Anonymity Properties of Two Network Coded Gossip Protocols

by

Colleen A. Josephson

S.B., Electrical Engineering and Computer Science, M.I.T., 2013

Submitted to the
Department of Electrical Engineering and Computer Science
in partial fulfillment of the requirements for the degree of

Masters of Engineering in Electrical Engineering and Computer Science

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

September 2014

Author . . . . . .
Department of Electrical Engineering and Computer Science
August 22, 2014

Certified by . . . . . .
Muriel Médard
Cecil H. Green Professor of Electrical Engineering, Thesis Supervisor
August 22, 2014

Accepted by . . . . . .
Prof. Albert R. Meyer
Chairman, Masters of Engineering Thesis Committee

Anonymity Properties of Two Network Coded Gossip Protocols

by

Colleen A. Josephson

Submitted to the Department of Electrical Engineering and Computer Science
on August 22, 2014, in partial fulfillment of the
requirements for the degree of
Masters of Engineering in Electrical Engineering and Computer Science

Abstract

This thesis documents the design and implementation of a new anonymous communications protocol, and an analysis of an existing protocol. NCGAB, proposed by Sergeev in 2013, efficiently implements broadcast over unicast and requires no pre-existing infrastructure. We propose a second protocol, CHAP, which extends NCGAB and is designed to use wireless broadcast capabilities as well as wired links. We show anonymity for some information-theoretic measures under certain assumptions regarding adversaries and traffic independence. Numerical results show that for some networks NCGAB fully anonymizes up to 90% of messages, with the remaining 10% having strong anonymity properties. NCGAB also improves up to 30% upon the baseline anonymity provided by a network coded gossip protocol not optimized for anonymity. We compare CHAP to NCGAB and show that CHAP is at least as anonymous as NCGAB and also exhibits interesting hierarchical separability that allows multiple anonymity protocols to operate simultaneously in different domains.

Thesis Supervisor: Muriel Médard
Title: Cecil H. Green Professor of Electrical Engineering

Acknowledgments

I thank my advisor, Prof. Muriel Médard, who worked hard to find me a place when I needed one. Her flexibility, understanding, and expertise made it possible for me to write this thesis with maximal enjoyment and minimal stress and anxiety, while still producing a document I am proud of. I also thank Prof. Moshe Schwartz, my co-advisor, who provided interesting ideas and helpful insights into academic culture. I wish him the best of luck in his new position at Ben-Gurion University in Be'er Sheva, Israel. I thank Kyle Miller for his unwavering support. He listened when I needed and ear, and provided helpful advice of both a technical and non-technical sort. Finally, I thank my friends and family who never doubted me and provided support and advice in various ways.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Security in computer networking has long been a concern, steadily increasing in importance as more and more aspects of our lives move online. Banking information, political views, and even a simple conversation are all things that we do not want others to eavesdrop upon. When people think about security, they often turn first to the idea of hiding *what* they are sending. This typically involves some sort of encryption technique so that only authorized parties can view the intended message. An often overlooked facet is anonymity, which aims to obscure *who* you are talking to. Anonymity is important because even if the content of your communications are hidden, the simple fact that two parties communicate at all can be very valuable information. For example, a government could keep tabs on who communicates with certain political dissidents and arrest those citizens, despite not knowing what they were saying to each other.

This thesis describes two anonymous broadcast protocols that use network coding. The first, NCGAB [20] (Network Coded Anonymous Gossip Protocol) efficiently implements broadcast over unicast and requires no pre-existing infrastructure. We propose a second, CHAP (Coded Heterogeneous Anonymity Protocol) that extends NCGAB and has wireless ad-hoc components as well as the ability to use wired links.

A heterogeneous network is a network that contains different types of nodes or uses multiple different protocols or communication media. For example, a campus LAN that contains both wired nodes and wireless nodes can be considered a heterogeneous

network.

An ad-hoc network is a decentralized wireless network that does not rely on existing Internet infrastructure to route packets. In most networks today, if two laptops in the same room want to communicate, they both forward their packets to a wireless access point to have them routed to the final destination. In an ad-hoc network, the router is bypassed, and the packets are sent directly to the destination, or another node that knows how to reach the destination. Ad-hoc networks are useful for situations where an adversary controls the network infrastructure. However, they require geographic proximity.

What if two groups of political dissidents wish to communicate, but they are hundreds of miles apart? Is there a way to use the infrastructure while still benefiting from ad-hoc properties? CHAP solves this problem by creating an anonymous overlay network on top of wireless routers. Nodes in the ad-hoc clouds can communicate with the router to reach other clouds.

In Chapter 2, we discuss relevant background knowledge, such as previous work on anonymity and what network coding is.

In Chapter 3 we introduce CHAP, the Coded Heterogeneous Anonymity Protocol. It is a network coded gossip protocol that has wireless ad-hoc components to use the capabilities of wireless networks, as well as wired components to increase the utility and range. This chapter also presents an in-depth theoretical analysis of the anonymity CHAP provides and presents other interesting properties.

Chapter 4 describes how the hierarchical nature of CHAP allows for flexibility inside the cloud without affecting the anonymity. It proposes a number of possible schemes to use, focusing on simplicity, throughput, or anonymity against in-cloud observers.

Chapter 5 describes the CHAP simulator, and how we use it to measure the anonymity CHAP provides. These measurements are compared against the theoretical bounds presented in chapter 3.

In chapter 6 we provide a summary of the work described in the thesis and its impact, and suggest future work.

Chapter 2

Background

2.1 Related Work

Anonymous communications seek to hide the source and destination of packets. Prior work can roughly be divided into three approaches: source address re-writing, broadcast, and DC-nets.

2.1.1 Source address re-writing

Source address re-writing obscures the true source address by using relay nodes to forward data for an originator. This is the most widely known anonymity technique. It is used in the popular Tor [9] network, and many others. Tor's Onion routing [18] is a multi-hop source address re-writing system that hides the address of intermediate relays under layers of encryption (see Fig. 2-1). At each relay, one layer is decrypted so the node can reveal the next forwarding address.

One weakness of this system is that it requires trusted Samaritan nodes to operate. If the traffic goes through nodes that are not trusted, an adversary can do traffic analysis to try and de-anonymize the source. Tor is *circuit switched* which means the packets in a session all travel the same path across the relays. An adversary knows both the source and destination if it can follow the path. Even if the adversary only knows the entry and exit nodes, it can do a timing analysis attack [23].

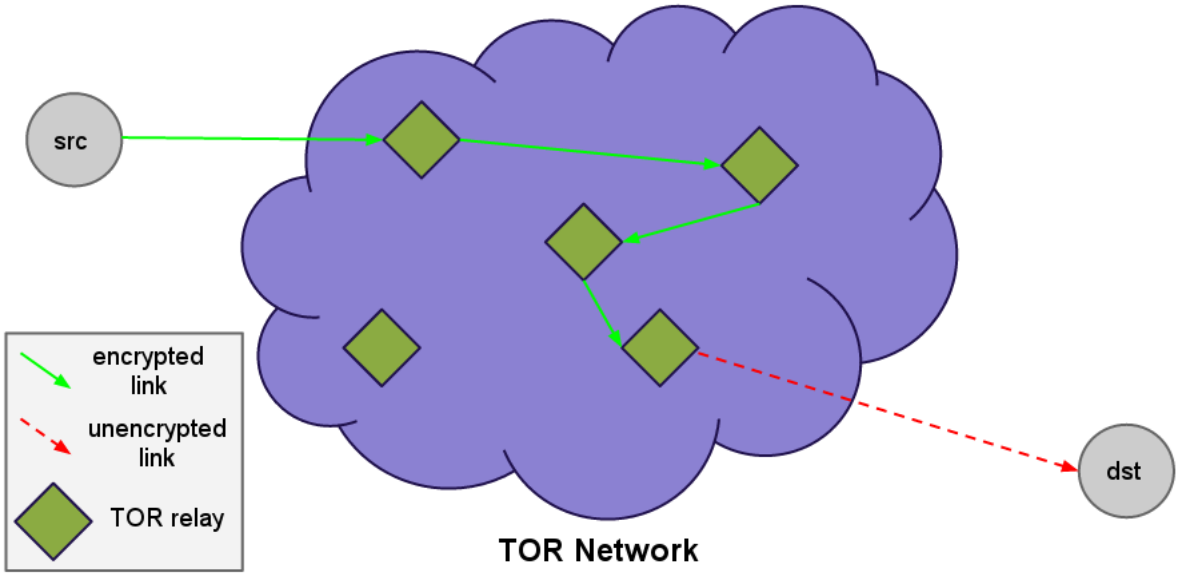


Figure 2-1: A graphical overview of how Tor works

Some attempts have been made to remove the need for trusted infrastructure. Katti et. al. in [14] use *information slicing* over a layered subgraph to remove the need for a trusted public key infrastructure and still provide anonymity as long as the adversary cannot intercept all channels. Chang et. al. expand upon this in [5], and use network coding (see Section 2.2) and other optimizations over a peer-to-peer overlay network to more formally characterize the information theoretic security.

Mixnets, proposed by Chaum in 1981 in [6] also route packets through a fixed path using layers of encryption, but it employs an additional measure against traffic analysis. Instead of the relays being FIFO queues, they collect some number of packets and forward them in a random order. This makes the end-to-end time non-constant, which makes traffic pattern matching difficult. Onion routing omits the mixing step despite the increased anonymity because it adds latency.

Although Tor is the most well-known source address re-writing anonymity system, many others exist such as the I2P [2] overlay network, Hordes [15], Tarzan [10], Crowds [19], and the single-hop anonymizing proxy Anonymizer [1].

2.1.2 Broadcast

Broadcast networks deliver messages to all participants. No receiving addresses are specified, so the recipients are intrinsically anonymous. One approach to hiding source addresses is to use dummy transmissions, as used by the P^5 protocol [21]. Dummy traffic makes it difficult to correlate two communicating nodes. If all nodes appear to be broadcasting at a constant rate, it is difficult or impossible for an observer to know when a message from one node spawns a response from another. Introducing dummy traffic makes traffic analysis more difficult, but it introduces undesirable overhead.

One problem with broadcast protocols is that they are difficult to implement in a unicast environment, as the internet currently is. Some programs, like Hordes [15], try to leverage the multicast functionality of IPv4. The most popular approach is building an overlay network, where messages are unicast to a subset of the participants.

Another approach to broadcast is gossip protocols, also known as epidemic protocols, where nodes re-transmit received messages in order to propagate the message further away from the starting node [13]. Network coding, discussed further in Section 2.2, sends linear combinations of messages instead of singular messages. This increases efficiency, as incoming packets are likely to have some new information if the packets were re-coded before re-transmission, while a non-coded gossip will almost always have some portion of the recipients receiving completely redundant information. NCGAB, introduced in [20] by Ivan Sergeev, is one example of an anonymous network coded gossip protocol.

The schemes above are designed specifically for communication across a wired medium. In [22] the authors explore the use of network coding on a wireless scheduling scheme for session anonymity. Overall, though, there has been relatively little research in the area of leveraging the inherently broadcast nature of wireless, and even less in the area of heterogeneous networks that feature both wired nodes and anonymous broadcast clouds.

2.1.3 DC-Nets

Dining cryptographer networks [7], known as DC-nets, use the XOR computation to anonymously communicate information. The seminal example is that three cryptographers are eating dinner, and are informed that someone has paid for dinner. This could be either the NSA or one of the cryptographers. To figure out who paid, they use a two stage protocol involving shared secrets in the first stage, and XORs or negated XORs of the shared secrets in the second stage. Herbivore [11] uses a combination of broadcast and DC-net techniques.

DC-nets did not spawn as much future work as Chaum's mixnet idea, perhaps because they are vulnerable to jamming by malicious players. They have received regenerated interest in recent years, however.

2.2 Network Coding

Network coding, first introduced in [3], is a procedure where nodes in a network combines multiple packets together instead of forwarding the unmodified packets individually (as is the current practice in typical networks). Linear network coding, unsurprisingly, combines the packets linearly, e.g. $5p_1 + 2p_2 + 3p_3$. Random linear network coding (RLNC) combines packets using randomly generated coefficients. RLNC is useful because coefficients can be generated using a decentralized algorithm, which is easier to implement and more robust. With high probability these combinations are linearly independent of previously received packets, or *innovative*. If an incoming packet is not innovative, that indicates that the new packet is merely a combination of packets that the node has seen before. In this case, the receiver continues collecting coded packets, or *degrees of freedom*, until there are enough to decode.

Network coding has many useful applications. Ho et. al. found in [12] that RLNC achieves the maximum throughput of a multicast network. It can also make the network more resilient to failures [16], and safeguard against eavesdropping [26]. In this research, we will be using network coding to increase anonymity and improve throughput.

Chapter 3

CHAP

3.1 Overview

Coded Heterogeneous Anonymity Protocol, or CHAP, is a network coded gossip protocol that has wireless ad-hoc components to use the capabilities of wireless networks, as well as wired components to increase the utility and range of the network.

The project grew from the desire use these properties of wireless networking to improve anonymity: the inherently broadcast nature of wireless communications, and the necessity of proximity to hear these communications. If many wireless units gather together, they can communicate device-to-device in ad-hoc mode, bypassing the internet infrastructure. This would require any adversary to be physically nearby to eavesdrop, which makes eavesdropping much more difficult.

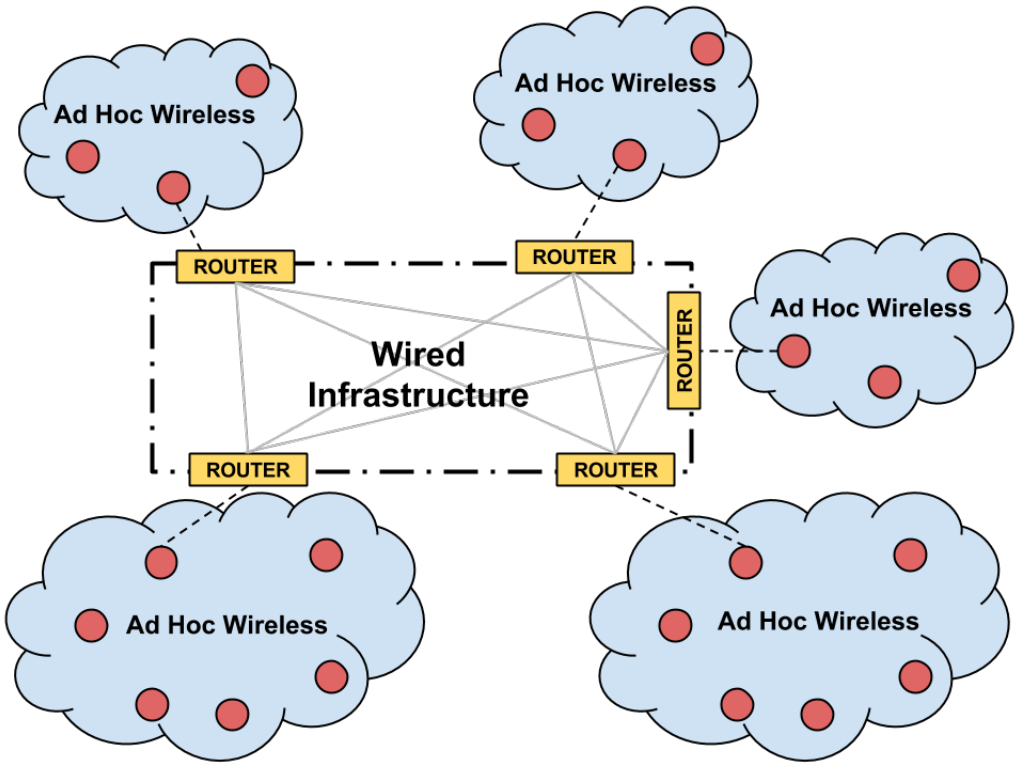


Figure 3-1: Example CHAP topology, with five ad-hoc clouds.

3.2 Design

A CHAP network is composed of *clouds* of wireless nodes operating as an ad-hoc network. This makes it impossible for an adversary to eavesdrop upon the participants inside the cloud unless it is physically present.

Although the physical proximity requirement makes eavesdropping difficult, it also limits the usefulness of the protocol, as it is desirable to communicate with people who are far away. This problem is solved by interconnecting the clouds. Clouds can connect to other clouds by designating one node as an *exit node*. The exit node communicates with the wireless access point to reach other clouds (see Fig. 3-2). All packets that enter or exit the cloud must go through the exit node.

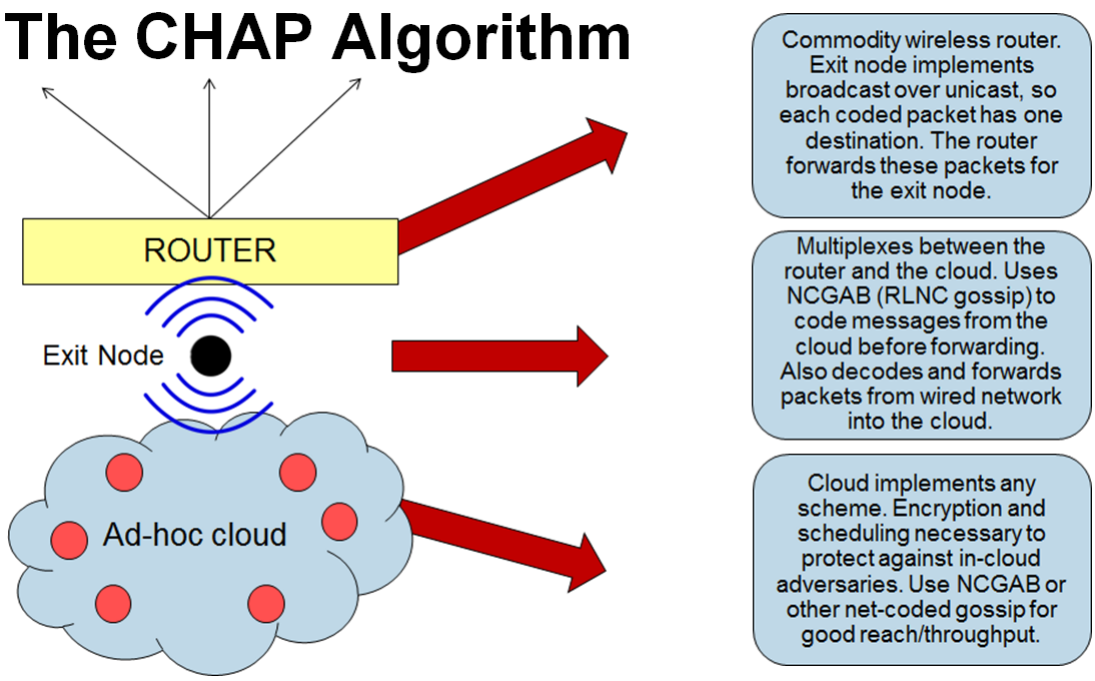


Figure 3-2: A graphical overview of the CHAP algorithm

The exit nodes are reachable over the Internet, which makes eavesdropping easier. The nodes behind the exit node are still invisible, though, unless an eavesdropper is physically present. In this way, the exit node acts as a scapegoat. We assume that the exit node is behind a network whose participation in an anonymity network does not compromise anonymity, such as publicly available wireless at a library or cafe. In networks like these, a NAT (network address translator) [24] is primarily for allowing multiple users to gain internet connectivity behind a single publicly reachable IP address, but has a side benefit that it avoid exposing the users directly to the Internet. This further reduces liability for the exit node.

The exit nodes communicate with each other via NCGAB, a peer-to-peer anonymous broadcast overlay network. NCGAB uses network coded gossip to broadcast to the participants, and is designed for networks of 10 to 100 nodes. CHAP is optimal when participants are interested in most or all of what other peers are contributing, because all contributions are broadcast to all peers. It is still possible to use the network in a situation where a peer is only interested in messages from one or two other peers, but it is still obliged to receive and gossip packets it is not necessarily

interested in.

The non-exit node, or *internal nodes*, of a cloud can use any scheme, but we suggest a network coded gossip broadcast scheme designed for high throughput and successful dissemination. Which specific protocol to use depends on the expected reach of the adversary and the performance requirements. This is discussed further in Chapter 4.

CHAP operates in the application layer and uses UDP. TCP would add unnecessary overhead, as network coding does not need acknowledgments for reliable delivery since participants request additional degrees of freedom if there are not enough packets to decode.

3.2.1 Definitions

Sender: The node that originally contributes a message to the network. A node that is gossiping a message that originated elsewhere is not the sender.

Receiver: The intended destination of a message, which could be all nodes.

Receiver anonymity: An observed message cannot be correlated with its intended destination. Broadcast networks have inherent receiver anonymity because all participants receive all messages.

Sender anonymity: An observed message cannot be correlated with its original sender.

Availability: Average percentage of peers that successfully receive and decode an inserted message.

Delay: The average number of gossip rounds it takes for an inserted message to be decoded by a peer, given success.

Message: An uncoded unit of data. CODE_SIZE of these are eventually encoded together using random linear network coding into a coded packet.

Packet: A unit of encoded data, made up of a random linear combination of CODE_SIZE messages.

Gossip: A message or packet that is being re-transmitted to spread it through the network.

3.2.2 NCGAB

This section describes the original implementation and application of NCGAB for a network of wired nodes. CHAP's exit nodes participate in NCGAB.

NCGAB is a network coded gossip anonymous broadcast protocol. It is an overlay network where the participants broadcast network coded packets to a random subset of peers. Retransmission of previously received messages and packets, or gossiping, is used so that peers more than one hop away can be reached. NCGAB uses random linear network coding, and does not require a cryptosystem.

In addition to forwarding existing linear combinations (*re-gossip*), NCGAB also forms *new-gossip*. NCGAB keeps a pool of decoded messages and mixes these in with new messages to form a never before seen linear combination. This increases the likelihood that gossip will contain useful information even if one of the constituent messages has been previously received.

The pool of previously received messages is occasionally pruned to ensure that it does not grow too large. If messages are not pruned, then the longer the system runs, the less useful the gossip is, since it becomes likely that old and no longer useful packets are being sent instead of new packets.

The CODE_SIZE is the number of messages linearly combined to make a network coded packet. In the default implementation, the coefficients are sent with each packet, which means that anybody who receives enough degrees of freedom can decode it.

NCGAB ensures receiver anonymity because it broadcasts messages to all participants. NCGAB ensures sender anonymity under certain conditions. For an omniscient adversary that can listen to all messages on all links, NCGAB cannot guarantee

anonymity as the adversary can determine which node first introduced a message. For a more limited adversary that can listen to a subset of the peers, and can only observe incoming *or* outgoing packets, but not both, NCGAB ensures some level of anonymity. The adversary cannot pinpoint which peer introduced the message, because it cannot see the entire network. If it observes a message leaving a particular node, it cannot assume that the message originated there, since it might be gossip. The adversary cannot listen to all the links, so it cannot be certain which node introduced the message.

3.3 Analysis

Properly implemented, CHAP can provide a strong level of anonymity. The strength of the anonymity depends on the reach of the adversary. For an adversary only able to listen to one peer, CHAP provides complete anonymity. For omniscient adversaries, CHAP cannot be fully anonymous, but there is an increased level of uncertainty. This section analyzes the anonymity CHAP provides.

We make a number of assumptions for this analysis. We will assume that the adversary can only listen to traffic on the wireline (see Fig. 3-4) in one direction. If an adversary can wiretap both the incoming and outgoing links, it can collect and decode all incoming packets and compare the decoded outgoing packets against this list to figure out whether or not the packet originated from that cloud. If this is an unreasonable assumption with (i.e. the links are inherently bidirectional), then this problem can be mitigated using encryption. Adversaries that can eavesdrop on the wireless communications are briefly discussed in Section 3.3.4. We also assume that the packet contents cannot be correlated to other packets. Since traffic is frequently correlated, this is a rather unrealistic assumption. It can be made more realistic by the use of a cryptosystem, but encryption is expensive computationally and temporally. We also assume that the traffic timing is not correlatable, but this assumption is reasonable as the protocol maintains sending patterns regardless of what it receives from other peers. These assumptions are difficult to guarantee on a real network, but

this simple first order approximation network model allows us to gain fundamental insights about the behavior of CHAP. We hope to relax these assumptions in future work.

The goal of CHAP is to make it impossible for an eavesdropper to tell that a packet originated from the jth node in cloud c_1, and had an intended destination of node l in cloud c_2.

The simplest case is when the network has a single cloud, c, as shown in Figure 3-3. Let $p^c_{j,l}$ be the probability that node j sends a packet to node l. $p^c_{j,l}$ is a function of ω_c, observations of the state made by the adversary from within the cloud. Let $X^c_{j,l}$ be the random variable associated with that probability. Anonymity is maximized when the entropy of $X^c_{j,l}$, $H(X^c_{j,l})$ is maximized. This happens when all outcomes are equally likely, so the source j and the destination l both have a $\frac{1}{N}$ chance of being a particular node, where N is the number of nodes in the cloud. This means that there is a $\frac{1}{\binom{N}{2}}$ chance of a particular pair of nodes being the source and destination of a transmission.

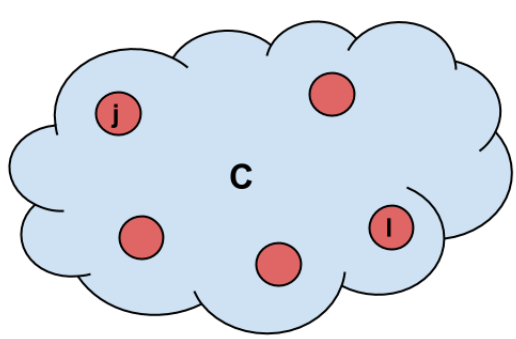


Figure 3-3: A CHAP topology consisting of a single cloud

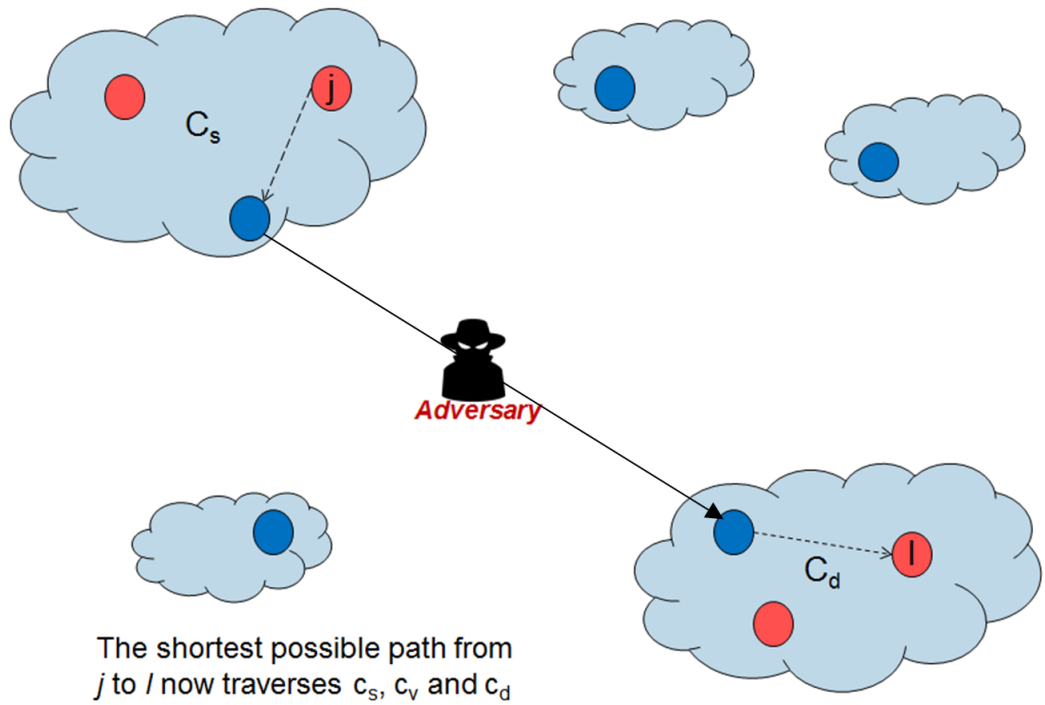


Figure 3-4: The primary threat model CHAP is designed to combat

Extending to cross-cloud communications (see Fig. 3-4), the probability that node j in cloud c_1 sends a packet to node l in cloud c_n, and the packet only traverses these two clouds, is $p_{j,l}^{c_1,c_n}$. If the packet traverses some intermediate clouds on its way to c_n then the probability becomes $p_{j,k,\ldots,l}^{c_1,c_2,\ldots,c_n}$. The entropy $H(X_{j,k,\ldots,l}^{c_1,c_2,\ldots,c_n})$, is equal to the sum of the conditional entropies:

$$H(X_{j,k,\ldots,l}^{c_1,c_2,\ldots,c_n}) = \sum_{i=1}^{n} H(X_j^{c_i} | X_{k,\ldots,l}^{c_1,\ldots,c_{i-1}}) \tag{3.1}$$

Above, $X_j^{c_i}$ is the random variable representing the probability that a node j in cloud c_i will receive a packet. It is worth pointing out that if we know that a packet arrives at a given node, the entropy of that term becomes zero, which decreases the overall joint entropy. The more information the adversary knows, the lower the joint entropy. The minimum entropy corresponds to a bit pipe, where the adversary is certain about every node the packet ever touches. The maximum value of this joint entropy is less than or equal to the sums of the individual entropies:

$$H(X_{j,k,\ldots,l}^{c_1,c_2,\ldots,c_n}) \leq \sum_{i=1}^{n} H(X_{j_i}^{c_i}) \tag{3.2}$$

The maximal entropy occurs when the individual $X_j^{c_i}$s are a uniform distribution, meaning that every node in a cloud is equally likely to be the one sending or receiving the packet. This reduces the probability of a packet's path to a simple expression. If each cloud has N_i nodes, then the probability that an observed packet goes from a particular node in c_1 to a particular node in c_2 is $\frac{1}{N_1 N_2}$.

This maximal entropy can be achieved in CHAP when there is a wireline adversary listening to a single wired link. CHAP's wired portion can be represented as another cloud, c_w, since the fact that the connections are wired does not change the information theoretic analysis. First, we assume that an adversary is only able to observe the traffic on the wired portion. We also assume that an exit node always has traffic available to send. The entropy of a CHAP communication originating from some node in c_s and destined for a node in c_d is $H(X_{j,k,l}^{c_s,c_w,c_d})$. The c_w term is included because all cross-cloud traffic must traverse the wired network.

$$H(X_{j,k,l}^{c_s,c_w,c_d}) = H(X_j^{c_s}|X_{k,l}^{c_w,c_d}) + H(X_k^{c_w}|X_{j,l}^{c_s,c_d}) + H(X_l^{c_d}|X_{j,k}^{c_s,c_w}) \tag{3.3}$$

To achieve the maximal entropy in equation 3.2, the entropy in each cloud must be independent of the other clouds. More intuitively, the traffic patterns in one cloud must not be correlated to the traffic patterns in another cloud. This independence is not unreasonable to assume, because CHAP disperses traffic without making considerations about the destination, and CHAP always has traffic to send. If a node i in c_s wishes to send a packet to a node j in c_d, the source cloud c_s will broadcast the packet to all clouds, not just c_d, therefore the traffic arriving at the clouds is independent of whether or not the intended destination actually resides in the cloud. Furthermore, because the source cloud c_s is always sending traffic, there is no increase in traffic when j inserts the packet, meaning that an adversary cannot easily tell the difference between when c_s is propagating new packets or simply regurgitating old packets. The adversary must inspect and decode each network coded packet, and

even then it is not simple to figure out if the constituent packets are innovative, since any never-before-seen packet could simply be forwarded gossip.

This independence means that the overall entropy is a simple sum of the individual entropies, satisfying 3.2 with equality.

$$H(X_{j,k,l}^{c_s,c_w,c_d}) = H(X_j^{c_s}) + H(X_k^{c_w}) + H(X_l^{c_d}) \tag{3.4}$$

The individual entropies are maximized when the participating node is equally likely to be any node in the network. For the case on an adversary observing a single link, it simple to come up with an expression for $p_{i,j,k}^{c_s,c_w,c_d}$, the probability that a packet originated from node i and is destined for k. If we let N_s be the number of nodes in c_s, N_d be the number of nodes in c_d and N_w be the number of nodes in c_w, then

$$p_{i,j,k}^{c_s,c_w,c_d} = \frac{1}{N_s N_d \binom{N_w}{2}} \tag{3.5}$$

In the equation above, the adversary must treat N_s and N_d as unknowns. Not only is it uncertain what the origin and destination clouds are, but the adversary also has no data on N_i for each cloud, since it is limited to observations on the wireline. Therefore, it is reasonable that Equation 3.5 assumes a uniform distribution for $X_j^{c_s}$ and $X_k^{c_d}$.

3.3.1 Network Coding

In the previous sections, we assumed that each packet has a single target destination. However, since network coding combines CODE_SIZE packets together, each packet now has up to CODE_SIZE destinations. However, since CODE_SIZE degrees of freedom are needed to decode a packet, the amortized analysis is the same as the uncoded case.

3.3.2 Hierarchical Anonymity

The CHAP system is a two-tier hierarchy, where the wireline is the top tier and the ad-hoc clouds are the lower tier (see Fig. 3-5). The CHAP hierarchy allows for limited liability by only requiring one node per cloud to be reachable by Internet. The exit nodes participate in NCGAB, and each ad-hoc node participates in their own internal anonymity protocol.[1] What protocol for in-cloud communications maximizes anonymity? Surprisingly, for a wireline-only adversary, as long as NCGAB is properly implemented, the level of anonymity cannot be improved or worsened by what happens inside the cloud. The nodes each cloud can use a modified NCGAB, implement their own anonymity scheme, or use none at all, and still achieve complete anonymity. This is a powerful statement because it allows us to look at intra-cloud and extra-cloud anonymity as separate problems.

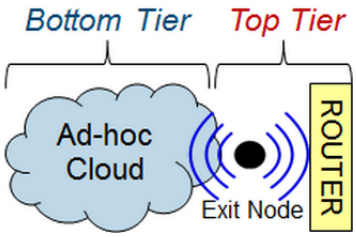


Figure 3-5: A CHAP network is a two-tier hierarchy

As long as the exit node can decode the scheme the rest of the cloud uses, then it can participate in NCGAB properly. If the exit node does not have enough data from the rest of the cloud to code a packet, it can incorporate re-gossip. The rest of the cloud can do anything short of connecting unauthorized to the wireline, including sending nothing at all [2] The clouds activities do not affect how successfully the exit node can implement NCGAB, so for the limited adversary, the anonymity provided by CHAP is at least good as NCGAB's.

[1]The exit nodes must simultaneously satisfy the role of an ad-hoc node and an NCGAB peer. In this way, the exit nodes are like pseudo-wired nodes–although they are not wired, they use the wireless access point to communicate with the wireless network on their behalf.

[2]This assumes that the rest of the network is generating enough traffic. If this is not the case, then the exit nodes can resort to generating packets full of random data. This will reduce throughput, since the data is useless, but will ensure that the protocol continues to operate successfully.

Not only does CHAP have anonymity as least as effective as NCGAB, it actually improves anonymity. This is because an adversary that can only observe the wired portion can only tell that traffic came from some cloud, and not which of the N nodes inside sent it. In fact, an observer cannot even determine the number of nodes in the cloud.

Although the hierarchical nature of CHAP has a positive impact on anonymity, the same is not necessarily true for the delay and availability. The cost of hierarchy will be discussed in Section 5.5.

3.3.3 Omniscient wireline adversaries

If an adversary is able to eavesdrop over all links on the wireline, it can pinpoint which cloud introduces a message. If the adversary is able to eavesdrop to traffic in both directions on any link, then it can pinpoint any traffic that originates from that cloud. For the original NCGAB protocol, this constitutes a loss of sender anonymity. In the case of CHAP, this reduces anonymity, since the adversary knows which exit node a message came from, but the actual origin node of the message is still unknown.

3.3.4 Adversaries inside the cloud

If the adversary is not limited to the wireline, the activities of the cloud do have an effect on anonymity. If the observed cloud has no anonymity scheme, the adversary knows that node k transmitted packet t. Packet t is transformed into coded packet t' at the exit node, as part of the NCGAB implementation. This t' is not necessarily guaranteed to contain t. When the adversary sees t' leave the router, it needs to first decode the packet. If the coefficients are known, then the adversary can decode the packet as soon as it has enough degrees of freedom, and eventually figure out which t' contained t. There is a $\frac{1}{\sum_i N_i}$ chance that packet t is destined to some node j. This system retains receiver anonymity but lose sender anonymity.

If the observed cloud *does* have an anonymity scheme we can simply multiply probabilities. If we let p_i be the probability that observed packet originated from node

k, then the probability of an observed transmission being from k to j is $\frac{1}{p_i N_d (N_w - 1)}$. If $p_i = \frac{1}{N_s}$, then this becomes $\frac{1}{N_s N_d (N_w - 1)}$, which looks very similar to Equation 3.5. The only difference is that the adversary has gained the information of which messages originate from that cloud, and how many nodes are in the clouds (the N_is are no longer unknown).

Considering this, one may wonder why a cloud may choose to forgo an in-cloud anonymity scheme. The core reasons are performance and throughput related. Anonymity is not free. Depending on how likely it is that an adversary can observe the cloud, the participants may choose to accept the increased risk to gain the performance benefits. In Chapter 4 we will look at what types of anonymity schemes are available to ad-hoc wireless networks, and what their costs and benefits are.

Chapter 4

Protocols inside Ad-hoc clouds

4.1 Overview

CHAP allows for flexibility in the clouds. The nodes inside the cloud do not need to implement NCGAB, and can instead use a protocol that is designed for metric of their choosing. The same anonymity will be provided to the wireline, regardless of what scheme the cloud implements.

In this chapter we suggest schemes designed for simplicity, anonymity, or throughput. We only provide code for the default implementation, though. It is a modified version of NCGAB described in Section 4.5.

4.2 Designing for simplicity

Simplicity is attractive because it means that the code is easy to implement and maintain. A simple solution is a great way to verify that the system is operating correctly, and then iteratively improve performance if desired. Below is a non-exhaustive list of simple schemes ordered from what we consider to be easiest to most difficult to implement.

Unicast: Each peer has a list of participating peers and sends them unicast packets. This means that each packet must go to the access point and then be re-sent.

This scheme is how most communication occurs on 802.11 networks, so most languages will have libraries with the bulk of the implementation done.

Broadcast: The ad-hoc peers broadcast each packet to other ad-hoc peers. The success depends on the network structure. If it is a clique network, where all peers can hear all peers, then this scheme will work very well. If not all peers can hear each other, then some peers will always miss what another peer says. This is solved by gossip.

Broadcast Gossip: The same as above, except received packets are re-broadcast. This increases the reach of the network, so that any node can eventually hear what another node says as long as there is not a partition. Broadcast gossip creates a few challenges, primarily with avoiding flooding and inefficiency. One helpful measure is to have a sliding window that expires messages after a certain number of forwards, which prevents data from bouncing around forever.

Network coded gossip: Network coded gossip increases efficiency, since nodes do not forward the packets verbatim, and instead re-code them. This removes the redundancy that is the primary source of inefficiency in gossip networks. NCGAB is an example of a network coded gossip scheme.

4.3 Designing for anonymity

While no actions the cloud takes can improve anonymity on the wireline, anonymity inside the cloud is a different matter. The anonymity NCGAB provides relies on the assumption that the cost of monitoring links increases at least linearly with the number of links. In a wireless network this is not the case, as transmissions are inherently broadcast, so any eavesdropper in monitor mode can easily listen to whatever any nearby node transmits. This means that NCGAB is not an effective anonymity solution for an ad-hoc wireless network. It may provide increased uncertainty for networks that are not cliques, but even a small number of well placed observers can defeat NCGAB.

One common approach to achieve anonymity in an ad-hoc wireless environment is by transmitting regularly at each node so that traffic analysis cannot be done. If a node does not have information to send during its slot, it sends a dummy packet. The wastefulness of sending dummy packets can be solved by using network coding. This approach and the benefits of network coding are described in detail in [22] by Sousa et. al.

One downside is that any sort of anonymity for the cloud requires a cryptosystem. The regular transmission approach makes traffic patterns look uniform, but the adversary can still do deep packet inspection to analyze the sources of requests and responses.

4.4 Designing for throughput

Maximizing the throughput, or even achieving acceptable throughput, is a major challenge for ad-hoc networks. Two typical approaches for improving network-wide throughput are to reduce overhead and to maximize the number of simultaneous transmissions. This spatial multiplexing approach can be combined with the techniques mentioned in 4.3.

For a gossip network, using network coding is a typical way to decrease overhead, since it increases the likelihood that some part of a coded packet is novel. Network coding can also be used instead of TCP or other transport layer protocols, which eliminates TCP overhead.

Spatial multiplexing requires analyzing which nodes can transmit simultaneously without interfering. In a clique network, no two nodes can transmit simultaneously. In more complex networks, two nodes that are physically distant or separated by an obstacle can often transmit simultaneously. Once the set of nodes that can transmit simultaneously is known, a schedule must be created. [17] describes a scheduling scheme for wireless gossip networks to maximize throughput, but it does not pair it with network coding. [22] describes spatial re-use for line networks in detail, and provides pseudo-code for a linear program that can be used to find satisfactory schedules

for arbitrary topologies.

Creating a schedule that yields good spatial re-use is hard. Furthermore, existing research assumes a static topology and the ability to compute the schedule offline. These assumptions are not realistic, however, since many ad-hoc networks have peers frequently joining and leaving and moving around. Possible future work includes coming up with a decentralized method for recomputing the schedule to incorporate topology changes.

4.5 Currently implemented CHAP scheme

Our simulator implementation of CHAP uses a networked coded gossip scheme similar to NCGAB. The nodes transmit in a TDM fashion to avoid collisions. We did not implement a scheme with spatial re-use at this time, as the topologies we work with (cliques, rings and grids) are relatively simple. We hope to look more at spatial reuse in future work.

It is important to note that this scheme, although based on NCGAB, will no longer provide good anonymity in a wireless medium unless a cryptosystem is used. The TDM nature will effectively mask traffic timing, but the contents still need to be hidden to have reasonable anonymity. Our simulator does not implement encryption.

4.6 Conclusion

The ability to separate the clouds from the wired network in CHAP gives implementors the flexibility to optimize for a number of metrics. Exploring these schemes, particularly the use of scheduling, is an interesting set of future work that combines research from wireless networking, mesh networks, and distributed systems.

Chapter 5

Simulation and Results

There are two simulators involved in this project. The original NCGAB simulator in [20] is about 900 lines of Python code, half of which is for plotting and logging. We added about 200 lines to collect statistics on anonymity, and removed 200 lines that generated statistics we no longer needed.

The CHAP simulator, which is based on the NCGAB simulator, is about 1600 lines of code. Among other things, we added support for a diverse set of heterogeneous topologies, the ability to run simulations on multiple topologies in one session, added hooks to simulate an adversary, and extended the plotting library to arbitrarily combine the plots of different datasets on one chart.

There is also supporting code to process the data into plots, and to quickly perform finite field math. In its entirety, the project is about 4000 lines of code. The source code is available in Appendix B.

5.1 Simulator Design

The simulators are object oriented, with the most basic network participant being a Peer. In the CHAP simulator, a Peer can be a WiredPeer or a wireless CloudPeer. A CloudPeer can be further specified as a CloudExitPeer, also known as an exit node. The Cloud object is a container for CloudPeers and one CloudExitPeer. A Network is a container for a set of Clouds. The simulation is run on a Network.

The different sub-types of Peers inherit most functionality from the base Peer class, and differ primarily in the **simulate()** method, which is responsible for sending out messages during each simulation round. Wireless and wired peers have different models for which nodes are reachable. A WiredPeer or CloudExitPeer can reach any other node of the same types. A CloudPeer can only reach its CloudExitPeer or other CloudPeers in the same cloud. For CloudPeers, the set of reachable peers is specified by an adjacency matrix.

The simulations assume that each attempted transmission is always successful. It would not be difficult to add the ability to simulate unreliable links, but it is omitted from this work for the sake of simplicity.

The **send()** function puts a coded packet onto the destination node's input queue. The destination runs the **solve()** function to attempt to decode any unsolved packets. For a WiredPeer or CloudExitPeer, if the packet is decoded, the constituent messages are added to the peer's Decoded Window. Undecoded packets are put into the Gossip Window. When the Peer wants to send a packet, it either forwards a packet as-is from the Gossip Window or creates a new linear combination from the Decoded Window.

The system is run in gossip rounds, where each Peer runs the operations to send and decode. The gossip round executes each Peer's activity in random order each round. The simulator randomly generates message contents, and tracks the origin location and time (in rounds) of all messages to calculate the availability and delay statistics. CloudExitPeers and WiredPeers implement NCGAB. CloudPeers implement a scheme described in Section 4.5.

The simulator also has code that acts as an adversary observer, which allows us to generate PMFs that convey the adversary's guesses about who the senders of the messages are. This work is explained in more detail in Section 5.3.

5.2 NCGAB Broadcast

Through the process of working with and modifying the original NCGAB simulator, we discovered an implementation detail that is somewhat unintuitive and has

important implications for CHAP.

The simulator's broadcast functionality does not actually broadcast the same packet to all peers. Instead, NCGAB sends unique linear combinations to the other peers. The broadcast aspect of NCGAB comes from the fact that all messages are received by all participants eventually, not that each peer sends identical packets out on each link.

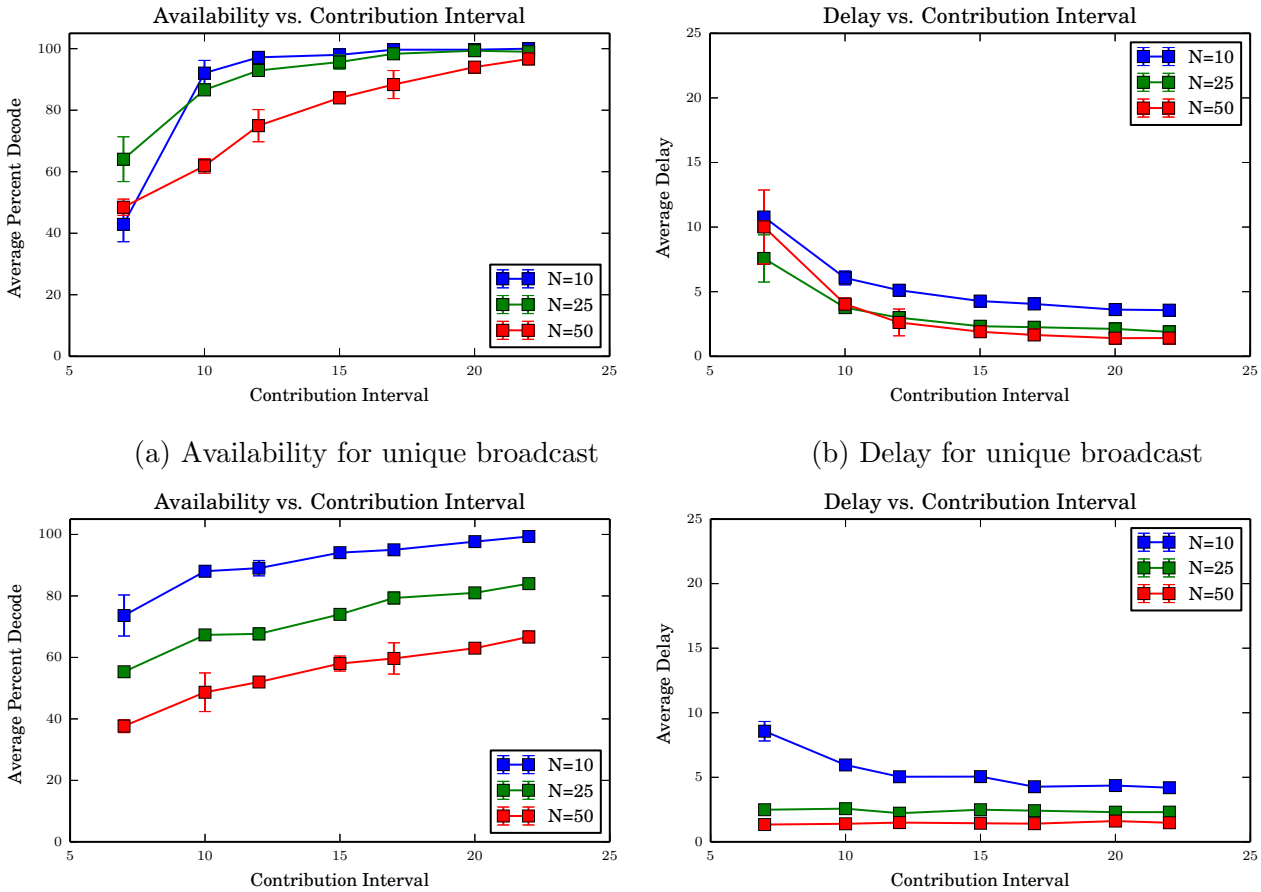
The detail is in lines 853 to 867 in the NCGAB code, found in Appendix A of [20]. It is excerpted below for convenience.

```
# Look up LOOKUP_PERCENT subset of peers on the network
dests = self.network.lookup_random(self.nid,
                                   int(self.simParams[ LOOKUP_PERCENT ]
                                       * self.simParams[ SIM_NUM_PEERS ]))
for d in dests:
    gossip = []
    if random.getrandbits(1):
        # Code new-gossip from our Decoded Window
        gossip.append(RLC(self.decoded_window.choose_random(
                          self.simParams[ CODE_SIZE ])))
    else:
         # Choose re-gossip from our Gossip Window
        if len(self.gossip_window.live_objects()) > 0:
            gossip.append(self.gossip_window.choose_random())

    # Transmit to the destination
    d.put( (self.nid, gossip) )
```

If we run NCGAB where the same packet is send to all peers, we see in Figure Figure 5-1 that it is more sensitive to the number of peers on the network, and is less scalable. The ability to send different linear combinations helps diversity and makes the peers able to decode more reliably, as evidenced by the improved availability.

This indicates that NCGAB may not perform as well on a wireless media such as the ad-hoc clouds, since this media forces the same packet to be transmitted to all peers.



(a) Availability for unique broadcast

(b) Delay for unique broadcast

(c) Availability for non-unique broadcast

(d) Delay for non-unique broadcast

Figure 5-1: The performance differences between sending the same linear combinations to each peer, and sending unique combinations

5.3 Measuring Anonymity

We wish to measure the anonymity CHAP and NCGAB provide. Here we define complete anonymity to be when the adversary's probability distributions for both the sender and the receiver are uniform, given any possible set of observations.

Receiver anonymity is trivial due to the broadcast nature of the protocol. Each peer receives every message. A peer not interested in a message can discard it. However, an adversary can gain information on which recipients were actually interested

in a message if the recipient sends a response that can be correlated to the message. For example, if a peer gets a copy of File A, and then sends a message suggesting edits to the file, then the adversary knows that whoever sent that message received File A and was interested in it. Any system that uses CHAP must be conscious to avoid this, or use a cryptosystem if it is unavoidable.

Sender anonymity is more difficult. Ideally, we want the PMF of possible senders for a message to be uniform, which means that the adversary is maximally uncertain. The PMF moves further away from a uniform distribution the more the reach of an adversary increases. Let the network be made up of N links. If the adversary is only able to eavesdrop upon one link, even if it sees a message come out of the link, the message is equally likely to have emerged from any of the other $N - 1$ links. If the adversary can listen to two links, and hears the same message from two links, it knows that the link that sent the message later was not the original sender since it must be re-gossip. There are still $N - 2$ possible senders though. If the adversary can listen to all but one link, the adversary knows that there are two possible links that originated the message.

Through observations, the adversary can narrow down the pool of possible senders. We have modified the NCGAB simulator to gather statistics on how much anonymity the protocols provide. We do this by creating a new Adversary class of peer, and modifying the existing Peer class to forward messages to the adversary if it is in the adversary's victim list. The software adversary adjusts its PMF continuously to account for observations it sees, and writes them to disk after each round. These PMFs can be used to calculate the entropy and see how close our network gets to the theoretical maximum.

5.3.1 Adversary implementation

The adversary collects and decodes the packets, and periodically calculate the PMFs for senders of particular messages and writes the results to disk. A basic PMF is derived by answering this question:

Given that message m was first observed leaving peer x at round t and last observed m anywhere at time $t + n$, where $n \leq$, `TTL_DECODE`, what is the probability that x was the original sender?

To create a simple PMF, we first must determine the pool of possible senders. Initially, this is all the peers known to the adversary. In our simulations, we assume the adversary is aware of all peers. Using observations, the adversary can further narrow down the pool of senders by figuring out whether or not it witnessed the first transmission of a message.

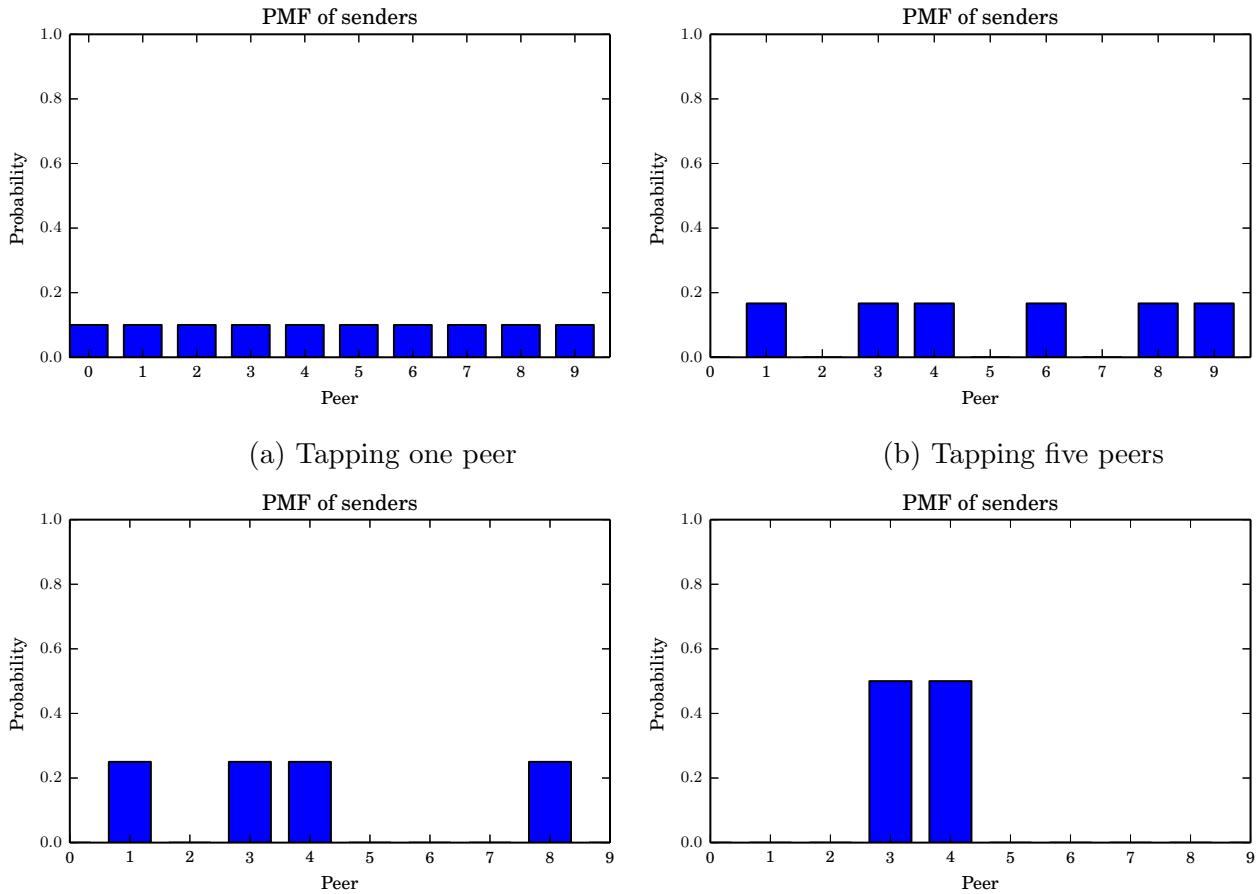
It is possible to immediately disqualify an observation from being the first. If the adversary has seen the message before, or same message is seen leaving more than one node in the same round[1], we know it must not have been the first instance since clearly these nodes are re-gossiping.

Observation 1. *If the adversary knows that it never witnessed the first transmission, then it is forced to use a uniform PMF for the distribution of possible senders. If it is possible that the adversary witnessed the first transmission, then a non-uniform PMF results.*

For the non-uniform PMF, we can immediately eliminate any nodes that are certain did not send the first transmission. The greater the adversary's reach, the more nodes it can eliminate. For example, in a network made up of N peers if the adversary is only able to eavesdrop upon one peer, even if it sees a message come out of the peer, the message is equally likely to have emerged from any of the other $N - 1$ peers so the PMF must be uniform. If the adversary hears the same message from two peers at different times, it knows that the peer that sent the message later was not the original sender since it must be re-gossip. There are still $N - 2$ possible senders though. If the adversary can listen to all but one peer, it knows that there are two possible peers that originated the message and can eliminate N-2 peers from the PMF. After the elimination step, we split the probability evenly across the possible

[1]How to determine rounds in an actual implementation will be much more difficult (or impossible), but since our simulator operates in discrete rounds we can use this rule.

senders. This yields a very predictable PMF[2], shown in Figure 5-2, that a function of how many peers the adversary can observe. Note that these calculations assume that the adversary is able to listen to the nodes for as long as the network has been active and never misses any transmissions.



(a) Tapping one peer

(b) Tapping five peers

(c) Tapping half the peers

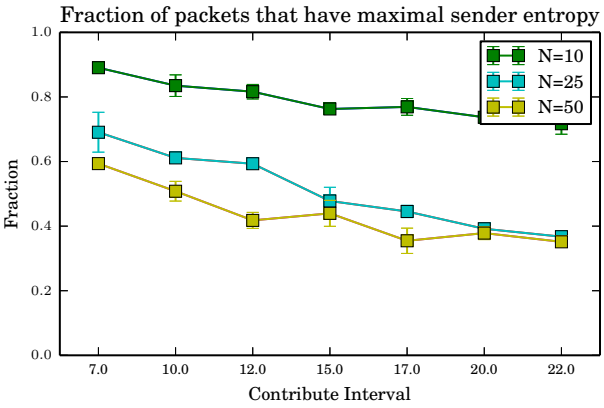
(d) Tapping all but one peer

Figure 5-2: Computed PMFs of senders for a packet in an NCGAB network of ten peers

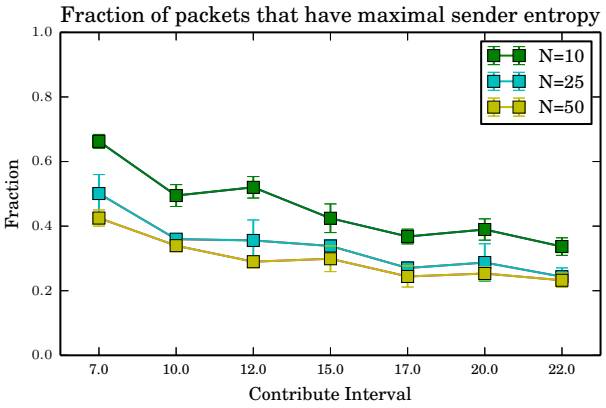
The PMFs for individual messages do not give a large-scale idea of how anonymous the network is. A better statistic is the percent of uniform PMFs. Recall that a uniform PMF occurs when all peers equally likely to be the sender, so this corresponds to maximum possible anonymity. The higher the percentage of uniform PMFs, the more uncertainty the adversary has about senders of packets. Figure 5-3 shows the

[2]This PMF is not directly sample or observation based, but it is derived from observations. A PMF may not be the best name, but we call it such because we lack a better name.

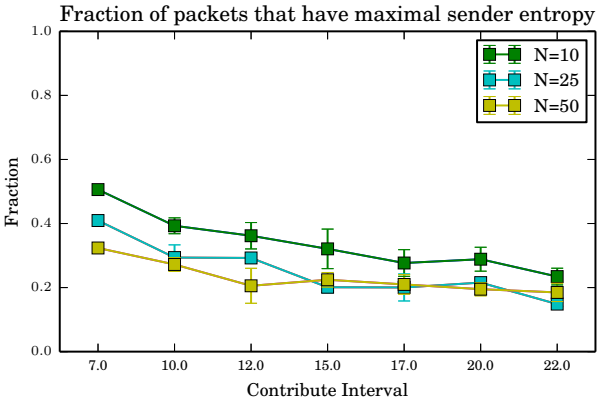
percent uniform PMFs for a variety of adversary reaches and network configurations. The *contribution* is a simulation configuration parameter, and it is the number of rounds a node must wait between contributing their own packets to the "to be coded" pool.



(a) Tapping 25% of peers



(b) Tapping 50% of peers



(c) Tapping 75% of peers

Figure 5-3: Percent of packets with maximal sender entropy for an NCGAB network with a variety of configurations. These statistics were compiled for 100 message simulation runs.

One may be surprised to see that the network of ten nodes seems to have the strongest anonymity. This is because for a larger network, 25% compromised is greater number of compromised peers than for a smaller network. If there is a static number of compromised nodes and a varying network size, then the larger networks perform comparably (see Figure 5-4). The absolute number of compromised nodes is clearly more important than the percent, although larger networks do tend to be

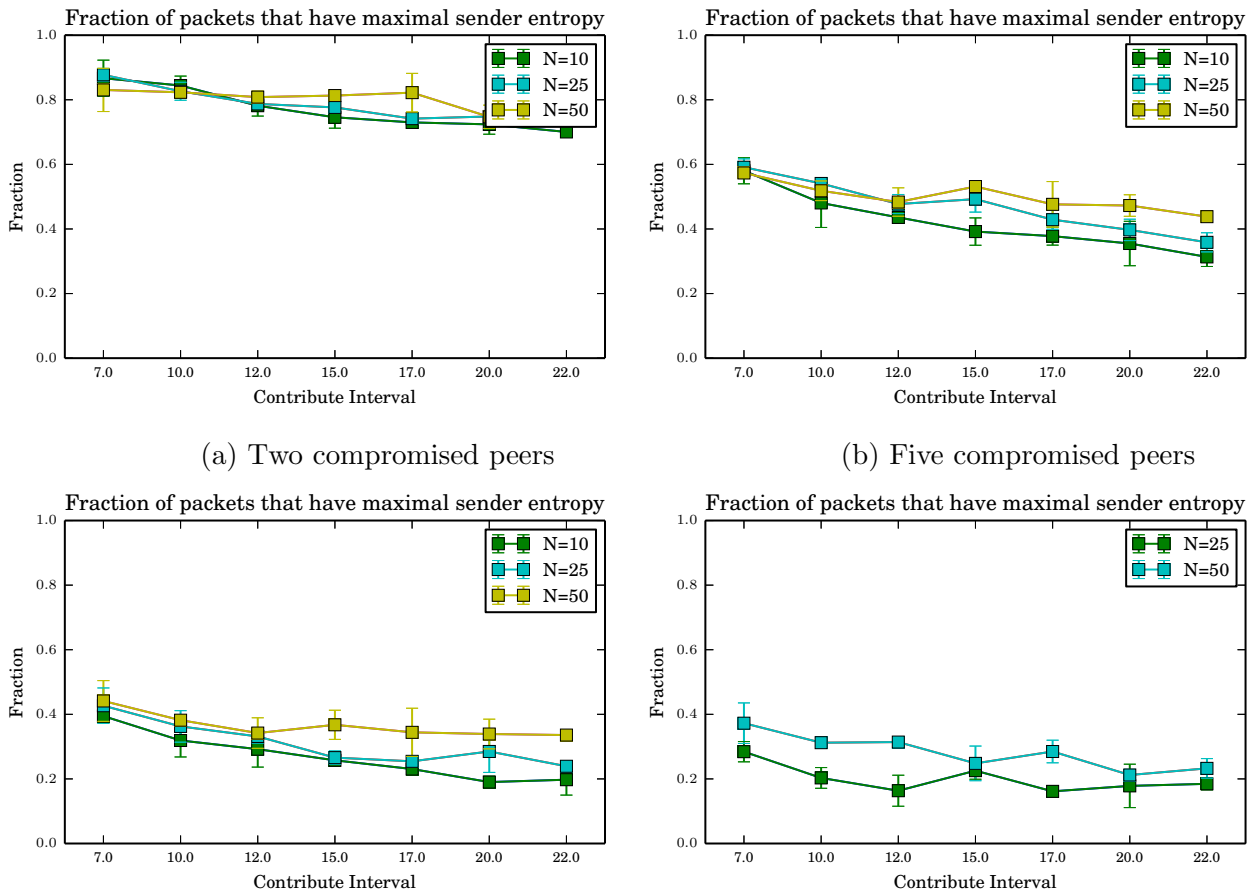
slightly more anonymous than smaller ones for the same number of compromised peers.

It may seem like the network performance degrades rapidly, since even compromising as few as 5 peers leads to only 50% of messages having maximal anonymity. However, even messages without maximal anonymity may still have strong sender uncertainty. Table 5.1 contains the probabilities of the adversary guessing the sender correctly for various network configurations. To guess, the adversary chooses the most likely sender from the PMF. This may be many senders, in which case the adversary chooses randomly among the equally likely options. Recall that the PMF is generated by evenly splitting the probability across a list of eligible senders, and the smallest possible list of eligible senders is a function of how many peers the adversary can listen to. This means that a network with many peers has more ways to split the liability. Thus, for a 50 node network with 25% of the nodes compromised, there will be fewer packets with maximal anonymity, but the adversary's probability of guessing the sender for packets without maximal anonymity will be much lower than for a network of 10 peers.

Probability of adversary guessing sender correctly			
	10 peers	**25 peers**	**50 peers**
25% Compromised	0.125	0.0526	0.026
50% Compromised	0.20	0.0769	0.04
75% Compromised	0.333	0.1428	0.077

Table 5.1: The best-case probability that an adversary guesses a sender correctly for a variety of NCGAB network configuration

It is important to realize that this analysis assumes that the messages are not correlated. However, requests and responses are sent in plain text, and the network coded packets are sent with the coefficients. This means that an adversary who solves the packets is able to do deep packet and potentially correlate packets. For example, if a file is being sent then information about which segment of the file is in the packet is necessary to re-assemble the file. If message passing, it may be necessary to include ordering data or other information to that peers know how messages are related to each other. If the users of the network are sending data that can be correlated, we



(a) Two compromised peers

(b) Five compromised peers

(c) 9 compromised peers

(d) 15 compromised peers

Figure 5-4: Fraction of packets with maximum sender entropy for NCGAB networks with a static number of compromised nodes, instead of a percent. These statistics were compiled for 100 message simulation runs.

recommend setting up a cryptosystem.

Expected packet lifetime

One problem with the PMF from Section 5.3.1 is that once a PMF is made for a particular message, the adversary does not incorporate any of the information from later observations to change the PMF.

One option for incorporating information from additional observations is using the expected packet lifetime in rounds (see Figure 5-5). If the expected packet lifetime is 35 rounds, and the adversary sees the packet on the network for 30 rounds, there are only 5 ways to fit those 30 rounds in the expected lifetime, so the adversary can

conclude that its first observation has a reasonable chance actually being the first time the packet was present on the network. Conversely, if it only sees the packet on the network for a total span of 5 rounds, then there are 30 ways the observation can fit inside the expected window. In this case, the adversary has much lower certainty that it witnessed the initial packet.

The bigger the reach of the adversary, the more useful this technique will be, since the more nodes the adversary can spy on the higher the chance that the adversary will observe a wide window of the packet life.

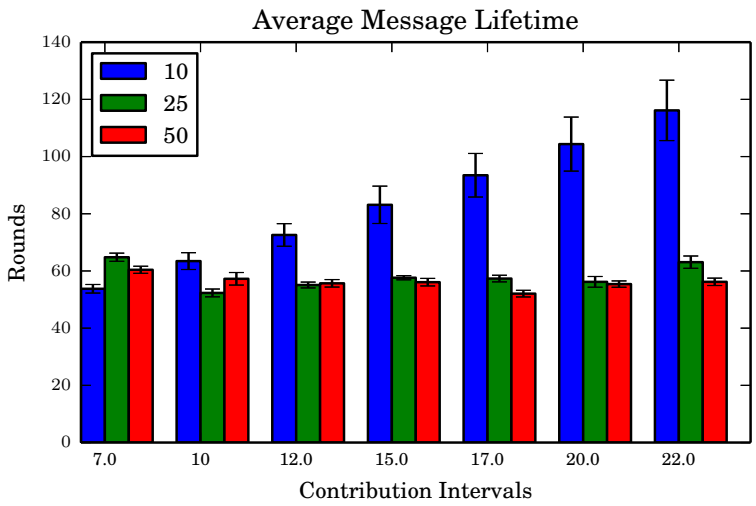


Figure 5-5: Average message lifetime for various contribution intervals in NCGAB for networks of 10, 25 and 50 peers.

This technique will give the adversary an idea of how likely it is that a particular appearance of a message is the first, but it will not narrow down the pool of possible senders. It is somewhat orthogonal to the approach outlined in Section 5.3.1.

While the message lifetime fit is a useful heuristic, it is also difficult to quantify. The message lifetime is not constant, so even if the adversary saw the packet for 35 rounds in a network with an average message lifetime of 35, it still cannot be certain that it saw the initial packet. Furthermore, it would be difficult for the adversary to calculate the average message lifetime unless it knows the configuration parameters of the network, or can observe a majority of the peers to statistically determine it.

Traffic pattern correlation

Information spread in gossip networks has been studied before [25]. For small networks, such as NCGAB, it should be possible to take advantage of the typical traffic pattern. The adversary would iterate through the possible dissemination trees, and then then compares its own subset to try and guess which tree fits its observations best.

It is possible to come up with an average probability that a particular peer receives a particular message in NCGAB. We say average because this probability depends on averages–we need to know the average gossip window and solved window sizes. Messages have a limited lifetime because of the ephemeral message window.

The `TTL_GOSSIP` parameter defines how many rounds a particular linear combination is eligible for forwarding. The default value is 10. `TTL_DECODE` defines how many rounds a single message is eligible for use in new linear combinations. The default value for this parameter is 30 rounds. The `LOOKUP_PERCENT` parameter is the percent of peers that a sender broadcasts to. The default is 0.25, which means that each peer randomly chooses a quarter of the peers to send messages to.

Figure 5-6 shows the average gossip and solved window sizes that we measured during simulations. The figure shows how the average window sizes change over time for networks of various sizes and contribution intervals. Using the average window sizes and the `LOOKUP_PERCENT` value, we have the steady state probabilities we need.

Using these probabilities in our adversary logic would be computationally computationally intensive and tricky to implement. Additionally, this technique relies on configuration parameters that the adversary would be unable to easily guess, such as `TTL_DECODE` and the contribute interval. Arguably, the adversary could use observations to make estimations of these parameters, but this is error prone and complex. As such, we leave further pursuit of this idea as future work.

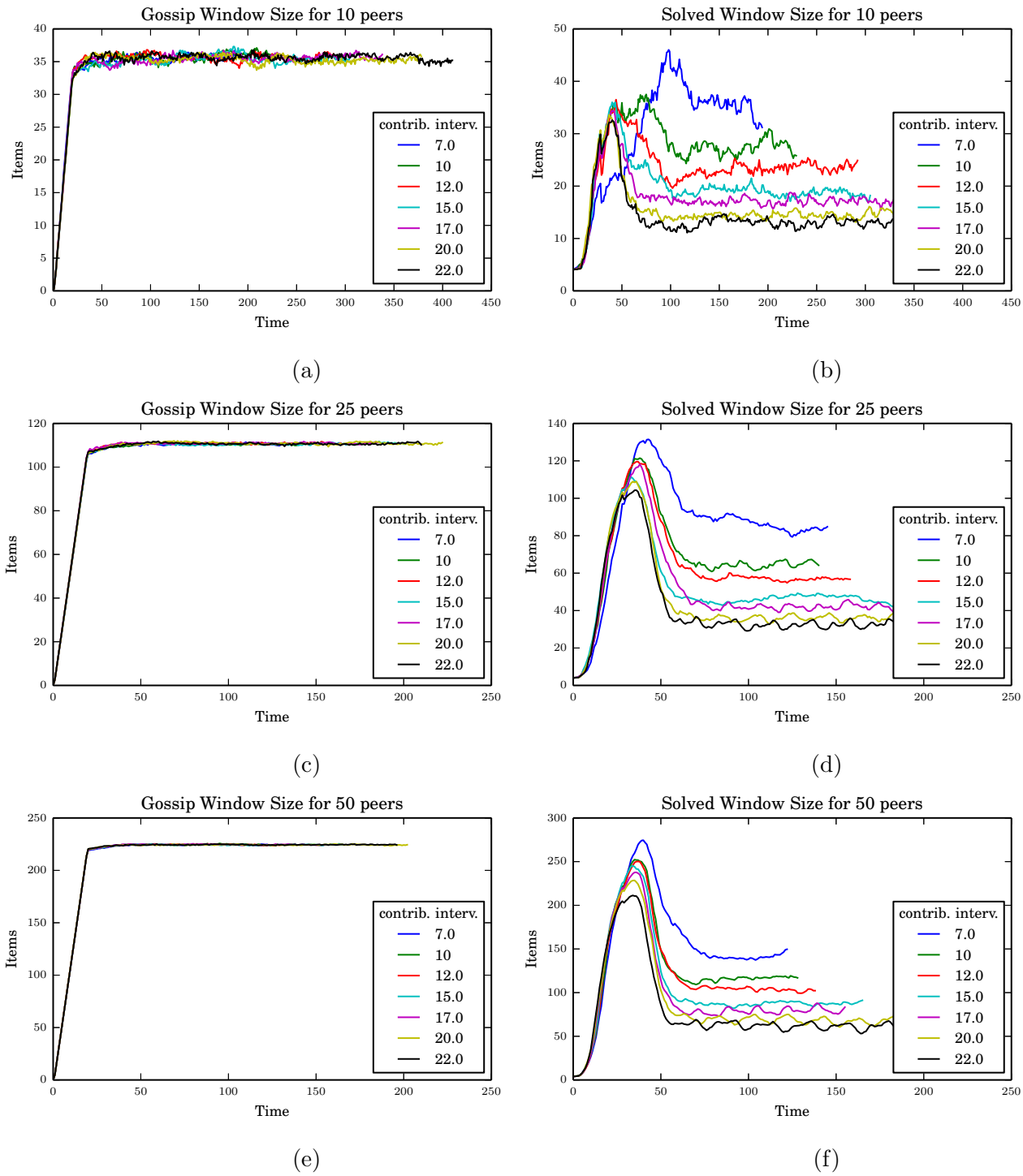


Figure 5-6: Average gossip and solved window sizes in NCGAB for a variety of network sizes and contribution intervals. The peaks in the solved window plots are due to the warm-up decodes that are used to fill the windows initially and bring the system to a steady state.

5.3.2 Probability of witnessing introduced message

What is the probability of an adversary witnessing the introduction of a contributed message in gossip it receives from a peer? This is an unresolved question from the NCGAB thesis that we now able to answer now that we have statistics on window sizes. The first part of answering this is finding the probability that a newly contributed message is sent during that round, which is

$$p_{new\_gossip} \frac{\binom{\texttt{AVG\_WINDOW\_SIZE}-1}{\texttt{CODE\_SIZE}-1}}{\binom{\texttt{AVG\_WINDOW\_SIZE}}{\texttt{CODE\_SIZE}}} \tag{5.1}$$

p_{new_gossip} is the probability that the peers codes new gossip during that round, instead of forwarding old gossip. Our NCGAB simulator has $p_{new\_gossip} = 0.5$. `AVG_WINDOW_SIZE` is the average solved window size for the network, which depends on the number of peers and the contribute interval (see Figure 5-6). `CODE_SIZE` is the number of packets that are randomly linearly combined to form a network coded packet.

Equation 5.1 is complicated looking, but is very similar to the probability of getting a particular playing card, c in a randomly chosen hand. The number of ways to pick a hand that includes that card is $\binom{51}{4}$, assuming a 52 card deck and a 5 card hand. This is because once we know we have c we choose the 4 other cards from the remaining 51. There are $\binom{52}{5}$ ways to choose any arbitrary hand. So the probability of getting a hand with our card in it is $\frac{\binom{51}{3}}{\binom{52}{4}}$. The only difference between this and Equation 5.1 is the additional p_{new_gossip} parameter.

As an example, let us look at a network with 25 peers and a contribute interval of 12. This has a steady-state `AVG_WINDOW_SIZE` of approximately 55. If we have a `CODE_SIZE` of 4, then the probability that a newly contributed message is sent out the same round is

$$0.5\frac{\binom{54}{3}}{\binom{55}{4}} = 0.0\overline{36} \tag{5.2}$$

This means that a newly introduced message has a approximately a 3.6% chance of being sent out in the same round. That makes the expected number of rounds

before a new message is sent out $\frac{1}{0.03\overline{6}} = 27.5$. We can use the contribute interval to find the average number of never-before-seen packets in the decode window: it is simply the expected number of rounds before a new message is sent out, divided by the contribute interval. For our example, that is $27.5/12 = 2.291\overline{6}$.

Using the average number of unseen messages, or `AVG_UNSEEN`, we can produce an expression for the probability that an adversary witnesses the contribution of a new message. As described above, we can determine `AVG_UNSEEN` using Equation 5.1 and the contribute interval. Then, in any arbitrary round, the probability of an adversary witnessing the introduction of one or more contributed message in an arbitrary round is

$$p_{new\_gossip}\left(1 - \frac{\binom{\texttt{AVG\_WINDOW\_SIZE}-\texttt{AVG\_UNSEEN}}{\texttt{CODE\_SIZE}}}{\binom{\texttt{AVG\_WINDOW\_SIZE}}{\texttt{CODE\_SIZE}}}\right) \tag{5.3}$$

The expression after p_{new_gossip} is 1-P[contributing zero new messages], which is equivalent to P[contributing one of more new messages]. Going back to the example we have been working on, the probability that an adversary witnesses the introduction of a new message in our network with `CODE_SIZE` 4 and contribute interval 12 is approximately 8%, as shown below in Equation 5.4.

$$0.5\left(1 - \frac{\binom{55-2.291\overline{6}}{4}}{\binom{55}{4}}\right) = 0.08035 \tag{5.4}$$

It may be possible to use this number to help quantify the expected packet lifetime approach to more reliably determine whether an observed message was the original transmission. We leave this pursuit to future work.

5.4 Comparison

In this section we compare NCGAB to two related protocols. The first protocol is a non-anonymity network coding protocol, similar to MORE [4]. The next is a modified version of NCGAB that introduces dummy packets. We compare the anonymity, delay and availability between these three protocols.

Baseline MORE-like protocol

We implemented a non-anonymity focused network coding system similar to MORE, which uses an ACK system to let the sender know when it has enough linear combinations to solve for a particular batch of messages[3]. The sender then does not include any solved messages in future transmissions. NCGAB differs from MORE in that it does keep including these solved messages, which increases diversity and makes the message appear from more nodes when re-gossiped.

Dummy packet protocol

Dummy packets are well-known technique for improving anonymity in networks. As [8] describes, dummy traffic is usually used to keep bandwidth usage constant on a channel so that an adversary has a difficult time doing traffic analysis. Since NCGAB and CHAP can use re-gossip to fill the channel gaps, dummy traffic is not actually needed. In fact, the lack of need for dummy traffic is what give NCGAB performance benefits over other gossip anonymity schemes.

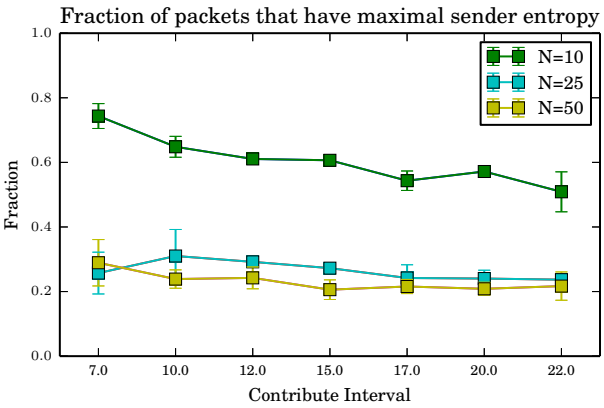
We wondered what effect adding dummy traffic back in would have on NCGAB/CHAP, so we implemented a modification to NCGAB that makes a certain percentage of outgoing packets into dummy packets.

5.4.1 Anonymity

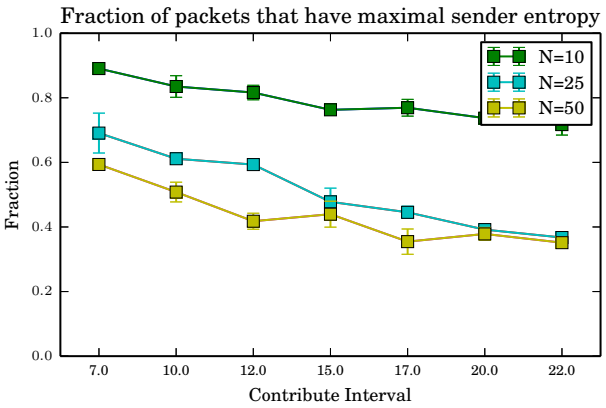
Looking at Figure 5-7 we see that NCGAB has stronger anonymity performance than the baseline MORE-like protocol, with as much as 30% more fully-anonymous messages. NCGAB also scales better, as evidenced by the decreased performance gap between N=10 and N=25. MORE is less anonymous because it excludes solved messages from coded transmissions.

Surprisingly, there is not much of an anonymity gain for the NCGAB with dummy packets. We expected that increasing dummy packets would have a similar effect to increasing the contribute interval, but this was not the case.

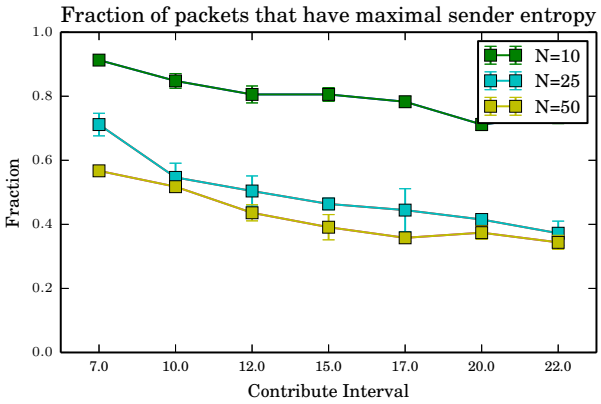
[3]Instead of implementing actual ACKs, we created an internal message passing system in the simulator.



(a) Baseline MORE-like protocol



(b) NCGAB



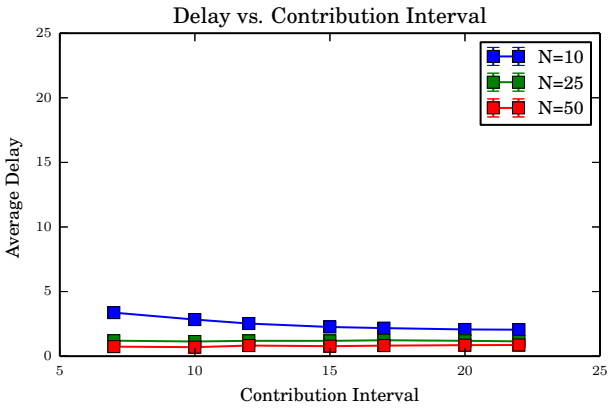
(c) 50% dummy NCGAB

Figure 5-7: Anonymity comparison across three protocols related to NCGAB using the percent of uniform PMFs as a metric. These statistics were compiled for an adversary with 25% reach and the simulation ran for 100 messages.

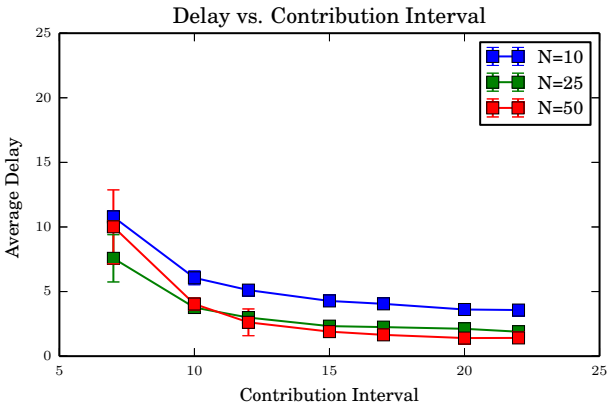
5.4.2 Delay and Availability

The delay for the baseline protocol is very low, most noticeably in the N=10 and N=25 networks. See Figure 5-8.The delay NCGAB with 50% dummy messages has a similar shape to the delay for the original NCGAB, but the dummy protocol actually has a larger separation between the N=10 and the other networks.

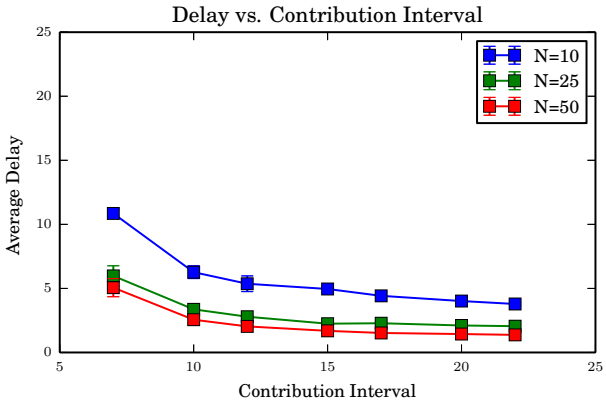
Looking at Figure 5-9, we see that the availability for NCGAB with dummy packets suffers a lot. This is possibly because the packets expire in the ephemeral message window before they have been decoded due to all the junk overhead. This could likely be solved by increasing the window sizes. The availability for MORE is



(a) Baseline MORE-like protocol



(b) NCGAB

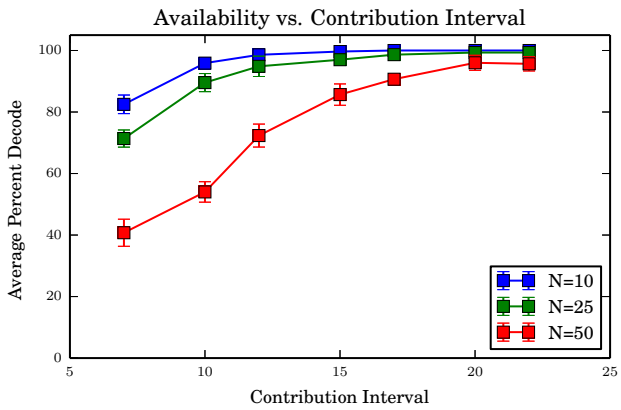


(c) 50% dummy NCGAB

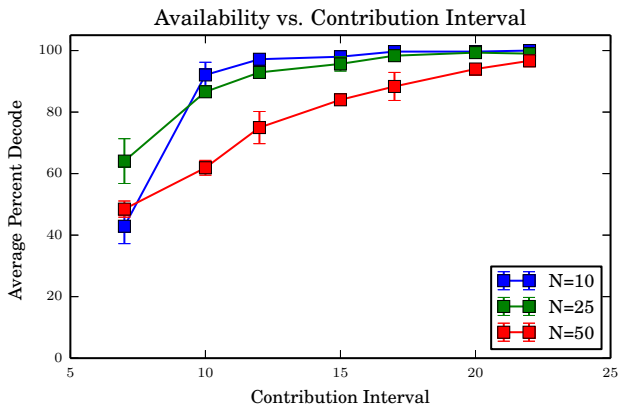
Figure 5-8: The average number of gossip rounds it takes for an inserted message to be decoded by a peer, given success, or delay, across three three network coded protocols: a baseline protocol, NCGAB, and NCGAB with dummy packets. The simulation ran for 100 messages.

generally higher than for NCGAB, especially for low contribution intervals. However, the two protocols perform quite similarly for a network of 50 nodes.

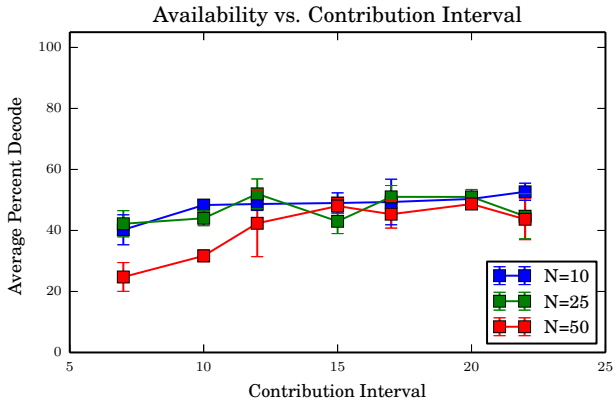
The primary cost of NCGAB is in delay. For higher contribution intervals and larger networks, the delay is not much worse than the MORE-like protocol, though, and the number of fully anonymized packets is about 30% higher for a network of 25 nodes. This suggests that NCGAB is a viable solution, particularly to applications that want good throughput but are not particularly sensitive to latency.



(a) Baseline MORE-like protocol



(b) NCGAB



(c) 50% dummy NCGAB

Figure 5-9: Average percentage of peers that successfully receive and decode an inserted message, or availability, across three network coded protocols: a baseline protocol, NCGAB, and NCGAB with dummy packets. The simulation ran for 100 messages.

5.5 CHAP

For the remainder of the chapter, we will focus on measuring the performance of CHAP. Since CHAP uses NCGAB, we already have a good idea of how anonymous CHAP is. In this section, we look deeper at how CHAP differs from NCGAB and how CHAP improves upon NCGAB's anonymity. We will also present the delay and availability for various CHAP topologies. We ran simulations on networks of 4, 8 and 12 clouds with each cloud having 6 nodes, including the exit node (see Fig. 5-10).

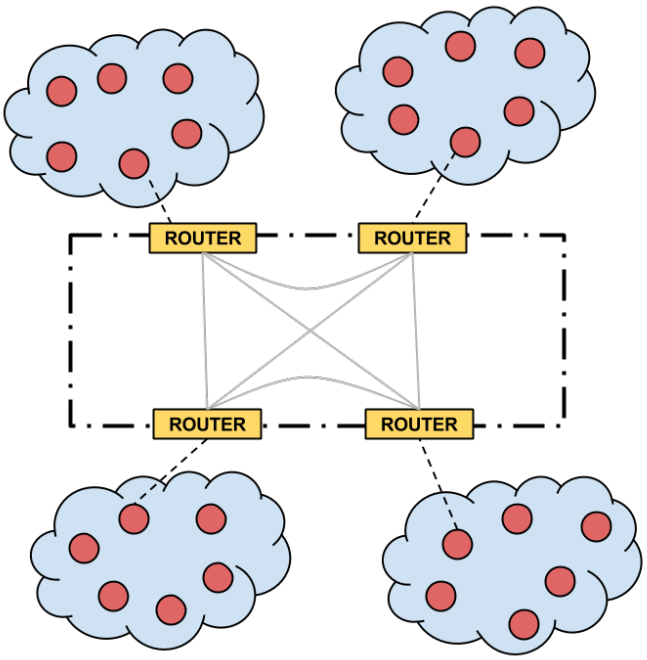


Figure 5-10: One of the CHAP topologies we ran simulations on, a network of four clouds with six nodes inside

5.5.1 CHAP Anonymity

CHAP uses NCGAB, and by the hierarchical separability presented in 3.3.2, we know that for a wireline-only adversary CHAP is at least as anonymous as NCGAB. Not only does CHAP have anonymity as least as effective as NCGAB, it actually improves anonymity. This is because an adversary that can only observe the wired portion can at most tell which cloud traffic comes from, and not which of the N nodes inside sent it. In fact, an observer cannot even determine the number of nodes in the cloud. Exit nodes are more vulnerable than the nodes inside the cloud since they communicate with the wireline, but they still have anonymity as strong NCGAB's.

In the case of an adversary inside a cloud, anonymity depends on what scheme the cloud is using. If there is no anonymity scheme, an adversary could pinpoint which peer introduces a message if it can listen to all wired links and at least one cloud. If there is an anonymity scheme, then the same adversary might be able to pinpoint which cloud sent the message, but not which peer in the cloud sent it. In future work, we hope to explore heterogeneous adversaries more.

We do not include any empirical anonymity measurements, since our simulator is not designed for a wireless adversary, and the wireline-only adversary would look very similar to the results already presented for NCGAB (except the PMFs would be for original cloud, not original sender). This is because the adversary is only able to eavesdrop upon the wired portion (the exit nodes), which communicate using NCGAB.

5.5.2 CHAP Delay and Availability

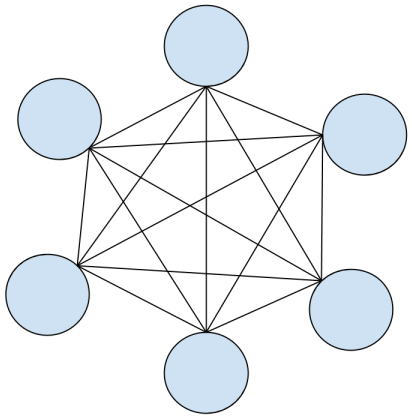
The variety of topologies possible in CHAP makes measuring delay and availability difficult, since there are so many variations to account for. The connectivity inside the ad-hoc clouds is represented by an adjacency matrix, where each row is a vector that represents the connectivity for a peer. Although the adjacency matrices gives us the ability to model arbitrary topologies inside the ad-hoc clouds, we have simplified the problem down to a few structured cases: the clique, the grid and the ring.

Clique

A clique (see Fig. 5-11) is a topology where all nodes can communicate with all other nodes. This means that if one node broadcasts a packet, all the other nodes can hear it.

Before looking at the performance of the entire CHAP topology, we will first look at the performance inside a single cloud. Figure 5-12 shows the delay and availability of 10, 15 and 20 node cliques. The performance is reasonable. CHAP is designed for clouds of 5-25 nodes; it supports more, but clearly the availability suffers the more nodes are added. The delay remains low for any number of nodes since all nodes can hear every transmission.

Next we look at the performance of cross-cloud communication. We ran simulations on networks of 4, 8 and 12 clouds with each cloud having 6 nodes, including the exit node (see Fig. 5-13). The delay does not display the negative correlation with contribute interval, thought it does not get much beyond 5 rounds. The availability,



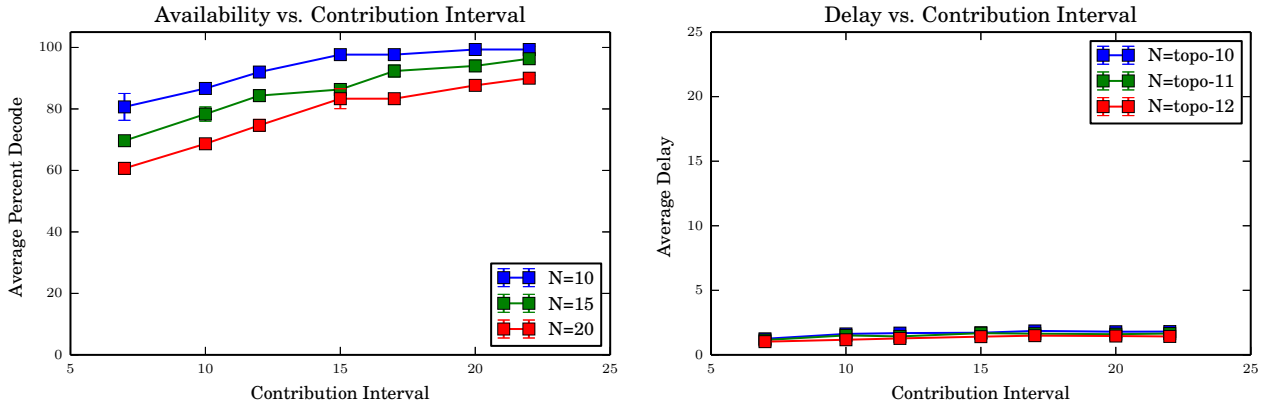
(a) The connectivity graph

$$\begin{bmatrix} 1 & 1 & 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 & 1 & 1 \end{bmatrix}$$

(b) Adjacency matrix

Figure 5-11: Connectivity graph and adjacency matrix of a 6 node clique

however, clearly suffers. In Section 5.5.3 we discuss a modification to the protocol that improves availability.



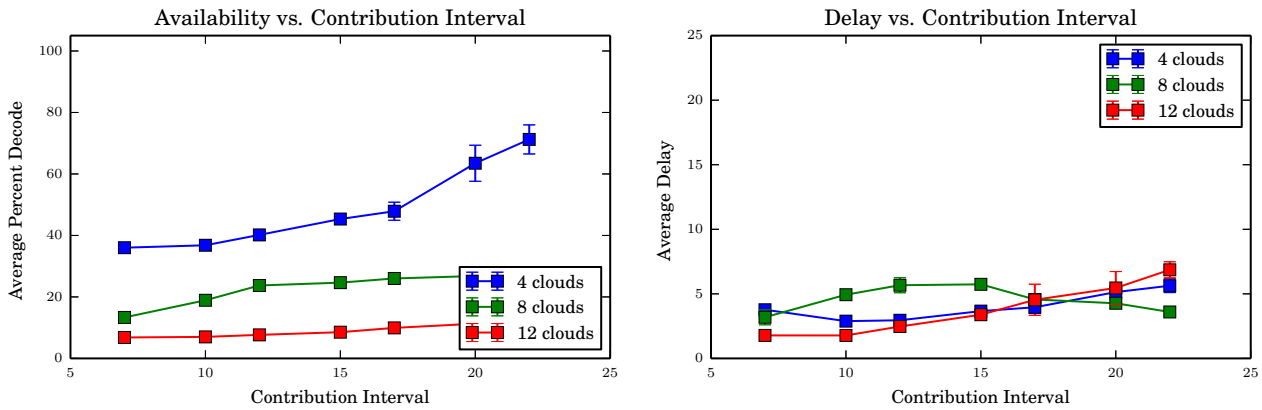
(a) Availability

(b) Delay

Figure 5-12: Availability and delay for cliques of 10, 15 and 20 peers. The simulation ran for 100 messages.

Ring

A ring topology arranges all of the nodes in a ring, and each node can talk to both of their neighbors but nobody else (see Fig. 5-14). Figure 5-15 shows the availability and delay for an ad-hoc cloud running an NCGAB-like protocol. As with the clique, adding peers shifts the availability down almost linearly. The delay for the circle



(a) Availability

(b) Delay

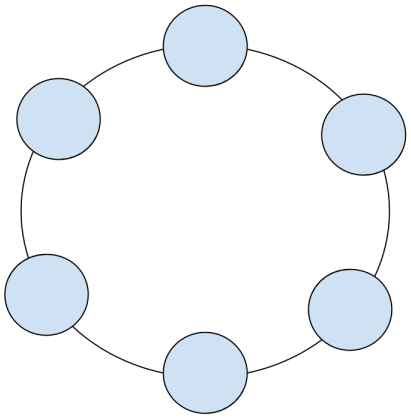
Figure 5-13: Availability and delay for CHAP networks of 4, 8 and 12 clique clouds, with each cloud having 6 nodes. The simulation ran for 100 messages.

does something interesting. Unlike all other networks, NCGAB or CHAP, the delay increases as the contribution interval increases for networks of 15 and 20. This is because the ring is a severely limiting topology, and each peer can only contact 2 other peers during a round. In other topologies, as the network peer list grows, more peers are reached during a broadcast. For a ring, adding peers just increases the amount of time it takes for a message to get across the ring. When the contribution interval is higher, perhaps the far away nodes are waiting longer to get needed degrees of freedom.

In Figure 5-16 we show the performance for cross-cloud communications of networks with 4, 8 and 12 clouds with each cloud having 6 nodes. Somewhat surprisingly, the performance is very similar to the network of clique clouds in Figure 5-13. We were expecting the performance of the ring network to be much worse, since the in-cloud availability is so much worse for rings than cliques. We believe the small size of the ring (6 nodes) is not large enough to see much performance detriment when compared to a clique.

Grid

In a grid topology, the nodes are arranged in a grid and an adjacency matrix dictates which of their neighbors they can talk to. Below is the adjacency matrix for a fully-

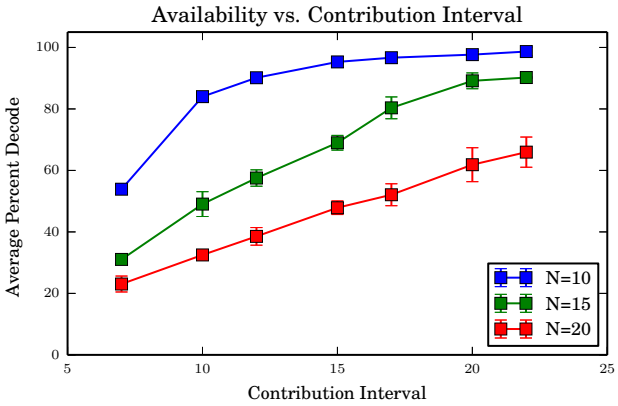


(a) The connectivity graph

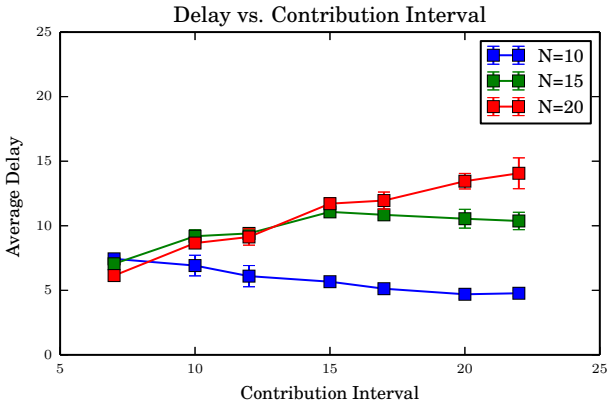
$$\begin{bmatrix} 0 & 1 & 0 & 0 & 0 & 1 \\ 1 & 0 & 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 & 0 & 1 \\ 1 & 0 & 0 & 0 & 1 & 0 \end{bmatrix}$$

(b) Adjacency matrix

Figure 5-14: Connectivity graph and adjacency matrix of a 6 node ring



(a) Availability



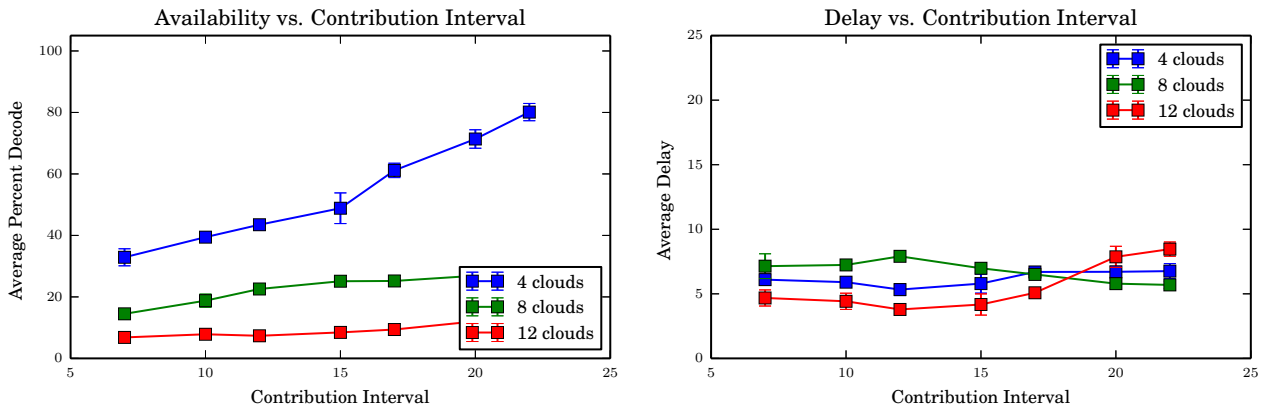
(b) Delay

Figure 5-15: Availability and delay for circles of 10, 15 and 20 peers. The simulation ran for 100 messages.

connected 3x3 grid (pictured graphically in Figure 5-17). It is also possible to have a grid with missing connections. Of the three structures, the grid most closely resembles what might be seen in a real ad-hoc network.

The delay and availability for fully connected grids are in Figure 5-18. Unsurprisingly, the availability is better than a ring, but not as good as a clique. The delay for the grid is similarly ranked.

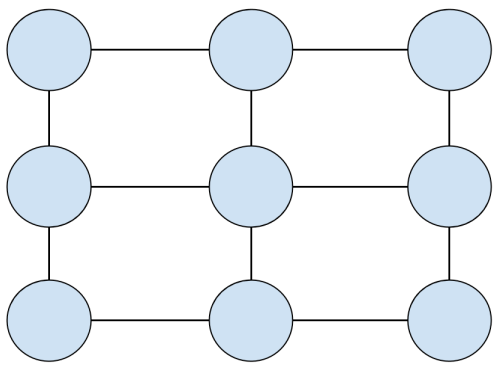
Figure 5-19 we shows the performance for cross-cloud communication. It is similar to the performance of the clique and ring CHAP networks presented earlier.



(a) Availability

(b) Delay

Figure 5-16: Availability and delay for CHAP networks of 4, 8 and 12 ring clouds, with each cloud having 6 nodes. The simulation ran for 100 messages.



(a) The connectivity graph

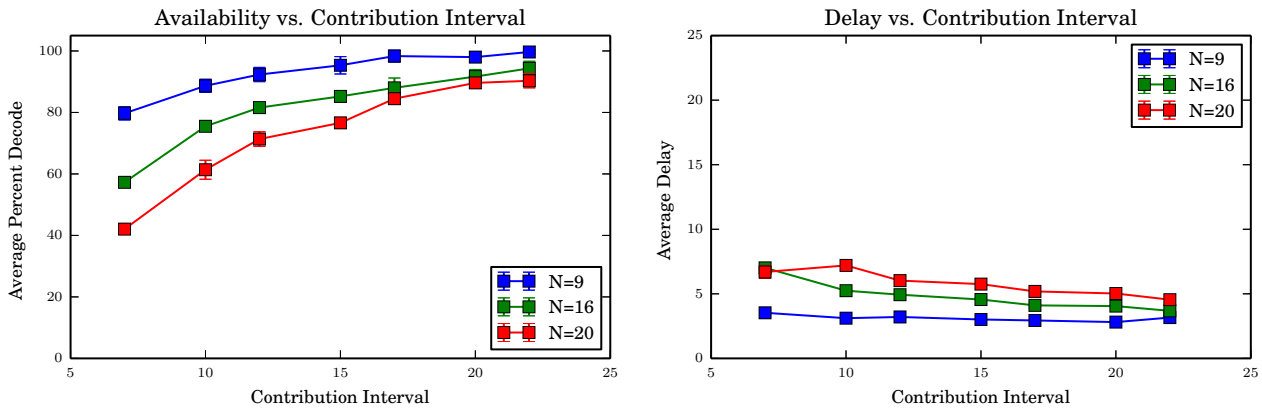
$$\begin{bmatrix} 0 & 1 & 0 & 1 & 0 & 0 \\ 1 & 0 & 1 & 0 & 1 & 0 \\ 0 & 1 & 0 & 0 & 0 & 1 \\ 1 & 0 & 0 & 0 & 1 & 0 \\ 0 & 1 & 0 & 1 & 0 & 1 \\ 0 & 0 & 1 & 0 & 1 & 0 \end{bmatrix}$$

(b) Adjacency matrix

Figure 5-17: Connectivity graph and adjacency matrix of a 3x3 grid

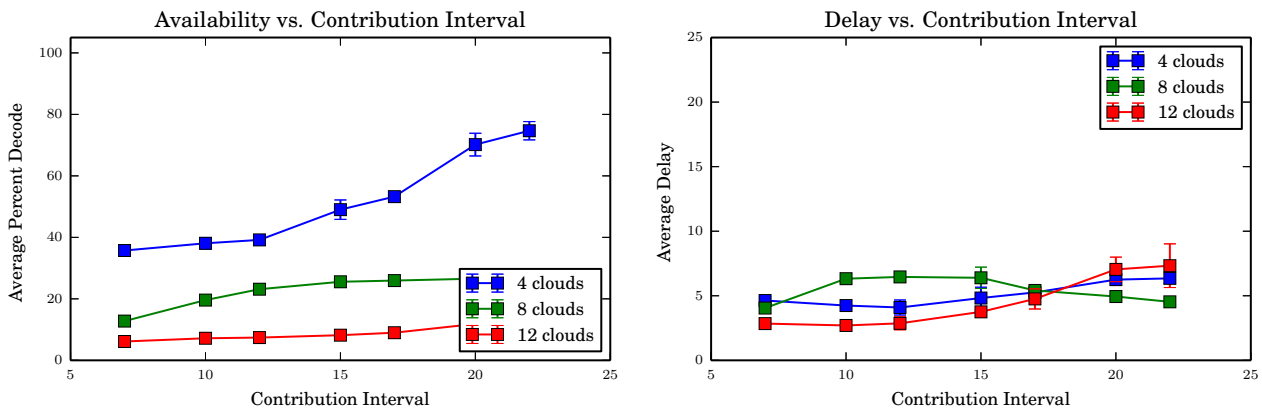
5.5.3 Contributions

So far we have assumed each node has packets to contribute when its contribute interval slot comes up. For a broadcast network of 15-50 nodes, however, it is unlikely that all nodes will have something that they want to say to all other nodes. Some peers will only be interested in receiving. We modified the simulator so that were able to set some number of nodes as contributors, and the others as receivers. The receivers still decode gossip and propagate both new and re-gossip. Looking at Figure 5-20, we see that lowering the number of contributing nodes drastically improves the performance



(a) Availability (b) Delay

Figure 5-18: Availability and delay for fully connected grids of 9, 16 and 20 peers. The simulation ran for 100 messages.

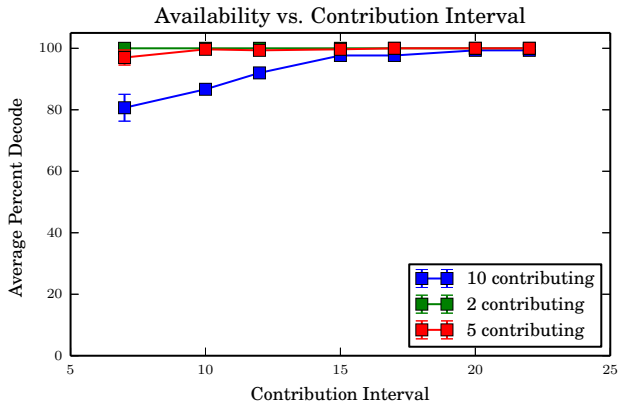


(a) Availability (b) Delay

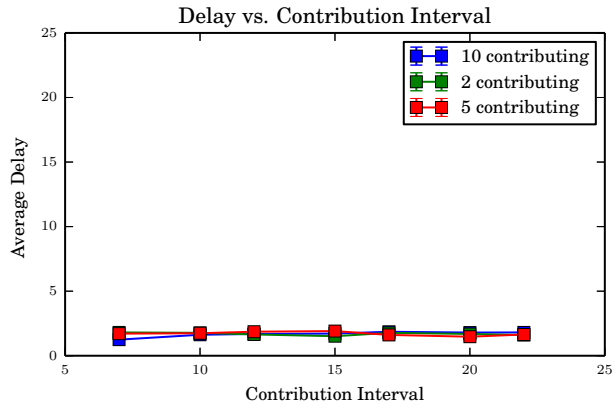
Figure 5-19: Availability and delay for CHAP networks of 4, 8 and 12 grid clouds, with each cloud having 6 nodes. The simulation ran for 100 messages.

for in-cloud communications.

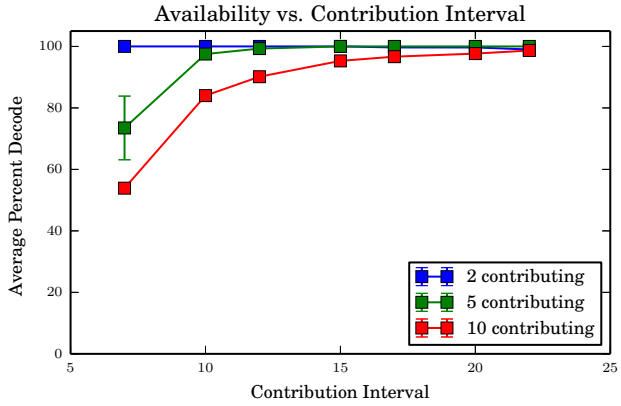
The delay and availability for cross-cloud communications shows improvement with a reduced number of unique contributors as well. The performance for clique, ring and grid CHAP networks with two nodes contributing is shown in Figure 5-21. These networks all have clouds with 6 peers inside, however we believe that for a small number of unique contributors per cloud the network could scale to 10 or 15 nodes per cloud without a drastic drop in performance, leading to an overall network range of 15-65 nodes.



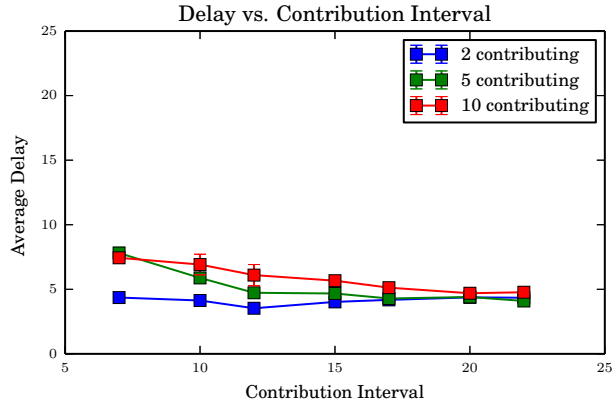
(a) Availability for a 10 node clique



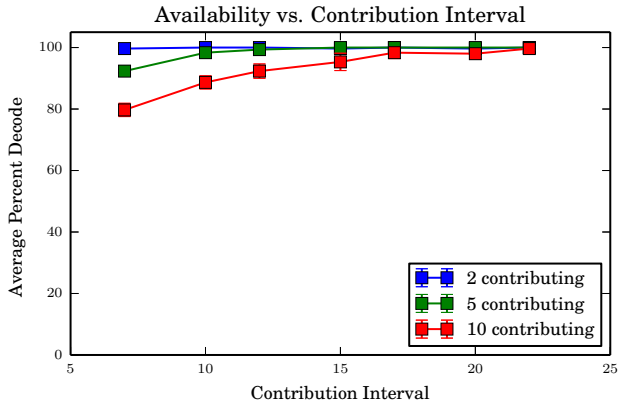
(b) Delay for a 10 node clique



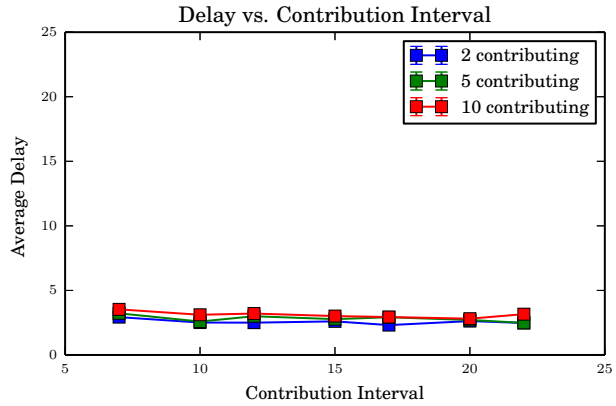
(c) Availability for a 10 node ring



(d) Delay for a 10 node ring

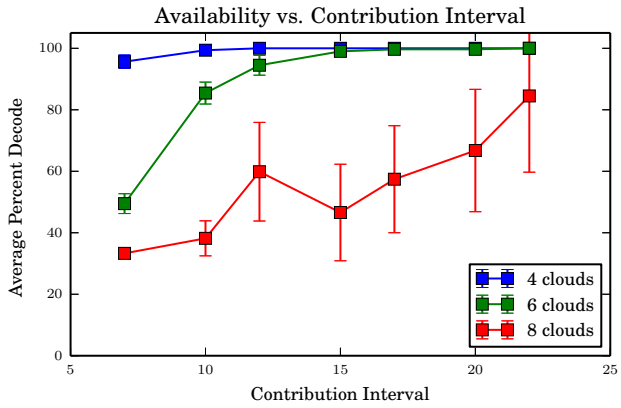


(e) Availability for a 9 node fully connected grid

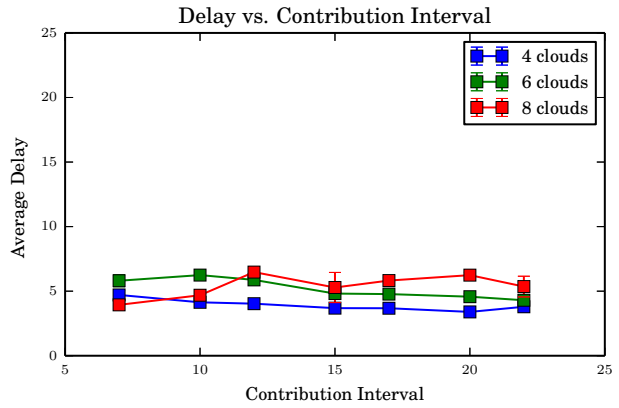


(f) Delay for a 9 node fully connected grid

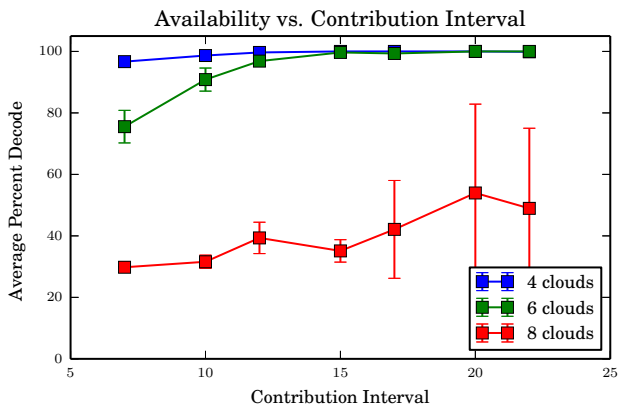
Figure 5-20: Availability and delay for clouds only with 2, 5 and 10 nodes contributing new messages. The simulations ran for 100 messages.



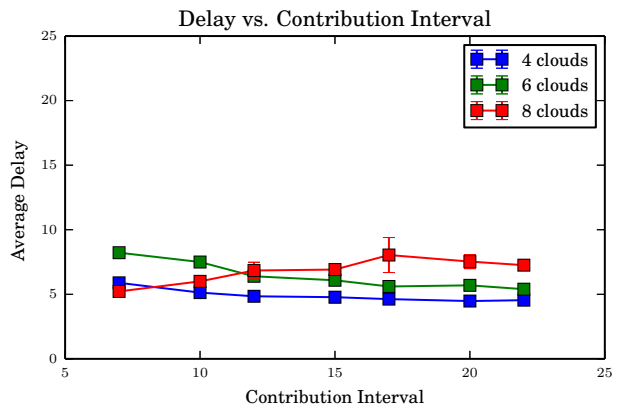
(a) Availability for a CHAP clique network



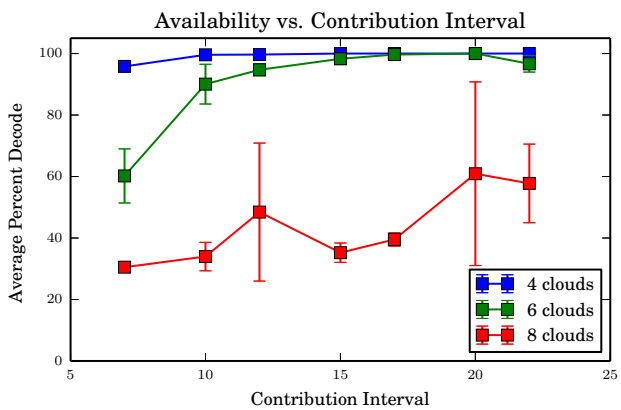
(b) Delay for a CHAP clique network



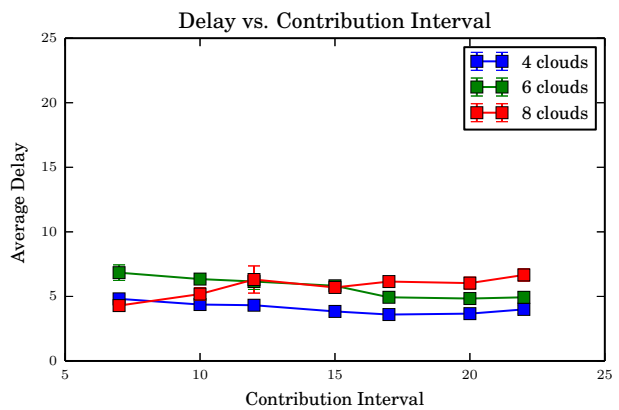
(c) Availability for a CHAP ring network



(d) Delay for a CHAP ring network



(e) Availability for a CHAP grid network



(f) Delay for a CHAP grid network

Figure 5-21: Availability and delay for CHAP networks with 4, 6 and 8 clouds with 2 nodes contributing new messages. The simulations ran for 100 messages.

5.5.4 Alternate topologies

The plots in this chapter have been for only a small subset of possible CHAP topologies. We have varied the number of nodes in the clouds, and the number of clouds present. It is certainly possible to have a set of clouds each having different numbers of nodes, or for the clouds to have different types of internal structures (e.g. some clouds are grids, some are cliques). There are many possible variations that we did not present. Although these may lead to interesting results, we believe that the topologies so far give a good picture of how CHAP scales and which ad-hoc cloud structures are the most favorable.

Additionally, although CHAP was designed as a protocol to connect ad-hoc clouds, it is possible to run the protocol on an arbitrary collection of both wired and wireless nodes. The wired links do not necessarily have to have an ad-hoc cloud behind them.

In this thesis, we have limited the scope to interconnecting ad-hoc clouds with the same number of nodes and the same internal topologies. A much wider range of topologies is possible, and it would be trivial for the reader to run the simulator on any topology of their choosing. We provided a few examples that the reader can use at the end of Appendix B.2.

5.5.5 LOOKUP_PERCENT modifications

In the original NCGAB implementation, `LOOKUP_PERCENT` is the percent of nodes that each peer sends packets to during a gossip round. The default implementation has this value set to 0.25. This is sufficient for networks of 10-100 nodes, but in a CHAP network typically much fewer nodes are participating in NCGAB. Limiting the `LOOKUP_PERCENT` to such a small fraction for networks smaller than ten nodes does not make sense. CHAP networks scale to 3-15 clouds, and each cloud usually has one exit node that runs NCGAB (though it is possible to have more exit nodes). With this in mind, we modified the LOOKUP_PERCENT value to change based on the number of peers, with the `LOOKUP_PERCENT` growing larger for smaller networks. See Table 5.2.

Number of NCGAB peers	LOOKUP_PERCENT
5 or fewer	100%
5 to 10	50%
10 or greater	25%

Table 5.2: CHAP LOOKUP_PERCENT values

5.6 Discussion

In this chapter, we discussed the simulator and how it measures anonymity, delay and availability. We presented these measurements for NCGAB as well as two related protocols. We see that although a simple MORE-like network coded gossip protocol does provide some level of anonymity, NCGAB's ephemeral message window does improve the anonymity provided. We also saw that having a fraction of the traffic as dummy packets is not very effective and is quite costly in terms of delay.

We presented the delay and availability for CHAP, and saw that, unsurprisingly, the network has a higher delay and lower availability because of the hierarchical nature. The loss in performance gives us an increase in anonymity against a wireline adversary, though, as discussed in Section 5.5.1. We did find that decreasing the number of unique contributors leads to significant improvements in performance for networks of 20-35 nodes.

Chapter 6

Conclusion

In Chapter 2 we discussed the three main types of current anonymity solutions, and also presented the techniques of network coding and gossip.

In Chapter 3 we presented CHAP, the Coded Heterogeneous Anonymity Protocol, which extends NCGAB and uses the inherently broadcast nature of wireless networks. We also provided an entropy-based analysis of the anonymity CHAP provides, and presented the interesting hierarchical separability property that allows the ad-hoc clouds to run a scheme designed for simplicity or throughput or anything else, while providing the same anonymity against a wireline-only adversary. This makes it possible to run multiple protocols in different domains simultaneously, which is something we would like to explore further in future work.

In Chapter 4 we briefly suggested a few ways to make use of the hierarchical separability of CHAP, such as designing for simplicity, throughput or anonymity. We believe that this chapter gives an interesting list of directions to pursue future work.

In Chapter 5 we present the NCGAB/CHAP simulator and its results. We showed that NCGAB fully anonymizes as much as 90% of messages, and is as much as 30% more anonymous that a basic non-anonymity focused network coding gossip protocol. This increased anonymity is in exchange for delay, but for certain configurations the delay only increases by two or three rounds. We presented performance statistics for CHAP and discussed why CHAP is at least as anonymous as NCGAB for a wireline-only adversary.

In the future, it would be interesting to compare how CHAP/NCGAB's anonymity compares to Tor's. It would also be interesting to attempt combining them, as it is possible to run NCGAB inside Tor's hidden services mode or even on the general Tor network. We would also like to see NCGAB/CHAP implemented in a more realistic environment, such as a testbed. It would also allow us to explore more arbitrary topologies and experience wireless noise and interference conditions. This would also expose unforeseen real-world implementation issues that would need to be solved before CHAP/NCGAB could be released to users.

We would also like to look at relaxing some of the assumptions made for the analysis of CHAP's anonymity in Chapter 3, such as the traffic contents not being correlated. We also want to provide a more formal analysis of how anonymous the system is when there is adversary presence in one or more ad-hoc cloud, as well as empirical measurements similar to those in Chapter 5.

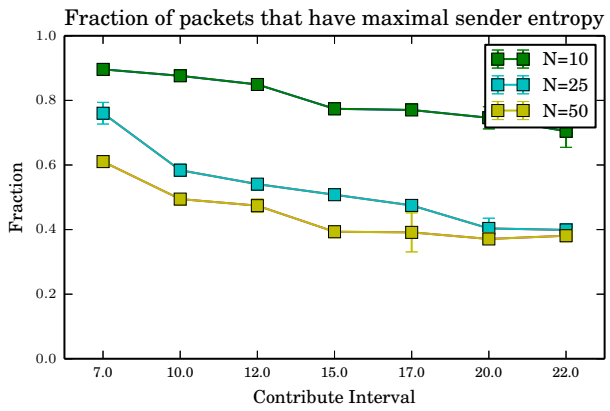
Furthermore, we outlined a number of additional adversarial techniques in Chapter 5 that we would like to explore more formally. The adversary we designed is quite simplistic, and we want to know how feasible the more sophisticated techniques we suggested would be. Also, much of our code assumes the adversary has accurate information on the system configuration parameters. It would be interesting to see how the adversary's performance compares when it has to estimate the parameters from observations instead.

Finally, we believe that anonymity in ad-hoc networks has a wealth of open questions. Although [22] suggests a linear program for maximal spatial re-use in an anonymous schedule, the program is only in pseudo-code. Furthermore, the existing solutions leave challenges remain to be solved, such as how to maintain anonymity when the schedule needs to be adjusted in real-time (such as nodes joining/leaving or moving positions). Ad-hoc networks themselves are also a very challenging research area, and although most wireless drivers do have an ad-hoc mode implemented, the implementations are generally poor and have low throughput due to lack of user demand.

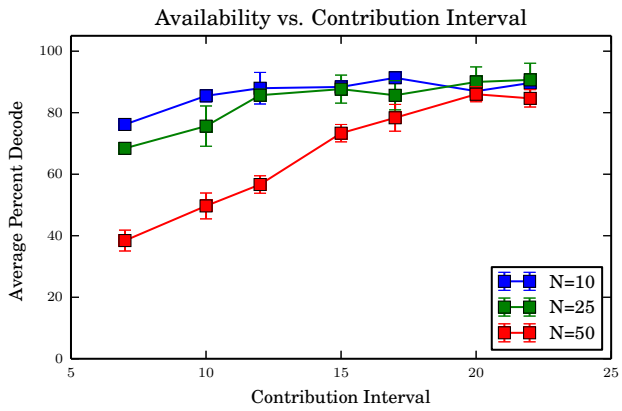
Appendix A

Figures

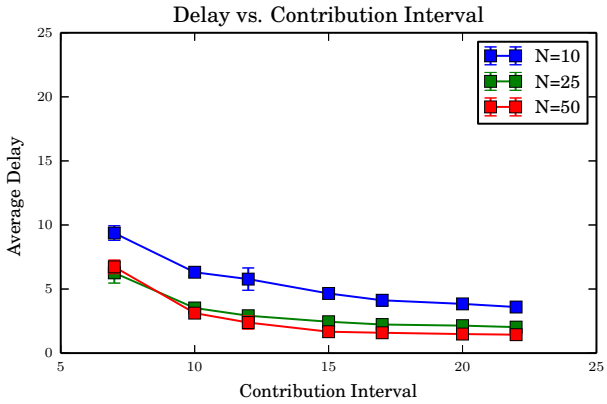
A.1 10% dummy packets



(a) Percent uniform



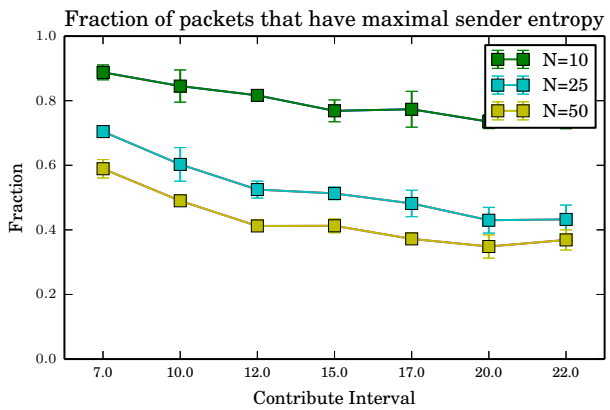
(b) Availability



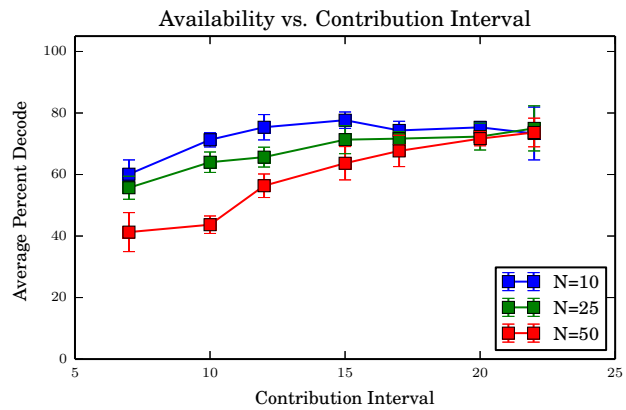
(c) Delay

Figure A-1: The performance differences by having 10% of the inserted packets be dummy packets.

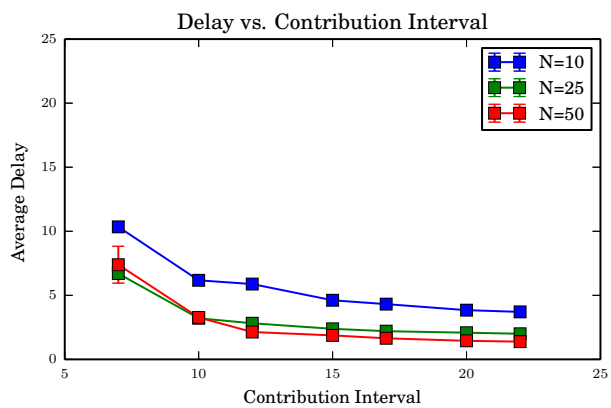
A.2 25% dummy packets



(a) Percent uniform



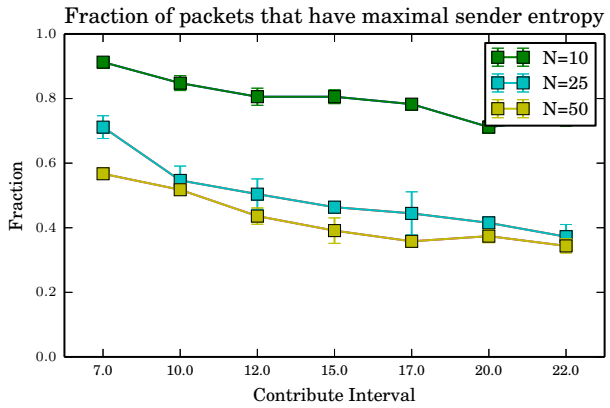
(b) Availability



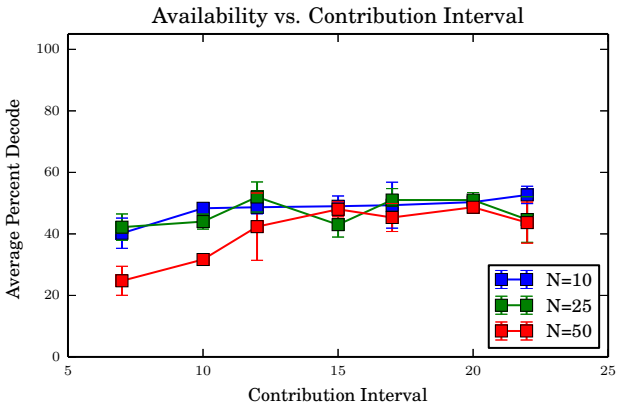
(c) Delay

Figure A-2: The performance differences by having 25% of the inserted packets be dummy packets.

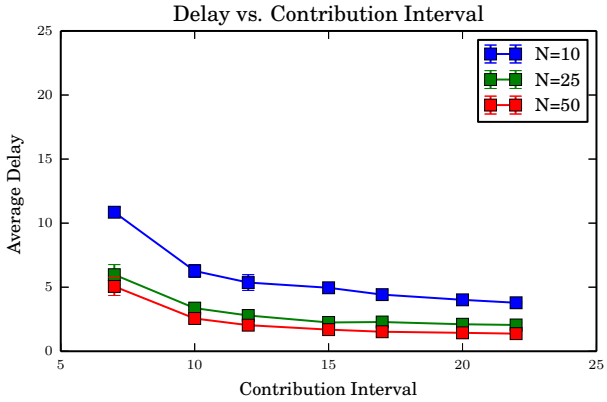
A.3 50% dummy packets



(a) Percent uniform



(b) Availability



(c) Delay

Figure A-3: The performance differences by having 50% of the inserted packets be dummy packets.

Appendix B

Simulator Code

B.1 ncgabsim.py

```
#!/usr/bin/env python2

# NCGAB Simulator- Ivan A. Sergeev + Colleen Josephson
# 2013/2014

import threading
import random
import json
import sys
import copy
import os

if sys.version_info.major == 2:
    import Queue as queue
else:
    import queue

import ff

import termcolor

##############################################################################

simConfig = 1

# Cooperative Peer Simulation

SimTemplate = {
    'NAME':                  "",
    'DESC':                  "",

    'LOOKUP_PERCENT':        0.25,
    'FRACTION_DUMMY':        0,
    'CODE_SIZE':             4,
    'TTL_DECODE':            30,
    'TTL_GOSSIP':            20,

    'SIM_NUM_PEERS':         0,
    'SIM_NUM_EVIL_PEERS':    0,
    'CONTRIBUTE_INTERVAL':   0,

    'SIM_WARMUP_DECODES':    50,
    'SIM_DURATION_INSERTS':  100,

    'SEED':                  0,

    'PRINT_LOG':             False,
}

if simConfig == 1:
    NumPeersSweep = [10, 25, 50]
```

```
52      ContributeIntervalSweep = [7.0, 10, 12.0, 15.0, 17.0, 20.0, 22.0]
53      SeedSweep = [0x1, 0x2, 0x3]
54
55      SimParamsList = []
56      for numPeers in NumPeersSweep:
57          for contributeInterval in ContributeIntervalSweep:
58              for seed in SeedSweep:
59                  name = "Sim-N%d-R%d-S%d" % (numPeers, contributeInterval, seed)
60                  params = copy.deepcopy(SimTemplate)
61                  params['NAME'] = name
62                  params['SEED'] = seed
63                  params['SIM_NUM_PEERS'] = numPeers
64                  params['CONTRIBUTE_INTERVAL'] = contributeInterval
65                  SimParamsList.append(params)
66
67  if simConfig == 7:
68      NumPeersSweep = [25]
69      ContributeIntervalSweep = [7.0, 10, 12.0, 15.0, 17.0, 20.0, 22.0]
70      SeedSweep = [0x1, 0x2, 0x3]
71
72      SimParamsList = []
73      for numPeers in NumPeersSweep:
74          for contributeInterval in ContributeIntervalSweep:
75              for seed in SeedSweep:
76                  name = "SimSelect2-N%d-R%d-S%d" % (numPeers, contributeInterval, seed)
77                  params = copy.deepcopy(SimTemplate)
78                  params['NAME'] = name
79                  params['SEED'] = seed
80                  #params['SELECTION'] = 'Uniform'
81                  params['SELECTION'] = 'Weighted'
82                  params['SIM_NUM_PEERS'] = numPeers
83                  params['CONTRIBUTE_INTERVAL'] = contributeInterval
84                  SimParamsList.append(params)
85
86  ##############################################################################
87
88  class Stats(object):
89      def __init__(self, simParams, simEventStop):
90          # Statistics collected
91          self.message_inserts = []
92          self.message_decodes = {}
93          self.matrix_reduces = []
94          self.window_sizes = []
95          self.last_message_exists = {}
96          self.message_life = {}
97          self.round_finished = 0
98
99          self.PMFs = {}
100
101         # Simulation parameters
102         self.simParams = simParams
103
104         self.simEventStop = simEventStop
105
106     #########################
107
108     def message_track(self, rnd, decoded_objects, gossip_objects):
109         if len(self.message_decodes) < self.simParams['SIM_WARMUP_DECODES']:
110             return
111         if len(self.message_inserts) < self.simParams['SIM_DURATION_INSERTS']:
112             return
113
114         # If the last message disappeared in the previous round, signal
115         # simulation stop
116         if rnd > 0 and (rnd-1) in self.last_message_exists:
117             if self.last_message_exists[rnd-1] == False:
118
119                 self.simEventStop.set()
120                 return
121
122         # If we already saw the last message in this round
123         if rnd in self.last_message_exists and self.last_message_exists[rnd] == True:
124                 return
125
126         # Look for all messages inserted inside peers on the network
127         for (rnd_inserted, _, last_pid) in self.message_inserts:
128             # Look for the inserted message in the decoded objects
129             for p in decoded_objects:
130                 if str(p) == last_pid:
131                     self.last_message_exists[rnd] = True
132                     return
133             # Look for the inserted message in the gossip objects
134             for lc in gossip_objects:
135                 for p in lc.messages:
136                     if str(p) == last_pid:
137                         self.last_message_exists[rnd] = True
138                         return
139             # Otherwise record that the message is 'dead'
140             if not last_pid in self.message_life:
141                 self.message_life[last_pid] = rnd - rnd_inserted
142
143         self.last_message_exists[rnd] = False
```

```
144
145     def message_insert(self, rnd, nid, pid):
146         if len(self.message_decodes) < self.simParams['SIM_WARMUP_DECODES']:
147             return
148
149         # If we have already collected our simulation number of inserts
150         if len(self.message_inserts) >= self.simParams['SIM_DURATION_INSERTS']:
151             return
152
153         self.message_inserts.append( (rnd, nid, pid) )
154
155     def message_decode(self, rnd, nid, pid):
156         # If we hadn't started recording this PID yet
157         if pid not in self.message_decodes:
158             self.message_decodes[pid] = []
159
160         # If this nid already decoded this pid in the past
161         if nid in [ n for (_, n) in self.message_decodes[pid] ]:
162             return
163
164         self.message_decodes[pid].append( (rnd, nid) )
165
166     def matrix_reduce(self, rnd, nid, numrows, numcols, numsolved):
167         self.matrix_reduces.append( (rnd, nid, numrows, numcols, numsolved) )
168
169     def window_size(self, rnd, nid, solved_size, gossip_size):
170         self.window_sizes.append( (rnd, nid, solved_size, gossip_size) )
171
172     def finished(self, rnd):
173         self.round_finished = rnd
174
175     #########################
176
177     def dump(self):
178         i = 0
179         while True:
180             path = "data/%s-%d.data" % (self.simParams['NAME'], i)
181             if not os.path.exists(path): break
182             i += 1
183
184         f = open(path, "w")
185         data = {'simParams': self.simParams,
186                 'message_inserts': self.message_inserts,
187                 'message_decodes': self.message_decodes,
188                 'matrix_reduces': self.matrix_reduces,
189                 'window_sizes': self.window_sizes,
190                 'round_finished': self.round_finished,
191                 'message_life': self.message_life,
192                 'PMFs': self.PMFs}
193         f.write(json.dumps(data))
194         f.close()
195
196         return path
197
198 ################################################################################
199
200 class Log(object):
201     def __init__(self, simParams):
202         self.elog = []
203         self.simParams = simParams
204
205     def log(self, rnd, etype, enid, emsg):
206         e = {}
207         e['time'] = rnd
208         e['type'] = etype
209         e['nid'] = enid
210         e['msg'] = emsg
211         ejson = json.dumps(e)
212         self.elog.append(ejson)
213
214         if self.simParams['PRINT_LOG']:
215             if etype == "join" or etype == "leave": print(termcolor.colored(ejson, "yellow"))
216             elif etype == "receive": print(termcolor.colored(ejson, "yellow"))
217             elif etype == "insert": print(termcolor.colored(ejson, "red"))
218             elif etype == "reduce": print(termcolor.colored(ejson, "blue"))
219             elif etype == "decode": print(termcolor.colored(ejson, "green"))
220             else: print(termcolor.colored(ejson, "white"))
221
222     def dump(self):
223         i = 0
224         while True:
225             path = "logs/%s-%d.log" % (self.simParams['NAME'], i)
226             if not os.path.exists(path): break
227             i += 1
228
229         f = open(path, "w")
230         for e in self.elog:
231             f.write(e + "\n")
232         f.close()
233
234         return path
235
```

```
236 ##############################################################################
237
238 class Network(object):
239     def __init__(self, simlog, simstats):
240         self.simlog = simlog
241         self.simstats = simstats
242         self.network = {}
243
244     def join(self, rnd, nid, q):
245         self.network[nid] = q
246         self.simlog.log(rnd, "join", nid, "")
247
248     def leave(self, rnd, nid):
249         if nid in self.network:
250             del self.network[nid]
251             self.simlog.log(rnd, "leave", nid, "")
252
253     def lookup_random(self, nid, n):
254         choices = list(self.network.keys())
255         choices.remove(nid)
256         random.shuffle(choices)
257         return [self.network[choices[i]] for i in range(n)]
258
259 ##############################################################################
260
261 class Message(object):
262     def __init__(self, pid):
263         self.pid = pid
264
265     def __str__(self):
266         return self.pid
267
268     def __eq__(self, other):
269         if (self.pid == other.pid):
270             return True
271         return False
272
273 class DummyMessage(Message):
274     def __init__(self):
275         Message.__init__(self, "R%04x" % random.getrandbits(32))
276
277 class RealMessage(Message):
278     def __init__(self, nid):
279         Message.__init__(self, "M%d/%04x" % (nid, random.getrandbits(32)))
280
281 class RLC(Message):
282     def __init__(self, messages):
283         coefs = [random.randint(0, 255) for p in messages]
284         pid = "LC/(" + ",".join([p.pid for p in messages]) + ")" + "/(" + \
285                       ",".join(["%02x" % c for c in coefs]) + ")"
286         Message.__init__(self, pid)
287         self.messages = messages
288         self.coefs = coefs
289
290
291 ##############################################################################
292
293 def choose_weighted_random(objects, scores):
294     scores_cdf = []
295     for s in scores[:]:
296         scores_cdf.append(sum(scores_cdf) + s)
297
298     random_score = random.random()*scores_cdf[-1]
299     for index in range(len(scores_cdf)):
300         if scores_cdf[index] > random_score:
301             break
302
303     return index
304
305 class Window(object):
306     def __init__(self, keep_expired = False):
307         self.window = []
308         self.ttl = {}
309         self.keep_expired = keep_expired
310
311     def add(self, p, ttl):
312         if type(p) == type(None) or p in self.window:
313             return False
314
315         self.window.append(p)
316         self.ttl[str(p)] = ttl
317         return True
318
319     def prune(self):
320
321         # Create a list of expired objects
322         expiredList = list(filter(lambda x: self.ttl[str(x)] == 0, self.window))
323
324         if self.keep_expired:
325             return expiredList
326
327         # Remove all expired objects from our window
```

```
        for e in expiredList:
            self.window.remove(e)
            del self.ttl[str(e)]

        return expiredList

    def tick(self):
        # Decrement the TTL for each object in the window
        for x in self.window:
            self.ttl[str(x)] = max(0, self.ttl[str(x)] - 1)

        return self.prune()

    def live_objects(self):
        return list(filter(lambda x: self.ttl[str(x)] > 0, self.window))

    def objects(self):
        return self.window[:]

    def __str__(self):
        s = "Window\n"
        for x in self.window:
            s += "\t" + str(x) + " TTL: " + str(self.ttl[str(x)]) + "\n"
        return s

class Decoded_Window(Window):
    def __init__(self):
        Window.__init__(self, keep_expired = True)

    def choose_random_uniform(self, n):
        choices = self.live_objects()
        random.shuffle(choices)
        return choices[0 : min(n, len(choices))]

    def choose_random(self, n):
        choices = self.live_objects()
        scores = [self.ttl[str(c)] for c in choices]

        n = min(n, len(choices))
        chosen = []
        for i in range(n):
            k = choose_weighted_random(choices, scores)
            chosen.append(choices[k])
            del choices[k]
            del scores[k]

        return chosen

    def prune(self):
        expiredList = Window.prune(self)
        return expiredList

class Gossip_Window(Window):
    def __init__(self):
        Window.__init__(self, keep_expired = False)
        # Keep active objects in a dictionary by source as well
        self.active_objects_by_source = {}

    def add(self, src, p, ttl):
        if Window.add(self, p, ttl) == True:
            # Create a new list for the source if it's not in our dictionary
            if src not in self.active_objects_by_source:
                self.active_objects_by_source[src] = []

            # Add the message to the source's list
            self.active_objects_by_source[src].append(p)
            return True

        return False

    def prune(self):
        expiredList = Window.prune(self)
        # For each expired object
        for e in expiredList:
            for src in self.active_objects_by_source:
                # If the object exists in this source's list
                if e in self.active_objects_by_source[src]:
                    # Delete it
                    self.active_objects_by_source[src].remove(e)
                    # Delete the source's list if it's empty now
                    if len(self.active_objects_by_source[src]) == 0:
                        del self.active_objects_by_source[src]
                    break
        return expiredList

    def choose_random_uniform(self):
        # Choose a random source
        source = random.choice(list(self.active_objects_by_source.keys()))

        # Gather a list of objects by this source
        objects = self.active_objects_by_source[source]

```

```
        return random.choice(objects)

    def choose_random(self):
        # Choose a random source
        source = random.choice(list(self.active_objects_by_source.keys()))

        # Gather a list of objects by this source
        choices = self.active_objects_by_source[source]
        scores = [self.ttl[str(c)] for c in choices]

        return choices[choose_weighted_random(choices, scores)]

    def fast_rref(self, m, b, x):
        done = False
        pi = 0

        # Iterate through each row
        for j in range(len(m)):
            # While we do not have a pivot for this row
            while m[j][pi] == 0:
                # Find a row below to swap with for a pivot at pi
                for k in range(j+1, len(m)):
                    if m[k][pi] != 0:
                        # Swap with this row
                        (m[j], m[k]) = (m[k], m[j])
                        (b[j], b[k]) = (b[k], b[j])
                        break

                # Increment pivot index if we could not find a row to swap with
                if m[j][pi] == 0:
                    pi += 1

                # If there is no pivots left, we're done reducing
                if pi == len(m[0]):
                    done = True
                    break

            if done:
                break

            # Divide through to have a pivot of 1
            m[j] = [ ff.FiniteFieldArray.ff_elem_div(m[j][i], m[j][pi]) for i in range(len(m[0])) ]

            # Eliminate above & below
            for k in range(len(m)):
                if k != j and m[k][pi] != 0:
                    m[k] = [ ff.FiniteFieldArray.ff_elem_sub(m[k][i], \
                      ff.FiniteFieldArray.ff_elem_mul(m[j][i], m[k][pi])) for i in range(len(m[0])) ]

            # Move onto the next pivot
            pi += 1
            # If there is no pivots left, we're done reducing
            if pi == len(m[0]):
                break

        solved = []
        used = []

        for i in range(len(m)):
            # If this row has only one non-zero entry
            reduced_coefs = [1*(m[i][j] != 0) for j in range(len(m[0]))]
            if sum(reduced_coefs) == 1:
                # Add the solution to our solved list
                solved.append(x[reduced_coefs.index(1)])
                # Add the decoded LC to our used list
                used.append(b[i])

        return (solved, used)

    def solve(self, decoded_window):
        unsolved_message_map = {}

        # Get a list of decoded messages
        decoded_messages = decoded_window.objects()

        # Make a list of undecoded linear combinations
        undecoded_lc = []
        for lc in self.live_objects():
            # If we've decoded this entire linear combination, don't add it
            if sum([p in decoded_messages for p in lc.messages]) == len(lc.messages):
                continue
            undecoded_lc.append(lc)

        # Put together a list of all decoded messages referenced by undecoded
        # linear combinations
        ref_decoded_messages = []
        for lc in undecoded_lc:
            for p in lc.messages:
                if p in decoded_messages and p not in ref_decoded_messages:
                    ref_decoded_messages.append(p)

        # Assemble x of mx=b
```

```
        #   x maps messages to column indices
        x = {}
        # Assign a col index to each referenced decoded message
        for p in ref_decoded_messages:
            x[str(p)] = len(x)
        # Assign a col index to each undecoded message
        for lc in undecoded_lc:
            for p in lc.messages:
                if str(p) not in x:
                    x[str(p)] = len(x)
                    unsolved_message_map[str(p)] = p

        # Assemble m and b of mx = b
        m = []
        b = []
        # Create a row for each solved message
        for p in ref_decoded_messages:
            r = [0] * len(x)
            r[x[str(p)]] = 1
            m.append(r)
            b.append(p)
        # Create a row for each linear combination
        for lc in undecoded_lc:
            r = [0] * len(x)
            for i in range(len(lc.messages)):
                pid = str(lc.messages[i])
                coef = lc.coefs[i]
                r[x[pid]] = coef
            m.append(r)
            b.append(lc)

        # Information about the size of this reduce attempt
        b_pids = [str(p) for p in b]
        m_numrows = len(m)
        m_numcols = len(m[0]) if m_numrows > 0 else 0

        (solved, used) = self.fast_rref(m, b, sorted(x, key=x.get))

        # Remove previously decoded objects from our solution
        for p in ref_decoded_messages:
            if str(p) in solved:
                solved.remove(str(p))
            if p in used:
                used.remove(p)

        # Map message ids to message objects in our solution
        for i in range(len(solved)):
            solved[i] = unsolved_message_map[solved[i]]

        # Information about the solution of this reduce attempt
        s_pids = [str(p) for p in solved]

        return (m_numrows, m_numcols, b_pids, s_pids, solved)

class Adversary_Gossip_Window(Gossip_Window):
    def __init__(self):
        #Gossip_Window.__init__(self)
        self.window_by_rnd = {}

    def add(self, rnd, src, p):
        if rnd not in self.window_by_rnd:
            self.window_by_rnd[rnd] = []

        self.window_by_rnd[rnd].append((src, p))
        return True


################################################################################

class Adversary(object):
    def __init__(self, victims, network, simlog, simstats, simParams):
        # Network, Log, Stats handles
        self.network = network
        self.victims = victims #the peers we can spy on
        self.simlog = simlog
        self.simstats = simstats

        # Simulation parameters
        self.simParams = simParams

        # Create a solved message window
        self.decoded_window = Gossip_Window()#Decoded_Window()

        # Create a gossip window
        self.gossip_window = Adversary_Gossip_Window()

        # Create an input queue
        self.queue = queue.Queue()

        #pmfs dict indexed by message pid (packet id). Contents for
        #each dict entry are a list of size numPeers, where each index
        #is a number from 0-1 and the sum of the whole list is 1
```

```
604         self.PMFs = {}
605
606     def put(self, p):
607         self.queue.put(p)
608
609     def simulate(self, rnd):
610         # Process all received gossip
611         while not self.queue.empty():
612             try: (src, gossip) = self.queue.get(False)
613             except queue.Empty: break
614
615             msgs = set()
616             #is there ever more than one LC?
617             for lc in gossip:
618                 self.gossip_window.add(rnd, src, lc.messages)
619                 for m in lc.messages:
620                     msgs.add(m)
621
622         self.makePMF(msgs, rnd)
623
624     def makePMF(self, msgs, rnd):
625         for m in msgs:
626             if m.pid in self.PMFs:
627                 #This implementation makes no changes after the initial PDF
628                 break
629             else:
630                 PMF = [0 for i in xrange(self.simParams['SIM_NUM_PEERS'])]
631                 msgs_per_peer = [p for src,p in self.gossip_window.window_by_rnd[rnd]]
632                 eligible = list(sorted(self.network.network.keys()))
633
634                 copies = 0
635                 for p in msgs_per_peer:
636                     if m in p:
637                         copies += 1
638
639                 #if we notice that the same message left more than one node in
640                 #the same round, we know it must not have been the first
641                 #instance, so remove our victim list from eligibles
642                 if copies > 1:
643                     for v in self.victims:
644                         eligible.remove(v)
645                 #divide probability evenly across list of eligible senders
646                 p = 1.0/float(len(eligible))
647                 for e in eligible:
648                     PMF[e] = p
649
650                 self.PMFs[m.pid] = PMF
651
652 ################################################################################
653
654 class Peer(object):
655     def __init__(self, nid, network, simlog, simstats, simParams, adversary):
656         # Our unique peer ID
657         self.nid = nid
658         # Network, Log, Stats handles
659         self.network = network
660         self.simlog = simlog
661         self.simstats = simstats
662         # Simulation parameters
663         self.simParams = simParams
664
665         self.ackLog = []
666
667         # the adversary
668         self.adversary = adversary
669         # Create an input queue
670         self.queue = queue.Queue()
671         # Create a solved message window
672         self.decoded_window = Decoded_Window()
673         # Create a gossip window
674         self.gossip_window = Gossip_Window()
675
676         # Initialize our window with dummy messages
677         for i in range(self.simParams['CODE_SIZE']):
678             self.decoded_window.add(DummyMessage(), self.simParams['TTL_DECODE'])
679
680         # Choose a random insert message timeout
681         self.insert_message_timeout = int(random.randint(0, int(self.simParams['CONTRIBUTE_INTERVAL'])))
682
683         # Join the network
684         self.network.join(0, nid, self.queue)
685
686     def simulate(self, rnd):
687         # Update the TTLs of our object windows
688         self.decoded_window.tick()
689         el = self.decoded_window.prune()
690         self.gossip_window.tick()
691         self.gossip_window.prune()
692         # Keep track of the last message disappearing
693         self.simstats.message_track(rnd, self.decoded_window.live_objects(),
694                                     self.gossip_window.live_objects())
695         # Log our window sizes
```

```
696         self.simstats.window_size(rnd, self.nid,
697                                   len(self.decoded_window.objects())-len(el),
698                                   len(self.gossip_window.objects()))
699
700         # Update our insert message timeout counter
701         if self.insert_message_timeout > 0:
702             self.insert_message_timeout -= 1
703
704         # Introduce a new message if our random insert timeout expired, but only
705         # if we've have a Decoded Window of at least CODE_SIZE
706         if self.insert_message_timeout == 0:
707             p = RealMessage(self.nid)
708             if random.random() <= self.simParams['FRACTION_DUMMY']:
709                 p = DummyMessage()
710             # Add a real message to our decoded window
711             if self.decoded_window.add(p, self.simParams['TTL_DECODE']):
712                 # Log the insert
713                 self.simlog.log(rnd, "insert", self.nid, str(p))
714                 self.simstats.message_insert(rnd, self.nid, str(p))
715
716             # Choose a new insert message timeout
717             self.insert_message_timeout = int(self.simParams['CONTRIBUTE_INTERVAL'])
718
719         # Process all received gossip
720         while not self.queue.empty():
721             try: (src, gossip) = self.queue.get(False)
722             except queue.Empty: break
723
724             # Add the linear combinations to our gossip window
725             for lc in gossip:
726                 self.gossip_window.add(src, lc, self.simParams['TTL_GOSSIP'])
727
728             # Log the receive
729             self.simlog.log(rnd, "receive", self.nid, "src: %d, gossip: %s" % (src, str([str(p) for p in gossip])))
730
731         # Try to solve some gossip
732         (m_numrows, m_numcols, b_pids, s_pids, solved) = self.gossip_window.solve(self.decoded_window)
733         # Log the reduce attempt
734         self.simlog.log(rnd, "reduce", self.nid, "%dx%d to %d" % (m_numrows, m_numcols, len(solved)))
735         self.simstats.matrix_reduce(rnd, self.nid, m_numrows, m_numcols, len(solved))
736         self.simlog.log(rnd, "reduce_info", self.nid, "%s to %s" % (str(b_pids), str(s_pids)))
737
738         # Add the decoded messages to our Decoded Window
739         for p in solved:
740             if self.decoded_window.add(p, self.simParams['TTL_DECODE']):
741                 # Log the decodes
742                 self.simlog.log(rnd, "decode", self.nid, str(p))
743                 if isinstance(p, RealMessage):
744                     self.simstats.message_decode(rnd, self.nid, str(p))
745
746         # Fill up Decoded Window with dummy messages if it is short
747         for i in range(self.simParams['CODE_SIZE'] - len(self.decoded_window.live_objects())):
748             self.decoded_window.add(DummyMessage(), self.simParams['TTL_DECODE'])
749
750         #create some gossip
751         gossip = self.makeGossip()
752
753         # Look up LOOKUP_PERCENT subset of peers on the network
754         dests = self.network.lookup_random(self.nid, int(self.simParams['LOOKUP_PERCENT']*self.simParams['SIM_NUM_PEERS']))
755         for d in dests:
756
757             ###################FAKE BROADCAST#####################
758             # #########comment out to do true broadcast###########
759             # gossip = self.makeGossip()
760             # ##################################################
761
762             # Transmit to the destination
763             d.put( (self.nid, gossip) )
764
765         #send to adversary
766         if self.nid in self.adversary.victims:
767             self.adversary.put((self.nid, gossip))
768
769     def makeGossip(self):
770         gossip = []
771         if random.getrandbits(1):
772             # Code new-gossip from our Decoded Window
773             gossip.append(RLC(self.decoded_window.choose_random(self.simParams['CODE_SIZE'])))
774         else:
775             # Choose re-gossip from our Gossip Window
776             if len(self.gossip_window.live_objects()) > 0:
777                 gossip.append(self.gossip_window.choose_random())
778
779         return gossip
780
781
782
783 ##############################################################################
784 def random_victims(npeers, percent, nvictims = 0):
785     choices = list(range(npeers))
786     random.shuffle(choices)
787     if nvictims != 0:
```

```
788         print "running for %i victims" % nvictims
789         return [choices[i] for i in xrange(nvictims)]
790     else:
791         return [choices[i] for i in xrange(int(npeers*percent))]
792
793 if __name__ == "__main__":
794     ff.FiniteFieldArray.ff_precompute()
795
796     # Make data and logs folders if they don't exist
797     if not os.path.exists("data/"): os.mkdir("data")
798     if not os.path.exists("logs/"): os.mkdir("logs")
799
800
801     for si in range(len(SimParamsList)):
802         simParams = SimParamsList[si]
803
804         # If we have already completed this simulation, skip it
805         if os.path.exists("data/%s-0.data" % simParams['NAME']):
806             continue
807
808         random.seed(simParams['SEED'])
809
810         # Simulation stop event set by simStats
811         simEventStop = threading.Event()
812
813         # Simulation objects
814         simStats = Stats(simParams, simEventStop)
815         simLog = Log(simParams)
816         simNetwork = Network(simLog, simStats)
817
818         simPeers = []
819         #create adversary
820         percent_victims = 0.25
821         nvicts = 0
822         victims = random_victims(simParams['SIM_NUM_PEERS'], percent_victims,
823                                  nvictims = nvicts)
824         adversary = Adversary(victims, simNetwork, simLog, simStats, simParams)
825         # Add cooperative peers to the network
826         for i in range(simParams['SIM_NUM_PEERS'] - simParams['SIM_NUM_EVIL_PEERS']):
827             simPeers.append(Peer(i, simNetwork, simLog, simStats, simParams, adversary))
828         # Add evil peers to the network
829         for i in range(simParams['SIM_NUM_EVIL_PEERS']):
830             if simParams['SIM_EVIL_PEER_TYPE'] == "inactive":
831                 evilPeer = EvilPeer_Inactive
832             elif simParams['SIM_EVIL_PEER_TYPE'] == "underdetermined":
833                 evilPeer = EvilPeer_Underdetermined
834             elif simParams['SIM_EVIL_PEER_TYPE'] == "decodable":
835                 evilPeer = EvilPeer_Decodable
836
837             simPeers.append(evilPeer(i + (simParams['SIM_NUM_PEERS'] -
838                                           simParams['SIM_NUM_EVIL_PEERS']),
839                                      simNetwork, simLog, simStats, simParams))
840
841         print("\nStarting simulation %d / %d: %s" % (si+1, len(SimParamsList), simParams['NAME']))
842
843         roundCount = 0
844         while True:
845             # Simulate the peers in a different order each round
846             random.shuffle(simPeers)
847             for n in simPeers:
848                 n.simulate(roundCount)
849
850             adversary.simulate(roundCount)
851
852             sys.stdout.write("\r%d, %d -- Round %d" % \
853                 (len(simStats.message_inserts), len(simStats.message_decodes), roundCount+1))
854             # Stop the simulation if we've collected enough data
855             if simEventStop.is_set():
856                 break
857
858             roundCount += 1
859
860         #insert the PMF data into stats
861         simStats.PMFs = adversary.PMFs
862
863         # Log the finish at this round count
864         simStats.finished(roundCount)
865         simLog.log(roundCount, "finish", 0, "")
866
867         print()
868
869         # Dump stats
870         print("Wrote stats to %s" % simStats.dump())
871         # Dump log
872         print("Wrote log to %s" % simLog.dump())
```

B.2 chapsim.py

```python
#!/usr/bin/env python2

# CHAP simulator, by Colleen Josephson (2014)
# based on the NCGAB Simulator by Ivan A. Sergeev

import threading
import random
import json
import sys
sys.path.insert(0, '/usr/lib64/python2.7/site-packages')
import copy
import os
import argparse

if sys.version_info.major == 2:
    import Queue as queue
else:
    import queue

import ff

import termcolor


################################################################################

simConfig = 1
clique = "clique"
circle = "circle"
grid = "grid"

# Cooperative Peer Simulation

SimTemplate = {
    'NAME':                     "",
    'DESC':                     "",
    'TOPOLOGY':                 "",

    'LOOKUP_PERCENT':           0.25,
    'CODE_SIZE':                4,
    'TTL_DECODE':               30,
    'TTL_GOSSIP':               10,

    'SIM_NUM_PEERS':            0,
    'SIM_NUM_EVIL_PEERS':       0,
    'CONTRIBUTE_INTERVAL':      0,

    'SIM_WARMUP_DECODES':       50,
    'SIM_DURATION_INSERTS':     100,

    'SEED':                     0,

    'PRINT_LOG':                False,
}

if simConfig == 1:
    #NumPeersSweep = [10, 25, 50]
    ContributeIntervalSweep = [7.0, 10, 12.0, 15.0, 17.0, 20.0, 22.0]
    SeedSweep = [0x1, 0x2, 0x3]

    SimParamsList = []
    #for numPeers in NumPeersSweep:
    for contributeInterval in ContributeIntervalSweep:
        for seed in SeedSweep:
            name = "Sim-CI%d-Seed%d" % (contributeInterval, seed)
            params = copy.deepcopy(SimTemplate)
            params['NAME'] = name
            params['SEED'] = seed
            params['SIM_NUM_PEERS'] = 0
            params['CONTRIBUTE_INTERVAL'] = contributeInterval
            SimParamsList.append(params)

if simConfig == 7:
    NumPeersSweep = [10, 15, 25]
    ContributeIntervalSweep = [7.0, 10, 12.0, 15.0, 17.0, 20.0, 22.0]
    SeedSweep = [0x1, 0x2, 0x3]

    SimParamsList = []
    for numPeers in NumPeersSweep:
        for contributeInterval in ContributeIntervalSweep:
            for seed in SeedSweep:
                name = "SimSelect2-N%d-R%d-S%d" % (numPeers, contributeInterval, seed)
                params = copy.deepcopy(SimTemplate)
                params['NAME'] = name
```

```
85                 params['SEED'] = seed
86                 #params['SELECTION'] = 'Uniform'
87                 params['SELECTION'] = 'Weighted'
88                 params['SIM_NUM_PEERS'] = numPeers
89                 params['CONTRIBUTE_INTERVAL'] = contributeInterval
90                 SimParamsList.append(params)
91
92  ##############################################################################
93
94  class Stats(object):
95      def __init__(self, simParams, simEventStop):
96          # Statistics collected
97          self.message_inserts = []
98          self.message_decodes = {}
99          self.matrix_reduces = []
100         self.window_sizes = []
101         self.last_message_exists = {}
102         self.round_finished = 0
103
104         # Simulation parameters
105         self.simParams = simParams
106
107         self.simEventStop = simEventStop
108
109     ##########################
110
111     def message_track(self, rnd, decoded_objects, gossip_objects):
112         if len(self.message_decodes) < self.simParams['SIM_WARMUP_DECODES']:
113             return
114         if len(self.message_inserts) < self.simParams['SIM_DURATION_INSERTS']:
115             return
116
117         # If the last message disappeared in the previous round, signal
118         # simulation stop
119         if rnd > 0 and (rnd-1) in self.last_message_exists:
120             if self.last_message_exists[rnd-1] == False:
121                 self.simEventStop.set()
122                 return
123
124         # If we already saw the last message in this round
125         if rnd in self.last_message_exists and self.last_message_exists[rnd] == True:
126                 return
127
128         # Look for all messages inserted inside peers on the network
129         for (_, _, last_pid) in self.message_inserts:
130             # Look for the inserted message in the decoded objects
131             for p in decoded_objects:
132                 if str(p) == last_pid:
133                     self.last_message_exists[rnd] = True
134                     return
135             # Look for the inserted message in the gossip objects
136             for lc in gossip_objects:
137                 for p in lc.messages:
138                     if str(p) == last_pid:
139                         self.last_message_exists[rnd] = True
140                         return
141
142         self.last_message_exists[rnd] = False
143
144     def message_insert(self, rnd, nid, pid):
145         if len(self.message_decodes) < self.simParams['SIM_WARMUP_DECODES']:
146             return
147
148         # If we have already collected our simulation number of inserts
149         if len(self.message_inserts) >= self.simParams['SIM_DURATION_INSERTS']:
150             return
151
152         self.message_inserts.append( (rnd, nid, pid) )
153
154     def message_decode(self, rnd, nid, pid):
155         # If we hadn't started recording this PID yet
156         if pid not in self.message_decodes:
157             self.message_decodes[pid] = []
158
159         # If this nid already decoded this pid in the past
160         if nid in [ n for (_, n) in self.message_decodes[pid] ]:
161             return
162
163         self.message_decodes[pid].append( (rnd, nid) )
164
165     def matrix_reduce(self, rnd, nid, numrows, numcols, numsolved):
166         self.matrix_reduces.append( (rnd, nid, numrows, numcols, numsolved) )
167
168     def window_size(self, rnd, nid, solved_size, gossip_size):
169         self.window_sizes.append( (rnd, nid, solved_size, gossip_size) )
170
171     def finished(self, rnd):
172         self.round_finished = rnd
173
174     #########################
175
176     def dump(self):
```

```
        i = 0
        while True:
            path = "%s/%s-%d.data" % (dataDir, self.simParams['NAME'], i)
            if not os.path.exists(path): break
            i += 1

        f = open(path, "w")
        data = {'simParams': self.simParams,
                'message_inserts': self.message_inserts,
                'message_decodes': self.message_decodes,
                'matrix_reduces': self.matrix_reduces,
                'window_sizes': self.window_sizes,
                'round_finished': self.round_finished}
        f.write(json.dumps(data))
        f.close()

        return path

################################################################################

class Log(object):
    def __init__(self, simParams):
        self.elog = []
        self.simParams = simParams

    def log(self, rnd, etype, enid, emsg):
        e = {}
        e['time'] = rnd
        e['type'] = etype
        e['nid'] = enid
        e['msg'] = emsg
        ejson = json.dumps(e)
        self.elog.append(ejson)

        if self.simParams['PRINT_LOG']:
            if etype == "join" or etype == "leave": print(termcolor.colored(ejson, "yellow"))
            elif etype == "receive": print(termcolor.colored(ejson, "yellow"))
            elif etype == "insert": print(termcolor.colored(ejson, "red"))
            elif etype == "reduce": print(termcolor.colored(ejson, "blue"))
            elif etype == "decode": print(termcolor.colored(ejson, "green"))
            else: print(termcolor.colored(ejson, "white"))

    def dump(self):
        i = 0
        while True:
            path = "%s/%s-%d.log" % (logDir, self.simParams['NAME'], i)
            if not os.path.exists(path): break
            i += 1

        f = open(path, "w")
        for e in self.elog:
            f.write(e + "\n")
        f.close()

        return path

################################################################################

class Network(object):
    def __init__(self, simlog, simstats):
        self.simlog = simlog
        self.simstats = simstats
        self.network = {}

    def join(self, rnd, nid, q):
        self.network[nid] = q
        self.simlog.log(rnd, "join", nid, "")

    def leave(self, rnd, nid):
        if nid in self.network:
            del self.network[nid]
            self.simlog.log(rnd, "leave", nid, "")

    def lookup_random(self, nid, n):
        choices = list(self.network.keys())
        choices.remove(nid)
        random.shuffle(choices)
        return [self.network[choices[i]] for i in range(n)]

    def get_lookup_percent(self):
        if len(self.network) <= 5:
            return 1*(len(self.network)-1)
        elif len(self.network) >= 10:
            return 0.25*len(self.network)
        else: return 0.5*len(self.network)

################################################################################

class Message(object):
    def __init__(self, pid):
        self.pid = pid

```

```
    def __str__(self):
        return self.pid

    def __eq__(self, other):
        if (self.pid == other.pid):
            return True
        return False

class DummyMessage(Message):
    def __init__(self):
        Message.__init__(self, "R%04x" % random.getrandbits(32))

class RealMessage(Message):
    def __init__(self, nid):
        Message.__init__(self, "M%d/%04x" % (nid, random.getrandbits(32)))

class RLC(Message):
    def __init__(self, messages):
        coefs = [random.randint(0, 255) for p in messages]
        pid = "LC/(" + ",".join([p.pid for p in messages]) + ")" + "/(" + \
                    ",".join(["%02x" % c for c in coefs]) + ")"
        Message.__init__(self, pid)
        self.messages = messages
        self.coefs = coefs

################################################################################

def choose_weighted_random(objects, scores):
    scores_cdf = []
    for s in scores[:]:
        scores_cdf.append(sum(scores_cdf) + s)

    random_score = random.random()*scores_cdf[-1]
    for index in range(len(scores_cdf)):
        if scores_cdf[index] > random_score:
            break

    return index

class Window(object):
    def __init__(self, keep_expired = False):
        self.window = []
        self.ttl = {}
        self.keep_expired = keep_expired

    def add(self, p, ttl):
        if type(p) == type(None) or p in self.window:
            return False

        self.window.append(p)
        self.ttl[str(p)] = ttl
        return True

    def prune(self):
        if self.keep_expired:
            return []

        # Create a list of expired objects
        expiredList = list(filter(lambda x: self.ttl[str(x)] == 0, self.window))

        # Remove all expired objects from our window
        for e in expiredList:
            self.window.remove(e)
            del self.ttl[str(e)]

        return expiredList

    def tick(self):
        # Decrement the TTL for each object in the window
        for x in self.window:
            self.ttl[str(x)] = max(0, self.ttl[str(x)] - 1)

        return self.prune()

    def live_objects(self):
        return list(filter(lambda x: self.ttl[str(x)] > 0, self.window))

    def objects(self):
        return self.window[:]

    def __str__(self):
        s = "Window\n"
        for x in self.window:
            s += "\t" + str(x) + " TTL: " + str(self.ttl[str(x)]) + "\n"
        return s

class Decoded_Window(Window):
    def __init__(self):
        Window.__init__(self, keep_expired = True)

    def choose_random_uniform(self, n):
        choices = self.live_objects()
```

```
        random.shuffle(choices)
        return choices[0 : min(n, len(choices))]

    def choose_random(self, n):
        choices = self.live_objects()
        scores = [self.ttl[str(c)] for c in choices]

        n = min(n, len(choices))
        chosen = []
        for i in range(n):
            k = choose_weighted_random(choices, scores)
            chosen.append(choices[k])
            del choices[k]
            del scores[k]

        return chosen

class Gossip_Window(Window):
    def __init__(self):
        Window.__init__(self, keep_expired = False)
        # Keep active objects in a dictionary by source as well
        self.active_objects_by_source = {}

    def add(self, src, p, ttl):
        if Window.add(self, p, ttl) == True:
            # Create a new list for the source if it's not in our dictionary
            if src not in self.active_objects_by_source:
                self.active_objects_by_source[src] = []

            # Add the message to the source's list
            self.active_objects_by_source[src].append(p)
            return True

        return False

    def prune(self):
        expiredList = Window.prune(self)
        # For each expired object
        for e in expiredList:
            for src in self.active_objects_by_source:
                # If the object exists in this source's list
                if e in self.active_objects_by_source[src]:
                    # Delete it
                    self.active_objects_by_source[src].remove(e)
                    # Delete the source's list if it's empty now
                    if len(self.active_objects_by_source[src]) == 0:
                        del self.active_objects_by_source[src]
                    break
        return expiredList

    def choose_random_uniform(self):
        # Choose a random source
        source = random.choice(list(self.active_objects_by_source.keys()))

        # Gather a list of objects by this source
        objects = self.active_objects_by_source[source]

        return random.choice(objects)

    def choose_random(self):
        # Choose a random source
        source = random.choice(list(self.active_objects_by_source.keys()))

        # Gather a list of objects by this source
        choices = self.active_objects_by_source[source]
        scores = [self.ttl[str(c)] for c in choices]

        return choices[choose_weighted_random(choices, scores)]

    def fast_rref(self, m, b, x):
        done = False
        pi = 0

        # Iterate through each row
        for j in range(len(m)):
            # While we do not have a pivot for this row
            while m[j][pi] == 0:
                # Find a row below to swap with for a pivot at pi
                for k in range(j+1, len(m)):
                    if m[k][pi] != 0:
                        # Swap with this row
                        (m[j], m[k]) = (m[k], m[j])
                        (b[j], b[k]) = (b[k], b[j])
                        break

                # Increment pivot index if we could not find a row to swap with
                if m[j][pi] == 0:
                    pi += 1

                # If there is no pivots left, we're done reducing
                if pi == len(m[0]):
                    done = True
```

```
                    break

            if done:
                break

            # Divide through to have a pivot of 1
            m[j] = [ ff.FiniteFieldArray.ff_elem_div(m[j][i], m[j][pi]) for i in range(len(m[0])) ]

            # Eliminate above & below
            for k in range(len(m)):
                if k != j and m[k][pi] != 0:
                    m[k] = [ ff.FiniteFieldArray.ff_elem_sub(m[k][i], \
                      ff.FiniteFieldArray.ff_elem_mul(m[j][i], m[k][pi])) for i in range(len(m[0])) ]

            # Move onto the next pivot
            pi += 1
            # If there is no pivots left, we're done reducing
            if pi == len(m[0]):
                break

        solved = []
        used = []

        for i in range(len(m)):
            # If this row has only one non-zero entry
            reduced_coefs = [1*(m[i][j] != 0) for j in range(len(m[0]))]
            if sum(reduced_coefs) == 1:
                # Add the solution to our solved list
                solved.append(x[reduced_coefs.index(1)])
                # Add the decoded LC to our used list
                used.append(b[i])

        return (solved, used)

    def solve(self, decoded_window):
        unsolved_message_map = {}

        # Get a list of decoded messages
        decoded_messages = decoded_window.objects()

        # Make a list of undecoded linear combinations
        undecoded_lc = []
        for lc in self.live_objects():
            # If we've decoded this entire linear combination, don't add it
            if sum([p in decoded_messages for p in lc.messages]) == len(lc.messages):
                continue
            undecoded_lc.append(lc)

        # Put together a list of all decoded messages referenced by undecoded
        # linear combinations
        ref_decoded_messages = []
        for lc in undecoded_lc:
            for p in lc.messages:
                if p in decoded_messages and p not in ref_decoded_messages:
                    ref_decoded_messages.append(p)

        # Assemble x of mx=b
        #   x maps messages to column indices
        x = {}
        # Assign a col index to each referenced decoded message
        for p in ref_decoded_messages:
            x[str(p)] = len(x)
        # Assign a col index to each undecoded message
        for lc in undecoded_lc:
            for p in lc.messages:
                if str(p) not in x:
                    x[str(p)] = len(x)
                    unsolved_message_map[str(p)] = p

        # Assemble m and b of mx = b
        m = []
        b = []
        # Create a row for each solved message
        for p in ref_decoded_messages:
            r = [0] * len(x)
            r[x[str(p)]] = 1
            m.append(r)
            b.append(p)
        # Create a row for each linear combination
        for lc in undecoded_lc:
            r = [0] * len(x)
            for i in range(len(lc.messages)):
                pid = str(lc.messages[i])
                coef = lc.coefs[i]
                r[x[pid]] = coef
            m.append(r)
            b.append(lc)

        # Information about the size of this reduce attempt
        b_pids = [str(p) for p in b]
        m_numrows = len(m)
        m_numcols = len(m[0]) if m_numrows > 0 else 0
```

```
        (solved, used) = self.fast_rref(m, b, sorted(x, key=x.get))

        # Remove previously decoded objects from our solution
        for p in ref_decoded_messages:
            if str(p) in solved:
                solved.remove(str(p))
            if p in used:
                used.remove(p)

        # Map message ids to message objects in our solution
        for i in range(len(solved)):
            solved[i] = unsolved_message_map[solved[i]]

        # Information about the solution of this reduce attempt
        s_pids = [str(p) for p in solved]

        return (m_numrows, m_numcols, b_pids, s_pids, solved)

##########################################################################################

class Peer(object):
    def __init__(self, nid, simlog, simstats, simParams, contributor = False):
        # Our unique peer ID
        self.nid = nid
        # Network, Log, Stats handles
        self.simlog = simlog
        self.simstats = simstats
        # Simulation parameters
        self.simParams = simParams

        # Create an input queue
        self.queue = queue.Queue()
        # Create a solved message window
        self.decoded_window = Decoded_Window()
        # Create a gossip window
        self.gossip_window = Gossip_Window()

        self.contributor = contributor

        # Initialize our window with dummy messages
        for i in range(self.simParams['CODE_SIZE']):
            self.decoded_window.add(DummyMessage(), self.simParams['TTL_DECODE'])

        # Choose a random insert message timeout
        self.insert_message_timeout = int(random.randint(0, int(self.simParams['CONTRIBUTE_INTERVAL'])))

    def send(self, dst, gossip):
        dst.queue.put((self.nid, gossip))

    def simulate(self, rnd):
        # Update the TTLs of our object windows
        self.decoded_window.tick()
        self.decoded_window.prune()
        self.gossip_window.tick()
        self.gossip_window.prune()
        # Keep track of the last message disappearing
        self.simstats.message_track(rnd, self.decoded_window.live_objects(), self.gossip_window.live_objects())
        # Log our window sizes
        self.simstats.window_size(rnd, self.nid, len(self.decoded_window.objects()), len(self.gossip_window.objects()))

        if self.contributor:
            # Update our insert message timeout counter
            if self.insert_message_timeout > 0:
                self.insert_message_timeout -= 1

            # Introduce a new message if our random insert timeout expired, but only
            # if we've have a Decoded Window of at least CODE_SIZE
            if self.insert_message_timeout == 0:
                p = RealMessage(self.nid)
                # Add a real message to our decoded window
                if self.decoded_window.add(p, self.simParams['TTL_DECODE']):
                    # Log the insert
                    self.simlog.log(rnd, "insert", self.nid, str(p))
                    self.simstats.message_insert(rnd, self.nid, str(p))

                # Choose a new insert message timeout
                #self.insert_message_timeout = int(random.expovariate(1.0 / float(self.simParams['CONTRIBUTE_INTERVAL'])))
                self.insert_message_timeout = int(self.simParams['CONTRIBUTE_INTERVAL'])

        # Process all received gossip
        while not self.queue.empty():
            try: (src, gossip) = self.queue.get(False)
            except queue.Empty: break

            # Add the linear combinations to our gossip window
            for lc in gossip:
                self.gossip_window.add(src, lc, self.simParams['TTL_GOSSIP'])

            # Log the receive
            self.simlog.log(rnd, "receive", self.nid, "src: %d, gossip: %s" % (src, str([str(p) for p in gossip])))

```

```
        # Try to solve some gossip
        (m_numrows, m_numcols, b_pids, s_pids, solved) = self.gossip_window.solve(self.decoded_window)
        # Log the reduce attempt
        self.simlog.log(rnd, "reduce", self.nid, "%dx%d to %d" % (m_numrows, m_numcols, len(solved)))
        self.simstats.matrix_reduce(rnd, self.nid, m_numrows, m_numcols, len(solved))
        self.simlog.log(rnd, "reduce_info", self.nid, "%s to %s" % (str(b_pids), str(s_pids)))

        # Add the decoded messages to our Decoded Window
        for p in solved:
            if self.decoded_window.add(p, self.simParams['TTL_DECODE']):
                # Log the decodes
                self.simlog.log(rnd, "decode", self.nid, str(p))
                if isinstance(p, RealMessage):
                    self.simstats.message_decode(rnd, self.nid, str(p))

        # Fill up Decoded Window with dummy messages if it is short
        for i in range(self.simParams['CODE_SIZE'] - len(self.decoded_window.live_objects())):
            self.decoded_window.add(DummyMessage(), self.simParams['TTL_DECODE'])

class WiredPeer(Peer):
    def __init__(self, nid, simNetwork, simLog, simStats, simParams, contributor = False):
        Peer.__init__(self, nid, simLog, simStats, simParams, contributor)
        # Join the network
        self.network = simNetwork
        #   def join(self, rnd, nid, q):
        self.network.join(0, nid, self)

    def simulate(self, rnd):
        Peer.simulate(self, rnd)
        # Look up LOOKUP_PERCENT subset of peers on the network
        dests = self.network.lookup_random(self.nid, int(self.network.get_lookup_percent()))
        for d in dests:
            gossip = []

            if random.getrandbits(1):
                # Code new-gossip from our Decoded Window
                gossip.append(RLC(self.decoded_window.choose_random(self.simParams['CODE_SIZE'])))
            else:
                # Choose re-gossip from our Gossip Window
                if len(self.gossip_window.live_objects()) > 0:
                    gossip.append(self.gossip_window.choose_random())

            # Transmit to the destination
            self.send(d, gossip)


class CloudPeer(Peer):
    def __init__(self, nid, simCloud, simLog, simStats, simParams, contributor = False):
        Peer.__init__(self, nid, simLog, simStats, simParams, contributor)
        self.cloud = simCloud
        self.cloud.join(0, self)

    def simulate(self, rnd):
        Peer.simulate(self, rnd)
        dests = self.cloud.getDests(self.nid)

        gossip = []
        #This part goes outside the for loop because the same gossip
        #will be broadcast to everyone.
        if random.getrandbits(1):
            # Code new-gossip from our Decoded Window
            gossip.append(RLC(self.decoded_window.choose_random(self.simParams['CODE_SIZE'])))
        else:
            # Choose re-gossip from our Gossip Window
            if len(self.gossip_window.live_objects()) > 0:
                gossip.append(self.gossip_window.choose_random())

        if self.cloud.adj == None: #assume clique
            for d in dests:
                if d.nid != self.nid:
                    # Transmit to the destination
                    self.send(d, gossip)
        else:
            #assumes the nodes in the cloud have contiguous NIDs
            adj = self.cloud.adj[self.nid - dests[0].nid]
            for i in xrange(len(adj)):
                if  adj[i] == 1:
                    self.send(dests[i], gossip)


class CloudExitPeer(Peer):
    def __init__(self, nid, simCloud, simNetwork, simLog, simStats, simParams):
        Peer.__init__(self, nid, simLog, simStats, simParams)
        self.cloud = simCloud
        self.cloud.join(0, self)
        self.network = simNetwork
        #def join(self, rnd, nid, q):
        self.network.join(0, nid, self)

    def simulate(self, rnd):
        Peer.simulate(self, rnd)

```

```
        ##cloud portion#################################
        dests = self.cloud.getDests(self.nid)
        gossip = []
        #This part goes outside the for loop because the same gossip
        #will be broadcast to everyone.
        if random.getrandbits(1):
            # Code new-gossip from our Decoded Window
            gossip.append(RLC(self.decoded_window.choose_random(self.simParams['CODE_SIZE'])))
        else:
            # Choose re-gossip from our Gossip Window
            if len(self.gossip_window.live_objects()) > 0:
                gossip.append(self.gossip_window.choose_random())

        for d in dests:
            # Transmit to the destination
            if d.nid != self.nid:
                self.send(d, gossip)
        ################################################

        ##wired portion##################################
        dests = self.network.lookup_random(self.nid, int(self.network.get_lookup_percent()))
        for d in dests:
            gossip = []

            if random.getrandbits(1):
                # Code new-gossip from our Decoded Window
                gossip.append(RLC(self.decoded_window.choose_random(self.simParams['CODE_SIZE'])))
            else:
                # Choose re-gossip from our Gossip Window
                if len(self.gossip_window.live_objects()) > 0:
                    gossip.append(self.gossip_window.choose_random())

            # Transmit to the destination
            self.send(d, gossip)
        ################################################

class Cloud(object):
    #an ad-hoc cloud of wireless hosts

    def __init__(self, nid, name, simlog, network, topo = clique, adj = None):
        self.hosts = []
        self.nid = nid
        self.name = name
        self.simlog = simlog
        self.topo = topo
        self.adj = adj #2

    #hosts join the cloud
    def join(self, gossip_round, h):
        #a host 'h' with nid 'nid' joins the network
        self.hosts.append(h)
        self.simlog.log(gossip_round, "join", h.nid, "")

    #hosts leave the cloud
    def leave(self, nid, gossip_round):
        for h in self.hosts:
            if h.nid == nid:
                h.remove(h)
                self.simlog.log(gossip_round, "leave", nid, "")

    def getDests(self, nid):
        s = sorted(self.hosts, key=lambda d: d.nid )
        return s

    #run simulate function on all hosts in the cloud
    def simulate(self, rnd):
        pass



################################################################################

if __name__ == "__main__":
    parser = argparse.ArgumentParser(description='Python network simulator')
    parser.add_argument('topologies', metavar='N', type=int, nargs='+',
                help='[REQUIRED] space-separated topology numbers to simulate (see code for topologies)')
    #parser.add_argument("--peers", "-p", metavar='P', type=int, nargs='+', help="[OPTIONAL] Specify the number of peers for the
          topology, if applicable")

    args=parser.parse_args()
    simTopologies = args.topologies
    peers = args.peers

    ff.FiniteFieldArray.ff_precompute()

    def generatePeers(topology, simLog, simNetwork, simStats, simParams):
        '''
        Topology 1: wired network
        '''
        if topology == -1:#1:
            simPeers = []
            if peers: n = peers.pop(0)
```

```
            else: n = 10
            for i in xrange(n):
                simPeers.append(WiredPeer(i, simNetwork, simLog, simStats,
                                          simParams, contributor = True))
            return simPeers

        '''
        Topology 10: wireless cloud w/ all contributing
        '''
        if topology == 10:
            simPeers = []
            cloud1 = Cloud(1, 'cloud1', simLog, simNetwork)
            if peers: n = peers.pop(0)
            else: n = 10

            for i in xrange(10):
                simPeers.append(CloudPeer(i, cloud1, simLog, simStats, simParams, contributor = True))
            return simPeers

        '''
        Topology 11: wireless cloud w/ all contributing
        '''
        if topology == 11:
            simPeers = []
            cloud1 = Cloud(1, 'cloud1', simLog, simNetwork)
            if peers: n = peers.pop(0)
            else: n = 15

            for i in xrange(15):
                simPeers.append(CloudPeer(i, cloud1, simLog, simStats, simParams, contributor = True))
            return simPeers

        '''
        Topology 12: wireless cloud w/ all contributing
        '''
        if topology == 12:
            simPeers = []
            cloud1 = Cloud(1, 'cloud1', simLog, simNetwork)
            if peers: n = peers.pop(0)
            else: n = 20

            for i in xrange(20):
                simPeers.append(CloudPeer(i, cloud1, simLog, simStats, simParams, contributor = True))
            return simPeers


        '''
        Topology 2: wireless cloud with two contributing
        '''
        if topology == 2:
            simPeers = []
            cloud1 = Cloud(1, 'cloud1', simLog, simNetwork)
            if peers: n = peers.pop(0)
            else: n = 10

            for i in xrange(10):
                simPeers.append(CloudPeer(i, cloud1, simLog, simStats, simParams))
            for i in xrange(2):
                simPeers[i].contributor = True
            return simPeers

        '''
        Topology 3: A circle with all nodes contributing
        '''
        if topology == 3:
            simPeers = []
            a = [[0,1,0,0,0,0,0,0,0,1],
                 [1,0,1,0,0,0,0,0,0,0],
                 [0,1,0,1,0,0,0,0,0,0],
                 [0,0,1,0,1,0,0,0,0,0],
                 [0,0,0,1,0,1,0,0,0,0],
                 [0,0,0,0,1,0,1,0,0,0],
                 [0,0,0,0,0,1,0,1,0,0],
                 [0,0,0,0,0,0,1,0,1,0],
                 [0,0,0,0,0,0,0,1,0,1],
                 [1,0,0,0,0,0,0,0,1,0]]

            cloud1 = Cloud(1, 'cloud1', simLog, simNetwork, adj = a)
            if peers: n = peers.pop(0)
            else: n = 10

            for i in xrange(10):
                simPeers.append(CloudPeer(i, cloud1, simLog, simStats, simParams,
                                          contributor = True))
            return simPeers

        '''
        Topology 4: A circle with all nodes contributing
        '''
        if topology == 4:
            simPeers = []
            a = [[0,1,0,0,0,0,0,0,0,0,0,0,0,1],
```

```
                [1,0,1,0,0,0,0,0,0,0,0,0,0,0],
                [0,1,0,1,0,0,0,0,0,0,0,0,0,0],
                [0,0,1,0,1,0,0,0,0,0,0,0,0,0],
                [0,0,0,1,0,1,0,0,0,0,0,0,0,0],
                [0,0,0,0,1,0,1,0,0,0,0,0,0,0],
                [0,0,0,0,0,1,0,1,0,0,0,0,0,0],
                [0,0,0,0,0,0,1,0,1,0,0,0,0,0],
                [0,0,0,0,0,0,0,1,0,1,0,0,0,0],
                [0,0,0,0,0,0,0,0,1,0,1,0,0,0],
                [0,0,0,0,0,0,0,0,0,1,0,1,0,0],
                [0,0,0,0,0,0,0,0,0,0,1,0,1,0],
                [0,0,0,0,0,0,0,0,0,0,0,1,0,1],
                [1,0,0,0,0,0,0,0,0,0,0,0,1,0]]

            cloud1 = Cloud(1, 'cloud1', simLog, simNetwork, adj = a)
            if peers: n = peers.pop(0)
            else: n = 15

            for i in xrange(15):
                simPeers.append(CloudPeer(i, cloud1, simLog, simStats, simParams,
                                          contributor = True))
            return simPeers


        '''
        Topology 5: A circle with all nodes contributing
        '''
        if topology == 5:
            simPeers = []
            a = [[0,1,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,1],
                 [1,0,1,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0],
                 [0,1,0,1,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0],
                 [0,0,1,0,1,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0],
                 [0,0,0,1,0,1,0,0,0,0,0,0,0,0,0,0,0,0,0,0],
                 [0,0,0,0,1,0,1,0,0,0,0,0,0,0,0,0,0,0,0,0],
                 [0,0,0,0,0,1,0,1,0,0,0,0,0,0,0,0,0,0,0,0],
                 [0,0,0,0,0,0,1,0,1,0,0,0,0,0,0,0,0,0,0,0],
                 [0,0,0,0,0,0,0,1,0,1,0,0,0,0,0,0,0,0,0,0],
                 [0,0,0,0,0,0,0,0,1,0,1,0,0,0,0,0,0,0,0,0],
                 [0,0,0,0,0,0,0,0,0,1,0,1,0,0,0,0,0,0,0,0],
                 [0,0,0,0,0,0,0,0,0,0,1,0,1,0,0,0,0,0,0,0],
                 [0,0,0,0,0,0,0,0,0,0,0,1,0,1,0,0,0,0,0,0],
                 [0,0,0,0,0,0,0,0,0,0,0,0,1,0,1,0,0,0,0,0],
                 [0,0,0,0,0,0,0,0,0,0,0,0,0,1,0,1,0,0,0,0],
                 [0,0,0,0,0,0,0,0,0,0,0,0,0,0,1,0,1,0,0,0],
                 [0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,1,0,1,0,0],
                 [0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,1,0,1,0],
                 [0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,1,0,1],
                 [1,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,1,0]]


            cloud1 = Cloud(1, 'cloud1', simLog, simNetwork, adj = a)
            if peers: n = peers.pop(0)
            else: n = 20

            for i in xrange(20):
                simPeers.append(CloudPeer(i, cloud1, simLog, simStats, simParams,
                                          contributor = True))
            return simPeers


        '''
        Topology 22: A clique with 5 nodes contributing
        '''
        if topology == 22:
            simPeers = []
            cloud1 = Cloud(1, 'cloud1', simLog, simNetwork)
            if peers: n = peers.pop(0)
            else: n = 10

            for i in xrange(10):
                simPeers.append(CloudPeer(i, cloud1, simLog, simStats, simParams))
            for i in xrange(5):
                simPeers[i].contributor = True
            return simPeers

        '''
        Topology 23: A circle with 5 nodes contributing
        '''
        if topology == 23:
            simPeers = []
            a = [[0,1,0,1,0,0,0,0,0],
                 [1,0,1,0,1,0,0,0,0],
                 [0,1,0,0,0,1,0,0,0],
                 [1,0,1,0,1,0,1,0,0],
                 [0,1,0,1,0,1,0,1,0],
                 [0,0,1,0,1,0,0,0,1],
                 [0,0,0,1,0,0,0,1,0],
                 [0,0,0,0,1,0,1,0,1],
                 [0,0,0,0,0,1,0,1,0]]

```

```
        cloud1 = Cloud(1, 'cloud1', simLog, simNetwork, adj = a)
        if peers: n = peers.pop(0)
        else: n = 9

        for i in xrange(9):
            simPeers.append(CloudPeer(i, cloud1, simLog, simStats, simParams))
        for i in xrange(5):
            simPeers[i].contributor = True
        return simPeers

    '''
    Topology 6: A fully-connected grid with all nodes contributing
    '''
    if topology == 6:
        simPeers = []
        a = [[0,1,0,1,0,0,0,0,0],
             [1,0,1,0,1,0,0,0,0],
             [0,1,0,0,0,1,0,0,0],
             [1,0,1,0,1,0,1,0,0],
             [0,1,0,1,0,1,0,1,0],
             [0,0,1,0,1,0,0,0,1],
             [0,0,0,1,0,0,0,1,0],
             [0,0,0,0,1,0,1,0,1],
             [0,0,0,0,0,1,0,1,0]]

        cloud1 = Cloud(1, 'cloud1', simLog, simNetwork, adj = a)
        if peers: n = peers.pop(0)
        else: n = 9

        for i in xrange(9):
            simPeers.append(CloudPeer(i, cloud1, simLog, simStats, simParams,
                                      contributor = True))
        return simPeers


    '''
    Topology 7: A fully-connected grid with all nodes contributing
    '''
    if topology == 7:
        simPeers = []
        ######1,2,3,4,5,6,7,8,9,0,1,2,3,4,5,6
        a = [[0,1,0,0,1,0,0,0,0,0,0,0,0,0,0,0],
             [1,0,1,0,0,1,0,0,0,0,0,0,0,0,0,0],
             [0,1,0,1,0,0,1,0,0,0,0,0,0,0,0,0],
             [0,0,1,0,0,0,0,1,0,0,0,0,0,0,0,0],
             [1,0,0,0,0,1,0,0,1,0,0,0,0,0,0,0],
             [0,1,0,0,1,0,1,0,0,1,0,0,0,0,0,0],
             [0,0,1,0,0,1,0,1,0,0,1,0,0,0,0,0],
             [0,0,0,1,0,0,1,0,0,0,0,1,0,0,0,0],
             [0,0,0,0,1,0,0,0,0,1,0,0,1,0,0,0],
             [0,0,0,0,0,1,0,0,1,0,1,0,0,1,0,0],
             [0,0,0,0,0,0,1,0,0,1,0,1,0,0,1,0],
             [0,0,0,0,0,0,0,1,0,0,1,0,0,0,0,1],
             [0,0,0,0,0,0,0,0,1,0,0,0,0,1,0,0],
             [0,0,0,0,0,0,0,0,0,1,0,0,1,0,1,0],
             [0,0,0,0,0,0,0,0,0,0,1,0,0,1,0,1],
             [0,0,0,0,0,0,0,0,0,0,0,1,0,0,1,0]]

        cloud1 = Cloud(1, 'cloud1', simLog, simNetwork, adj = a)
        if peers: n = peers.pop(0)
        else: n = 16

        for i in xrange(16):
            simPeers.append(CloudPeer(i, cloud1, simLog, simStats, simParams,
                                      contributor = True))
        return simPeers


    '''
    Topology 8: A fully-connected grid with all nodes contributing
    '''
    if topology == 8:
        simPeers = []
        ######1,2,3,4,5,6,7,8,9,0,1,2,3,4,5,6,7,8,9,0
        a = [[0,1,0,0,0,1,0,0,0,0,0,0,0,0,0,0,0,0,0,0],
             [1,0,1,0,0,0,1,0,0,0,0,0,0,0,0,0,0,0,0,0],
             [0,1,0,1,0,0,0,1,0,0,0,0,0,0,0,0,0,0,0,0],
             [0,0,1,0,1,0,0,0,1,0,0,0,0,0,0,0,0,0,0,0],
             [0,0,0,1,0,0,0,0,0,1,0,0,0,0,0,0,0,0,0,0],
             [1,0,0,0,0,0,1,0,0,0,1,0,0,0,0,0,0,0,0,0],
             [0,1,0,0,0,1,0,1,0,0,0,1,0,0,0,0,0,0,0,0],
             [0,0,1,0,0,0,1,0,1,0,0,0,1,0,0,0,0,0,0,0],
             [0,0,0,1,0,0,0,1,0,1,0,0,0,1,0,0,0,0,0,0],
             [0,0,0,0,1,0,0,0,1,0,0,0,0,0,1,0,0,0,0,0],
             [0,0,0,0,0,1,0,0,0,0,1,0,0,0,0,1,0,0,0,0],
             [0,0,0,0,0,0,1,0,0,0,0,1,0,1,0,0,0,1,0,0],
             [0,0,0,0,0,0,0,1,0,0,0,0,1,0,1,0,0,0,1,0],
             [0,0,0,0,0,0,0,0,1,0,0,0,0,1,0,1,0,0,0,1],
             [0,0,0,0,0,0,0,0,0,1,0,0,0,0,1,0,0,0,0,0],
             [0,0,0,0,0,0,0,0,0,0,1,0,0,0,1,0,1,0,0,0],
             [0,0,0,0,0,0,0,0,0,0,0,1,0,0,0,1,0,1,0,0],
             [0,0,0,0,0,0,0,0,0,0,0,0,1,0,0,0,1,0,1,0],
```

```
                [0,0,0,0,0,0,0,0,0,0,0,0,0,1,0,0,0,1,0,1],
                [0,0,0,0,0,0,0,0,0,0,0,0,0,0,1,0,0,0,1,0]]

            cloud1 = Cloud(1, 'cloud1', simLog, simNetwork, adj = a)
            if peers: n = peers.pop(0)
            else: n = 20

            for i in xrange(20):
                simPeers.append(CloudPeer(i, cloud1, simLog, simStats, simParams,
                                          contributor = True))
            return simPeers

        '''
        Topology 13: four clique clouds with 6 nodes (one exit)
        '''
        if topology == 13:
            simPeers = []
            #create cloud
            cloud1 = Cloud(1, 'cloud1', simLog, simNetwork)
            cloud2 = Cloud(2, 'cloud2', simLog, simNetwork)
            cloud3 = Cloud(3, 'cloud3', simLog, simNetwork)
            cloud4 = Cloud(4, 'cloud4', simLog, simNetwork)
            # Add cooperative peers to the network
            ##########cloud 1################################
            for i in xrange(5):
                simPeers.append(CloudPeer(i, cloud1, simLog, simStats, simParams))
            simPeers[2].contributor = True
            simPeers.append(CloudExitPeer(5, cloud1, simNetwork, simLog, simStats,
                                          simParams))

            ##########cloud 2################################
            for i in xrange(6,11):
                simPeers.append(CloudPeer(i, cloud2, simLog, simStats, simParams))

            simPeers[7].contributor = True
            simPeers.append(CloudExitPeer(11, cloud2, simNetwork, simLog, simStats,
                                          simParams))

            ##########cloud 3################################
            for i in xrange(12,17):
                simPeers.append(CloudPeer(i, cloud3, simLog, simStats, simParams))
            simPeers[13].contributor = True
            simPeers.append(CloudExitPeer(17, cloud3, simNetwork, simLog, simStats,
                                          simParams))


            ##########cloud 4################################
            for i in xrange(18,23):
                simPeers.append(CloudPeer(i, cloud4, simLog, simStats, simParams))
            simPeers[19].contributor = True
            simPeers.append(CloudExitPeer(23, cloud4, simNetwork, simLog, simStats,
                                          simParams))

            return simPeers

        '''
        Topology 14: 6 clique clouds with 6 nodes each (one exit)
        '''
        if topology == 14:
            simPeers = []
            #create cloud
            cloud1 = Cloud(1, 'cloud1', simLog, simNetwork)
            cloud2 = Cloud(2, 'cloud2', simLog, simNetwork)
            cloud3 = Cloud(3, 'cloud3', simLog, simNetwork)
            cloud4 = Cloud(4, 'cloud4', simLog, simNetwork)
            cloud5 = Cloud(5, 'cloud5', simLog, simNetwork)
            cloud6 = Cloud(6, 'cloud6', simLog, simNetwork)
            # Add cooperative peers to the network
            ##########cloud 1################################
            for i in xrange(5):
                simPeers.append(CloudPeer(i, cloud1, simLog, simStats, simParams))
            simPeers[2].contributor = True
            simPeers.append(CloudExitPeer(5, cloud1, simNetwork, simLog, simStats,
                                          simParams))

            ##########cloud 2################################
            for i in xrange(6,11):
                simPeers.append(CloudPeer(i, cloud2, simLog, simStats, simParams))
            simPeers[7].contributor = True
            simPeers.append(CloudExitPeer(11, cloud2, simNetwork, simLog, simStats,
                                          simParams))

            ##########cloud 3################################
            for i in xrange(12,17):
                simPeers.append(CloudPeer(i, cloud3, simLog, simStats, simParams))
            simPeers[13].contributor = True
            simPeers.append(CloudExitPeer(17, cloud3, simNetwork, simLog, simStats,
                                          simParams))


            ##########cloud 4################################
            for i in xrange(18,23):
```

```
            simPeers.append(CloudPeer(i, cloud4, simLog, simStats, simParams))
        simPeers[19].contributor = True
        simPeers.append(CloudExitPeer(23, cloud4, simNetwork, simLog, simStats,
                                      simParams))

        ##########cloud 5################################
        for i in xrange(24,29):
            simPeers.append(CloudPeer(i, cloud5, simLog, simStats, simParams))
        simPeers[25].contributor = True
        simPeers.append(CloudExitPeer(29, cloud5, simNetwork, simLog, simStats,
                                      simParams))

        ##########cloud 6################################
        for i in xrange(30,35):
            simPeers.append(CloudPeer(i, cloud6, simLog, simStats, simParams))
        simPeers[31].contributor = True
        simPeers.append(CloudExitPeer(35, cloud6, simNetwork, simLog, simStats,
                                      simParams))
        return simPeers

    '''
    Topology 15: 8 clique clouds with 6 nodes each (one exit)
    '''
    if topology == 15:
        simPeers = []
        #create cloud
        cloud1 = Cloud(1, 'cloud1', simLog, simNetwork)
        cloud2 = Cloud(2, 'cloud2', simLog, simNetwork)
        cloud3 = Cloud(3, 'cloud3', simLog, simNetwork)
        cloud4 = Cloud(4, 'cloud4', simLog, simNetwork)
        cloud5 = Cloud(5, 'cloud5', simLog, simNetwork)
        cloud6 = Cloud(6, 'cloud6', simLog, simNetwork)
        cloud7 = Cloud(7, 'cloud7', simLog, simNetwork)
        cloud8 = Cloud(8, 'cloud8', simLog, simNetwork)
        # Add cooperative peers to the network
        ##########cloud 1################################
        for i in xrange(5):
            simPeers.append(CloudPeer(i, cloud1, simLog, simStats, simParams))
        simPeers[2].contributor = True
        simPeers.append(CloudExitPeer(5, cloud1, simNetwork, simLog, simStats,
                                      simParams))

        ##########cloud 2################################
        for i in xrange(6,11):
            simPeers.append(CloudPeer(i, cloud2, simLog, simStats, simParams))
        simPeers[7].contributor = True
        simPeers.append(CloudExitPeer(11, cloud2, simNetwork, simLog, simStats,
                                      simParams))

        ##########cloud 3################################
        for i in xrange(12,17):
            simPeers.append(CloudPeer(i, cloud3, simLog, simStats, simParams))
        simPeers[13].contributor = True
        simPeers.append(CloudExitPeer(17, cloud3, simNetwork, simLog, simStats,
                                      simParams))


        ##########cloud 4################################
        for i in xrange(18,23):
            simPeers.append(CloudPeer(i, cloud4, simLog, simStats, simParams))
        simPeers[19].contributor = True
        simPeers.append(CloudExitPeer(23, cloud4, simNetwork, simLog, simStats,
                                      simParams))

        ##########cloud 5################################
        for i in xrange(24,29):
            simPeers.append(CloudPeer(i, cloud5, simLog, simStats, simParams))
        simPeers[25].contributor = True
        simPeers.append(CloudExitPeer(29, cloud5, simNetwork, simLog, simStats,
                                      simParams))

        ##########cloud 6################################
        for i in xrange(30,35):
            simPeers.append(CloudPeer(i, cloud6, simLog, simStats, simParams))

        simPeers.append(CloudExitPeer(35, cloud6, simNetwork, simLog, simStats,
                                      simParams))

        ##########cloud 7################################
        for i in xrange(36,41):
            simPeers.append(CloudPeer(i, cloud7, simLog, simStats, simParams))
        simPeers[37].contributor = True
        simPeers.append(CloudExitPeer(41, cloud7, simNetwork, simLog, simStats,
                                      simParams))

        ##########cloud 8################################
        for i in xrange(42,47):
            simPeers.append(CloudPeer(i, cloud8, simLog, simStats, simParams))
        simPeers[43].contributor = True
        simPeers.append(CloudExitPeer(47, cloud8, simNetwork, simLog, simStats,
                                      simParams))

```

```
            return simPeers

        '''
        Topology 16: 4 ring/grid clouds with 6 nodes (one exit)
        '''
        if topology == 16:
            simPeers = []
            ### ring adj matrix ###
            # a = [[0,1,0,0,0,1],
            #      [1,0,1,0,0,0],
            #      [0,1,0,1,0,0],
            #      [0,0,1,0,1,0],
            #      [0,0,0,1,0,1],
            #      [1,0,0,0,1,0]]

            ##fully connected grid ###
            a = [[0,1,0,0,0,1],
                 [1,0,1,0,1,0],
                 [0,1,0,0,0,1],
                 [1,0,0,0,1,0],
                 [0,1,0,1,0,1],
                 [0,0,1,0,1,0]]

            #create cloud
            cloud1 = Cloud(1, 'cloud1', simLog, simNetwork, adj = a)
            cloud2 = Cloud(2, 'cloud2', simLog, simNetwork, adj = a)
            cloud3 = Cloud(3, 'cloud3', simLog, simNetwork, adj = a)
            cloud4 = Cloud(4, 'cloud4', simLog, simNetwork, adj = a)

            # Add cooperative peers to the network
            ##########cloud 1################################
            for i in xrange(5):
                simPeers.append(CloudPeer(i, cloud1, simLog, simStats, simParams))
            simPeers[2].contributor = True
            simPeers.append(CloudExitPeer(5, cloud1, simNetwork, simLog, simStats,
                                          simParams))

            ##########cloud 2################################
            for i in xrange(6,11):
                simPeers.append(CloudPeer(i, cloud2, simLog, simStats, simParams))
            simPeers[7].contributor = True
            simPeers.append(CloudExitPeer(11, cloud2, simNetwork, simLog, simStats,
                                          simParams))

            ##########cloud 3################################
            for i in xrange(12,17):
                simPeers.append(CloudPeer(i, cloud3, simLog, simStats, simParams))
            simPeers[13].contributor = True
            simPeers.append(CloudExitPeer(17, cloud3, simNetwork, simLog, simStats,
                                          simParams))


            ##########cloud 4################################
            for i in xrange(18,23):
                simPeers.append(CloudPeer(i, cloud4, simLog, simStats, simParams))
            simPeers[19].contributor = True
            simPeers.append(CloudExitPeer(23, cloud4, simNetwork, simLog, simStats,
                                          simParams))

            return simPeers

        '''
        Topology 17: 6 ring/grid clouds with 6 nodes (one exit)
        '''
        if topology == 17:
            simPeers = []
            ### ring adj matrix ###
            # a = [[0,1,0,0,0,1],
            #      [1,0,1,0,0,0],
            #      [0,1,0,1,0,0],
            #      [0,0,1,0,1,0],
            #      [0,0,0,1,0,1],
            #      [1,0,0,0,1,0]]

            ### fully connected grid ###
            a = [[0,1,0,0,0,1],
                 [1,0,1,0,1,0],
                 [0,1,0,0,0,1],
                 [1,0,0,0,1,0],
                 [0,1,0,1,0,1],
                 [0,0,1,0,1,0]]

            #create cloud
            cloud1 = Cloud(1, 'cloud1', simLog, simNetwork, adj = a)
            cloud2 = Cloud(2, 'cloud2', simLog, simNetwork, adj = a)
            cloud3 = Cloud(3, 'cloud3', simLog, simNetwork, adj = a)
            cloud4 = Cloud(4, 'cloud4', simLog, simNetwork, adj = a)
            cloud5 = Cloud(5, 'cloud5', simLog, simNetwork, adj = a)
            cloud6 = Cloud(6, 'cloud6', simLog, simNetwork, adj = a)
            # Add cooperative peers to the network
            ##########cloud 1################################
            for i in xrange(5):
```

```
            simPeers.append(CloudPeer(i, cloud1, simLog, simStats, simParams))
        simPeers[2].contributor = True
        simPeers.append(CloudExitPeer(5, cloud1, simNetwork, simLog, simStats,
                                      simParams))

        ##########cloud 2################################
        for i in xrange(6,11):
            simPeers.append(CloudPeer(i, cloud2, simLog, simStats, simParams))
        simPeers[7].contributor = True
        simPeers.append(CloudExitPeer(11, cloud2, simNetwork, simLog, simStats,
                                      simParams))

        ##########cloud 3################################
        for i in xrange(12,17):
            simPeers.append(CloudPeer(i, cloud3, simLog, simStats, simParams))
        simPeers[13].contributor = True
        simPeers.append(CloudExitPeer(17, cloud3, simNetwork, simLog, simStats,
                                      simParams))


        ##########cloud 4################################
        for i in xrange(18,23):
            simPeers.append(CloudPeer(i, cloud4, simLog, simStats, simParams))
        simPeers[19].contributor = True
        simPeers.append(CloudExitPeer(23, cloud4, simNetwork, simLog, simStats,
                                      simParams))

        ##########cloud 5################################
        for i in xrange(24,29):
            simPeers.append(CloudPeer(i, cloud5, simLog, simStats, simParams))
        simPeers[25].contributor = True
        simPeers.append(CloudExitPeer(29, cloud5, simNetwork, simLog, simStats,
                                      simParams))

        ##########cloud 6################################
        for i in xrange(30,35):
            simPeers.append(CloudPeer(i, cloud6, simLog, simStats, simParams))
        simPeers[31].contributor = True
        simPeers.append(CloudExitPeer(35, cloud6, simNetwork, simLog, simStats,
                                      simParams))
        return simPeers

    '''
    Topology 18: 8 ring/grid clouds with 6 nodes (one exit)
    '''
    if topology == 18:
        simPeers = []
        ### ring adj matrix ###
        # a = [[0,1,0,0,0,1],
        #      [1,0,1,0,0,0],
        #      [0,1,0,1,0,0],
        #      [0,0,1,0,1,0],
        #      [0,0,0,1,0,1],
        #      [1,0,0,0,1,0]]

        ### fully connected grid###
        a = [[0,1,0,0,0,1],
             [1,0,1,0,1,0],
             [0,1,0,0,0,1],
             [1,0,0,0,1,0],
             [0,1,0,1,0,1],
             [0,0,1,0,1,0]]

        #create cloud
        cloud1 = Cloud(1, 'cloud1', simLog, simNetwork, adj = a)
        cloud2 = Cloud(2, 'cloud2', simLog, simNetwork, adj = a)
        cloud3 = Cloud(3, 'cloud3', simLog, simNetwork, adj = a)
        cloud4 = Cloud(4, 'cloud4', simLog, simNetwork, adj = a)
        cloud5 = Cloud(5, 'cloud5', simLog, simNetwork, adj = a)
        cloud6 = Cloud(6, 'cloud6', simLog, simNetwork, adj = a)
        cloud7 = Cloud(7, 'cloud7', simLog, simNetwork, adj = a)
        cloud8 = Cloud(8, 'cloud8', simLog, simNetwork, adj = a)

        # Add cooperative peers to the network
        ##########cloud 1################################
        for i in xrange(5):
            simPeers.append(CloudPeer(i, cloud1, simLog, simStats, simParams))
        simPeers[2].contributor = True
        simPeers.append(CloudExitPeer(5, cloud1, simNetwork, simLog, simStats,
                                      simParams))

        ##########cloud 2################################
        for i in xrange(6,11):
            simPeers.append(CloudPeer(i, cloud2, simLog, simStats, simParams))
        simPeers[7].contributor = True
        simPeers.append(CloudExitPeer(11, cloud2, simNetwork, simLog, simStats,
                                      simParams))

        ##########cloud 3################################
        for i in xrange(12,17):
            simPeers.append(CloudPeer(i, cloud3, simLog, simStats, simParams))
        simPeers[13].contributor = True
```

```
1464            simPeers.append(CloudExitPeer(17, cloud3, simNetwork, simLog, simStats,
1465                                          simParams))
1466
1467
1468            ##########cloud 4################################
1469            for i in xrange(18,23):
1470                simPeers.append(CloudPeer(i, cloud4, simLog, simStats, simParams))
1471            simPeers[19].contributor = True
1472            simPeers.append(CloudExitPeer(23, cloud4, simNetwork, simLog, simStats,
1473                                          simParams))
1474
1475            ##########cloud 5################################
1476            for i in xrange(24,29):
1477                simPeers.append(CloudPeer(i, cloud5, simLog, simStats, simParams))
1478            simPeers[25].contributor = True
1479            simPeers.append(CloudExitPeer(29, cloud5, simNetwork, simLog, simStats,
1480                                          simParams))
1481
1482            ##########cloud 6################################
1483            for i in xrange(30,35):
1484                simPeers.append(CloudPeer(i, cloud6, simLog, simStats, simParams))
1485            simPeers[31].contributor = True
1486            simPeers.append(CloudExitPeer(35, cloud6, simNetwork, simLog, simStats,
1487                                          simParams))
1488
1489            ##########cloud 7################################
1490            for i in xrange(36,41):
1491                simPeers.append(CloudPeer(i, cloud7, simLog, simStats, simParams))
1492            simPeers[37].contributor = True
1493            simPeers.append(CloudExitPeer(41, cloud7, simNetwork, simLog, simStats,
1494                                          simParams))
1495
1496            ##########cloud 8################################
1497            for i in xrange(42,47):
1498                simPeers.append(CloudPeer(i, cloud8, simLog, simStats, simParams))
1499            simPeers[43].contributor = True
1500            simPeers.append(CloudExitPeer(47, cloud8, simNetwork, simLog, simStats,
1501                                          simParams))
1502            return simPeers
1503
1504        i = 0
1505        #run each topology
1506        print simTopologies
1507        for t in simTopologies:
1508
1509            ##make new directory, adding one to suffix if already exists
1510            while (os.path.exists("data%i"%i) or os.path.exists("logs%i"%i)):
1511                i+=1
1512            dataDir = "data" + str(i)
1513            logDir= "logs" + str(i)
1514            os.mkdir("data%i"%(i))
1515            os.mkdir("logs%i"%(i))
1516
1517            #run each set of parameters
1518            for si, simParams in enumerate(SimParamsList):
1519                random.seed(simParams['SEED'])
1520                # Simulation stop event set by simStats
1521                simEventStop = threading.Event()
1522                # Simulation objects
1523                simStats = Stats(simParams, simEventStop)
1524                simLog = Log(simParams)
1525                simNetwork = Network(simLog, simStats)
1526
1527                simPeers = generatePeers(t, simLog, simNetwork, simStats, simParams)
1528                simParams['SIM_NUM_PEERS'] = len(simPeers)
1529                simParams['TOPOLOGY'] = "topo-"+str(t)
1530
1531                print("\nStarting simulation %d / %d: %s" % (si+1, len(SimParamsList), simParams['NAME']))
1532
1533                roundCount = 0 #reset round count for each topo
1534                #the actual simulation loop
1535                while True:
1536                    # Simulate the peers in a different order each round
1537                    random.shuffle(simPeers)
1538                    for n in simPeers:
1539                        n.simulate(roundCount)
1540
1541                    sys.stdout.write("\r%d, %d -- Round %d" % \
1542                                     (len(simStats.message_inserts), len(simStats.message_decodes), roundCount+1))
1543                    # Stop the simulation if we've collected enough data
1544                    if simEventStop.is_set():
1545                        break
1546
1547                    roundCount += 1
1548                # Log the finish at this round count
1549                simStats.finished(roundCount)
1550                simLog.log(roundCount, "finish", 0, "")
1551                print()
1552                # Dump stats
1553                print("Wrote stats to %s" % simStats.dump())
1554                # Dump log
1555                print("Wrote log to %s" % simLog.dump())
```

```
        print "------------------------End Topo %i--------------------"%t
    print "All simulations finished"
```

B.3 ncgab_stats_process.py

```
#!/usr/bin/python3

# Plotter for NCGAB Simulator Results
# Colleen Josephson/Ivan Sergeev
# 2013/2014

import sys
import json
import numpy
import pylab
from operator import itemgetter, attrgetter
##############################################################################

def avg(x):
    return (sum(x) / float(len(x))) if len(x) > 0 else 0.0

def ci_95(x):
    std = numpy.std(x)
    ci = 1.96*std/numpy.sqrt(len(x))
    return ci

class StatsProcess():
    def __init__(self, filename):
        f = open(filename)
        data = json.loads(f.read())
        print("Loaded %s %s" % (data['simParams']['NAME'], data['simParams']['DESC']))
        self.simParams = data['simParams']
        self.message_inserts = data['message_inserts']
        self.message_decodes = data['message_decodes']
        self.message_life = data['message_life']
        self.matrix_reduces = data['matrix_reduces']
        self.window_sizes = data['window_sizes']
        self.PMFs = data['PMFs']
        f.close()

    #########################

    def dump_decodes(self):
        for i in range(len(self.message_inserts)):
            # Print time, node id, and message id of insert
            (itime, nid, pid) = self.message_inserts[i]
            print("[%.6f]: nid %d, insert %s" % (itime, nid, pid))
            # If this message was decoded by others
            if pid in self.message_decodes and len(self.message_decodes[pid]) > 0:
                # Count number of decodes
                ndecodes = len(self.message_decodes[pid])
                # Print time and node id of the decode
                for (dtime, nid) in self.message_decodes[pid]:
                    print("\t[%.6f]: nid %d, decode" % (dtime, nid))
                print("\n\tnumber of decodes: %d / %d = %.2f %%\n" % (ndecodes, \
                     self.simParams['SIM_NUM_PEERS'] - 1 - self.simParams['SIM_NUM_EVIL_PEERS'], \
                     100.0 * float(ndecodes)/(self.simParams['SIM_NUM_PEERS']-1-self.simParams['SIM_NUM_EVIL_PEERS'])))

    def dump_reduces(self):
        for (itime, nid, m, n, ns) in self.matrix_reduces:
            print("[%.6f]: nid %d, reduce %dx%d to %d" % (itime, nid, m, n, ns))

    def dump_window_sizes(self):
        for (t, nid, ss, gs) in self.window_sizes:
            print("[%.6f]: nid %d, solved size %d, gossip size %d" % (t, nid, ss, gs))

    #########################

    def compute_delays(self):
        delay_min = []
        delay_max = []
        delay_avg = []

        # For each inserted message
        for (itime, _, pid) in self.message_inserts:
            # If it was decoded
            if pid in self.message_decodes and len(self.message_decodes[pid]) > 0:
                # Collect the delays to decoding
                delays = [ (dtime - itime) for (dtime, _) in self.message_decodes[pid] ]
                # Add it to our min, max, avg tables
                delay_min.append( delays[0] )
```

```
77                delay_max.append( delays[-1] )
78                delay_avg.append( sum(delays) / float(len(delays)) )
79
80            return (delay_min, delay_max, delay_avg)
81
82        def compute_pdecodes(self):
83            times = []
84            pids = []
85            pdecode = []
86
87            # For each inserted message
88            for (time, _, pid) in self.message_inserts:
89                times.append(time)
90                pids.append(pid)
91                # If the message was decoded
92                if pid in self.message_decodes and len(self.message_decodes[pid]) > 0:
93                    if len(self.message_decodes[pid]) > self.simParams['SIM_NUM_PEERS']:
94                        print("num decodes = %s, peers = %i"%(len(self.message_decodes[pid]),
95                                                              self.simParams['SIM_NUM_PEERS']))
96                    # Calculate the percent of other nodes that decoded it
97                    pdecode.append( 100.0 * \
98                        (float((len(self.message_decodes[pid]))) / \
99                        float(self.simParams['SIM_NUM_PEERS']-self.simParams['SIM_NUM_EVIL_PEERS'])))
100
101               else:
102                   pdecode.append(0.0)
103
104           return (times, pids, pdecode)
105
106       def compute_avg_delay(self):
107           (_, _, delay_avg) = self.compute_delays()
108           return (numpy.mean(delay_avg), ci_95(delay_avg))
109
110       def compute_avg_pdecode(self):
111           (_, _, pdecode) = self.compute_pdecodes()
112           return (numpy.mean(pdecode), ci_95(pdecode))
113
114       #########################
115
116       def print_stats(self):
117           print(self.simParams['NAME'])
118
119           print("\tNum inserts: %d" % len(self.message_inserts))
120
121           (_, _, pdecode) = self.compute_pdecodes()
122           avg_pdecode = avg(pdecode)
123           print("\tDecode Percent: %.2f\t" % avg_pdecode)
124
125           (delay_min, delay_max, delay_avg) = self.compute_delays()
126           avg_delay_min = avg(delay_min)
127           avg_delay_max = avg(delay_max)
128           avg_delay_avg = avg(delay_avg)
129           print("\tAvg Delay: %.6f" % avg_delay_avg)
130           print("\tMin Delay: %.6f" % avg_delay_min)
131           print("\tMax Delay: %.6f" % avg_delay_max)
132
133           print ()
134
135       #########################
136
137       def plot_pdecodes_vs_time(self):
138           (times, pids, pdecode) = self.compute_pdecodes()
139           avg_pdecode = avg(pdecode)
140
141           pylab.figure()
142           pylab.plot(times, pdecode, 's')
143           pylab.vlines(times, 0, pdecode)
144           pylab.hlines(avg_pdecode, times[0], times[-1])
145           pylab.title('Final Decoding of Inserted messages')
146           pylab.xlabel('Time')
147           pylab.ylabel('Percent Peers Decode')
148           pylab.ylim([0, 101])
149
150
151   ##############################################################################
152
153   class StatsPlot():
154       def __init__(self, plotFilePrefix, statsProcList):
155           self.statsProcList = statsProcList
156           self.plotFilePrefix = plotFilePrefix
157
158       def plot_pdecode_vs_throughput(self):
159           data = {}
160           for sim in self.statsProcList:
161               params = sim.simParams
162               numPeers = params['SIM_NUM_PEERS']
163               contributeInterval = params['CONTRIBUTE_INTERVAL']
164               if numPeers not in data:
165                   data[numPeers] = {}
166               if contributeInterval not in data[numPeers]:
167                   data[numPeers][contributeInterval] = []
168               data[numPeers][contributeInterval].append(sim.compute_avg_pdecode()[0])
```

```
        pylab.figure()
        for numPeers in sorted(data.keys()):
            x = sorted(data[numPeers].keys())
            y = [ numpy.mean(data[numPeers][r]) for r in x]
            ci = [ ci_95(data[numPeers][r]) for r in x]
            pylab.errorbar(x, y, yerr=ci, fmt="s-", label="N=%d" % numPeers)
        pylab.xlabel('Contribution Interval')
        pylab.xlim([5, 25])
        pylab.ylabel('Average Percent Decode')
        pylab.ylim([0, 105])
        pylab.title('Availability vs. Contribution Interval')
        pylab.legend(loc='lower right')
        pylab.savefig(self.plotFilePrefix + "-availability.eps")

    def plot_delay_vs_throughput(self):
        data = {}
        for sim in self.statsProcList:
            params = sim.simParams
            numPeers = params['SIM_NUM_PEERS']
            contributeInterval = params['CONTRIBUTE_INTERVAL']
            if numPeers not in data:
                data[numPeers] = {}
            if contributeInterval not in data[numPeers]:
                data[numPeers][contributeInterval] = []
            data[numPeers][contributeInterval].append(sim.compute_avg_delay()[0])

        pylab.figure()
        for numPeers in sorted(data.keys()):
            x = sorted(data[numPeers].keys())
            y = [ numpy.mean(data[numPeers][r]) for r in x]
            ci = [ ci_95(data[numPeers][r]) for r in x]
            pylab.errorbar(x, y, yerr=ci, fmt="s-", label="N=%d" % numPeers)
        pylab.xlabel('Contribution Interval')
        pylab.xlim([5, 25])
        pylab.ylabel('Average Delay')
        pylab.ylim([0, 25])
        pylab.title('Delay vs. Contribution Interval')
        pylab.legend()
        pylab.savefig(self.plotFilePrefix + "-delay.eps")

    def plot_pdecode_vs_evil_nodes(self):

        data = {}
        for sim in self.statsProcList:
            params = sim.simParams
            numPeers = params['SIM_NUM_PEERS']
            percentEvil = 100.0 * params['SIM_NUM_EVIL_PEERS'] / float(params['SIM_NUM_PEERS'])
            if numPeers not in data:
                data[numPeers] = {}
            if percentEvil not in data[numPeers]:
                data[numPeers][percentEvil] = []
            data[numPeers][percentEvil].append(sim.compute_avg_pdecode()[0])

        pylab.figure()
        for numPeers in sorted(data.keys()):
            x = sorted(data[numPeers].keys())
            y = [ numpy.mean(data[numPeers][r]) for r in x]
            ci = [ ci_95(data[numPeers][r]) for r in x]
            pylab.errorbar(x, y, yerr=ci, fmt="s-", label="N=%d" % numPeers)
        pylab.xlabel('Percent Attackers')
        pylab.xlim([-1, 31])
        pylab.ylabel('Average Percent Decoded')
        pylab.ylim([0, 105])
        pylab.title('Availability vs. Percent Attackers')
        pylab.legend(loc='lower right')
        pylab.savefig(self.plotFilePrefix + "-availability.eps")

    def plot_delay_vs_evil_nodes(self):
        data = {}
        for sim in self.statsProcList:
            params = sim.simParams
            numPeers = params['SIM_NUM_PEERS']
            percentEvil = 100.0 * params['SIM_NUM_EVIL_PEERS'] / float(params['SIM_NUM_PEERS'])
            if numPeers not in data:
                data[numPeers] = {}
            if percentEvil not in data[numPeers]:
                data[numPeers][percentEvil] = []
            data[numPeers][percentEvil].append(sim.compute_avg_delay()[0])

        pylab.figure()
        for numPeers in sorted(data.keys()):
            x = sorted(data[numPeers].keys())
            y = [ numpy.mean(data[numPeers][r]) for r in x]
            ci = [ ci_95(data[numPeers][r]) for r in x]
            pylab.errorbar(x, y, yerr=ci, fmt="s-", label="N=%d" % numPeers)
        pylab.xlabel('Percent Attackers')
        pylab.xlim([-1, 31])
        pylab.ylabel('Average Delay')
        pylab.ylim([0, 25])
        pylab.title('Delay vs. Percent Attackers')
        pylab.legend()
```

```
261         pylab.savefig(self.plotFilePrefix + "-delay.eps")
262
263     def plot_window_sizes_vs_time(self):
264         peers = {}
265
266         for sim in self.statsProcList:
267             params = sim.simParams
268             numPeers = params['SIM_NUM_PEERS']
269             contributeInterval = params['CONTRIBUTE_INTERVAL']
270             time = []
271             solved_size = []
272             gossip_size = []
273
274             i = 0
275             while i*numPeers < len(sim.window_sizes):
276                 t = sim.window_sizes[i*numPeers][0]
277                 if t not in time:
278                     time.append(t)
279                 blockSS = [ss for (t, nid, ss, gs) in sim.window_sizes[i*numPeers:(i+1)*numPeers]]
280                 blockGS = [gs for (t, nid, ss, gs) in sim.window_sizes[i*numPeers:(i+1)*numPeers]]
281                 solved_size.append(avg(blockSS))
282                 gossip_size.append(avg(blockGS))
283                 i += 1
284
285             if numPeers not in peers:
286                 peers[numPeers] = {}
287             peers[numPeers][contributeInterval] = (time, solved_size, gossip_size)
288
289         for numPeers in sorted(peers.keys()):
290             pylab.figure()
291             for ci in peers[numPeers].keys():
292                 pylab.plot(peers[numPeers][ci][0], peers[numPeers][ci][1],
293                            label = ci)
294                 pylab.title('Solved Window Size for %i peers' % numPeers)
295             pylab.xlabel('Time')
296             pylab.ylabel('Items')
297             pylab.legend(loc='lower right', title="contrib. interv.")
298             pylab.savefig("SolvedWindowSize%ipeers.eps" % numPeers)
299
300             pylab.figure() #plot Gossip Window
301             for ci in peers[numPeers].keys():
302                 pylab.plot(peers[numPeers][ci][0], peers[numPeers][ci][2],
303                            label = ci)
304             pylab.title('Gossip Window Size for %i peers' % numPeers)
305             pylab.xlabel('Time')
306             pylab.ylabel('Items')
307             pylab.legend(loc='lower right', title="contrib. interv.")
308             pylab.savefig("GossipWindowSize%ipeers.eps" % numPeers)
309
310     def print_message_lifetime(self):
311         peers = {}
312         for sim in self.statsProcList:
313             params = sim.simParams
314             numPeers = params['SIM_NUM_PEERS']
315             contributeInterval = params['CONTRIBUTE_INTERVAL']
316
317
318             if numPeers not in peers:
319                 peers[numPeers] = {}
320
321             peers[numPeers][contributeInterval] = (avg(sim.message_life.values()),
322                                                    ci_95(sim.message_life.values()))
323
324         ngroups = len(peers[peers.keys()[0]])
325         ticks = []
326         pylab.figure()
327         color_index = 0
328         off = 0
329         for numPeers in sorted(peers.keys()):
330             colors = "bgrcmykw"
331             index = numpy.arange(ngroups)
332             bar_width = 0.25
333             data = []
334             err = []
335
336             ticks = sorted(peers[numPeers].keys())
337             for ci in sorted(peers[numPeers].keys()):
338                 data.append(peers[numPeers][ci][0])
339                 err.append(peers[numPeers][ci][1])
340
341             #pylab.errorbar(x, y, yerr=ci, fmt="s-", label="N=%d" % numPeers)
342             pylab.bar(index+off, data, bar_width,
343                       color=colors[color_index],
344                       label=str(numPeers), yerr = err,
345                       ecolor = 'k')
346             color_index += 1
347             off += bar_width
348
349         pylab.ylabel('Rounds')
350         pylab.xlabel('Contribution Intervals')
351         pylab.title('Average Message Lifetime')
352         pylab.xticks(index + bar_width, [str(t) for t in ticks])
```

```
353         pylab.legend(loc="upper left")
354         #pylab.ylabel('Avg. Life in Rounds')
355
356     def plot_PMFs(self):
357         peers = {}
358         for sim in self.statsProcList:
359             params = sim.simParams
360             numPeers = params['SIM_NUM_PEERS']
361             contributeInterval = params['CONTRIBUTE_INTERVAL']
362             if numPeers not in peers:
363                 peers[numPeers] = ([],[])
364             percentUni, maxes = peers[numPeers]
365
366             maxp = 0
367             numUniform = 0
368             for s in sim.PMFs.values():
369                 if s[0] == 1.0/float(numPeers):
370                     numUniform += 1
371                 else:
372                     for p in s:
373                         maxp = max(maxp, p)
374             maxes.append((maxp, float(contributeInterval)))
375             percentUni.append((float(numUniform)/float(len(sim.PMFs)),
376                                float(contributeInterval)))
377             peers[numPeers] = (percentUni, maxes)
378
379         #Plot percent uniform
380         pylab.figure()
381         for numPeers in sorted(peers.keys()):
382             percentUni, maxes = peers[numPeers]
383             labels = list(set([i[1] for i in sorted(percentUni,
384                                                     key=itemgetter(1))]))
385             percentUni = [i[0] for i in sorted(percentUni, key=itemgetter(1))]
386             x = []
387             y = []
388             ci = []
389             for l in xrange(len(labels)):
390                 x.append(l+1)
391                 y.append(avg(percentUni[l*3:l*3+3]))
392                 ci.append(ci_95(percentUni[l*3:l*3+3]))
393             pylab.plot(x, y)
394             pylab.errorbar(x, y, yerr=ci, fmt="s-", label="N=%d" % numPeers)
395
396         pylab.title('Fraction of packets that have maximal sender entropy')
397         pylab.xlabel('Contribute Interval')
398         pylab.ylabel('Fraction')
399         pylab.ylim([0,1])
400         pylab.xlim([0.5, 7.5])
401         pylab.xticks(xrange(1,8), labels)
402         pylab.legend(loc='upper right')
403
404 ##############################################################################
405
406 if __name__ == "__main__":
407     if len(sys.argv) < 3:
408         print("Usage: <plot output prefix> <simulation data files ...>")
409         sys.exit(1)
410
411     # From http://www.scipy.org/Cookbook/Matplotlib/LaTeX_Examples
412     fig_width_pt = 350#253.0
413     inches_per_pt = 1.0/72.27
414     golden_mean = (numpy.sqrt(5)-1.0)/2.0
415     fig_width = fig_width_pt*inches_per_pt
416     fig_height = fig_width*golden_mean
417     fig_size = [fig_width,fig_height]
418     params = {'backend': 'ps',
419               'axes.labelsize': 9,
420               'text.fontsize': 9,
421               'legend.fontsize': 9,
422               'xtick.labelsize': 7,
423               'ytick.labelsize': 7,
424               'text.usetex': True,
425               'figure.figsize': fig_size,
426               'figure.subplot.bottom': 0.15,
427               'font.size': 9,
428               'font.family': 'serif'}
429     pylab.rcParams.update(params)
430
431     plotFilePrefix = sys.argv[1]
432     simsToProcess = sys.argv[2:]
433     if len(simsToProcess) == 1:
434         statsProc = StatsProcess(simsToProcess[0])
435         statsProc.print_stats()
436         #statsProc.plot_pdecodes_vs_time()
437         statsProc.plot_window_sizes_vs_time()
438         pylab.show()
439
440     elif len(simsToProcess) > 1:
441         statsProcList = [ StatsProcess(p) for p in simsToProcess ]
442         p = StatsPlot(plotFilePrefix, statsProcList)
443         '''
444         Set which plots to generate below.
```

```
        '''
        #p.plot_pdecode_vs_throughput()
        #p.plot_delay_vs_throughput()
        #p.plot_window_sizes_vs_time()
        p.print_message_lifetime()
        #p.plot_PMFs()
        pylab.show()
```

B.4 ff.py

```
# Finite Field Arithmetic Implementation

class FiniteFieldArray():
    ### Predefined finite field characteristics
    FIELD_SIZE = 8
    PRIMITIVE_POLY = 0x1B
    GENERATOR = 0x03

    ##########################################################################

    ELEMENT_NUM = 0
    ELEMENT_MASK = 0
    ff_exp_table = []
    ff_log_table = []
    ff_sqrt_table = []

    ff_mul_table = {}
    ff_div_table = {}

    ##########################################################################

    ### Precomputation of finite field exponent, logarithm, and square root
    ### tables for fast multiplication, division, and square root

    @staticmethod
    def ff_precompute():
        # Setup constants associated with this field size
        FiniteFieldArray.ELEMENT_NUM = 2**FiniteFieldArray.FIELD_SIZE
        FiniteFieldArray.ELEMENT_MASK = FiniteFieldArray.ELEMENT_NUM-1
        MSB_MASK = (FiniteFieldArray.ELEMENT_MASK >> 1)+1

        # Slow finite field multiplication used to precompute exponent
        # and logarithm tables
        def ff_slow_mul(a, b):
            p = 0
            for i in range(FiniteFieldArray.FIELD_SIZE):
                if (b & 0x1):
                    p ^= a
                if (a & MSB_MASK):
                    a <<= 1
                    a &= FiniteFieldArray.ELEMENT_MASK
                    a ^= FiniteFieldArray.PRIMITIVE_POLY
                else:
                    a <<= 1
                    a &= FiniteFieldArray.ELEMENT_MASK
                b >>= 0x1
            return p

        # Precompute exponent and logarithm table
        FiniteFieldArray.ff_exp_table = [0] * FiniteFieldArray.ELEMENT_NUM
        FiniteFieldArray.ff_log_table = [0] * FiniteFieldArray.ELEMENT_NUM
        FiniteFieldArray.ff_exp_table[0] = 0x01
        for i in range(1, FiniteFieldArray.ELEMENT_NUM):
            FiniteFieldArray.ff_exp_table[i] = \
                ff_slow_mul(FiniteFieldArray.ff_exp_table[i-1], FiniteFieldArray.GENERATOR)
            FiniteFieldArray.ff_log_table[FiniteFieldArray.ff_exp_table[i]] = i

        # Slow finite field square root used to precompute square root table
        def ff_slow_sqrt(x):
            for i in range(FiniteFieldArray.ELEMENT_NUM):
                if FiniteFieldArray.ff_elem_mul_slow(i, i) == x:
                    return i
            raise ValueError("%d has no squareroot in (2**%d, 0x%02X, 0x%02X)!" \
                    % (x, FiniteFieldArray.PRIMITIVE_POLY, FiniteFieldArray.GENERATOR))

        # Precompute square root table
        FiniteFieldArray.ff_sqrt_table = [0] * FiniteFieldArray.ELEMENT_NUM
        for i in range(FiniteFieldArray.ELEMENT_NUM):
            try:
                FiniteFieldArray.ff_sqrt_table[i] = ff_slow_sqrt(i)
            except ValueError:
                FiniteFieldArray.ff_sqrt_table[i] = None
```

```
        # Precompute multiplication table
        for a in range(FiniteFieldArray.ELEMENT_NUM):
            FiniteFieldArray.ff_mul_table[a] = {}
            for b in range(FiniteFieldArray.ELEMENT_NUM):
                FiniteFieldArray.ff_mul_table[a][b] = FiniteFieldArray.ff_elem_mul_slow(a, b)

        # Precompute division table
        for a in range(FiniteFieldArray.ELEMENT_NUM):
            FiniteFieldArray.ff_div_table[a] = {}
            for b in range(FiniteFieldArray.ELEMENT_NUM-1):
                FiniteFieldArray.ff_div_table[a][b+1] = FiniteFieldArray.ff_elem_div_slow(a, b+1)

    ##########################################################################
    ##########################################################################

    ### Element-wise finite field addition, subtraction, multiplication,
    ### and division

    @staticmethod
    def ff_elem_add(a, b):
        return a ^ b

    @staticmethod
    def ff_elem_sub(a, b):
        return a ^ b

    @staticmethod
    def ff_elem_mul_slow(a, b):
        if a == 0 or b == 0: return 0
        # a * b = exp(log(a)) * exp(log(b)) = exp(log(a) + log(b))
        exp_index = FiniteFieldArray.ff_log_table[a] + FiniteFieldArray.ff_log_table[b]
        exp_index %= FiniteFieldArray.ELEMENT_MASK
        return FiniteFieldArray.ff_exp_table[exp_index]

    @staticmethod
    def ff_elem_mul(a, b):
        return FiniteFieldArray.ff_mul_table[a][b]

    @staticmethod
    def ff_elem_div_slow(a, b):
        if (b == 0): raise ZeroDivisionError
        # a / b = a * (1/b)
        b_inverse = FiniteFieldArray.ff_exp_table[FiniteFieldArray.ELEMENT_MASK - FiniteFieldArray.ff_log_table[b]]
        return FiniteFieldArray.ff_elem_mul(a, b_inverse)

    @staticmethod
    def ff_elem_div(a, b):
        if (b == 0): raise ZeroDivisionError
        return FiniteFieldArray.ff_div_table[a][b]

    @staticmethod
    def ff_elem_sqrt(a):
        return FiniteFieldArray.ff_sqrt_table[a]

    ##########################################################################

    @staticmethod
    def ff_bytearray_to_elemarray(x):
        if FiniteFieldArray.FIELD_SIZE != 8: raise \
            ValueError("ff_bytearray_to_elemarray unimplemented for FF 2**%d" % \
            FiniteFieldArray.FIELD_SIZE)
        return list(x)

    @staticmethod
    def ff_elemarray_to_bytearray(x):
        if FiniteFieldArray.FIELD_SIZE != 8: raise \
            ValueError("ff_elemarray_to_bytearray unimplemented for FF 2**%d" % \
            FiniteFieldArray.FIELD_SIZE)
        return bytearray(x)

    ##########################################################################
    ##########################################################################

    def __init__(self, x = None):
        self.raw_bytes = bytearray([])
        if x is not None:
            raw_bytes = bytearray(x)
            if len(raw_bytes) % (FiniteFieldArray.FIELD_SIZE/8) != 0:
                raise ValueError("invalid byte array size for FF 2**%d" % FiniteFieldArray.FIELD_SIZE)
            self.raw_bytes = raw_bytes

    ### Overloaded operators for scalar multiplication, scalar divison,
    ### array addition, array subtraction, equality, and string
    ### representation

    # FiniteFieldArray[i] = scalar
    def __getitem__(self, index):
        return self.raw_bytes[index]

    # len(FiniteFieldArray) = len(self.raw_bytes)
    def __len__(self):
```

```
        return len(self.raw_bytes)

    # FiniteFieldArray + FiniteFieldArray = FiniteFieldArray
    def __add__(self, other):
        if not isinstance(other, FiniteFieldArray): raise TypeError("other is not a FiniteFieldArray")
        elif len(self.raw_bytes) != len(other.raw_bytes): raise ValueError("other is not same size")

        my_elemarray = FiniteFieldArray.ff_bytearray_to_elemarray(self.raw_bytes)
        other_elemarray = FiniteFieldArray.ff_bytearray_to_elemarray(other.raw_bytes)
        sum_elemarray = [FiniteFieldArray.ff_elem_add(my_elemarray[i], \
                    other_elemarray[i]) for i in range(len(my_elemarray))]
        return FiniteFieldArray(FiniteFieldArray.ff_elemarray_to_bytearray(sum_elemarray))

    # FiniteFieldArray - FiniteFieldArray = FiniteFieldArray
    def __sub__(self, other):
        return self.__add__(other)

    # FiniteFieldArray * FiniteFieldArray = scalar (inner product)
    # FiniteFieldArray * scalar = FiniteFieldArray
    def __mul__(self, other):
        if isinstance(other, FiniteFieldArray):
            if len(self.raw_bytes) != len(other.raw_bytes):
                raise ValueError("other is not same size")

            my_elemarray = FiniteFieldArray.ff_bytearray_to_elemarray(self.raw_bytes)
            other_elemarray = FiniteFieldArray.ff_bytearray_to_elemarray(other.raw_bytes)
            x = 0
            for i in range(len(my_elemarray)):
                x = FiniteFieldArray.ff_elem_add(x, FiniteFieldArray.ff_elem_mul(my_elemarray[i], other_elemarray[i]))
            return x
        elif isinstance(other, int):
            if other < 0 or other > FiniteFieldArray.ELEMENT_NUM-1:
                raise ValueError("other is outside of finite field")

            my_elemarray = FiniteFieldArray.ff_bytearray_to_elemarray(self.raw_bytes)
            mul_elemarray = [FiniteFieldArray.ff_elem_mul(my_elemarray[i], other) for i in range(len(my_elemarray))]
            return FiniteFieldArray(FiniteFieldArray.ff_elemarray_to_bytearray(mul_elemarray))
        else:
            raise TypeError("unknown other")

    # FiniteFieldArray * FiniteFieldArray = scalar (inner product)
    # scalar * FiniteFieldArray = FiniteFieldArray
    def __rmul__(self, other):
        return self.__mul__(other)

    # FiniteFieldArray / scalar = FiniteFieldArray
    def __truediv__(self, other):
        if not isinstance(other, int): raise TypeError("other is not a scalar integer")
        elif other < 0 or other > FiniteFieldArray.ELEMENT_NUM-1: raise ValueError("other is outside of finite field")

        my_elemarray = FiniteFieldArray.ff_bytearray_to_elemarray(self.raw_bytes)
        div_elemarray = [FiniteFieldArray.ff_elem_div(my_elemarray[i], other) for i in range(len(my_elemarray))]
        return FiniteFieldArray(FiniteFieldArray.ff_elemarray_to_bytearray(div_elemarray))

    # FiniteFieldArray == FiniteFieldArray
    def __eq__(self, other):
        if not isinstance(other, FiniteFieldArray): return False
        return self.raw_bytes == other.raw_bytes

    # str(FiniteFieldArray)
    def __str__(self):
        hexstr = ""
        for i in range(len(self.raw_bytes)):
            hexstr += ("%02x" % self.raw_bytes[i]) + " "
            #if i == 0: continue
            #elif (i+1) % 64 == 0 and i+1 != len(self.raw_bytes): hexstr += "\n"
            #elif (i+1) % 2 == 0: hexstr += " "
        return hexstr

def ff_rref(m, b):
    # m is an array of FiniteFieldArray rows
    # b is an array of the right hand column

    pi = 0

    # Iterate through each row
    for j in range(len(m)):
        # While we do not have a pivot for this row
        while m[j][pi] == 0:
            # Find a row below to swap with for a pivot at pi
            for k in range(j+1, len(m)):
                if m[k][pi] != 0:
                    # Swap with this row
                    (m[j], m[k]) = (m[k], m[j])
                    (b[j], b[k]) = (b[k], b[j])
                    break

            # Increment pivot index if we could not find a row to swap with
            if m[j][pi] == 0:
                pi += 1

            # If there is no pivots left, we're done reducing
```

```
            if pi == len(m[0]):
                return (m, b)

        # Divide through to have a pivot of 1
        m[j] /= m[j][pi]

        # Eliminate above & below
        for k in range(len(m)):
            if k != j and m[k][pi] != 0:
                m[k] = m[k] - (m[j]*m[k][pi])

        # Move onto the next pivot
        pi += 1
        # If there is no pivots left, we're done reducing
        if pi == len(m[0]):
            break

    return (m, b)

def ff_solutions(a, x, b):
    (ap, bp) = ff_rref(a, b)

    solved = []
    used = []
    for i in range(len(ap)):
        r = ap[i]
        # If this row has only one non-zero entry
        reduced_coefs = [r[j] != 0 for j in range(len(r))]
        if sum(reduced_coefs) == 1:
            # Add the solution to our solved list
            solved.append(x[reduced_coefs.index(True)])
            # Add the decoded LC to our used list
            used.append(b[i])

    return (solved, used)

def ff_rref_test():
    m = [  FiniteFieldArray([33, 247, 109, 71, 139]), \
           FiniteFieldArray([97, 221, 102, 127, 72]), \
           FiniteFieldArray([101, 126, 186, 90, 103]), \
           FiniteFieldArray([59, 234, 145, 197, 122]), \
           FiniteFieldArray([75, 9, 213, 9, 3])]
    b = [1,2,3,4,5]
    (mp, bp) = ff_rref(m, b)
    for i in range(len(mp)):
        print(mp[i], "\t", b[i])
    print("")

    m = [  FiniteFieldArray([33, 247, 109, 71, 139]), \
           FiniteFieldArray([97, 221, 102, 127, 72]), \
           FiniteFieldArray([101, 126, 186, 90, 103]), \
           2*FiniteFieldArray([97, 221, 102, 127, 72])]
    b = [1,2,3,4]
    (mp, bp) = ff_rref(m, b)
    for i in range(len(mp)):
        print(mp[i], "\t", b[i])
    print("")

    m = [  FiniteFieldArray([0, 247, 109, 71, 139]), \
           FiniteFieldArray([97, 221, 102, 127, 72]), \
           FiniteFieldArray([101, 126, 186, 90, 103]), \
           2*FiniteFieldArray([97, 221, 102, 127, 72])]
    b = [1,2,3,4]
    (mp, bp) = ff_rref(m, b)
    for i in range(len(mp)):
        print(mp[i], "\t", b[i])
    print("")

    m = [  FiniteFieldArray([0, 0, 0, 71, 139]), \
           FiniteFieldArray([0, 0, 1, 127, 72])]
    b = [1,2]
    (mp, bp) = ff_rref(m, b)
    for i in range(len(mp)):
        print(mp[i], "\t", b[i])
    print("")
```

Bibliography

[1] Anonymizer. http://www.anonymizer.com/.

[2] I2p: Invisible internet project. http://www.i2p2.de/.

[3] Rudolf Ahlswede, Ning Cai, Shuo yen Robert Li, and Raymond W. Yeung. Network information flow. *IEEE TRANSACTIONS ON INFORMATION THEORY*, 46(4):1204–1216, 2000.

[4] Szymon Chachulski, Michael Jennings, Sachin Katti, and Dina Katabi. More: A network coding approach to opportunistic routing. 2006.

[5] Christopher Chang, Tracey Ho, and Michelle Effros. Peer-to-peer anonymous networking using coding. In *IEEE 50th Annual Allerton Conference on Communication, Control, and Computing*, 2012.

[6] D. Chaum. Untraceable electronic mail, return addresses, and digital pseudonyms. *Communications of the ACM*, 24:84–88, 1981.

[7] D. Chaum. The dining cryptographers problem: Unconditional sender and recipient untraceability. *Journal of Cryptology*, 1:65–75, 1988.

[8] Claudia Diaz and Bart Preneel. Taxonomy of mixes and dummy traffic. *Proceedings of I-NetSec04: 3rd Working Conference on Privacy and Anonymity in Networked and Distributed Systems*, 2004.

[9] R. Dingledine, N. Mathewson, and P. Syverson. Tor: The second generation onion router. In *Proceedings of the 13th Usenix Security Symposium*, 2004.

[10] M. J. Freedman, E. Sit, J. Cates, and R. Morris. Introducing tarzan: a peer-to-peer anonymizing network layer. 2002.

[11] S. Goel, M. Robson, M. Polte, and E. Sirer. Herbivore: A scalable and efficient protocol for anonymous communication. 2003.

[12] T. Ho, M. Medard, R. Koetter, D. Karger, M. Effros, J. Shi, and B Leong. A random linear network coding approach to multicast. *IEEE Transactions on Information Theory*, 52(10):4413–4430, 2006.

[13] JoAnne Holliday, Robert Steinke, Divyakant Agrawal, and Amr El Abbadi. Epidemic algorithms for replicated databases. *IEEE TRANSACTIONS ON KNOWLEDGE AND DATA ENGINEERING*, 15(3):2003, 2003.

[14] Sachin Katti, Dina Katabi, and Katay Puchala. Slicing the onion: Anonymous routing without pki. In *Fourth Workshop on Hot Topics in Networks ACM*, pages 665–674, 2005.

[15] Brian Neil Levine and Clay Shields. Hordes: A multicast based protocol for anonymity. *Journal of Computer Security*, 10:213–240, 2002.

[16] Desmond S. Lun, Muriel Medard, and Michelle Effros. On coding for reliable communication over packet networks. In *42nd Annual Allerton Conference on Communication, Control, and Computing*, 2004.

[17] Eytan Modiano, Devavrat Shah, and Gil Zussman. Maximizing throughput in wireless networks via gossiping. *ACM SIGMETRICS Performance Evaluation Review*, 34(1):27–38, 2006.

[18] Michael G. Reed, Paul F. Syverson, and David M. Goldschlag. Anonymous connections and onion routing. *IEEE Journal on Selected Areas in Communications*, 16:482–494, 1998.

[19] M. K. Reiter and A. D. Rubin. Crowds: Anonymity for web transactions. *ACM Transactions on Information and System Security*, 1:66–92, 1997.

[20] Ivan Sergeev. *Anonymous Broadcast with Network Coded Gossip*. Massachusetts Institute of Technology, Cambridge, MA, 2014.

[21] Rob Sherwood, Bobby Bhattacharjee, and Aravind Srinivasan. P5: A protocol for scalable anonymous communication. In *Proceedings of the 2002 IEEE Symposium on Security and Privacy*, May 2002.

[22] H. Sousa-Pinto, D. Lucani, and J. Barros. Hide and code: Session anonymity in wireless line networks with coded packets. In *Information Theory and Applications Workshop (ITA)*, pages 262–268, 2012.

[23] Paul Syverson. Practical vulnerabilities of the tor anonymity network. *Advances in Cyber Security: Technology, Operation, and Experiences*, page 60, 2013.

[24] Andrew S. Tanenbaum. *Computer Networks*, pages 452–455. The Art of Computer Programming. Prentice Hall, fifth edition, 2011.

[25] S. Verma and W. T. Ooi. Controlling gossip protocol infection pattern using adaptive fanout. In *Proceedings of IEEE International Conference on Distributed Computing Systems*, pages 665–674, 2005.

[26] J. P. Vilela, L. Lima, and J. Barros. Lightweight security for network coding. In *Proceedings of IEEE International Conference on Communications (ICC)*, pages 1750–1754, 2008.