many more randomized algorithms that could be studied and experimented with using the same implementation testbed we have developed. Building on the same testbed, we could investigate other important

distributed algorithms such as minimum spanning tree construction for efficient multicasting, and finding not only the minimum cut but also the exact edges that the minimum cut is composed of — this would aid the network administrative functionality of load balancing and network monitoring for resilience after link or node failures. We could also investigate gossip algorithms for other types of data aggregation.

Bibliography

[1] Romas Aleliunas et al. “Random walks, universal traversal sequences, and the complexity of maze problems”. In: *20th Annual Symposium on Foundations of Computer Science (sfcs 1979)*. IEEE. 1979, pp. 218–223.

[2] Tuncer Can Aysal et al. “Broadcast gossip algorithms for consensus”. In: *IEEE Transactions on Signal processing* 57.7 (2009), pp. 2748–2761.

[3] Albert-Laszlo Barabasi and Reka Albert. “Emergence of scaling in random networks”. In: *science* 286.5439 (1999), pp. 509–512.

[4] Burton H Bloom. “Space/time trade-offs in hash coding with allowable errors”. In: *Communications of the ACM* 13.7 (1970), pp. 422–426.

[5] Flavio Bonomi et al. “An improved construction for counting bloom filters”. In: *Algorithms–ESA 2006*. Springer, 2006, pp. 684–695.

[6] Stephen Boyd et al. “Randomized gossip algorithms”. In: *IEEE/ACM Transactions on Networking (TON)* 14.SI (2006), pp. 2508–2530.

[7] Andreas Briese. *bbloom: a bitset bloom filter for go/golang*. https://github.com/AndreasBriese/bbloom.

[8] Andrei Broder and Michael Mitzenmacher. “Network applications of bloom filters: A survey”. In: *Internet mathematics* 1.4 (2004), pp. 485–509.

[9] Colin Cooper and Alan Frieze. “The cover time of random regular graphs”. In: *SIAM Journal on Discrete Mathematics* 18.4 (2005), pp. 728–740.

[10] Colin Cooper and Alan Frieze. "The cover time of the preferential attachment graph". In: *Journal of Combinatorial Theory, Series B* 97.2 (2007), pp. 269–290.

[11] George F Coulouris, Jean Dollimore, and Tim Kindberg. *Distributed systems: concepts and design*. pearson education, 2005.

[12] Alan Demers et al. "Epidemic algorithms for replicated database maintenance". In: *ACM SIGOPS Operating Systems Review* 22.1 (1988), pp. 8–32.

[13] Marco Dorigo and Christian Blum. "Ant colony optimization theory: A survey". In: *Theoretical computer science* 344.2-3 (2005), pp. 243–278.

[14] Li Fan et al. "Summary cache: a scalable wide-area web cache sharing protocol". In: *IEEE/ACM Transactions on Networking (TON)* 8.3 (2000), pp. 281–293.

[15] Uriel Feige. "A tight lower bound on the cover time for random walks on graphs". In: *Random Structures and Algorithms* 6.4 (1995), pp. 433–438.

[16] Uriel Feige. "A tight upper bound on the cover time for random walks on graphs". In: *Random Structures and Algorithms* 6.1 (1995), pp. 51–54.

[17] Wu-chang Feng et al. "The BLUE active queue management algorithms". In: *IEEE/ACM Transactions on Networking (ToN)* 10.4 (2002), pp. 513–528.

[18] Michael L Fredman and Robert Endre Tarjan. "Fibonacci heaps and their uses in improved network optimization algorithms". In: *Journal of the ACM (JACM)* 34.3 (1987), pp. 596–615.

[19] Agata Fronczak, Piotr Fronczak, and Janusz A Hołyst. "Average path length in random networks". In: *Physical Review E* 70.5 (2004), p. 056110.

[20] Ayalvadi J Ganesh, A-M Kermarrec, and Laurent Massoulié. "Peer-to-peer membership management for gossip-based protocols". In: *IEEE transactions on computers* 52.2 (2003), pp. 139–149.

[21] Michelle Girvan and Mark EJ Newman. "Community structure in social and biological networks". In: *Proceedings of the national academy of sciences* 99.12 (2002), pp. 7821–7826.

[22] Christos Gkantsidis, Milena Mihail, and Amin Saberi. "Random walks in peer-to-peer networks". In: *IEEE INFOCOM 2004*. Vol. 1. IEEE. 2004.

[23] Adrien Guille and Hakim Hacid. "A predictive model for the temporal dynamics of information diffusion in online social networks". In: *Proceedings of the 21st international conference on World Wide Web*. ACM. 2012, pp. 1145–1152.

[24] Ali E Abdallah Cliff B Jones and Jeff W Sanders. "Communicating sequential processes". In: (2005).

[25] David Kempe, Alin Dobra, and Johannes Gehrke. "Gossip-based computation of aggregate information". In: *44th Annual IEEE Symposium on Foundations of Computer Science, 2003. Proceedings.* IEEE. 2003, pp. 482–491.

[26] David Kempe and Jon Kleinberg. "Protocols and impossibility results for gossip-based communication mechanisms". In: *The 43rd Annual IEEE Symposium on Foundations of Computer Science, 2002. Proceedings.* IEEE. 2002, pp. 471–480.

[27] David Kempe, Jon Kleinberg, and Alan Demers. "Spatial gossip and resource location protocols". In: *Journal of the ACM (JACM)* 51.6 (2004), pp. 943–967.

[28] Scott Kirkpatrick, C Daniel Gelatt, and Mario P Vecchi. "Optimization by simulated annealing". In: *science* 220.4598 (1983), pp. 671–680.

[29] Donald E Knuth. *Art of computer programming, volume 1: Fundamental algorithms*. Addison-Wesley Professional, 2014.

[30] Abhishek Kumar, Jun Xu, and Jia Wang. "Space-code bloom filter for efficient per-flow traffic measurement". In: *Selected Areas in Communications, IEEE Journal on* 24.12 (2006), pp. 2327–2339.

[31] Ching Law and K-Y Siu. "Distributed construction of random expander networks". In: *IEEE INFOCOM 2003. Twenty-second Annual Joint Conference of the IEEE Computer and Communications Societies (IEEE Cat. No. 03CH37428)*. Vol. 3. IEEE. 2003, pp. 2133–2143.

[32] Qin Lv et al. "Search and replication in unstructured peer-to-peer networks". In: *Proceedings of the 16th international conference on Supercomputing*. ACM. 2002, pp. 84–95.

[33] Michael Mitzenmacher. "Compressed bloom filters". In: *IEEE/ACM Transactions on Networking (TON)* 10.5 (2002), pp. 604–612.

[34] Mark EJ Newman, Steven H Strogatz, and Duncan J Watts. "Random graphs with arbitrary degree distributions and their applications". In: *Physical review E* 64.2 (2001), p. 026118.

[35] Jae Dong Noh and Heiko Rieger. "Random walks on complex networks". In: *Physical review letters* 92.11 (2004), p. 118701.

[36] Ori Rottenstreich, Yossi Kanizo, and Isaac Keslassy. "The variable-increment counting Bloom filter". In: *INFOCOM, 2012 Proceedings IEEE*. IEEE. 2012, pp. 1880–1888.

[37] Antony Rowstron and Peter Druschel. "Pastry: Scalable, decentralized object location, and routing for large-scale peer-to-peer systems". In: *IFIP/ACM International Conference on Distributed Systems Platforms and Open Distributed Processing*. Springer. 2001, pp. 329–350.

[38] Alex C Snoeren et al. "Single-packet IP traceback". In: *IEEE/ACM Transactions on Networking (ToN)* 10.6 (2002), pp. 721–734.

[39] Ion Stoica et al. "Chord: A scalable peer-to-peer lookup service for internet applications". In: *ACM SIGCOMM Computer Communication Review* 31.4 (2001), pp. 149–160.

[40] Spyros Voulgaris, Daniela Gavidia, and Maarten Van Steen. "Cyclon: Inexpensive membership management for unstructured p2p overlays". In: *Journal of Network and systems Management* 13.2 (2005), pp. 197–217.

[41] Duncan J Watts and Steven H Strogatz. "Collective dynamics of small-world networks". In: *nature* 393.6684 (1998), p. 440.

[42] Ben Y Zhao et al. "Tapestry: A resilient global-scale overlay for service deployment". In: *IEEE Journal on selected areas in communications* 22.1 (2004), pp. 41–53.

[43] Qi George Zhao et al. "Finding global icebergs over distributed data sets". In: *Proceedings of the twenty-fifth ACM SIGMOD-SIGACT-SIGART symposium on Principles of database systems*. ACM. 2006, pp. 298–307.

[44] Ming Zhong, Kai Shen, and Joel Seiferas. "Non-uniform random membership management in peer-to-peer networks". In: *Proceedings IEEE 24th Annual Joint Conference of the IEEE Computer and Communications Societies.* Vol. 2. IEEE. 2005, pp. 1151–1161.