Collaborative Approaches to Probabilistic Reasoning in Network Management

By

Benjamin J. Ewy

Submitted to the graduate degree program in the Department of Electrical Engineering & Computer Science and the Graduate Faculty of the University of Kansas in partial fulfillment of the requirements for the degree of Doctor of Philosophy

Chairperson Dr. Joseph B. Evans

Dr. Gary J. Minden

Dr. Victor S. Frost

Dr. Arvin Agah

Dr. Bozenna Pasik-Duncan

Date defended: May 19, 2014
The Dissertation Committee for Benjamin J. Ewy certifies that this is the approved version of the following dissertation:

Collaborative Approaches to Probabilistic Reasoning in Network Management

Chairperson Dr. Joseph B. Evans

Date approved: May 19, 2014
Abstract

Tactical networks, networks designed to facilitate command and control capabilities for militaries, have key attributes that differ from the Internet. Characterizing, modeling, and exploiting our understanding of these differences is the focus of this research.

The differences between tactical and commercial networks can be found primarily in the areas of access bandwidth, access diversity, access latency, core latency, subnet distribution, and network infrastructure. In this work we characterize and model these differences. These attributes affect research into issues such as overlays, service discovery, and server selection among others, as well as the deployment of services and systems in tactical networks. Researchers traditionally struggle with measuring, analyzing, or testing new ideas on tactical networks due to a lack of direct access, and thus this characterization is crucial to evolving this research field.

In this work we develop a topology generator that creates realistic tactical networks that can be visualized, analyzed, and emulated. Topological features including geographically constrained line of sight networks, high density low bandwidth satellite networks, and the latest high bandwidth on-the-move networks are captured. All of these topological features can be mixed to create realistic networks for many different tactical scenarios. A web based visualization tool is developed, as well as the ability to export topologies to the Mininet network virtualization environment.

Finally, state-of-the-art server selection algorithms are reviewed and found to perform poorly for tactical networks. We develop a collaborative algorithm tailored to the attributes of tactical networks, and utilize our generated networks to assess the algorithm, finding a reduction in utilized bandwidth and a significant reduction in client to server latency as key improvements.
Acknowledgements

I would like to thank my committee members, Dr. Minden, Dr. Frost, Dr. Agah, and Dr. Pasik-Duncan, and especially my advisor, Dr. Evans for all of their guidance and support. Joe, thank you for being supportive for so many years, and being such a good friend.

I would like to thank Mike Swink for assistance and advice, and Mark Redman, Mike Anglo, and Kyle Manship for assistance collecting data in a difficult environment.

Finally, I would like to thank Tom, Rita, Bob, Connie, Monica, and Jordan for making all of this possible and providing me with so much support and love.
# Contents

Abstract iii  

## 1 Introduction 1  
1.1 Motivation ........................................... 1  
1.2 Organization ....................................... 4  

## 2 Background 5  
2.1 Network Architecture Evolution .................... 5  
   2.1.1 Traditional Telecommunications Network ........ 6  
   2.1.2 Commercial Internet ............................ 6  
   2.1.3 Active Networks ................................. 7  
   2.1.4 PlanetLab ....................................... 8  
   2.1.5 GENI ........................................... 8  
   2.1.6 Software Defined Networks ..................... 9  
   2.1.7 Cognitive Networks ............................. 9  
2.2 Network Management ................................ 11  
   2.2.1 Overview ...................................... 11  
   2.2.2 Domain Based Network Management ............... 12  
      2.2.2.1 Background ................................ 12  
      2.2.2.2 Collecting Network Management Data ........ 13  
      2.2.2.3 Topology Discovery ....................... 14
4.3 Tactical TopGen - New Capabilities ........................................ 66
  4.3.1 JavaScript Visualization Export ..................................... 67
  4.3.2 Edge Property Management .......................................... 68
  4.3.3 Mininet emulation support ........................................... 69

4.4 Tactical TopGen New Routing Modules ................................. 71
  4.4.1 Tactical Core Network ............................................... 71
  4.4.2 WIN-T Region ...................................................... 75
  4.4.3 Satellite Region ..................................................... 80
  4.4.4 Tactical Topology Generator Conclusion .......................... 81

5 Server Selection ............................................................. 82
  5.1 TIGR Tactical Overlay Data Patterns ................................. 82
  5.2 Common Distributed Architectures .................................... 84
    5.2.1 Distributed Databases ........................................... 85
    5.2.2 Data Grids ...................................................... 85
    5.2.3 Content Distribution Networks .................................. 85
    5.2.4 Peer-to-Peer .................................................... 86
    5.2.5 TIGR Distribution Comparison .................................. 86
  5.3 State-of-the-Art Server Selection .................................... 87
    5.3.1 Content Distribution Network Server Selection ............... 88
    5.3.2 Peer-to-Peer Network Server Selection ........................ 89
  5.4 A Tactical Server Selection Algorithm ............................... 90
  5.5 Tactical Server Selection Evaluation ................................ 95
    5.5.1 Evaluation Methodology ......................................... 95
    5.5.2 Implementation Effectiveness .................................... 96
  5.6 Tactical Server Selection Conclusion ................................ 102
6 Contributions

6.1 Contributions & Publications .............................................. 103
6.2 Other Lessons Learned ....................................................... 104
6.3 Future Work ................................................................. 105
6.4 Conclusions ................................................................. 106

A Source Code for Tools and Utilities ........................................ 133

B Network Studies .............................................................. 167
List of Figures

1.1 Collaborative control architecture [5] ................................................. 2
1.2 Time sense of algorithms [12] ............................................................... 3
2.1 Long distance network [18] ................................................................. 6
2.2 Active networks [19] ......................................................................... 7
2.3 GENI concepts [35] ......................................................................... 8
2.4 Software defined networks [37] .......................................................... 9
2.5 Research concepts explored by CogNet [46] ...................................... 10
2.6 Global control plane architecture [46] ............................................... 11
2.7 Failure localization dependency graph [70] ...................................... 15
2.8 Network with aliases present ............................................................. 18
2.9 Network with merged aliases ............................................................ 19
2.10 Alias inference via graph analysis .................................................... 20
2.11 Unresponsive nodes in traceroute ..................................................... 20
2.12 Chord routing [14] .......................................................................... 24
2.13 Pastry overlay routing [14] ............................................................... 25
3.1 Representative tactical network architecture [155] .......................... 29
3.2 TIGR overlay topology in Iraq at its largest [17] ............................... 33
3.3 TIGR application level networking elements .................................... 34
3.4 Network access bandwidth comparison .......................................... 35
3.5 WIN-T Increment 2 soldiers network extension vehicle [164] .................. 36
3.6 WIN-T Increment 1 satellite truck [165] ............................................. 37
3.7 WIN-T Increment 2 tactical communications node [164] .................... 37
3.8 Layer 1/Layer 2 protocol delays ....................................................... 38
3.9 Serialization delays ................................................................. 39
3.10 Propagation delays ................................................................. 40
3.11 Histogram of all pairs ICMP RTT’s ................................................. 41
3.12 CDF of all pairs ICMP RTT’s ....................................................... 41
3.13 Internet data and model samples histogram ..................................... 44
3.14 Internet data and model samples CDF ............................................. 45
3.15 Kullback-Leibler divergence illustration ......................................... 48
3.16 Tactical data and model samples histogram .................................... 50
3.17 Tactical data and model samples CDF ............................................. 51
3.18 Blue force tracking network architecture [182] ................................ 51
3.19 WIN-T subnet design ............................................................... 52
3.20 Brigade subnet distribution ......................................................... 52
3.21 Typical Internet two way path ..................................................... 54
3.22 Typical tactical network two way path ......................................... 54
3.23 Iraq tactical network topology .................................................... 55
3.24 Afghanistan tactical network topology .......................................... 56
3.25 Partial iPlane PlanetLab network topology .................................... 56
3.26 Internet hop count ................................................................. 57
3.27 Hop count with best fit Gaussian ............................................... 57
3.28 Tactical hop count ............................................................... 57
3.29 Hop count with best fit Gaussian ............................................... 57

4.1 Waxman model graph [192] ......................................................... 60
4.2 Transit-Stub model graph [196] ..................................................... 61
5.13 Comparison of different "good enough" values and impact on system redirects . . . 100
5.14 Comparison of tactical algorithm versus P2P probing . . . . . . . . . . . . . . . . 101
# List of Tables

<table>
<thead>
<tr>
<th>Table</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>2.1</td>
<td>TMN FCAPS model</td>
<td>11</td>
</tr>
<tr>
<td>3.1</td>
<td>Internet mixture components</td>
<td>43</td>
</tr>
<tr>
<td>3.2</td>
<td>Ten component mixture assessment</td>
<td>48</td>
</tr>
<tr>
<td>3.3</td>
<td>Tactical mixture components</td>
<td>49</td>
</tr>
</tbody>
</table>
List of Listings

4.1 JavaScript Array Format .................................................. 67
4.2 Mininet Configuration File Example ................................. 70
4.3 Mininet Configuration File Example ................................. 73
4.4 WIN-T Mininet Configuration ............................................ 79
4.5 Satellite Region Mininet Configuration ......................... 81
A.1 JS Visualization ......................................................... 133
A.2 Server Selection .......................................................... 135
A.3 Iplane Diameter .......................................................... 144
A.4 Tactical Diameter ......................................................... 145
A.5 Tactical Delay Mixture Model ............................ 146
A.6 Kullbak-Liebler Example ............................................... 148
A.7 Topology To Server Selection ........................................ 150
A.8 Satellite Connector ......................................................... 152
A.9 WIN-T Connector ........................................................ 156
A.10 Tactical Core Connector ............................................... 162
Chapter 1

Introduction

1.1 Motivation

Tactical Networks - telecommunication systems designed to support situational awareness, collaborative planning, command, and control functions for military units of all sizes, and across all environments, have properties that vary sufficiently from the commercial Internet that many common network management approaches do not apply. Tactical networks are unreliable for many reasons; for example, power disruptions or difficult terrain can cause outages. Hence link outages and disruption are the norm rather than the exception. Additionally, tactical networks typically have less bandwidth than the commercial Internet, and more delay due to the inherent technical limitations of providing the communications service in a hostile and ever changing environment.

Network managers typically have little visibility into end users’ experience, and the feedback they receive is often limited to delayed complaints. The experience of the end user depends not just on the network providing their last hop connectivity, but heavily on the performance of the internetwork making up the links between the end user and the resources they are accessing. Attempts by the end user to determine what went wrong for a given connection are often met with a lack of reproducibility (due to the temporal or locality based nature of the problem). Monitoring the performance of the end-to-end system is challenging given the diverse nature of typical tacti-
cal networks, which span multiple generations of technologies, multiple access technologies, and extraordinary operating conditions.

Future generations of networks [1][2][3] have a need for more distributed and comprehensive network management functionality, as the number of devices and complexity of technology continues to expand, with a continuing proportional decrease in experienced network operators to manage these networks. A long term vision of a solution for this scenario was presented [4] to illustrate the issues and objectives for future research. Figure 1.1 represents a proposed architecture [5] that utilizes distributed sensing and control functionality, and a collaborative control mechanism to allow network management functionality to service the very diverse networks of the future.

The collaborative control mechanism is expected to leverage a variety of cognitive [6] approaches and machine learning [7][8] techniques to provide the needed functionality, and has inspired research [9][10] and calls [11] for continued research in these areas. Figure 1.2 from [12] shows the range of characteristics and the wide variety of algorithms that are being brought to bear
In addition to the development of the cognitive algorithms for performing the collaborative command and control features, the task of relaying measurements to the control logic and its coordination must be carefully considered. A distributed infrastructure must be developed and deployed which scales to the necessary network size, while minimizing the impact of the measurement coordination on the network itself. The data relayed for the measurements must be a small percentage of the traffic on the network itself, and allow end users to automatically share network performance information. Collaborative control algorithms can then process the information from across the collection of networks in aggregate. The development and deployment of structured peer-to-peer network overlays [13][14][15][16][17] has provided a unique technology for the distribution of network measurements in a scalable fashion.

By developing and deploying these technologies we enable the creation of network management capabilities that can automatically provide services for the end user. Actions taken automatically in response could include:

- The ability to select a the best server replica when a web site is unavailable.
- Select from different first hop networks (WiFi, Cellular, Satellite, etc.) when one becomes
unreliable or unavailable completely.

- Predict what should be happening, and focus additional resources on the situation when a deviation from this prediction occurs.

As a step towards the vision described above, we hypothesize that a collaborative system performing network management functions using probabilistic reasoning will take less aggregate network resources, and result in better accuracy, than if each network node acts independently. A distributed approach to network management taking advantage of a peer-to-peer communication infrastructure will be able to provide network management capabilities in a constrained tactical network. Probabilistic reasoning to handle the many forms of uncertainty and algorithms tailored to the unique networking environments seen in tactical networks can address these problems.

1.2 Organization

The rest of this document is organized as follows: In Chapter 2 we provide background information on network architecture, network management tasks, and network technologies pertinent to the tasks researched in this work. We discuss background information for collecting measurements, topology discovery, and failure localization for two styles of network management, domain based, and end-to-end based. Background on collaboration using peer-to-peer protocols, data base synchronization, and other distributed synchronization methods is detailed. Chapter 3 presents a characterization of tactical networks based on numerous network studies. Chapter 4 reviews synthetic topology generation approaches, and presents a synthetic topology generator for tactical networks matching the characteristics from Chapter 3. Chapter 5 reviews the state of the art in server selection techniques and the limitations of those approaches in tactical networks, and then presents a load balancing algorithm tailored for the constraints of the tactical network, and evaluates its effectiveness. Chapter 6 concludes with contributions. Appendix A provides source code for the software developed during this research. Appendix B details the data sources used in this study.
Chapter 2

Background

In this chapter we review the evolution of network architecture, from the traditional telecommunications network, to the packet forwarding Internet, and through a series of architectures exploring new approaches to support change at many levels of the network. We cover network management in general along with the differences between domain based versus end-to-end based management, and focus on the areas of measurement collection, topology discovery, and fault localization. Finally, we discuss collaboration approaches for distributed systems, with an emphasis on using this capability to perform distributed measurement collection and network management processing.

2.1 Network Architecture Evolution

The architecture of a network directly influences how you deploy and manage new services. There has been a general progression over time from rigid services controlled centrally to more flexible services allowing easier deployments with more end-user control. There is a natural tension between security and control on one side, and the flexibility to deploy new services by either the network provider or end-user on the other side. There is a cyclic nature to many of these approaches as ideas that were not feasible due to hardware or software constraints become feasible over time, allowing the actual implementation of ideas previously proposed with the newly available enabling technology. Often new ideas are developed in testbeds that are overlayed onto the
existing network for experimentation. As ideas are validated the capabilities get pushed down over time into the deployed commercial networks. As these capabilities become real, capabilities at higher levels of abstraction become possible, and the research focus moves to a new level.

### 2.1.1 Traditional Telecommunications Network

The traditional telecommunications network had clearly separated switching elements and control links as depicted in Figure 2.1. Call setup (the control plane) happened using dedicated signaling processors, and dedicated links between switches. A centralized database provided common access to key information for management such as phone numbers. The overall architecture made it difficult to deploy new services onto the network.

### 2.1.2 Commercial Internet

As the commercial Internet grew to the forefront of communications technology, the architecture became one where the control plane and data plane were merged onto the same network elements and links. Routing protocol messages flowed over the same links that the customer data traversed.
The architecture supports easier end-user deployment of new services, but has limitations in the processing capabilities of the routers themselves, as well as security and reliability issues due to user data and control data being on the same links.

2.1.3 Active Networks

One network research focus that began in the 1990’s was trying to solve problems with integrating new services into existing architectures, and this resulted in a concept called Active Networks [19]. The main idea is that nodes in the network can perform operations on the data in the packets instead of just manipulating the packet headers as in a traditional router network. Examples of the kind of processing that could be performed are gateway functionalities between a legacy network and a network based on a new architecture, firewall type functionality where data is sanitized as it flows through a node, or network management tasks such as injecting diagnostic applications into packets destined to neighbor routers to try to localize a fault. The concept as shown in Figure 2.2 from [19] supported two main approaches, a capsule approach [20] that had the packets themselves carry code to be used to operate on the packet, which gives the end users ultimate control over the system, and a programmable switch approach [21] that allows administrators to download programs via the management plane to the network nodes that can then operate on the transiting packets. The Active Networks research changed the way people thought about networks and showed the way to network virtualization and programmability.
2.1.4 PlanetLab

PlanetLab [22] is a distributed set of virtual machines that support experiments by allocating slices of the distributed network to researchers. It has been used for many experiments such as resource discovery [23], distributed hash tables [24], wide area storage [25], and numerous network measurement studies [26] [27] [28]. PlanetLab has also served as a testbed for a distributed management system [29]. PlanetLab led the way toward virtualized network testbeds used by a distributed community.

2.1.5 GENI

The Global Environment for Network Innovations [30] is an experimental infrastructure designed to enable research of radical ideas in networking, to enable the fundamental reassessment of network architecture to address problems [31] such as security [32], availability, the transition to data centric models of service, and intermittent as opposed to end-to-end connectivity. GENI extends the approaches of PlanetLab by having virtualizable and programmable resources [33] that can be partitioned among users as illustrated in Figure 2.3, but allows for experiments at scale, with long running operational experience [34] of networking at all levels of the network stack.
2.1.6 Software Defined Networks

Software Defined Networks (SDN) [36] are in many ways a realization of the Active Networks programmable switch concept. The main focus in SDN is the decoupling of the control plane from the data plane, in particular making it possible for the control plane to reside on physically different hardware. The main API for communicating between the control plane and data plane has been standardize as OpenFlow as shown in Figure 2.4 [37]. OpenFlow allows remote (either centralized or distributed) control over the forwarding tables in a network of switches, and is supported by a wide variety of network hardware vendors [38]. SDN has supported research into network management [39], scalability [40], performance [41], and software languages to support describing network tasks and policies [42][43]. The broad range of recent research has included novel multicast distribution schemes [44], and energy efficient routing in data centers [45]. Much of the work in the SDN area is now directed towards techniques for managing large distributed systems, as the base technology has proven itself in the data center and campus environments.

2.1.7 Cognitive Networks

The CogNet [46] architecture is a framework which grew out of cognitive radio research. Cognitive Radio research [47] [48] [49] developed algorithms for sensing spectrum conditions and modifying
radio parameters to optimize particular criteria. The CogNet effort expanded this approach to include a framework for the sharing of measured data at all levels of the networking stack and performing different types of algorithms on this cross layer data to handle tasks appropriate to each level as shown in Figure 2.5. An example might be a very fast algorithm to manipulate physical layer parameters in response to noise, while also having a slow running algorithm that is calculating the optimal routing topology for the wireless network given the interference seen at each node.

The CogNet architecture is designed around the following capabilities:

- Spectrum Agility
- Fast PHY adaption
- Spectrum Coordination [50]
- Dynamic MAC layer [51]
- Ad-hoc network formation through auto-configuration
- Service Discovery and other network layer protocols [52]
- A Global Control Plane to serve as a cross layer and cross node repository and interface to manage each of these capabilities [53]
One of the central premises behind CogNet was that the Global Control Plane is essentially a management overlay that allows management processes running on any node, and operating at any layer of the network stack, to have a common API to access collected measurements to use in its calculations as shown in Figure 2.6.

### 2.2 Network Management

#### 2.2.1 Overview

Network Management is a term that covers a number of distinct but related activities. The International Telecommunications Union defines the Telecommunications Management Network (TMN) [54] model to encourage standard ways for performing network management, and further defines the FCAPS model. FCAPS is an acronym which stands for the elements commonly associated with network management, Fault, Configuration, Accounting, Performance, and Security, as detailed in Table 2.1.

<table>
<thead>
<tr>
<th>Fault</th>
<th>Configuration</th>
<th>Accounting</th>
<th>Performance</th>
<th>Security</th>
</tr>
</thead>
<tbody>
<tr>
<td>alarm handling</td>
<td>system turn-up</td>
<td>track service usage</td>
<td>data collection</td>
<td>control access</td>
</tr>
<tr>
<td>trouble detection</td>
<td>network provisioning</td>
<td>bill for services</td>
<td>report generation</td>
<td>enable functions</td>
</tr>
<tr>
<td>trouble correction</td>
<td>auto-discovery</td>
<td>data analysis</td>
<td>access logs</td>
<td></td>
</tr>
<tr>
<td>test and acceptance</td>
<td>back up and restore</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>network recovery</td>
<td>database handling</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Table 2.1: TMN FCAPS model
performance management. Throughout this document we will use a number of terms defined as follows. A fault is the specific cause of a failure, and is often not directly observable by the network management system. A fault will cause a number of symptoms which are observable, and can be both negative – i.e. suggestive that a particular fault exists, or positive, i.e. counter evidence that a particular fault does not exist. Fault localization is the act of determining the specific fault given the observed symptoms. Once a fault has been localized, corrective action can be taken such as notifying responsible administrators, or using alternate providers of the resource. Fault management is the act of recognizing, localizing, and correcting faults. The topology of a network is defined by the configuration and links between nodes of the network. An accurate knowledge of the topology is essential for determining the dependencies required in fault localization. Performance data can be used for diagnosis as well as trend estimation to allow predictive fault diagnosis and forecast future requirements.

We next focus on providing background information on key tasks associated with network management including topology discovery, fault detection, performance measurement, and fault localization. There are two traditionally accepted methods for performing network management functions, a domain based approach, and an end-to-end based approach. There are a number of tradeoffs in how each approach performs tasks that affect the accuracy and deployability of a system based on these different techniques. See [55][56] for a list of network management tools. Through the course of this background review we will document and make a case for particular choices that are most applicable to a widely deployed, highly scalable, distributed system operating in a tactical network, like the one motivating our work.

2.2.2 Domain Based Network Management

2.2.2.1 Background

The domain based management approach is the historically predominant method for managing networks, especially legacy networks such as TDM networks, but is also widely used in enterprise and ISP type networks. Domain based networks are so named because all of the participating
network elements must belong to the same administrative domain as the management system. This is typically enforced via passwords, access control lists, and other methods to limit the visibility of the network management data to centralized network management systems. The network elements and the network management station form a hierarchy with the management station collecting data, and performing fault management. Researchers [57] [58] [59] have developed techniques for federated data from multiple administrative domains and managing the trust issues resulting from incomplete or untrustworthy data.

2.2.2.2 Collecting Network Management Data

In order to participate in domain based management network elements must be “manageable”, meaning that they have a control plane set of functionality typically implemented in software, and this software is remotely accessible. This requirement adds cost to network elements, and it is not unusual for smaller enterprise environments to have a mix of managed and unmanaged network elements due to cost constraints.

There are two main types of data collection in domain based networks, active polling, and event driven methods. Active polling is periodically done by the network management station to collect information for performance and billing purposes. This polling often occurs using the Simple Network Management Protocol [60] (SNMP) or web based interfaces. Information such as bytes in and out on interfaces can be used for trending and billing, and thresholds can be checked to verify parameters are operating within desired levels. Effort must be made to poll at proper intervals so that counters of pertinent information (typically 32 bit) do not roll over during the sample window, a situation more common with modern high speed network interfaces.

Event driven messages are often configured on network elements to send notification of a failure or unusual condition to the network management station. SNMP traps are the primary source of these types of messages, and can be configured to send notifications in the presence of interfaces becoming unplugged, losing synchronization with peers, exceeding environmental thresholds, and so forth.
2.2.3 Topology Discovery

Accurate knowledge of the topology for the network being managed is key to many network management tasks, especially fault management. Historically, careful record keeping was the only choice, but as networks became more complex, automated solutions were required. A number of commercial systems [61][62][63] provide support for deriving layer three topologies using the routing tables available on layer three devices via SNMP.

Layer two topology discovery has not had a standard method for representing topology information in devices, and solutions have often been vendor specific using extensions to SNMP Management Information Bases (MIBs). An IETF RFC was developed [64] to provide a consistent interface for this type of information, but it is not widely supported at this time. Breitbart et al [65] developed an algorithmic solution to merge together universally supported management information to get a complete topology across a heterogeneous mix of layer two and three devices. This algorithm and similar SNMP data based solutions are able to learn the topology only when the device supports SNMP (a managed device) and when it is in the administrative domain of the system doing the topology discovery.

2.2.4 Failure Localization

Failure localization is a widely studied area of network management, and many different approaches have been taken. Simple systems with well defined failures can use deterministic techniques, but the majority of research is on the use of stochastic approaches to handle ambiguity of alarms, inconsistent information, and other uncertain data. The range of approaches applied to failure localization covers much of computer science, with different techniques having advantages over others given certain conditions or assumptions.

Rule based systems [66][67], case based systems [68], and neural networks [69] have all been used for fault localization with varying degrees of success. These approaches often work in specific domains, but typically struggle to handle updates to the system knowledge as new equipment or dependencies are added, requiring retraining. Model based approaches that separate out the
network topology and operation in a modular fashion are more expandable. These approaches rely on an inference using rules that allow the system to express and test correlation, frequency, origin, and then take actions such as filtering of alarms or escalating the problem for explicit testing.

Another family of solutions involves modeling failures using graph theory. In these approaches a directed acyclic graph is used to model the system. Nodes represent failures or symptoms, and vertices between them represent causal relationships, typically with some conditional probability. In order to create these graphs the dependencies between the symptoms and equipment must be known ahead of time, and are typically derived from topology and heuristics. An example failure model from [70] is shown in Figure 2.7.

Codebook approaches have been used with success [71][72], and they model failure event propagation as a communications signal, with coding techniques (Hamming distance for deterministic models, log likelihood for probabilistic approaches) used to decode the location of the failure given the signal transmitted (the symptoms). Unpredictable behavior when there are overlapping failures and the need to recreate the entire codebook when the system changes are drawbacks to this approach.

Bayesian Belief Networks (BBNs) [73] are another approach using graphs, and they can be solved using either an exact message passing approach [74] or an approximation. Exact inference
using belief networks has been done with the fault model being automatically generated by logs of alarms using data mining techniques [75], and then failure localization using message passing to calculate the belief that a particular fault occurred given the observed symptoms. Typically, failure localization systems [76][77][78] only use negative symptoms, although some have been extended to include positive symptoms [79]. Localization in Bayesian belief networks has been shown to be NP-hard in general [80], but has been used successfully in specific topologies. The current state of the art in these systems [81] requires $O(n^5)$ effort where $n$ is the number of nodes in the graph, and has been shown to scale to hundreds of nodes in realistic systems using hierarchical approaches [82][83]. For larger systems it is often necessary to use a stochastic simulation based inference to approximate the overall inference.

### 2.2.3 End-to-End Based Network Management

#### 2.2.3.1 Background

The end-to-end based management approach is the primary method end users have for diagnosing problems, but is often used as a complement to domain based solutions, where a network monitoring station serves as one of the end points, and probes end services such as routers or web servers. The data which can be collected using these techniques is limited in scope compared to traditional domain approaches, due to the lack of authentication. Typically, this results in tests using specific management plane protocols such as the Internet Control Message Protocol (ICMP) [84], but tests of the normal data path of the network are common as well. Security concerns and implementations to address these concerns often further limit the availability of directly measured observations, providing incentives for inferences of indirectly observable variables using alternate measurements.
2.2.3.2 Collecting Network Management Data

The collection of end-to-end measurement data can be broken down into two main categories, direct observations, and indirect measurements. Direct observations [85] provide the most accurate information, and are typically things such as the round trip time (RTT) between two end systems utilizing ICMP or TCP handshakes. Other examples include application response times using application specific queries such as HTTP GETs [86], or identification of intermediate layer three nodes using hop count limited probes [87]. Responses to ICMP latency probes, or hop limited probes requires cooperation from the end system as well as intermediate systems for full data collection. Some nodes do not respond to these probes, most likely a decision relating to a perceived security benefit, and this practice is expected to increase. One-way measurements of path loss and latency can be performed as well, but require cooperating software deployed at the far station, and can also introduce non-trivial time synchronization requirements [88].

Measurements intended to provide information for an indirectly observed quantity are another category of measurement data. Measurements of delay covariance between hosts using multicast [89] or unicast [90][91] traffic are an example of indirectly measuring the amount of shared network links between hosts. Measurements of end-to-end path loss can be used to infer link losses [92]. Setting up specific packet patterns can be used to measure available bandwidth on a per link basis using techniques like variable packet size probing [93], packet pair dispersion [94][95], and self-loading periodic streams [96]. Most of these approaches have variations that require cooperation from the other end, or can be single ended and work from the noisier round trip type measurements, and there are even hybrids that use both active and passive measurements [97]. Passive measurements [98] are becoming more important for many areas because with the adoption of IPv6, it is impossible to actively probe every subnet, for example.

2.2.3.3 Topology Discovery

Topology discovery in end-to-end management systems follows the same two categories of measurement techniques. Direct measurement of intermediate nodes using hop count limited packets
(traceroute) is the most widely used [99]. The technique works by sending hop limited packets into a network and seeing which nodes return the error message, and in so doing identify themselves. Traceroutes [100] from a set of distributed servers [101] [102] [103] [104] can be stitched together, to provide more complete topologies than the trees resulting from a single node’s mapping efforts. There have been a number of approaches that use various algorithmic [105] and heuristic [106] approaches to limit the amount of probing while recovering as much data as possible. Shortcomings of hop limited probe techniques include the “alias” problem of collapsing independent nodes that are in reality multiple interfaces of a single router, and the limitation that only layer three devices can be identified. Figure 2.8 shows two traceroute paths from a tactical network, one in each direction between two servers, without the aliases being identified and merged. Figure 2.9 shows the same network with the aliases identified and merged.

There are a number of approaches to performing IP alias resolution, and they fall into the following categories:

- ICMP/IP ID [107] [108] [106] [109] [110] [111] [112]
- IPV6 IP fragmentation [113]
- DNS [114]
- Multicast/IGMP [115] [116]
The ICMP/IP ID approaches take advantage of implementation details in today’s routers which respond to probes of addresses on different interfaces with packets that have common information in them to identify which IP addresses are likely to be on the same router. A similar approach that works for IPv6-based networks takes advantage of information leakage in how IP fragmentation is performed. DNS-based approaches take advantage of heuristics and common naming schemes and perform reverse DNS queries for all IP addresses and try to find common patterns that suggest addresses are the same physical router. For example, the hostnames swc-lx-2n1.arsalon.net and swc-lx-1n1.arsalon.net represent the same physical router "swc-lx" and correspond to different interfaces 2n1 and 1n1. Different ISPs have their own naming conventions but rules have been developed for the major ones. Multicast approaches utilize IGMP messages instead of ICMP to elicit multicast topology information from multicast enabled routers. Graph analysis techniques involve approaches for traversing the graph from different directions and finding common stubs, which implies that the last hops before the common stub are most likely the same router. Figure 2.10 shows two transits of the graph each with a common stub of C-D. That implies that nodes B and E are most likely the same physical router. As mentioned, not all layer three devices will respond to probes, as shown in Figure 2.11, but their presence can typically be inferred from the TTL changes.

Topology discovery can also be performed using inference and indirect observations. This ap-
A bottom up agglomerative hierarchical clustering method [122] can be used to pick two nodes with high similarity, group them into a cluster, then repeat until there is a single cluster. When we compare clusters to nodes, we can choose to use complete link (longest distance merged), average link (average distance merged), or single link (shortest distance merged) as the basis for the similarity check. This is a greedy algorithm, and can make some non-globally optimum choices leading to inaccuracies.

Another approach [123] utilizes a Markov Chain Monte Carlo simulation to determine a most likely topology given the collection of similarity metrics, by searching the space of all possible trees and calculating the probability of a particular tree given the observations.
2.2.3.4 Failure Localization

Failure localization in end-to-end management approaches again follows the two categories of measurement systems. Directly measuring services with approaches like traceroute can often find intermediate nodes with symptoms of the problem, without identifying the problem specifically. It is typically difficult to disambiguate the cause of failure as it could be that the last hop router is down, the end node is down, or they could simply be blocking measurement probes. Another limitation is that the hop limited probes return information about the path from the client to the server, but the dominant direction of transfer is typically from the server back to the client. Asymmetric routes occur in 30% [124] of the paths, and losses in the backwards direction can be difficult to localize using these techniques. Wawrzoniak et al [10] utilize an approach of tracerouting simultaneously from multiple sources to a destination which is a party in an end-to-end loss. The results of the traceroutes are merged and analyzed offline to identify likely causes and locations for the impairment. Research [125] has also shown that active probing can be useful for localizing home WLAN problems to determine if there are signal to noise problems, hidden terminals, or congestion in the local WiFi network.

Inference of link level loss from path level measurements is a common objective of tomography approaches [89] [92]. An important requirement of these approaches is that an accurate topology must be known a priori before the localization can be performed. It is also possible to use path-level round trip measurements to characterize the type of failure [126].

2.3 Collaboration Using Overlays

The main focus of collaboration in this work is the sharing of measured or sensed data from a distributed set of nodes, so that algorithms that want to perform network management functionality have a broad set of data from which to work. In general, methods that scale easily, allow a common API for finding data, and support the easy addition and subtraction of participating nodes are attractive.


2.3.1 Structured Peer-to-Peer

Peer-to-peer networks provide a mechanism to aggregate distributed resources such as storage space and computing time without requiring complicated configuration or cross-domain administrative rights. Unstructured peer-to-peer networks provide loose mechanisms to join and search the network, but rely on broadcast mechanisms to find resources, and as such ultimately lack scalability. Structured peer-to-peer networks have strict joining procedures to control and manage the topology of the network, and are able to utilize this topology to provide a fast search often \( O(\log \# \text{ of nodes}) \). A number of surveys and overviews [13][14] provide more detailed background. Structured peer-to-peer networks have been used to provide a wide range of services including content delivery networks [127][128], next generation Domain Name Services [129], mobility services [130], file sharing [131], service discovery [132], rich query data structures [133][134], efficient multicast [135], and other tasks.

2.3.2 Consistent Hashing

Karger et al [136] developed the key concept that enabled structured peer-to-peer networks, although the original application was to relieve traffic hot spots on the Internet. The breakthrough involved the development of consistent hash functions. A typical hash table is implemented as an array of some fixed size (often a prime number). A hash function maps keys to an ID, which is a pointer to the location where a value is stored. The array’s positions are called hash buckets, and a good hash function is needed to distribute the keys uniformly across the ID space. If more than one key maps to an ID, a collision results, and must be handled. Even with a good hash function, it is typical to start having collisions (and thus poor performance due to the collision resolution function) at greater than 75% to 80% full. A typical solution to this is to grow the hash table, i.e., increase the ID space, and pick a correspondingly different hash function that can distribute keys across the larger space. \( h(x) = (ax + b) \mod m \) is a common hash function, where \( m \) is number of buckets (size of the array), and \( a \) and \( b \) are application dependent.

The problem results from what happens when we grow the number of buckets. If the hash
function is modified to add additional storage, then every value stored in the array needs to get moved to a new location to reflect the new mapping. To minimize this shuffling, many applications use as large an array as memory permits, but this does not work well for applications where the storage amount needs to be adjusted frequently, such as Karger’s effort to replicate web content on demand for hot spots. Consistent hashing is the solution to creating a hash function that when array buckets are added or subtracted, just a few items move. Consistent hashing works by spreading the buckets randomly and uniformly over an ID space (for example a unit interval, often this is implemented as a ring) using a normal hash function. Each bucket is responsible for the space to midway between it and its neighbors, or alternatively, behind it until the next bucket. When a key is mapped using the hash function to the ID space, the bucket whose ID is closest to it is responsible for the key’s storage. New buckets are inserted into the ID space (uniformly with a good hash function), and it collects the values from the nearest buckets from which it is taking over ID space. In order to find a particular key we are able to use a binary search to move to the next nearest bucket to the key, resulting in an average $O(\log n)$ number of routing hops where $n$ is the number of buckets.

Consistent hashing algorithms can then be used to build distributed hash tables, which are used in many overlays. Nodes in a distributed hash table partition the ID space, and the corresponding values are inserted, based on their key, such that a node is like a hash table bucket. Since these nodes are often end-user computers, it is important for them to be able to come and go without having to shuffle the buckets all over the network, so consistent hashing is crucial to insuring the overhead of the key maintenance does not impact the overall system.

### 2.3.3 Randomly Distributed Overlays

One of the earliest and most widely used DHT based overlays was Chord [16]. Chord uses consistent hashing to assign nodes and data items a 128-bit key. Each node is responsible for the data items with keys in the range from its own key to its successor node’s key. When a new node joins the system, its IP address is passed through the hash function to generate its key, and then data
items indexed by keys between the new node and its successor are transferred to the new node. Since the consistent hash function is random, the placement of the nodes in the overlay is also random. Figure 2.12 from [14] shows an example routing table for a Chord network.

Chord keeps what is called the finger table of size $\log n$ with an entry for a node $\frac{1}{2}$ way around the ring, $\frac{1}{4}$ way around the ring, etc. A successor list of the next several nodes following in the ring is also kept in case neighbors go away unexpectedly.

2.3.4 Topologically Aware Overlays

Topologically aware overlays typically add additional complexity to the join and leave steps, as well as additional data structures to make sure the neighbors in terms of the peer-to-peer address space are close in proximity in the IP network space. Topologically aware systems like Pastry [15] and Tapestry [137] are able to implement multicast efficiently. In general, multicast trees have the desired property that nodes in successively smaller subtrees are increasingly near each other in
Pastry adds a neighborhood set to allow it to communicate efficiently with its neighbors, and a tree routing structure that uses prefixes and longest match routing, as shown in Figure 2.13 from [14].

2.3.5 Collaboration using other methods

In addition to overlays there are other approaches to share measurements including

- Database replication
- Publishing of results to a central server
• Content replication

Database replication techniques [138] [139] are available to mirror data from one server to others. This is usually done for improved reliability and performance. Applications have been developed for smart phones [140] and home broadband routers [141] to perform measurements and publish the results to a central server for analysis. This crowdsourcing of the measurement provides for a very diverse set of measurements without adding significant burden to any one end node. Content replication provides a more finely grained method of selecting what data you want replicated between a set of servers, and has been used with success [142] in particular in military networks with significant limitations on bandwidth availability.

2.4 Properties of Scalable Network Management Systems

The taxonomy and review presented in this chapter serves to document a number of tradeoffs possible in designing network management systems. We highlight several keys ones below, and present arguments for preferred properties in scalable network management systems.

• Active versus Passive measurements – For many types of measurements, passively recording normal traffic flows can be more representative of the desired traffic experience than an artificial flow which may be treated differently by the network. This comes at the expense of not controlling which measurements you can take at a particular time, typically resulting in significantly longer times to collect the same data, or relying on smaller sets of data with less temporal correlation. Passive measurements do conserve bandwidth and power which can be important considerations in wireless networks.

• Single-ended versus Double-ended – Measurements that can be performed by a single host, and do not require cooperation with another endpoint increase deployability of a technique, at the expense of round trip effects such as asymmetric paths or bandwidths impacting the measurements.
• Algorithm approach – Event-driven algorithms which are incremental, and can build on previous inferences as they get more information are a natural fit.

• End-to-end versus Domain – Domain-based solutions typically can provide significantly more information, but only to parties with administrative privileges, and only for devices that support explicit management. End-to-end solutions handle cross-network approaches better.

• Use of positive and negative symptoms for localization – Most methods use indications of problems alone to determine the location of a failure. It is useful to include positive information to help reduce the search space.

• Probabilistic versus Deterministic – Uncertainty is present in many forms in network management systems, and can come from measurement error, cross-traffic biases, and insufficient data, among many other sources. Deterministic solutions, and the assumptions they require to operate in these situations are suitable for only the simplest of networks.

A number of key tradeoffs in architectural organization, data collection, reasoning algorithms, measurement methodologies, and collaboration frameworks were identified. The ability to assess the most useful choice for these methods in a number of different environments is crucial to the management of our ever increasing networks. The research presented in this work was analyzed and developed using these metrics to assess the appropriateness, accuracy, scalability, and the network overhead of the designs.
Chapter 3

Tactical Network Characterization

3.1 Overview

Tactical networks are telecommunication systems designed to support situational awareness, collaborative planning, command, and control functions for military units of all sizes, and across all environments. Tactical networks are designed to be picked up and moved to wherever they are needed. They have multiple redundant network technologies that support different capabilities in different conditions. The main requirements are the support for:

- Voice [143]
- Low bit rate data messages (alerts, position updates) [144]
- Chat type capabilities [145]
- Real time white board type collaboration [146]
- Real time video streaming from sensor platforms [147]
- Multimedia sharing of pictures and documents [148]

In order to provide these capabilities in diverse environments, in a rapidly deployable fashion with infrastructure that is transportable, the following technologies are used:
- Ad hoc self-healing networks [149] [150]
- VHF [143] [151]
- UHF [152]
- Microwave line of sight (LOS) [153]
- L, Ku, Ka, and X band satellite connections [154]
- Fiber and other wired connections for long term outposts

These connections are provided redundantly to the different layers of command that exist in the military. Figure 3.1 from [155] shows a typical network layout for a deployed brigade.

The military deploys in hierarchical elements of command known as echelons with each layer getting its own infrastructure, so that an echelon can be deployed independently of others. In the
US Army, the typical hierarchy that is deployed as independent elements is:

- Company - around 200 people led by a Captain. In Iraq in 2007 during the surge [156] companies were deployed out individually to Combat Outposts (COPs).

- Battalion - commonly made up of 4 companies and a headquarters staff, led by a Major.

- Brigade - commonly made up of 4 combat battalions and supporting units, and led by a Colonel.

In the US Army, a brigade is typically the smallest completely self-sufficient unit that is deployed for extended periods, and is the size of unit that is rotated in and out of the theater of operations. When a brigade deploys, the various headquarters set up their infrastructure at wherever they will be operating, and are linked together. There is a typical daily planning cycle [157] where the upcoming missions are planned and resourced, and this planning is performed in a hierarchical manner. Brigade staff are given a broad directive, and determine which lower echelon resources to dedicate to the task. The lower echelon headquarters staff are informed of their specific piece of the mission, and plan the operation in more detail. Intelligence is consulted, and detailed plans are assembled and shared with the higher echelon commands, until a complete detailed plan is presented, reviewed, and approved [158]. As part of the planning cycle, multimedia collaboration between the planners is critical, and the staff needs to be able to share maps, voice, PowerPoints, etc. between all elements. An operations cycle is the execution of the plan and requires real-time detailed coordination of all elements deployed. This will include position updates of every vehicle performing the mission, chat and voice messages detailing specific instructions during the mission, and requests for assistance or clarification up and down the command hierarchy. Crucial for all of these actions is functional network connectivity including all the way out to the commanders performing the operation on the move, and for these forward commanders to be able to communicate with individual lower level units like squads and platoons who might be dismounted and operating on foot.
There are a number of information systems that utilize the communications infrastructure and provide tools and resources for the planning and execution of missions. A number of key applications include:

- **DCGS-A** - An intelligence repository focused on the headquarters staff intelligence teams at the battalion and above echelons [159].

- **CPOF** - The Command Post Of the Future is a real-time collaboration tool that allows for shared voice, whiteboard, and maps amongst commanders at the battalion and above [160].

- **TIGR** - The Tactical Ground Reporting System is a map-based tool for planning and execution and provides a repository of planning and patrol reports of past missions at the company and higher echelons [17].

### 3.2 TIGR - A Tactical Network Overlay

The Tactical Ground Reporting System (TIGR) [17] is an application overlay that facilitates collaboration and information sharing at the company and above. The Defense Advanced Research Projects Agency (DARPA) began the development efforts in 2005, and introduced the system to a single brigade in Iraq in the beginning of 2007. Its use spread to all US Army units in Iraq and Afghanistan, as well as Kuwait, the Philippines, South Korea, and the Horn of Africa. The system enables collection and dissemination of fine-grained information on people, places, and events, and offers a media-rich view of the battlefield with digital photos, videos, and high-resolution geo-spatial imagery.

TIGR addresses a series of challenging requirements - the need to efficiently distribute and search large amounts of data, and the need to support multimedia attachments, using bandwidth-efficient thumbnails and compressed versions of the multimedia. TIGR fills a gap between Operations and Intelligence by providing a theater-wide distribution of patrol reports, significant actions and other tactical information.
TIGR utilizes a general information model that is simple and intuitive. The main elements are:

- **Events** - Represented as icons on the map, these describe enemy actions as well as friendly activities such as meeting with a local administrator.

- **Places** - Represented as icons on the map which denote a physical location such as a traffic checkpoint, school, or minefield.

- **Reports** - These are summaries that pull together Events and Places with additional information to capture a particular mission or patrol performed by a company.

- **Tasks** - These are used for mission planning and execution and capture the intended tasks to be performed.

- **Collections** are containers which can be used to group other objects, such as a collection of all patrols by a particular company in June.

TIGR is a distributed cloud-based overlay designed to operate on disadvantaged networks. TIGR uses persistent TCP connections over existing network assets – tactical network assets under unit control and the strategic network – as available. The TIGR overlay consists of servers located at large Forward Operating Bases (FOBs), as well as remote outposts such as Joint Security Stations (JSSs) and Coalition Outposts (COPs), and on-the-move Command vehicles such as company commanders’ vehicles. A snapshot of the Iraq server topology when at its most distributed level is depicted in Figure 3.2. The server distribution is critical to providing reliable functionality on tactical networks. Tactical networks are unreliable for many reasons; for example, power disruptions or difficult terrain can cause outages. Hence link outages and disruption are the norm rather than an exception. By placing servers close to the users at the edge, soldiers can continue to exercise TIGR functionality even when the WAN connectivity is interrupted. Once the network link is restored, TIGR’s synchronization mechanism automatically exchanges the data generated during an outage.
Figure 3.2: TIGR overlay topology in Iraq at its largest [17]
TIGR has a policy driven synchronization mechanism which provides fine grained control over what types of information need to be shared to anywhere in the tactical network. In order to efficiently use a wide variety of heterogeneous networks of various capabilities, TIGR has implemented many networking concepts at the application level. Effectively, TIGR is an advanced policy controlled network implemented at the application layer which is overlayed onto the tactical network. TIGR utilizes both priority and round robin queues as well as a general open loop token bucket to manage pacing of data into the tactical network, as illustrated in Figure 3.3.

3.3 Access Bandwidth

The access link (also known as the last mile) for tactical networks typically has much less bandwidth than the Internet. Figure 3.4 illustrates the comparison between tactical networks, the fixed Internet, and on the move (cellular 3G/4G) Internet. Of special note is that the y-axis is log scale. The tower based infrastructure of the mobile Internet provides 2-4 Mbps outdoors and 1-3.5 Mbps indoors for 3G technology and 5-8 Mbps both outdoors and indoors for 4G technology [161]. The fixed infrastructure options for home and enterprise use range from traditional dial-up modems (2% of US households in 2013 [162]) to Gigabit/s connections [163]. For tactical networks, the
headquarters access links are the only links with bandwidth of the same order of magnitude as the commercial options [141], while the vehicle and dismounted soldier options are all worse than traditional dial-up from a bandwidth perspective.

### 3.4 Access Diversity

The vehicles and portable headquarter facilities used in tactical networks typically have more than one access link to the network. The multiple network connections provide redundancy and different capabilities depending on the environment. In general, the connections trade off bandwidth for accessibility. The higher bandwidth connections may only work within line of sight of a tower, while the satellite connections that are highly available (unless there is a terrain obstacle such as a steep cliff between the vehicle and the satellite) are lower bandwidth. Figure 3.5 shows a typical Company Commanders vehicle as provided by the WIN-T Increment 2 [164] capability deployed.
by the US Army, and the large number of antennas representative of the access diversity.

Figure 3.6 shows a typical Ku band dish representative [165] of the satellite capabilities available to portable headquarters. Figure 3.7 shows the line of sight tower capabilities also available to portable headquarters.

The presence of the multiple redundant access technologies impacts the average vertex count for the network graph as compared to the Internet. The ability to model the transition of an access link from one technology to another, the need for an application to recognize dramatic changes in the connectivity capabilities and accommodate them are important points of consideration.

### 3.5 Access Latency

Delays in the first hop come from four main area’s, Layer 1/Layer 2 protocol delays, serialization delays, propagation delays, and cryptographic delays. Figure 3.8 shows the inherent Layer 1 and Layer 2 protocol delays for the most affected technologies. Note that the y-axis increases by a factor of two each increment. DSL access links commonly utilize interleaving to make the data less susceptible to bursts of noise and this interleaving can add 20-40 ms to the first hop delay [166]. HSPA based 3G mobile networks have up to a 2 second delay on the first connection of a
Figure 3.6: WIN-T Increment 1 satellite truck [165]

Figure 3.7: WIN-T Increment 2 tactical communications node [164]
data session, and then 75-150 ms round trip delay added to the first hop for remainder of the session [167]. With LTE 4G based mobile networks the first hop is a 50 ms delay for the first connection and 10-15 ms round trip delay added for subsequent packets [168]. Ad hoc wireless networks such as the Wideband Networking Waveform (WNW) utilize a TDM based slot assignment mechanism [150] which can introduce 130-250 ms to the first hop delay.

With the speeds of most commercial network access links serialization is usually not a consideration, but for the mobile tactical networks serialization can be a factor. Figure 3.9 shows serialization delay for a range of speeds (note both axis are logarithmic). The serialization delay is calculated simply as $\frac{\text{Size}}{\text{Rate}}$ where $\text{Size}$ is the packet size to be transmitted in bits and $\text{Rate}$ is the access bandwidth in bits per second.
Propagation delay is also typically not a major element in first hop delay, as most first hops are within the distance to the horizon, but in tactical networks the first hop can be a satellite bounce to a ground station far away. Figure 3.10 shows the two way speed of light propagation delay for common distances. A tactical satellite connection commonly has a 600 ms round trip delay associated with the propagation.

Finally, another access latency component unique to tactical networks is that some of the wireless links have a delay induced by the overhead of performing encryption and decryption at each end of the link. All of these delay mechanisms impact modeling of the access link for edge nodes to the network, and are crucial for accurately modeling the performance as experienced by the user.

### 3.6 Core Latency

Latency between the core elements (i.e., not the first hop) of a tactical network also differs compared to the Internet. An all pairs ICMP RTT measurement (Study 1 Appendix B) between PlanetLab nodes was performed as part of the iPlane project [104] in December 2012, and a similar set
of measurements (Study 2 Appendix B) was made between all TIGR servers participating in an overlay on a tactical network in Afghanistan also in December 2012.

Figure 3.11 shows the histogram for the iPlane data and the tactical network data. Figure 3.12 shows the cumulative distribution function for the same data. The tactical network data has a significantly longer tail. Of particular interest are the "harmonics" present at multiples of a round trip time of a satellite hop; there are one hop, two hop, and four hop samples in the study. There is less data in the intermediate delay range of 200 ms to 500 ms for the tactical network, and this is representative of the fact that the tactical network is geographically concentrated in Afghanistan, for the most part, with some satellite hop extensions. Delays in the core of the Internet are strongly correlated with geographic distance [169] and the PlanetLab servers are much more geographically distributed than the tactical servers.

In order to quantify the distributions of these server to server core network latencies, a mixture model [170] of distributions was selected, because we clearly have data from multiple different distributions, i.e., locally connected, cross-country, worldwide, and satellite delayed. A mixture model allows us to model the set of observations without knowing the membership of subsets. An
Figure 3.11: Histogram of all pairs ICMP RTT’s

Figure 3.12: CDF of all pairs ICMP RTT’s
example of a mixture model for the likelihood of a set of single dimensional Gaussians [171] can be expressed as a sum of \( k \) models with a mixing proportion \( a_k \) as in Equation 3.1.

\[
P(x|\mu, \sigma^2) = \sum_{i=1}^{X} \sum_{k=1}^{K} a_k \frac{1}{\sqrt{2\pi \sigma_k^2}} \exp\left(-\frac{(x_i - \mu_k)^2}{2\sigma_k^2}\right)
\]  

(3.1)

We experimented with a number of different distributions including gammas, exponentials, and Gaussians, and used Expectation Maximization (EM) to find the distributions that fit the data best. EM [172] is a common tool for finding the maximum likelihood estimator of parameters for a mixture model. In a typical Maximum Likelihood Estimation (MLE) analysis the log likelihood of the observed data is maximized given the parameter \( P(x|\theta) \). In some situations, it is possible to simply take the derivative of the log likelihood and set it equal to zero, and check if it is a local or global maximum. EM allows the same results for situations when the pdf is not available in closed form, or the derivative is difficult/intractable. EM works by taking the observed data \( y \) and iteratively guessing the complete data \( x \) and finding the parameters \( \theta \) that maximize \( P(x|\theta) \) by trying to find the maximum likelihood of \( \theta \) given \( y \). The algorithm [173] works as follows:
Table 3.1: Internet mixture components

<table>
<thead>
<tr>
<th>Component</th>
<th>Mixing Ratio</th>
<th>Parameter One</th>
<th>Parameter Two</th>
</tr>
</thead>
<tbody>
<tr>
<td>$k_0$</td>
<td>$a_0 = .2$</td>
<td>$\mu = 270$</td>
<td>$\sigma = 73$</td>
</tr>
<tr>
<td>$k_1$</td>
<td>$a_1 = .24$</td>
<td>$\mu = 160$</td>
<td>$\sigma = 33$</td>
</tr>
<tr>
<td>$k_2$</td>
<td>$a_2 = .56$</td>
<td>$\lambda = .011$</td>
<td></td>
</tr>
</tbody>
</table>

1. Make a initial guess of $\theta$

2. Given the observed data $y$, assume that $\theta$ is correct and calculate how likely the data is $x$ i.e. calculate $P(x|y, \theta)$

3. Make a new guess of $\theta$ and using our estimate of the probability distribution from the previous step, integrating over all values of $x$ and weighting log $P(x|\theta)$ by the probability of $P(x|y, \theta)$

4. Make a new guess at $\theta$ to maximize the function in the previous step

5. Repeat until convergence

Just like MLE, EM can converge on local maxima, so it is important to run numerous simulations with different random starting points and keep the best one.

After experimentation, distributions were developed for both the Internet and tactical network scenarios. For the Internet server to server ICMP data a mixture of one exponential and two Gaussians resulted in a good fit. Table 3.1 shows the mixing properties and distribution parameters for the functions.

A histogram of the server to server data collected from the Internet data set and random samples pulled from the derived mixture model developed above is shown in Figure 3.13, followed by a cumulative delay function plot for both in Figure 3.14.

Internet delays have been shown to be largely correlated with geographic distance [169], and studies have found similar peaks [174] at 45ms, 135ms and 295ms, as also seen in our data, and
Figure 3.13: Internet data and model samples histogram
Figure 3.14: Internet data and model samples CDF
which our model successfully captures. These delay peaks were found to be clusterings of geographically related hosts in [174] of which 45% of the nodes were in North America, 35% of the nodes were in Europe, and 9% of the nodes were in Asia, with the remaining 11% distributed outside of those regions.

The goodness of fit for the model was evaluated using two techniques, a $\chi^2$ test [175], and the Kullback-Leibler divergence [176].

The $\chi^2$ goodness of fit technique [177] is based on the sum of differences of observed and expected observations, squared and divided by the expected value.

$$\chi^2 = \sum_{i=1}^{n} \frac{(O_i - E_i)^2}{E_i} \tag{3.2}$$

This value is then compared to a $\chi^2$ distribution with the correct degrees of freedom to determine the overall goodness of fit. The number of degrees of freedom is $(N - p - 1)$ where $N$ is the number of non-empty bins and $p$ is the number of parameters estimated from the same data. The smaller the $\chi^2$ critical value the better, and when two distributions are identical the critical value will be zero. The expected distribution should have no zeroes in it, and should have five or more samples in each bin for the metric to be accurate [178]. Our data sets had problems with both of these requirements, which results in inaccurate $\chi^2$ critical values, so we looked to alternative measures of fit.

The Kullback-Leibler (KL) divergence is a measure of the relative entropy between two distributions and has been proposed as a better measure for mixture models [179]. If you have data generated from distribution $P$ a lower bound on the average code length in bits required to state the data generated by $P$ is given by $H(P)$, the entropy of $P$,

$$H(P) = \sum_{i=1}^{n} p_i \log_2 \frac{1}{p_i} \tag{3.3}$$

If $P$ is unknown, and we instead assume the data was generated by a known function $Q$, then the average number of bits required to encode the data with this less efficient function will be
larger. The KL divergence is a measure of this inefficiency. The KL divergence is non-symmetric, non-negative, and equal to zero if \( P = Q \). Both \( P \) and \( Q \) must sum to 1, and if \( Q_i = 0 \) then \( P_i \) must \( = 0 \) for all \( i \). The KL divergence of \( Q \) from \( P \) is then calculated as

\[
D_{KL}(P||Q) = \sum_{i=1}^{n} P_i \log_2 \frac{P_i}{Q_i}
\]  

While a closed form solution does not exist for solving for the Kullback-Leibler divergence for a mixture of Gaussians [180], we can use a Monte-Carlo approach to discretely determine the divergence via simulation.

To illustrate the KL divergence for a simple case, we explore four Gaussians with the same \( \mu \) of 100, and a variety of standard deviations (\( \sigma \)). Assuming \( P \) is a Gaussian with \( \sigma = 10 \), we calculate the KL divergence of each of the other three Gaussians with respect to \( P \) using the code from Listing A.6. The results are shown in Figure 3.15 where it can be seen that as the value of \( \sigma \) gets closer to \( P \)'s value of ten, the KL divergence goes towards zero. The KL divergence for a distribution with \( \sigma = 60 \) from the distribution with \( \sigma = 10 \) was 1.85 bits, while the divergence from the distribution with \( \sigma = 30 \) was .9 bits, and the divergence of the distribution with \( \sigma = 11 \) was only .008 bits.

The KL divergence provided a useful metric for evaluating the impact of adding more components to the mixture, similar to how entropy is used as a metric for determining that a MCMC simulation has stabilized. For example, if adding another Gaussian component resulted in negligible improvement in the divergence, or if it made it worse, then that new component is not providing utility. Table 3.2 shows an EM run with ten Gaussians, with the KL divergence being calculated with the introduction of each component. It can be seen that adding component 6 produces a minimal improvement (KL divergence went from .143 to .141), and adding component 10 actually makes it worse (the KL divergence went from .016 to .253).

For the tactical network server to server ICMP data the best fit to the data was found by EM to be a mixture of six Gaussians. Table 3.3 shows the mixing ratios and parameters of the functions.

A histogram of the server data collected from the tactical network data set and random samples
Figure 3.15: Kullback-Leibler divergence illustration

<table>
<thead>
<tr>
<th>Component</th>
<th>µ</th>
<th>σ</th>
<th>KL Divergence (bits)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>228</td>
<td>.1</td>
<td>1.261</td>
</tr>
<tr>
<td>2</td>
<td>1117</td>
<td>.1</td>
<td>0.281</td>
</tr>
<tr>
<td>3</td>
<td>579</td>
<td>11</td>
<td>0.266</td>
</tr>
<tr>
<td>4</td>
<td>2516</td>
<td>60</td>
<td>0.163</td>
</tr>
<tr>
<td>5</td>
<td>160</td>
<td>53</td>
<td>0.143</td>
</tr>
<tr>
<td>6</td>
<td>147</td>
<td>50</td>
<td>0.141</td>
</tr>
<tr>
<td>7</td>
<td>178</td>
<td>55</td>
<td>0.086</td>
</tr>
<tr>
<td>8</td>
<td>72</td>
<td>18</td>
<td>0.036</td>
</tr>
<tr>
<td>9</td>
<td>184</td>
<td>54</td>
<td>0.016</td>
</tr>
<tr>
<td>10</td>
<td>52</td>
<td>13</td>
<td>0.253</td>
</tr>
</tbody>
</table>

Table 3.2: Ten component mixture assessment
from the derived mixture model developed above is shown in Figure 3.16, followed by a cumulative delay function plot for both in Figure 3.17. The code that does the Expectation Maximization and plots these figures is found in Listing A.5. This mixture model provides insights into the composition of links that we will utilize in the next chapter.

### 3.7 Subnet Size Distribution

The distribution of subnets at the edge is also different between the Internet and tactical networks. Most residential Internet connections are a single IP address where the home network behind the router is expected to use Network Address Translation (NAT) to access the Internet [141]. Smartphones accessing 3G/4G networks are similar, where a single address is assigned to the phone and NAT is performed by the carrier [181]. On the tactical network there are a wide variety of edge subnet assignments. For the majority of vehicles that are satellite connected, they are assigned an address out of one of several large flat /16 networks, each capable of having 64,382 hosts. These vehicles are all one satellite hop away from the same default router, as part of the Blue Force Tracking network [182] as seen in Figure 3.18.

Figure 3.19 shows a representative WIN-T Increment 2 subnet layout such as what would be used for the network in Figure 3.1. The headquarters subnets are either /24s or /25s, and the commanders vehicles are /29s. Figure 3.20 shows the distribution of all of the data subnets allocated to a brigade.

In general, edges in the Internet are much more likely to have a single “routable” IP address,
Figure 3.16: Tactical data and model samples histogram
Figure 3.17: Tactical data and model samples CDF

Figure 3.18: Blue force tracking network architecture [182]
Figure 3.19: WIN-T subnet design

Figure 3.20: Brigade subnet distribution
and employ NAT to access the globally routed network, while in tactical networks we have two extremes, either small subnets that are all globally routable (no NAT necessary) or very large flat subnets attached via satellite, a scenario that just does not exist in the Internet today. In the Internet it is not uncommon to assume that all addresses within a /24 boundary are located in the same geographic region and reached by a similar access link. In the tactical network we have seen that adjacent /29 networks can be distributed much further apart, and use completely different access links.

### 3.8 Network Infrastructure

There are a number of fundamental differences in the network architecture of a tactical network as compared to the commercial Internet. They can be summarized as follows:

- No peering
- Relocatable IP address blocks
- Inconsistent DNS usage

There are not multiple competing network operators, so there is no need for peering points such as IXPs and the resulting additional border type routers with their complicated policy routing. Since there are no economic motivations for it, there is no hot potato routing [183], so routes between two points are much more likely to be symmetrical. Figure 3.21 shows a typical pair of routes between two hosts on the Internet, where the only common hops are the routers closest to each node. Figure 3.22 shows a similar trace between two hosts on a tactical network, and in this case, there is only one hop that is not common in the two paths. Note in both of these diagrams alias router interfaces have been merged.

Figure 3.23 shows a router level topology of the links that were used by the TIGR overlay in Iraq in 2008 (Study 3 Appendix B). The visualization of the network is done using the JavaScript

53
Figure 3.21: Typical Internet two way path

Figure 3.22: Typical tactical network two way path
Figure 3.23: Iraq tactical network topology

based canvas implementation in Listing A.1 by the author to allow inspection and manipulation of the network nodes.

Figure 3.24 shows a router level topology of the links that were used by the TIGR overlay in Afghanistan in 2013 (Study 4 Appendix B) after the draw-down had begun, and the US Army had consolidated their units into a few larger bases.

What you see in common in both of these tactical topologies is edges that are either clustered around a common router feeding a high latency satellite distribution point, or a small chain of low latency line of sight or fiber type connections running to new geographic regions. In general, there are relatively few router hops that are at the same general location (i.e., chains of routers with 1 ms or less of latency). The ones that do exist are at hubs related to the command structure deployed in these environments.

Figure 3.25 shows a partial router level topology of the links that were used by the iPlane PlanetLab overlay on the Internet in the 2012 measurements found in Study 1 Appendix B.

Two key items related to topology are of note, first, that the average end-to-end hop count is much higher for the Internet case, as expected, and second, long chains of low latency hops leading to the edges. These are artifacts of the distribution structure common with ISPs and their border
Figure 3.24: Afghanistan tactical network topology

Figure 3.25: Partial iPlane PlanetLab network topology
A histogram of the end-to-end hop count for the Internet data was produced using the code in Listing A.3 and the result is seen in Figure 3.26. In Figure 3.27 we fit a Gaussian to the data shown in Figure 3.26 to improve our understanding of it, and the best fit determined by EM was a Gaussian with a $\mu = 15$ and $\sigma = 4.95$.

A histogram for the combined tactical data was produced using the code in Listing A.4 and the result is seen in Figure 3.28. In Figure 3.29 we fit a Gaussian to the data shown in Figure 3.28, and the best fit determined by EM was a Gaussian with a $\mu = 6.51$ and $\sigma = 1.63$, which is under half...
the size of the mean Internet hop count.

The network blocks assigned to a brigade travel with the brigade, so these large network blocks relocate from the military backbones in the US, to training centers for the units pre-deployment training, and then to the tactical theater as the unit deploys overseas. The configuration, setup, and transportation of the entire infrastructure is done as part of regular garrison training, then again at one of the pre-deployment training centers such as NTC or JRTC [184], and then in the tactical theater when deployed. The closest analogy on the Internet is when a large enterprise with its own dedicated address block switches from one ISP to another for service.

Domain Name Service (DNS) [185] is something that is considered essential for the Internet, and while there are certainly DNS servers integrated into the tactical network, many of the tactical servers are not kept up to date due to the rate of change of equipment and personnel. It is not uncommon for users to bookmark raw IP addresses for their servers.

### 3.9 Tactical Network Characterization Conclusion

Tactical networks, designed to facilitate rapid deployed command and control communications for militaries, have many properties that differ from the Internet. We have explored how the access links in tactical networks typically have less bandwidth, higher latencies, less address translation, and more redundancy than typical Internet access links. Properly modeling and accounting for these access links directly impacts the end user experience of network services. We have additionally seen that the tactical core network has a smaller diameter due to covering less geography and to lack of IXP type network constructs, along with a significantly different latency profile as compared to the Internet.
Chapter 4

Network Topology Generation

4.1 Overview

A network’s topology is a representation of the nodes and edges along with any relevant properties that make up a graph. There are different levels of network topology abstraction used by researchers. For example AS level topologies may be used to research the impact of policy changes or economic choices on the Internet [186] [187]. Router level topologies are used to enable development, testing, and evaluation of new applications and technologies [188] [189].

There has been significant research on network topology generators, and their evolution has occurred as the general understanding of the Internet’s structure has improved. The simplest generators are ones that generate a random graph. The most common of these were based on the Waxman model [190], which is a variation of the Erdos-Renyi random graph [191]. Graphs based on this model exhibit a locality feature, nodes close together have a higher probability of being interconnected. Nodes are placed at random in a two dimensional space of maximum distance $L$, and the the probability of any particular edge is found by:

$$P(u,v) = \beta \exp\left(-\frac{d(u,v)}{L\alpha}\right)$$ (4.1)

where $d(u,v)$ is the Euclidean distance between nodes $u$ and $v$, and $\beta$ and $\alpha$ are constants.
Figure 4.1: Waxman model graph [192]
Figure 4.1 from [192] shows a representative random graph $G(n, \frac{2}{n})$ where $n$ is 1000.

The next stage of topology generators attempted to mimic structural components found in real world networks, in particular redundancy and hierarchy, in what is called the transit-stub model [193] [194]. In this type of model, routers are assigned to a domain, and then domains are further classified as either transit or stub. Nodes are more likely to be interconnected to other nodes in their same domain, and stubs are more likely to have an external connection to a transit domain than another stub. These properties introduce hierarchy to the graph. These approaches were incorporated into a software package called the Georgia Tech Internetwork Topology Models (GT-ITM) [195], and Figure 4.2 from [196] shows an example graph constructed using this technique.

The next stage of topological model was brought on by observations that Internet topologies exhibited power-law degree characteristics [197] at different levels. Several new topology generators [198] [199] [200] were created that targeted reproducing the observed statistics as closely as possible. The most common approach for generating graphs matching the power-law statistics was an algorithm called preferential attachment. Preferential attachment works by sequentially attaching new nodes to the graph, and for each new node attached, it prefers to connect to nodes with many existing connections. The probability of a particular edge connected to new node $u$ is
found by:

$$P(u,v) = \frac{k_v}{\sum_j k_j}$$  \hspace{1cm} (4.2)

where $k_v$ is the degree of existing node $v$, and $j$ is all existing nodes.

Graphs produced with this algorithm have the power-law node degree properties observed in [197] and an example of one of these power-law properties can be seen in Figure 4.3 from [201] where the x-axis is number of links connected to a router, and the y-axis is the count of routers with that property.

The purely statistics driven models received some push back in 2004 [202], when a *First Principles* based approach was proposed. The first principles were based on technology constraints, and argue that the current and near term router technology places some boundaries on overall bandwidth capabilities of a router, as well as the number of physical interfaces with which it can interconnect to other routers. Li et al [202] introduce a feasibility region that shows what is likely practical and suggest limiting properties such as node degree based on these feasibility regions.
Figure 4.4 from [202] illustrates the concept.

The current state of the art topology generators [203] utilize further understandings of the underlying business impacts (presence of Access and Intra AS tiers, IXP peering points, and geographic regions) and the first principles of router technological constraints (backplane bandwidth and number of physical links possible at a given link rate) to build very realistic Internet topologies. Figure 4.5 shows a representative model of the elements in a typical Internet infrastructure including multiple Autonomous Systems (AS’s) with their respective access layers, distribution layers, and border routers, meeting at IXP peering points with core routers interconnected to other geographic regions with similar constructs.

There are a number of common metrics used for comparing topologies. Some of the simpler ones include the diameter of the network, clustering coefficient, assortivity, and the node degree distribution, i.e., how many nodes have a particular number of edges. When you represent the connections between nodes in a graph as a normalized Laplacian matrix, the eigenvalues and their counts make up the Laplacian spectrum [204] of a network. These graph spectra have been shown [205] to be applicable to network graphs at many different layers. Li et al in [202] introduced a
structure metric

\[ s(g) = \sum_{i,j \text{ connected}} d_i d_j \quad \text{where } d_i \text{ is degree of node } i \quad (4.3) \]

which is a measure of how structured or "hub-like" a graph is. Essentially this is the likelihood that a particular graph was created randomly, where the more structure the less likely. This metric is often normalized by the largest \( s(g) \) possible in the graph to have

\[ S(g) = \frac{s(g)}{s_{\text{max}}} \quad (4.4) \]

Figure 4.6 from [202] shows two graphs with identical node rank versus node degree distributions, but clearly different structures. The left one has a \( S(g) \) of .2 and is much more likely to be man made, versus the graph on the right which has a \( S(g) \) of .6 and looks much more random.

In the next section we will do an analysis of the approaches necessary to modify state-of-the-art Internet topology generators to instead produce tactical network topologies.
4.2 TopGen

TopGen [203] is a generic, extensible topology generator, written in C# [206], with source code available [207]. TopGen was developed by Ingo Scholtes while at the University of Trier, and supports defining different router types such as core, border or access, each with their own number of links, and backplane bandwidths that can be tailored to the current technology as shown in Figure 4.4.

Router connectivity is very flexible in TopGen, and it was designed from the start to support different interconnection modules. It comes with several random graph generators like Erdos-Renyi, and Barabasi-Albert, as well as structural models such as trees, meshes, and finally, a first principles model that generates structured Internet topologies such as that found in Figure 4.5.

Regionality is implemented so that you can specify a number of regions, each with their own independent topologies, and then have the regions interconnected using a topology more realistic for situations like undersea fiber interconnecting continent regions.

Finally, TopGen supports Graph Analysis by implementing tools for calculating network diam-
eter, and generating PDF, CDF, and CCDF plots of the vertex degree and likelihood distribution, as well as the clustering coefficient, assortivity, and average vertex degree of the topology.

Figure 4.7 shows the normal user interface with parameters selected to generate an Internet accurate topology.

4.3 Tactical TopGen - New Capabilities

We modified and extended TopGen to create Tactical TopGen. A number of general capabilities were added to the TopGen framework to increase its utility. Some of these new capabilities are for exporting the topologies generated by TopGen in new formats so they can be interacted with by additional third party tools. The other new capabilities are related to increasing the native handling of edges to include property assignment and management of these properties, primarily delay and
4.3.1 JavaScript Visualization Export

A method to visualize and inspect topologies in the web browser was developed, and then TopGen was extended to be able to export the topologies it generates in a JavaScript array format that the web tool can incorporate.

The exported format is a simple JavaScript file containing a single array that is parsed and rendered by the visualization program. The Tactical TopGen output format consists of an array with each row describing an edge between two nodes, and the delay in milliseconds on that edge. A partial example is shown in Listing 4.1 where the array row traces[0] reflects an edge with 80 ms delay from Node ‘A’ to Node ‘I’.

```
Listing 4.1: JavaScript Array Format

var traces = new Array();
traces[0] = ['A', 80, 'I'];
traces[1] = ['A', 80, 'S'];
traces[2] = ['A', 80, 'K'];
traces[3] = ['B', 80, 'S'];
...
```

In addition to the JavaScript file filled with traces, TopGen now also exports a delays file which has a row for the best path between every two nodes in the entire topology, along with the delay and number of hops on that path.
The renderer was designed to import the external file with the nodes and edges, parse the array, creating nodes and edges in the web browser, handling duplicates and updates as necessary. The renderer utilizes a force directed graph layout library written in JavaScript called Springy.js [208]. Springy uses a physics based force spring methodology to minimize the overlap of nodes, and proportionally spaces them related to a distance metric provided by the user. Springy’s primary capability is that it provides an abstraction for graph manipulation and for calculating the layout. In our renderer we use the $\log_2$ of the delay on an edge as the spring length at rest in the simulation. Figure 4.8 shows a 20 node network with a random mesh topology generated by TopGen to demonstrate the export capability. Listing A.1 contains our routines developed to read in the TopGen generated topology array and render them.

### 4.3.2 Edge Property Management

TopGen as originally written had router properties (such as the number of interfaces and backplane bandwidth) but did not have any properties associated with generated edges. In order to generate topologies with useful edge labels such as delay and maximum link bandwidth, we modified the
Edge and Vector classes and extended the Graph.AddEdge and Router.GetEdgeDelay methods. We then modified the connector classes to be able to allow the routing modules to see and incorporate these parameters into their routing layout. These properties are now assigned to edges when topologies are generated and can be viewed in the multiple new export formats.

4.3.3 Mininet emulation support

Mininet [209] is a network emulator that allows the researcher to create a virtual network composed of hosts and forwarding elements, which can be configured in a wide variety of topologies. The virtual hosts run Linux, and regular Linux software can be run on them. The links between hosts and forwarding elements can be assigned delay, bandwidth, and loss. Normal networking utilities such as ping and traceroute can be utilized to investigate the network.

An example of the Mininet configuration file format that TopGen can now export is show in in Listing 4.2. There are two types of lines in the comma separated value formatted file. The first is a list of nodes, with a unique name by which they are referred, and an IP address that should be assigned to the node. The second type of configuration lines are links that list a node, the IP address assigned to that end of the link, another node, with its IP address, a link bandwidth in Mbps, a link delay, and a link loss rate.
Listing 4.2: Mininet Configuration File Example

node, h1, 10.0.1.1/24
node, h2, 10.0.2.1/24
node, h3, 10.0.3.1/24
node, h4, 10.0.4.1/24
link, h1, 10.0.1.1/24, h2, 10.0.1.2/24, 0.128, 500, 0.1
link, h2, 10.0.2.1/24, h3, 10.0.2.2/24, 25, 20, 0.01
link, h3, 10.0.3.1/24, h4, 10.0.3.2/24, 0.128, 500, 0.1

These configuration files can then be loaded and run (Figure 4.9) using a Mininet topology instantiation script which we developed and the virtual network can then be accessed with terminals and other typical interfaces.

```
mininet@mininet-vm:~$ sudo ./tactical_topo.py
net.ipv4.ip_forward = 1
net.ipv4.conf.all.mtu = 0
*** Adding controller
*** Reading topology
*** Building topology
*** Configuring hosts
AJSLeaf ISatLeaf JSatLeaf JSatLeaf AJSLeaf JSatLeaf ISatLeaf JSatLeaf JSatLeaf AJSLeaf JSatLeaf ISatLeaf JSatLeaf JSatLeaf AJSLeaf JSatLeaf ISatLeaf JSatLeaf JSatLeaf AJSLeaf JSatLeaf ISatLeaf JSatLeaf JSatLeaf AJSLeaf JSatLeaf ISatLeaf JSatLeaf JSatLeaf AJSLeaf JSatLeaf
*** Starting controller
*** Starting 1 switches
s1
*** Testing connectivity
*** Running CLI
```
4.4 Tactical TopGen New Routing Modules

4.4.1 Tactical Core Network

The tactical core network module creates topologies representative of the relatively static (as opposed to on-the-move) networks found in Iraq and Afghanistan in the 2000s. The main properties captured are a smaller average end-to-end hop count than the Internet due to the smaller region being transited and the lack of peering, and the core latency distribution with the properties analyzed previously and presented in Table 3.3.

We take a first principles approach to generating the tactical core network. We use the groupings of latency model we derived in Section 3.6 to inform the ratios of different types of links to each other. For example, we sum the mixture ratios of the three components modeling satellite
connections, to find that 7% of all paths should have a satellite delay component. We model satellite delays as integer multiples of satellite hops as opposed to a random number, because there cannot be 1.5 satellite hops. The composition of routers and links in a tactical core network can be decomposed as follows. There are a number of routers which are "close" to each other, typically one or fewer milliseconds away with high bandwidths. These are most commonly at larger bases which have more infrastructure and are often all in the same facility interconnected directly. There are a number of network links that are approximately 10-30 ms in delay with bandwidth ranging from 10-50 Mbps which are line of sight microwave point to point links. Finally, we have some edge connections which are one or more satellite hops away from the core with latencies of 300 ms, 600 ms, or 1200 ms, and bandwidths of 1-4 Mbps.

A pictorial representation is shown in Figure 4.11.

For our tactical core module, we define the following routers:

- Tactical Core - interconnect to other cores, and edges, as well as to other regions’ core routers
- Leaf - interconnect to other edges over short delay links or cores over 10 ms delay links
- Satellite Edge - interconnect to core routers or other satellite edge routers

A preferential attachment scheme is used to determine which nodes connect to others in the network. The satellite edge routers are interconnected to core routers which interconnect to other core or edge routers until the proper ratios of latencies are achieved using the delay components we have derived. The diameter analysis from Section 3.8 introduces constraints such that we tend to grow a path until it has a number of hop counts equal to six, then make it less likely to grow larger than that. The source code for the tactical core generator is found in Listing A.10.
Figure 4.12: Tactical core topology visualization

Figure 4.12 generated by Tactical TopGen is representative of a tactical core network topology such as previously detailed in Figures 3.23 and 3.24.

Listing 4.3 shows parts of the Mininet export of a generated tactical core. Example line 1 shows a core to satellite edge connection with 300 ms of delay and 1 Mbps of bandwidth. Example line 2 shows a core to core connection with 30 Mbps of bandwidth and 10 ms of delay. Example line 3 shows a leaf to leaf connection with a 100 Mbps bandwidth and a delay of 1 ms.

<table>
<thead>
<tr>
<th>Listing 4.3: Mininet Configuration File Example</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>EXAMPLE LINE 1</strong> − link, L(TC Core), 10.0.0.28/30, CS(TC SAT), 10.0.0.29/30, 1000, 300, .01</td>
</tr>
<tr>
<td><strong>EXAMPLE LINE 2</strong> − link, L(TC Core), 10.0.1.5/30, A(TC Core), 10.0.1.6/30, 30000, 10, .01</td>
</tr>
<tr>
<td><strong>EXAMPLE LINE 3</strong> − link, BN(TC Leaf), 10.0.1.99/30, W(TC Leaf), 10.0.1.100/30, 100000, 1, .01</td>
</tr>
</tbody>
</table>
We generated ten networks ranging from 50 to 400 routers and extracted the end-to-end hop count and end-to-end latency data from them for use in assessing the accuracy of modeling the measured tactical network data. Figure 4.13 shows the measured hop counts in Iraq and Afghanistan versus the hop counts in our ten generated topologies and it can be seen that the generated topologies have more short hop networks, but that the overall spread of hop lengths is very comparable.

Figure 4.14 shows the histogram of the measured latencies in Iraq and Afghanistan versus the latencies in our ten generated topologies. It can be seen that there are more low latency paths in the measured network data than in the generated topologies, but that the non-satellite cluster density is similar. There are three generated satellite hop clusters but most different is that the largest delay cluster is at approximately 2 seconds as opposed to the 2.5 seconds as measured in the tactical network. This difference was a deliberate decision to limit the generator to build networks with only one or two satellite bounces and this limited the number of delay components that could be
Figure 4.14: Tactical core latency measured versus generated 

Figure 4.15 shows the CDF of the measured latencies in Iraq and Afghanistan versus the laten-
cies in our ten generated topologies. The data shows clearly the lower number of short delay paths
in the generated dataset, but matches very well after 200 ms.

4.4.2 WIN-T Region

A WIN-T Increment 2 network configuration is typically composed of three networks of different
technologies that tie together different echelons. This results in a capability so that each com-
pany, battalion, and brigade commander has connectivity with every other commander over two
redundant networks, a highly available, long delay satellite connection, and a lower delay, higher
bandwidth (possibly) multi-hop line of sight network. Additionally, each company commander
Figure 4.15: Tactical core latency measured versus generated
has a third independent network for communicating with his own platoon leaders.

Beginning at the company level and working our way up the echelons, we assume there are four platoons per company, and that each officer has a sergeant second in command, and that each of these leaders is in a separate vehicle, resulting in a total of ten vehicles with leaders communicating within each company. This company level network utilizes the Soldier Radio Waveform (SRW) [210] that provides an ad hoc wireless mesh. The SRW vehicles share approximately 100 kbps [211] with each other, with delays varying depending on how many hops are being relayed through. Each company has their own independent SRW network that is not shared with other companies. The two company commanders’ vehicles also have connectivity to each of the other two networks, a battalion command network, and a brigade command network.

The battalion network is based on the Wideband Networking Waveform (WNW) [210] or Advanced Networking Wideband Waveform (ANW2) waveform [212] and is an ad hoc wireless mesh with a shared bandwidth of between 1-4 Mbps [213], and with delays as high as 250 ms one way [150] through multiple layer two hops. The battalion network consists of the two commanders vehicles from each company in the battalion, plus the battalion commanders vehicle, and two infrastructure nodes, the battalion Point of Presence (POP) and the battalion Tactical Communications Node (TCN). These same vehicles all have access to the Brigade command network, which is a higher availability satellite connection, but with longer delays. In addition to the vehicles from the other battalions in the brigade, there is an additional brigade level TCN and a core router for connections to other regions.

Our topology generator models this collection of networks by defining five router types:

- **Win-T Truck** - Platoon Leader and Platoon Sergeant’s vehicles only on SRW
- **Command Truck** - Company Commander and Company XO vehicles on all three nets
- **Battalion TCN** - Used for the Battalion Commander, Battalion POP, and Battalion TCN, connected to battalion and brigade nets
- **Brigade TCN** - Satellite hub for the brigade network
• Brigade Core - Used to interconnect this WIN-T region with other networks

The router types are interconnected in the following fashion to accurately recreate the WIN-T Increment 2 network. The Win-T Truck and Command Truck routers for each company are interconnected with a full mesh, link bandwidth of 100 kbps, and a delay picked from a Gaussian with $\mu = 100$ ms, and a $\sigma = 40$ ms. The Command Truck routers from each company (4 companies per battalion by default) plus the Battalion TCN routers (3 per battalion) are interconnected by a full mesh to replicate the battalion command network. These links have 1 Mbps of bandwidth and a delay picked from a Gaussian with $\mu = 200$ ms and a $\sigma = 80$ ms. All of the battalion network routers are also connected to the brigade network (along with the by default 3 other battalions per brigade). The brigade network consists of a hub and spoke topology, with each Command Truck (32 total for 4 companies in each of 4 battalions) having a 64 kbps link with a 300 ms delay back to the Brigade TCN, and the Battalion TCN nodes (12 total for 4 battalions) having 1 Mbps links with 300ms delay back to the Brigade TCN. The Brigade Core router is connected to the Brigade TCN with a 1 ms connection with 100 Mbps of bandwidth. All of these parameters including number of units per echelon and the speeds and delays of networks are parameterized and can be modified either for non-standard deployments, new network technologies that are introduced, or if funding decisions change the expected order of battle and allocation of equipment.

Figure 4.16 generated by Tactical TopGen is representative of a single company’s connection to all three command networks in a typical WIN-T Increment 2 deployment such as previously detailed in Figure 3.19. In this network, the standard deviation was set to zero for illustration to more clearly delineate the three networks. The bottom mesh is the company command net for a single company with a latency of 100 ms. The middle mesh is between the two command trucks of that single company, and the battalion assets, with a latency of 200 ms. Finally at the top is the brigade level hub, with the core attached with a low latency high bandwidth connection, and a 300 ms latency to each of the spokes. The source code for the WIN-T topology generator is in Listing A.9.

Listing 4.4 shows elements of the WIN-T module configuration. Example line 1 shows the
Brigade TCN connected to the Brigade core router with a 100 Mbps (100000 kbps) link with 1 ms delay. Example line 2 shows a company command truck connected to the brigade TCN via the 64 kbps 300 ms delay satellite link. Example line 3 shows the company command truck connected to the battalion TCN with 1 Mbps (1000 kbps) 200 ms delay WNW network. Example line 4 shows the company command truck connected to a platoon leader with a 100 kbps, 100 ms delay SRW network.

Listing 4.4: WIN-T Mininet Configuration

EXAMPLE LINE 1 – link, N(Brigade TCN)/Brig1, 10.0.0.1/30,
O(Brigade Core)/Brig1, 10.0.0.2/30, 100000, 1, .01
EXAMPLE LINE 2 – link, J(Command Truck)/Brig1, 10.0.0.29/30,
N(Brigade TCN)/Brig1, 10.0.0.30/30, 64, 300, .01
EXAMPLE LINE 3 – link, J(Command Truck)/Brig1, 10.0.1.33/26,
M(Battalion TCN)/Brig1, 10.0.1.34/26, 1000, 200, .01
EXAMPLE LINE 4 – link, H(WIN-T Truck)/Brig1, 10.0.2.65/27,
J(Command Truck)/Brig1, 10.0.2.66/27, 100, 100, .01
4.4.3 Satellite Region

The satellite region module produces a topology component that matches Figure 3.18. This network topology is representative of a centralized Network Operations Center [182] housing a core router that is the forwarding element both between individual nodes on a low bandwidth satellite L Band connection and between these satellite connected nodes and other regions of the tactical network. The region is designed to allow you to specify how many brigades worth of vehicles are to be simulated, and for each brigade, 500 end nodes are added to the core router as edge nodes in the network. The latency for this connection defaults to a typical satellite hop one way delay of 300 ms, but can be modified as desired. The bandwidth for these connections is set to the very small amount of 5 kbps, reflecting the limited capacity shared equally amongst these vehicles. There are two types of routers defined in a Satellite region, a Sat Core and a Sat Leaf. All Sat Leaf routers are connected to the Sat Core, and the Sat Core is allowed to interconnect to other regions’ core routers. The source code to the satellite generator is found in Listing A.8. Figure 4.17 shows a visualization of a fifty node subset of a Satellite region generated by Tactical TopGen.

Listing 4.5 shows elements of the satellite module configuration. Example line 1 shows a instantiation of one of the leaf nodes with the very large flat /16 address space. Example line 2 shows the link between a leaf and the core router, with link properties of 5 kbps, and 300 ms of delay.
Listing 4.5: Satellite Region Mininet Configuration

EXAMPLE LINE 1 − node, AN(Sat Leaf)/Sat1, 10.0.0.153/16
EXAMPLE LINE 2 − link, B(Sat Leaf)/Sat1, 10.0.0.1/16, A(Sat Core)/Sat1, 10.0.0.2/16, 5, 300, .01

4.4.4 Tactical Topology Generator Conclusion

We developed an extensible tactical topology generator with three key modules that can be combined to model a wide variety of tactical networks. The core region produces networks that look like the relatively static deployments of facilities such as occurred in Iraq in 2006 - 2010 and Afghanistan in 2007 - 2013. While these networks evolved over those periods, for example as the number of outposts increased or decreased, the time scale of these changes was relatively slow. The Satellite region connector produces a network which matches the majority of low bandwidth vehicles that are networked while mobile. The WIN-T region generator produces a network representative of what is found for a brigade deployed with WIN-T Increment 2 capabilities in all of the commanders’ vehicles down to the company level. This generator reflects the capabilities of US Army brigades deployed in 2014 or later, with their significantly improved on-the-move tactical network capabilities. The three modules can be combined in many ways to produce unique networks tailored to a number of different possible deployment scenarios.
Chapter 5

Server Selection

In this chapter, we outline the data flow of a tactical overlay, and discuss distributed system architectures. We review the state of the art server selection algorithms on the Internet, and consider their effectiveness in a tactical network. We finally present and evaluate an algorithm that reduces the overhead of the server selection algorithm on the network while effectively routing users to a good server.

5.1 TIGR Tactical Overlay Data Patterns

In the TIGR tactical overlay we have two main patterns of data flow, the background replication of tactical content between servers, and the client to and from server transmission of each particular user session.

On average over the course of a day, 15 MB of data flows to each TIGR server in the overlay from other parts of the overlay. This is the new content created elsewhere in the overlay that is then synchronized with all servers. Servers can be offline at some locations for some number of days (for example an extended power outage), and when they come back online, they have a larger data set to synchronize. Figure 5.1 is a histogram for all servers in the overlay showing the daily bytes received.

A typical user session, i.e., bytes flowing between a client and one server in the overlay, is on
Figure 5.1: TIGR server to server data flow

Figure 5.2: TIGR client imagery resolution requests
the order of 5 MB, which is made up of a mix of JavaScript code, pictures, and high resolution (sub meter, as illustrated in Figure 5.2) satellite and aerial imagery.

When the overlay in Afghanistan was 25 servers, there were 425 sessions per day on average, which is over 2 GB per day between browsers and servers. In TIGR today, users bookmark the server that they hope is the "best" server for them. A data set was collected (Study 5) of the time it took for a round trip AJAX call over TCP from each user’s browser to all servers in the overlay. Figure 5.3 shows the client to server RTT TCP delay, and Figure 5.4 is the same data with a log scale so we can see the data in the tail better. Approximately 6.25% of all user sessions have over two seconds round trip time delay in their user session. This not only results in a poor user experience, but also wastes bandwidth by sending a user session over backbone links unnecessarily. In a one week period this is almost 1 GB of extra data traversing the WAN.

5.2 Common Distributed Architectures

There are a number of typical architectures for large distributed systems [214], and in this section we compare and contrast them with the TIGR tactical overlay, focusing on the attributes related to how these systems scale and how they allocate resources. We also discuss the administrative models, for example whether there is a single authority controlling the resources versus a consortium,
and the performance objectives.

5.2.1 Distributed Databases

Distributed databases [215] are collections of servers acting as independent nodes to perform queries of data sets. Each local node responds to global queries with a consistent interface, so that the distribution is typically transparent to the user of the service. The system spreads database resources for reliability and performance to speed query processing [216]. A distributed database is typically managed by a single responsible authority.

5.2.2 Data Grids

A data grid [217] is a set of geographically distributed storage and computing resources connected by high speed networks. They most commonly are used for scientific applications [218]. They are administered by a virtual organization made by bringing together storage, database, and computational resources from different real organizations, often with shared interests such as different teams analyzing the same very large data set. The overall focus of providing the data grid is on the performance gain from pooling of resources for very large tasks.

5.2.3 Content Distribution Networks

Content Distribution Networks (CDNs) [214] are a collection of resources that provide content replication so that there are cached copies of files located close to end users. They are managed by a single authority, but there are many independent customers of the CDN service. The performance goals are focused on meeting customer performance requirements [128], rather than strictly finding the best resource for the end user. This allows for engineering tradeoffs to minimize cost while meeting customer requirements.
5.2.4 Peer-to-Peer

Peer-to-peer (P2P) networks [219] allow for the sharing of resources, most commonly files, without a central authority. Nodes participating in a P2P network collaborate to perform searches, find other peers, and retrieve content [131]. Peer-to-peer networks have much higher rates of nodes joining and leaving the system as compared to other distributed systems, and the main goal of the network is to share resources to most efficiently find and download files.

5.2.5 TIGR Distribution Comparison

The TIGR overlay is administratively a single authority like CDN and distributed database systems. Individual node performance is sufficient for the tasks assigned, so there is not a need for splitting queries or processing amongst nodes for improved performance. TIGR differs from a CDN in that there are no other customers to balance across finite resources. TIGR does try to get the best performance for the end user, similar to P2P systems, but TIGR does not have the churn of resources like P2P systems. Figure 5.5 shows graphically the comparison between TIGR and other distributed systems with respect to node churn, administrative authority, resource limitations, and performance objectives.
5.3 State-of-the-Art Server Selection

Server selection is the task of assigning a particular server to fulfill a particular user request. For purposes of this work, we will focus on global server selection, as opposed to intra-site selection. Figure 5.6 shows the concept of global versus intra-site selection. If resources are distributed to multiple locations, global server selection directs the user to the best site to fulfill his request. Within a site, it is common to have multiple local servers each of which can fulfill a request. Intra-site techniques often take into account the state of a client’s session and near real-time loading of the servers to pick a particular local server, and these approaches are equally applicable to server farms located in a tactical network as to the Internet.

Global server selection has a number of different forms that work in certain environments and we focus on two main components of the server selection process, the request routing algorithm, and the request routing mechanism. The request routing algorithm is the process by which a server is selected for a particular request. There are a number of different approaches to this, including:

- Non-adaptive – the simplest approaches utilize techniques like round robin assignment to assign each incoming server request.
- Adaptive – there are a number of systems that try to approximate latency to the client, and they differ in how they measure and distribute those results.
- Network coordinate systems - coordinate systems are developed that approximate the latency
between network nodes. Then distances can be calculated given two nodes’ positions in the coordinate system. There are simulation based approaches [220] as well as landmark based approaches [221].

The request routing mechanism is the method by which a user is sent the choice of the server selection algorithm. There are a number of common methods used to do this, including:

- Round robin DNS based – this basic approach returns a list of addresses, and the common implementation [222] is that the resolver will pick from that list in a round robin fashion.

- Global DNS based - this approach alters the DNS response based on the server selection algorithm.

- Anycasting – this network address space approach [223] requires dedicated IP address space which is announced equally from every site, so that routing policies will direct the requests to the closet site.

- URL rewriting – this approach is most common for partial site resources, for example videos on a site. The HTTP server will generate new URLs in response to the selection algorithm when transmitting the page.

- HTTP redirection – this approach [224] uses a HTTP 307 response code to tell a browser that the URL is at a different location. This only works for web traffic, and has the disadvantage of making the request routing a multiple request transaction.

### 5.3.1 Content Distribution Network Server Selection

Google’s state-of-the-art algorithm for selecting servers [225] begins with redirecting a small percentage of all web traffic to alternate servers in order to collect data. Google maintains a database with a bin for each BGP AS prefix advertised on the Internet, which roughly corresponds to the assumption that every user on a particular ISP has the same connectivity to Google’s infrastructure. TCP headers are recorded at all servers, and the round trip time from a particular network to
that server is recorded. As new data is collected the database is updated with the best server for a 
particular AS. This information is then used by Google’s DNS servers to decide which IP address 
to respond with on a name lookup.

Several aspects of this approach do not work well on tactical networks. The first is that because 
DNS is not consistently used, we need a different routing mechanism. Second, the data bin size of 
an AS prefix does not work, as not only are there not unique ISPs, but as shown, very small adjacent 
network blocks can have radically different access links in tactical networks. Third, redirecting a 
tactical user session just to collect data wastes critical bandwidth on the tactical backbones. Fourth, 
it is much less likely that the DNS server doing lookups for a particular client on a tactical network 
has a useful location relationship to that client on tactical networks versus the Internet. Finally, 
the rate of TIGR sessions is several orders of magnitude lower than Google requests, so the rate at 
which you can build accurate tables for all address blocks to all servers is dramatically slower.

5.3.2 Peer-to-Peer Network Server Selection

A common algorithm used in BitTorrent P2P clients is the Vivaldi [226] network coordinate sys-
tem. Each client is assigned a number of neighbors when it connects, where some are estimated 
to be close to the new client and some estimated to be far away. The client probes each of these 
networks and measures the RTT to them. The results are shared with a distributed simulation of a 
neighboring model.

The mass-spring model correlates to minimizing the mean squared error of the delay estimates. 
If we have $L_{ij}$ be the actual RTT between nodes $i$ and $j$, and $x_i$ is the coordinate assumed to be 
node $i$, then the errors in the coordinates can be expressed as

$$E = \sum_i \sum_j (L_{ij} - |x_i - x_j|)^2$$  \hspace{1cm} (5.1)

Hooke’s Law states that the force that a spring between nodes $i$ and $j$ exerts on node $i$ is
\[ F_{ij} = (L_{ij} - |x_i - x_j|) \times u(x_i - x_j) \] (5.2)

which is the displacement of the spring from rest multiplied by the direction of the force on \( i \). It can be seen that when the force on the nodes is minimized it is analogous to minimizing the RTT error. At each step of the algorithm, nodes are moved a small distance in the coordinate space in the direction of \( F_i \) and then we recompute the forces. When the model converges it has updated coordinates for all of the nodes. Once the coordinates for a new node have converged, whenever it needs to talk to any other node, it can estimate the distance using a distance function and pick “close” nodes from which to download.

There are aspects of the Vivaldi algorithm that are not ideal for tactical networks. Typically the spring tension model requires probing 8 - 32 neighbors for accuracy [227], which is expensive if done every time a TIGR user connects to the overlay. Because there is not a single authority managing the P2P overlay, there is no consistent data store, so probing must be done at every connection. Additionally, in the TIGR overlay case, the end users do not need to be able to estimate their distance to all other TIGR end users, just to the servers.

### 5.4 A Tactical Server Selection Algorithm

We utilize concepts from both the CDN and the P2P approaches and the constraints observed in the tactical network to develop a server selection algorithm tailored to tactical networks. We incorporate the following main features:

- Collaboration between servers to share measurement data in a historical archive of probes that is eventually consistent throughout the overlay
- A spring tension model to provide a mechanism to estimate delays between users and servers in the absence of actual measurements
- A *good enough* concept to limit probing when a threshold has been crossed
- Random probing when the best predicted server is known to not be good enough from historical data

- HTTP redirection to route users to new servers

The algorithm incorporates a spring tension model to model a coordinate space for estimating delays between servers and clients. When a server is added to the overlay, it performs an ICMP probe of all of the other servers, and the results of these probes are shared throughout the overlay. Each server then updates its spring tension model with the new data. These servers in general have low churn as compared to a peer-to-peer network, and this data is relatively stable. As servers change addresses, are added or dropped from the overlay, and periodically, the measurements are repeated and kept synchronized throughout the overlay. This full mesh of server to server latencies forms the foundation of the coordinate system.

When a client connects to a server, it collects a RTT sample for that server, and shares it with the server. The server looks into its historical records to see if it has data about the best server for the /25 subnet of which the client is a member, as well as updates the record with the current probe from that /25 to itself. The new data is shared throughout the overlay in a manner so that the overlay’s historical records are eventually consistent amongst all servers.

One of the concepts incorporated in the algorithm is that of a good enough parameter, which specifies that if a delay is less than the good enough value then we do not perform additional probing to look for a better server. If there are no records for the client’s subnet to a server that is good enough, the algorithm calculates the shortest distance between the client and all servers using the coordinate system from the spring tension model, and selects the closest server. If there is no historical information available about the the delay from this subnet to this server, the client is directed to probe the server. If the best calculated server has historical information suggesting that it is not good enough, we assume the spring model is not sufficiently accurate, and we randomly pick a server that does not have historical data for the client’s subnet and probe it so that the algorithm explores new parts of the network. As each new probe occurs, the spring model and historical database is updated.
Listing 5.4 shows a pseudo-code representation of the server selection algorithm.

```java
while (probes) {
    if (this probe is good enough) {
        leave user on this server
    } else {
        if (historical has good enough) {
            redirect to best good enough server
        } else if (calculatedBestFromModel is not in historical) {
            generate probe to calculatedBestFromModel server
        } else if there are known servers not in historical {
            generate probe to random server not in historical
        } else {
            redirect to best historical server
        }
    }
    store this probe in historical
    update simulation to steady state
}
```

The algorithm is currently implemented in JavaScript and incorporates a visualization of the system state during the simulation. Listing A.2 contains the source code for the simulator.

An example of the kinds of improvement possible as a result of the server selection algorithm can be seen in Figure 5.7. In this example, 200 host connections were simulated, of which 65 had zero improvement because they were good enough already, 17 were able to realize between 20% to 80% of the best possible server latency, and 118 were able to be directed to the very best server
Figure 5.7: Realized improvement example

The iterative nature of the algorithm will find a good enough server if one exists, by continuing to probe until one is found, and then from that point forward use its historical information to immediately redirect the host to the good enough server. Figure 5.8 shows the mechanics of the algorithm as new user sessions connect starting from an empty historical database. At first there are some sessions that are good enough and others that cause probes to be generated to learn more about the network. As information about the network is developed and stored, the number of hosts that are redirected to a server via historical information starts increasing. After the first 25 sessions, enough information is gathered in this case to always know how to get to a good enough server, and no further probes are generated.
Figure 5.8: Server selection mechanics
5.5 Tactical Server Selection Evaluation

5.5.1 Evaluation Methodology

We utilized the tactical topology generator to generate a number of representative tactical core networks. One of the output files that is created contains the sum of all delays on the path with the least number of hops between every two nodes in the network. If two paths have identical hop counts, the one with the smallest total end-to-end latency is selected. The code in Listing A.7 parses this information, and then generates data for use in evaluating the load balancing algorithm. The first data structures created are a randomly selected 10% of the nodes which are declared to be servers. We next randomly select 20% of the nodes and declare them to be users (the two groups can overlap). We output a JavaScript array file containing the servers, and their end-to-end delay to every other server. This is the steady state set of relationships with which the server selection algorithm starts. The server selection simulation reads in this server matrix, and converges to a steady state system with precise locations in the delay space for each server.

For a reference point in the assessment, for each user the tool looks at their best paths to every server, and then finds the server with the least delay for that particular server. This set of data is output as a JavaScript associative list that allows for easy lookup of best server and latency by user, and this information is utilized in the evaluation statistics of the server selection simulation to see how close to the absolute best server the simulation was able to come.

The simulation next needs a stream of user connections to servers, so the tool randomly outputs entries with their latencies. For the course of a simulation, we run until every user has connected to every server once, then compare the result of the server selection with the best possible server for that user. Finally, the simulation needs to know valid delays to use when it chooses to redirect a user to a server for which the selection algorithm does not yet know the latency. The data set for this is a JavaScript associative list composed of more lists with every possible user to server path.

All of this data is read into the simulation, and the server selection algorithm performs the initial server address space simulation and reaches steady state. Then individual user sessions are
read in, and the selection algorithm decides if they are "good enough" to stay, if there are known better ones to which to redirect, or if we need to try new servers with probes until a good enough server is found. This user session is tracked, and metrics are recorded about whether it was initially good enough (the selection algorithm did no work), if there was historical information that got to the good enough state, (i.e., the collaboration of probe history was the reason for gain), or how many random redirects were necessary before the session was "good enough". It also compares the delay of the good enough choice to the best possible delay for that user.

5.5.2 Implementation Effectiveness

We utilized the topologies generated as part of the validation in Section 4.4.1 to generate input files for the server selection simulation.

Figure 5.9 shows ten servers selected from a one hundred node tactical core topology loaded into the simulation at an initial steady state, and Figure 5.10 shows the simulation complete after twenty user nodes have been added and each user has connected to all ten servers. Figure 5.10 is a graphical representation of where the user nodes are located in the delay space compared to the servers.
In this simulation, there were two hundred simulated user sessions where a user connected to a random server and metrics were collected including the initial delay at the start of the session, the selected server’s delay, the best possible server’s delay, and how many redirect probes were generated.

Figure 5.11 shows the average improvement in ms in the latency given by using our selected server, and the improvement of the best possible server. The average difference is 6-8 ms in these 10 simulation runs, composed of two different generated networks with 50, 100, 200, 300, and 400 nodes. The variation seen is largely due to the presence of a satellite connected user in a particular simulation. If there are one or more satellite connected users in a particular simulation we have a larger average latency in general therefore more room for improvement. The differences between the algorithm’s selection and the perfect selection are a function of the "good enough" value used in the simulation, and this is explored in more detail later in this section.

Figure 5.12 shows the absolute latencies before and after the simulation. The red bars are the number of milliseconds of the initial probe from user to server. The blue bars represent the latency between the server that the algorithm selected and the user.

Figure 5.13 shows that as we vary the "good enough" server selection parameter, we can affect
Figure 5.11: Average improvement versus best possible improvements for ten simulations
Figure 5.12: Initial latency and selected server latency per user session
Figure 5.13: Comparison of different "good enough" values and impact on system redirects

how often the algorithm probes the network. With a relatively loose bound of 100 ms being close
enough for a server, in 200 user sessions, 187 of them were acceptable with no additional probes,
and there were only 24 probes generated to find good enough for the remainder. Tightening the
"good enough" parameter to 50 ms, and then 30 ms shows a decrease in acceptable initial probes,
and a corresponding increase in the generated network probes.

Figure 5.14 shows a comparison between the tactical server selection algorithm and P2P al-
gorithms that have to probe with every connection because they do not have a historical archive.
Figure 5.14: Comparison of tactical algorithm versus P2P probing
5.6 Tactical Server Selection Conclusion

We developed a flexible server selection algorithm that utilizes a spring tension model to inform the selection process of candidate best servers rather than using purely random sampling. The incorporation of a "good enough" parameter allows the algorithm to balance getting the client to the perfect server with reduced network load, and quicker user session initiation. We utilized the tactical topology generator to create data sets for evaluating the performance of the server selection algorithm, and found that for representative tactical networks, the algorithm quickly learned a network’s delay properties, and directed a client to a server that was often close to the perfect choice, while minimizing unnecessary probing and taking advantage of collaboration with other servers to utilize their probe data.
Chapter 6

Contributions

6.1 Contributions & Publications

This dissertation has documented in detailed fashion the unique environment tactical networks operate within, and has developed collaborative probabilistic solutions to an important network management task – server selection, as well as provided tools so that other researchers can explore this problem space without access to these unique networks. In particular we provided:

- Tactical network characterization - quantitatively and qualitatively captured differences in networking fundamentals between tactical networks and the Internet.

- Tactical network extensions to generators for topology research - enabling researchers without direct access to tactical networks to explore how new protocols and systems work in tactical environments.

- Tactical network server selection algorithm - incorporating fundamental limitations of tactical networks into a probabilistic load balancing algorithm to reduce bandwidth and improve the user experience.

A list of publications produced during the course of this research includes:
6.2 Other Lessons Learned

- Configuration management is a fundamental problem with many military systems and is exacerbated by the need to reduce support staff. Tactical systems need to be remotely configurable and be able to be integrated into configuration management systems to reduce human
error, and simplify deployments. The transition to virtual machines is a helpful start along this path.

- The Department of Defense procurement process continues to have challenges [228]. A National Research Council report [229] highlighted the development of TIGR as something that "should be evaluated in depth for lessons learned that can be deployed across the DOD IT system acquisition community".

- Integration of networking management and capabilities is critical for a tactical networking application. Many large monolithic architectures [228] [230] compartmentalize networking functionality into blocks such as “QoS” or “management”, and then applications and services just take a hands off abstracted view of the real network. There are often missed opportunities or requirements when applications are developed abstracted away from the real operating environment. In order to overcome these sorts of problems, more black boxes are brought in such as TCP accelerators. Instead, understanding the behavior of a system with respect to its network usage, and then understanding the capabilities of the particular transports being used can be very valuable. TIGR incorporated networking concepts directly into the system, i.e., the TIGR overlay has a queuing system and a traffic shaper overlayed onto the networks below. If TIGR can take advantage of native networking capabilities it does, but where they are not available it provides its own. TIGR’s ability to manage its own impact on the network, as well as its ability to monitor its own usage and availability were crucial in getting it adopted for use in tactical networks.

### 6.3 Future Work

This research has laid a foundation for the future exploration of a number of areas including:

- Dynamic aspects of load balancing and cloud replication - the detailed delay models in this work were primarily collected from the relatively static tactical core networks. While
provisions have been made to allow discovery of new servers and better network links, and
the making stale of any old data, more work to assess it in a truly mobile force-on-force
environment such as the twice yearly NIE exercise is needed to assess its real world viability.

- While our topology models capture delay and bandwidth, they do not currently model loss
  in the tactical network, and its addition would be valuable for network modeling.

- The server selection algorithm currently only utilizes delay between the end user and server,
  and could become more effective by incorporating loss and bandwidth information as well.

- The server selection algorithm could incorporate more detailed models of the tactical topol-
  ogy other than the end-to-end delay probes and assess if this more detailed model impacts
  the server selection performance positively. In order for this to be possible, an effective
  anti-aliasing approach that did not impact the tactical network would need to be developed.

- Software defined networking is an enabler for deployment and management of rapidly evolv-
  ing tactical networks. The ability to roll out new capabilities from a centralized, configura-
  tion managed facility will radically change network deployments in the tactical environment.
  Porting the server selection algorithm to a SDN framework, where it can take advantage of
  the observed network conditions and direct users to the best server automatically is an excit-
  ing area of future effort.

6.4 Conclusions

We have developed and presented original ideas and insights into how tactical networks differ
from the Internet, and the impact of those differences. We provided a solid understanding of the
tactical network environment and developed tools that enable future research by the community
at large, most importantly those without direct access to tactical networks. We demonstrated how
this understanding can inform a design for a collaborative networked service, reducing bandwidth
usage and improving the user experience. This work provides a foundation for continued research into designing, managing and deploying services in a tactical environment.
References


130


[224] V. Cardellini, M. Colajanni, and S. Y. Philip, “Dynamic load balancing on web-server sys-

and J. Gao, “Moving beyond end-to-end path information to optimize cdn performance,” in
Proceedings of the 9th ACM SIGCOMM conference on Internet measurement conference.

system,” in *ACM SIGCOMM Computer Communication Review*, vol. 34, no. 4. ACM,


[228] A. Feickert, *Army’s Future Combat System (FCS); Background and Issues for Congress*.

[229] C. on Improving Processes, P. for the Acquisition, and T. of Information Technologies in
the Department of Defense; National Research Council, *Achieving Effective Acquisition of
Information Technology in the Department of Defense*. The National Academies Press,

Appendix A

Source Code for Tools and Utilities

Listing A.1: JS Visualization

```javascript
var HostByLabel = function (label, graph) {
    for (var j = 0; j < graph.nodes.length; j++) {
        if (graph.nodes[j].data.label === label) {
            return graph.nodes[j];
        }
    }
    return null;
}

var HostAlreadyExists = function (name, graph) {
    for (var j = 0; j < graph.nodes.length; j++) {
        if (graph.nodes[j].data.label === name) {
            // console.log('found dup');
            return true;
        }
    }
    return false;
}

var EdgeAlreadyExists = function (source, target, graph) {
    for (var j = 0; j < graph.edges.length; j++) {
        if (graph.edges[j].source.id === source.id && graph.edges[j].target.id === target.id) {
            console.log('found matching edge');
            return true;
        }
        if (graph.edges[j].source.id === target.id && graph.edges[j].target.id === source.id) {
            console.log('found opposite edge');
            return true;
        }
    }
    return false;
}
```

133
```javascript
var BuildEdge = function (firsthop, secondhop, latency, graph) {

    // check and see if this edge exists already.
    // compare two hops, edge label, and directionality
    var edgelength = 1;

    if (latency !== 1) {
        edgelength = Math.log(latency) / Math.log(2);
    }

    if (!EdgeAlreadyExists(HostByLabel(firsthop, graph), HostByLabel(secondhop, graph), graph)) {
        graph.newEdge(HostByLabel(firsthop, graph), HostByLabel(secondhop, graph), { label: latency, length: edgelength });
    }

    for (var k = 0; k < traces.length; k++) {
        for (var i = 0; i < traces[k].length; i++) {
            if (i % 2 === 0) {
                // console.log(traces[k][i]);
                if (HostAlreadyExists(traces[k][i], graph) !== true) {
                    console.log('adding node '+ traces[k][i]);
                    graph newNode({ label: traces[k][i] });
                }
            }
            if (i !== 0) {
                BuildEdge(traces[k][(i - 2)], traces[k][i], traces[k][(i - 1)], graph);
            }
        }
    }
}
```
Listing A.2: Server Selection

```javascript
var graph = new Graph();

graph.locations = new Object();
graph.historicalProbes = new Object();
graph.serverNames = new Array();
graph.clientNames = new Array();
graph.referencedLatency = new Object();
graph.lastHostAdded = "none";
graph.currentLatencyScale = 0;
graph.goodEnough = 50;
graph.goodEnoughCounter = 0;
graph.historicalRedirectCounter = 0;
graph.generatedNewProbeCounter = 0;
graph.newProbeNeeded = "";
graph.sessionNewProbeCounter = 0;
graph.sessionHistoricalCounter = 0;
graph.sessionStartingProbe = 0;
graph.sessionInProgress = false;
var delayscale = true;

for (var k = 0; k < traces.length; k++) {
    for (var i = 0; i < traces[k].length; i++) {
        if (i % 2 === 0) {
            //console.log(traces[k][i]);
            if (HostAlreadyExists(traces[k][i], graph) !== true) {
                if (i === 0) {
                    //console.log('adding first node ' + traces[k][i]);
                    graph.newNode({ label: traces[k][i], type: 'server' });
                    graph.serverNames.push(traces[k][i]);
                } else if (i === traces[k].length - 1) {
                    //console.log('adding last ' + traces[k][i]);
                    graph.newNode({ label: traces[k][i], type: 'server' });
                    graph.serverNames.push(traces[k][i]);
                } else {
                    //console.log('adding node ' + traces[k][i]);
                    graph.newNode({ label: traces[k][i], type: 'router' });
                }
            }
        }
    }
}
```
if (i !== 0) {
    if (k === 0) {
        graph.referencelatency.s1 = traces[k][(i-2)];
        graph.referencelatency.s2 = traces[k][i];
        graph.referencelatency.latency = traces[k][(i-1)];
    }
    if (EdgeAlreadyExists(HostByLabel(traces[k][(i-2)], graph), HostByLabel(traces[k][i], graph), graph, false) != true) {
        BuildEdge(traces[k][(i-2)], traces[k][i], traces[k][(i-1)], graph, delayscale, false);
    }
}
}

jQuery(function() {
    var springy = jQuery('#springydemo').springy({
        graph: graph,
        nodeSelected: function(node) {
            console.log('Node selected: ' + JSON.stringify(node.data));
        }
    });
});
</script>
<canvas id="springydemo" width="1240" height="640" />
</body>
</html>

/**
 * Returns a random integer between min and max
 * Using Math.random() will give you a non-uniform distribution!
 */
function getRandomInt(min, max) {
    return Math.floor(Math.random() * (max - min + 1)) + min;
}

var HostByLabel = function(label, graph) {
    for (var j = 0; j < graph.nodes.length; j++) {
        if (graph.nodes[j].data.label === label)
            return graph.nodes[j];
    }
    return null;
}

var CalculatePixelsPerMS = function(graph) {
    graph.referencelatency
    var distanceSquared = Math.pow(graph.locations[graph.referencelatency.s1].x - graph.locations[graph.referencelatency.s2].x, 2) + Math.pow(graph.locations[graph.referencelatency.s1].y - graph.locations[graph.referencelatency.s2].y, 2);
    var distance = Math.sqrt(distanceSquared);
    return distance / graph.referencelatency.latency;
}
var BestHistorySample = function (subnet, graph) {

var bestx, bestlatency = 10000;
var result = new Object();

//console.log('samples for subnet '+ subnet);
for (var x in graph.historicalProbes[subnet]) {
    //console.log(' server '+ x + ' latency '+graph.historicalProbes[subnet][x]);
    if (graph.historicalProbes[subnet][x] < bestlatency) {
        bestlatency = graph.historicalProbes[subnet][x];
        bestx = x;
    }
}
//console.log(' best historical for '+subnet+' is '+bestx+ ' with latency '+graph.historicalProbes[subnet][bestx]);
result.serverName = bestx;
result.latency = graph.historicalProbes[subnet][bestx];
return result;
}

var BestPossible = function (subnet) {

var bestx, bestlatency = 10000;
var result = new Object();
for (var i in bestserver[subnet]) {
    result.serverName = i;
    result.latency = bestserver[subnet][i];
}
//console.log('bestservername is '+result.serverName);
return result;
}

var PrintHistorySamples = function (subnet, graph) {

console.log('samples for subnet '+ subnet);
for (var x in graph.historicalProbes[subnet]) {
    console.log(' server '+ x + ' latency '+graph.historicalProbes[subnet][x]);
}
}

var PrintBestServer = function (subnet, graph) {

if (typeof(graph.locations[subnet]) === "undefined") {
    return;
}
var bestj, bestdistance = 10000;
for (var j = 0; j < graph.servernames.length; j++) {
    var distanceSquared = Math.pow(graph.locations[subnet].x - graph.locations[graph.servernames[j]].x, 2) +
    Math.pow(graph.locations[subnet].y - graph.locations[graph.servernames[j]].y, 2);
    var distance = Math.sqrt(distanceSquared);
    //console.log(subnet + ' to ' + graph.servernames[j] + ' is ' +distance);
    if (distance < bestdistance) {
        bestj = j;
        bestdistance = distance;
    }
}
}
//console.log(subnet + "'s best server is ' + graph.servernames[bestj] + ' with estimated latency of "+(bestdistance/graph.currentlatencyScale).toFixed(2) + ' ms ('+bestdistance.toFixed(0) + ' pixels');
return graph.servernames[bestj];
}

var PrintDistances = function (graph) {
var keys = [];
for (var key in graph.locations) {
if (graph.locations.hasOwnProperty(key)) {
keys.push(key);
}
}
for (var j = 0; j < keys.length; j++) {
for (var k = j+1; k < keys.length; k++) {
// calculate the distance using the Pythagorean Theorem (a^2 + b^2 = c^2)
var distanceSquared = Math.pow(graph.locations[keys[j]].x - graph.locations[keys[k]].x, 2) +
Math.pow(graph.locations[keys[j]].y - graph.locations[keys[k]].y, 2);
var distance = Math.sqrt(distanceSquared);
console.log(keys[j] + ' to ' + keys[k] + ' is estimated '+ (distance/graph.currentlatencyScale).toFixed(2) + ' ms ('+distance.toFixed(0) + ' pixels');
}
}
}

var HostAlreadyExists = function (name, graph) {
for (var j = 0; j < graph.nodes.length; j++) {
if (graph.nodes[j].data.label === name ){
//console.log('found dup ');
return true;
}
return false;
}

var EdgeAlreadyExists = function (source, target, graph, allowopposite) {
for (var j = 0; j < graph.edges.length; j++) {
if (graph.edges[j].source.id === source.id && graph.edges[j].target.id === target.id){
//console.log('found matching edge ');
return true;
}
if (!allowopposite && graph.edges[j].source.id === target.id && graph.edges[j].target.id === source.id){
//console.log('found opposite edge ');
return true;
}
}
return false;
}

var IPtoSlash25 = function (IP) {
var nAddr = new Array();
var nMask = new Array(255,255,255,128);
var a = IP.split('.'),
nAddr[0] = a[0];
nAddr[1] = a[1];
nAddr[2] = parseInt(a[2]);
nAddr[3] = parseInt(a[3]);

}

var AddHost = function (sample, graph) {
  // steps to do.
  // check whether this probe is good enough
  // if good enough, store this probe in historical
  // need to come up with a good enough distance measure - compute a scale based on longest distance, look up delay that
  // generated it, then normalize saying 100ms is good enough.
  // if not good enough, see who the best server is for this host via historical
  // if no historical good enough, generate random probe (pushing it onto top of hosts list so it happens first, and increment a
  // counter of number of probes generated)
  // update graph
  // this will then take care of itself as the algorithm will continue on its own
  // var clientsubnet = IPtrsList25(sample[0]);
  var clientsubnet = sample[0];
  var server = sample[2];
  var latencysample = sample[1];

  if (!graph.sessionInProgress) {
    graph.sessionStartingProbe=latencysample;
  }

  // clientsubnethistory is a hash of server latency samples
  var clientsubnethistory = new Object();
  // console.log('new history object for clientsubnet '+clientsubnet+' to server '+server+' of '+latencysample+' ms');
  if (typeof(graph.historicalprobes[clientsubnet]) !== 'undefined') {
    clientsubnethistory = graph.historicalprobes[clientsubnet];
  }

  clientsubnethistory[server]=latencysample;

  // historicalprobes is a hash of clientsubnet histories indexed by clientsubnet
  graph.historicalprobes[clientsubnet] = clientsubnethistory;

  for (var i = 0; i < sample.length; i++) {
    if (i % 2 === 0) {
      if (HostAlreadyExists(clientsubnet, graph) !== true) {
        if (i === 0) {
          console.log('adding subnet '+clientsubnet+' to server '+server+' with latency of '+latencysample);
          graph.newNode({label: clientsubnet, type: 'host'});
          graph.clientnames.push(clientsubnet);
        }
      }
    }
    if (i % 2 !== 0) {
      if (EdgeAlreadyExists(HostByLabel(clientsubnet, graph), HostByLabel(sample[i], graph), graph, false) !== true) {
        BuildEdge(clientsubnet, server, latencysample, graph, delayscale, false);
      } else {
        UpdateEdge(clientsubnet, server, latencysample, graph, delayscale, false);
      }
    }
  }
}
var UpdateEdge = function (firsthop, secondhop, latency, graph, delayScale) {
    console.log('update an edge');
    var edgeLength = latency;
    if (typeof delayScale === 'undefined') {
        delayScale = true;
    }
    if (latency !== 1 && delayScale) {
        edgeLength = Math.log(latency) / Math.log(2);
    }
    for (var j = 0; j < graph.edges.length; j++) {
        if (graph.edges[j].source.id === HostByLabel(firsthop, graph).id && graph.edges[j].target.id === HostByLabel(secondhop, graph).id) {
            //console.log('found matching edge to change length from ' + graph.edges[j].data.length);
            graph.removeEdge(graph.edges[j]);
            //console.log('edge dropped');
        }
    }
};

var BestPossibleResult = function (subnet) {
    //console.log('Get Best Possible Result for subnet');
    return subnet;
};

var BestHistoryResult = function (subnet, graph) {
    var bestHistoryResult = BestHistorySample(subnet, graph);
    //console.log('Get Best History Result for subnet');
    return bestHistoryResult;
};

var BestHistorySample = function (subnet, graph) {
    var bestHistorySample = subnet;
    //console.log('Get Best History Sample for subnet');
    return bestHistorySample;
};

var BestPossible = function (subnet) {
    var bestPossible = subnet;
    return bestPossible;
};

if (latencySample <= graph.goodEnough && subnet != server) {
    var bestPossibleResult = BestPossible(subnet);
    //console.log('Server is good enough, stay, inc goodenough');
    console.log('−−', 'server: ' + subnet + ' has a good enough connection to ' + server + ' so staying put ');
    console.log('−−', 'session: ' + subnet + ' is now ' + server + ' in ' + subnet + '');
    console.log('−−', 'session history ' + subnet + ' complete, picked ' + server + ' with latency ' + latencySample + ' best was ' + bestPossibleResult.serverName + ' with latency ' + bestPossibleResult.latency);
    graph.goodEnoughCounter = graph.goodEnoughCounter + 1;
    graph.sessionHistoryCounter = graph.sessionHistoryCounter + 1;
    graph.historyRedirectCounter = graph.historyRedirectCounter + 1;
    graph.sessionHistorySampleCounter = graph.sessionHistorySampleCounter + 1;
    graph.sessionInProgress = true;
    console.log('−−', 'session starting Probe ' + subnet + ' latency ' + latencySample + ' best was ' + bestPossibleResult.serverName + ' with latency ' + bestPossibleResult.latency);
}

if (BestHistoryResult.latency <= graph.goodEnough) {
    // else if a historical server is good enough redirect to it − redirecttohistoricalgoodenough
    console.log('−−', 'session history ' + subnet + ' is being redirected to historical best server ' + BestHistoryResult.serverName);
    graph.sessionHistoryCounter = graph.sessionHistoryCounter + 1;
    graph.historyRedirectCounter = graph.historyRedirectCounter + 1;
    graph.sessionHistorySampleCounter = graph.sessionHistorySampleCounter + 1;
    hosts.unshift((subnet, redirects[subnet][BestHistoryResult.serverName], BestHistoryResult.serverName));
    graph.sessionInProgress = true;
}

else {
    // else request a real sample to calculated best server and put it a top of queue, inc had to generate a new probe
    console.log('−−', 'session starting Probe ' + subnet + ' latency ' + latencySample + ' best was ' + bestPossibleResult.serverName + ' with latency ' + bestPossibleResult.latency);
    graph.newProbeNeeded = subnet;
    graph.sessionInProgress = true;
    console.log('−−', 'session starting Probe ' + subnet + ' latency ' + latencySample + ' best was ' + bestPossibleResult.serverName + ' with latency ' + bestPossibleResult.latency);
    
    } else if (subnet == server) {
        // also captures the case where we know a server is on your subnet, redirect to it and be done.
        var bestPossibleResult = BestPossible(subnet);
        console.log('−−', 'session history ' + subnet + ' is being redirected to historical best server ' + BestHistoryResult.serverName);
        graph.sessionHistoryCounter = graph.sessionHistoryCounter + 1;
        graph.historyRedirectCounter = graph.historyRedirectCounter + 1;
        graph.sessionHistorySampleCounter = graph.sessionHistorySampleCounter + 1;
        console.log('−−', 'session starting Probe ' + subnet + ' latency ' + latencySample + ' best was ' + bestPossibleResult.serverName + ' with latency ' + bestPossibleResult.latency);
    
    } else if (BestHistoryResult.latency == graph.goodEnough) {
        // else if a historical server is good enough redirect to it − redirecttohistoricalgoodenough
        console.log('−−', 'session history ' + subnet + ' is being redirected to historical best server ' + BestHistoryResult.serverName);
        graph.sessionHistoryCounter = graph.sessionHistoryCounter + 1;
        graph.historyRedirectCounter = graph.historyRedirectCounter + 1;
        graph.sessionHistorySampleCounter = graph.sessionHistorySampleCounter + 1;
        hosts.unshift((subnet, redirects[subnet][BestHistoryResult.serverName], BestHistoryResult.serverName));
        graph.sessionInProgress = true;
    }

else {
    // else request a real sample to calculated best server and put it a top of queue, inc had to generate a new probe
    console.log('−−', 'session starting Probe ' + subnet + ' latency ' + latencySample + ' best was ' + bestPossibleResult.serverName + ' with latency ' + bestPossibleResult.latency);
    graph.newProbeNeeded = subnet;
    graph.sessionInProgress = true;
    console.log('−−', 'session starting Probe ' + subnet + ' latency ' + latencySample + ' best was ' + bestPossibleResult.serverName + ' with latency ' + bestPossibleResult.latency);
    
    } else if (BestHistoryResult.latency == graph.goodEnough) {
        // else if a historical server is good enough redirect to it − redirecttohistoricalgoodenough
        console.log('−−', 'session history ' + subnet + ' is being redirected to historical best server ' + BestHistoryResult.serverName);
        graph.sessionHistoryCounter = graph.sessionHistoryCounter + 1;
        graph.historyRedirectCounter = graph.historyRedirectCounter + 1;
        graph.sessionHistorySampleCounter = graph.sessionHistorySampleCounter + 1;
        console.log('−−', 'session starting Probe ' + subnet + ' latency ' + latencySample + ' best was ' + bestPossibleResult.serverName + ' with latency ' + bestPossibleResult.latency);
    
    } else if (BestHistoryResult.latency <= graph.goodEnough) {
        // else if a historical server is good enough redirect to it − redirecttohistoricalgoodenough
        console.log('−−', 'session history ' + subnet + ' is being redirected to historical best server ' + BestHistoryResult.serverName);
        graph.sessionHistoryCounter = graph.sessionHistoryCounter + 1;
        graph.historyRedirectCounter = graph.historyRedirectCounter + 1;
        graph.sessionHistorySampleCounter = graph.sessionHistorySampleCounter + 1;
        console.log('−−', 'session starting Probe ' + subnet + ' latency ' + latencySample + ' best was ' + bestPossibleResult.serverName + ' with latency ' + bestPossibleResult.latency);
    
    } else if (subnet == server) {
        // also captures the case where we know a server is on your subnet, redirect to it and be done.
        var bestPossibleResult = BestPossible(subnet);
        console.log('−−', 'session history ' + subnet + ' is being redirected to historical best server ' + BestHistoryResult.serverName);
        graph.sessionHistoryCounter = graph.sessionHistoryCounter + 1;
        graph.historyRedirectCounter = graph.historyRedirectCounter + 1;
        graph.sessionHistorySampleCounter = graph.sessionHistorySampleCounter + 1;
        console.log('−−', 'session starting Probe ' + subnet + ' latency ' + latencySample + ' best was ' + bestPossibleResult.serverName + ' with latency ' + bestPossibleResult.latency);
    
    } else if (BestHistoryResult.latency == graph.goodEnough) {
        // else if a historical server is good enough redirect to it − redirecttohistoricalgoodenough
        console.log('−−', 'session history ' + subnet + ' is being redirected to historical best server ' + BestHistoryResult.serverName);
        graph.sessionHistoryCounter = graph.sessionHistoryCounter + 1;
        graph.historyRedirectCounter = graph.historyRedirectCounter + 1;
        graph.sessionHistorySampleCounter = graph.sessionHistorySampleCounter + 1;
        console.log('−−', 'session starting Probe ' + subnet + ' latency ' + latencySample + ' best was ' + bestPossibleResult.serverName + ' with latency ' + bestPossibleResult.latency);
    
    } else {
        // else request a real sample to calculated best server and put it a top of queue, inc had to generate a new probe
        console.log('−−', 'session starting Probe ' + subnet + ' latency ' + latencySample + ' best was ' + bestPossibleResult.serverName + ' with latency ' + bestPossibleResult.latency);
        graph.newProbeNeeded = subnet;
        graph.sessionInProgress = true;
        console.log('−−', 'session starting Probe ' + subnet + ' latency ' + latencySample + ' best was ' + bestPossibleResult.serverName + ' with latency ' + bestPossibleResult.latency);
    }

}
graph.newEdge(HostByLabel(firshop, graph), HostByLabel(secondhop, graph), {label: latency, length: edgeLength, directional: false});
return true;
}
}
}

var BuildEdge = function (firshop, secondhop, latency, graph, delayscale, allowopposite) {
  // check and see if this edge exists already.
  // compare two hops, edge label, and directionality
  var edgeLength = latency;
  if (typeof delayscale === "undefined") {
    delayscale = true;
  }
  if (typeof allowopposite === "undefined") {
    allowopposite = false;
  }
  if (latency !== 1 && delayscale) {
    edgeLength = Math.log(latency) / Math.log(2);
  }
  graph.newEdge(HostByLabel(firshop, graph), HostByLabel(secondhop, graph), {label: latency, length: edgeLength, directional: false});
}

// stop simulation when energy of the system goes below a threshold
if (t.totalEnergy() < 0.5) {
  t._started = false;
}

// bje added
  t.graph.currentlatencyScale = CalculatePixelsPerMS(t.graph);
  if (t.graph.lasthostadded === "none") {
    console.log('Reached steady state ' + t._starte());
    console.log('latency scale is ' + t.graph.currentlatencyScale.toFixed(2) + ' pixels per ms');
    PrintDistances(t.graph);
    console.log('Counts: GoodEnough=' + t.graph.goodEnoughCounter + ' HistRedirect=' + t.graph.historicalRedirectCounters + ' GenerateNewProbe=' + t.graph.generatedNewProbeCounter);
  } else if (t.graph.lasthostadded !== "done") {
    console.log('Reached steady state ');
    console.log('latency scale is ' + t.graph.currentlatencyScale.toFixed(2) + ' pixels per ms');
    if (t.graph.newProbeNeeded !== "") {
      var calculatedBestServer = PrintBestServer(t.graph.lasthostadded, t.graph);
      var bestisinhistorical = false;
      // if calculated is in historical.
      for (var x in t.graph.historical probes) {
        if (x == calculatedBestServer) {
          console.log("calbest (+calculatedBestServer+) is in historical, but must not be good enough")
          bestisinhistorical = true;
          break;
        }
      }
      if (!bestisinhistorical) {

```javascript
console.log('generating new probe for ' + t.graph.newProbeNeeded + ' to ' + calculatedBestServer + ' delay ' + redirects[t.graph.newProbeNeeded][calculatedBestServer] + t.graph.sessionInProgress = true;

hosts.unshift([t.graph.newProbeNeeded, redirects[t.graph.newProbeNeeded][calculatedBestServer], calculatedBestServer]);

if (t.graph.newProbeNeeded == t.graph.sessionInProgress)

}else {

console.log('need to generate new probe to random not in historical '');

// loop through list of all servers

// loop through historical servers
var shortlist = new Array();
var foundit = false;
for (var i = 0; i < t.graph.servernames.length; i++) {
for (var j in t.graph.historicalProbes[t.graph.newProbeNeeded]) {

if (t.graph.servernames[i] == j) {

foundit = true;
break;
}
}
if (!foundit) {
shortlist.push(t.graph.servernames[i]);
}

foundit = false;

console.log('short ' + shortlist);
if (shortlist.length == 0) {

// then we just send to best possible and end session.
var gotobest = BestPossible(t.graph.newProbeNeeded);
t.graph.sessionHistoricalCounter = t.graph.sessionHistoricalCounter + 1;
t.graph.historicalRedirectCounter = t.graph.historicalRedirectCounter + 1;
console.log('−− ' + t.graph.sessionStartingProbe + ' ' + gotobest.serverName + ' ' + gotobest.redirect); console.log(t.graph.newProbeNeeded + ' session complete, picked "+gotobest.serverName+" with latency "+gotobest.latency+" best was "+gotobest.serverName+" with latency "+gotobest.latency+"; t.graph.newProbeNeeded = "";

//hosts.unshift([t.graph.newProbeNeeded, gotobest.latency, gotobest.serverName]);

}else {

var rand = shortlist[Math.floor(Math.random() * shortlist.length)];
hosts.unshift([t.graph.newProbeNeeded, redirects[t.graph.newProbeNeeded][rand], rand]);
t.graph.sessionInProgress = true;
}

}

// then it isn’t good enough or we would have
// gone already. So instead fo random to something not in historical
// need to know how many servers exist, which ones aren’t in historical for this subnet – a little function, get back a list of ones not in historical, then randomly pick one
// if all servers are in historical and none good enough, just redirect to best. //which would be call BestHistorySample

//hosts.unshift([t.graph.newProbeNeeded, getRandomInt(75, 150), calculatedBestServer]);
t.graph.newProbeNeeded = "";
}

else {

// console.log('I think this is the end of a session ');
t.graph.sessionStartingProbe = 0;

```

t.graph.sessionNewProbeCounter = 0;
t.graph.sessionHistoricalCounter = 0;
t.graph.sessionInProgress = false;
}
} else if (t.graph.lasthostadded == "done") {
    console.log('simulation complete ' + Date());
    console.log('Counters: GoodEnough=' + t.graph.goodEnoughCounter + '
HistRedirect=' + t.graph.historicalRedirectCounter + 
GenerateNewProbe=' + t.graph.generatedNewProbeCounter);
}

if (hosts.length > 0) {
    AddHost(hosts.shift(), t.graph);
} else {
    t.graph.lasthostadded = "done";
}

if (typeof(done) !== 'undefined') { done(); }
} else {
    Layout.requestAnimationFrame(step);
}
Listing A.3: Iplane Diameter

```python
#!/opt/local/bin/python2.7

import matplotlib.pyplot as plt
import numpy as np
import csv
import sys

def read_tigr_delay_data(filename):
    
    # Parameters:
    # 'filename': name of the file that contains Iris data set.

    delays = []

    with open(filename) as fd:
        reader = csv.DictReader(fd)
        for row in reader:
            # delay column is data
            delays.append(float(row['Hops']))

    return delays

if len(sys.argv) != 2:
    print("usage: ", str(sys.argv[0]), " filename")
    sys.exit()
else:
    filename = str(sys.argv[1])

x = read_tigr_delay_data(filename)

hist, bins = np.histogram(x, bins=30)
cumulative = np.cumsum(hist)
total = np.sum(hist)
cumulative2 = np.true_divide(cumulative, total)
normalized = np.true_divide(hist, total)

# print hist
width = 0.7*(bins[1]-bins[0])
center = (bins[-1]+bins[1:])/2
plt.bar(center, normalized, align = 'center', width = width)
plt.xlabel('Hop Count')
plt.ylabel('Internet End to End Hop Count Histogram')
plt.show()
```

144
import matplotlib.pyplot as plt
import numpy as np
import csv
import sys

def read_tigr_delay_data(filename):
    
    :Parameters:
    - `filename`: name of the file that contains Iris data set.
    
    delays = []

    with open(filename) as fd:
        reader = csv.DictReader(fd)
        for row in reader:
            # delay column is data
            delays.append(float(row['Hops']))

    return delays

if len(sys.argv) != 2:
    print("usage:", str(sys.argv[0]), "filename")
    sys.exit()
else:
    filename = str(sys.argv[1])

x = read_tigr_delay_data(filename)

hist, bins = np.histogram(x, bins=30)
cumulative = np.cumsum(hist)
total = np.sum(hist)
cumulative2 = np.true_divide(cumulative, total)
normalized = np.true_divide(hist, total)

# print(hist)
width = 0.7*(bins[1]-bins[0])
center = (bins[-1]+bins[1:])/2

plt.bar(center, hist, align='center', width=width)
plt.bar(center, normalized)
plt.xlim(xmin=0, xmax=30)
plt.xticks(xmin=0, xmax=30)
plt.xlabel('Hop Count')
plt.ylabel('Iris Hop Count')
plt.title('Tactical End to End Hop Count Histogram')
plt.show()
Listing A.5: Tactical Delay Mixture Model

```python
#!/opt/local/bin/python2.7

def read_tigr_delay_data(filename):
    """: Parameters:
    - 'filename': name of the file that contains Iris data set.
    ""
    delays = []

    with open(filename, 'rU') as fd:
        reader = csv.DictReader(fd)
        for row in reader:
            # delay column is data
            if float(row['G']) != -1.0:
                delays.append(float(row['G']))  # column G is icmp rtt, H is http rtt

        return delays

if len(sys.argv) != 2:
    print "usage:", str(sys.argv[0]), "filename"
    sys.exit()
else:
    filename = str(sys.argv[1])

y = read_tigr_delay_data(filename)
x = np.true_divide(y, 100)  # convert from microseconds to ms

hist, bins = np.histogram(x, bins=100)
cumulative = np.cumsum(hist)
total = np.sum(hist)
cumulative2 = np.true_divide(cumulative, total)
normalized = np.true_divide(hist, total)
width = 0.7*(bins[1]-bins[0])
center = (bins[-1]+bins[1:])/2

data = mixture.DataSet()
data.fromList(x)

# start them off with something arbitrary (probably based on a guess from the figure)
n1 = mixture.NormalDistribution(30, 20)
```

146
n2 = mixture.NormalDistribution(300, 60)
n3 = mixture.NormalDistribution(600, 40)
n4 = mixture.NormalDistribution(1200, 60)
n5 = mixture.NormalDistribution(50, 60)
n6 = mixture.NormalDistribution(2500, 60)
m = mixture.MixtureModel(6, [0.2, 0.3, 0.1, 0.1, 0.2, 0.1], [n1, n2, n3, n4, n5, n6])

# perform expectation maximization
# m.EM(data, 40, .1)
# this runs 100 simulations with random starting values
m.randMaxEM(data, 100, 40, 0.1, silent=True)

print m

# now make a bunch of samples using the found distributions
inc = 0;
for comp in m.components:
    if hasattr(comp.distList[0], 'mu'):
        print 'comp is normal'
        mix = np.concatenate([mix, np.random.normal(comp.distList[0].mu, comp.distList[0].sigma, [5000 * m.pi[inc]])])
    else:
        # first component is Exponential
        print 'c0 is not normal'
        mix = np.concatenate([mix, np.random.exponential(1/comp.distList[0].lambda, [5000 * m.pi[inc]])])
    inc = inc + 1

# then graph the sampled data against the original

sample_hist, bins = np.histogram(mix, bins)
cumulative_sample = np.cumsum(sample_hist)
total_sample = np.sum(sample_hist)
cumulative2_sample = np.true_divide(cumulative_sample, total_sample)
normalized_sample = np.true_divide(sample_hist, total_sample)

plt.bar(center, normalized_sample, align='center', width=width, color='r', label='Samples from Derived Mass Function')
plt.bar(center, normalized, align='center', width=width, label='Tactical Network Measurements', alpha=0.5)
plt.xlim(xmax=3000)
plt.xlabel('Round Trip Time in ms')
plt.title('Tactical Network Server to Server Latency Histogram')
plt.legend()
plt.show()

# print cumulative2
plt.plot(bins[:-1], cumulative2_sample, color='r', label='Samples from Derived Mass Function')
plt.plot(bins[:-1], cumulative2, label='Tactical Network Measurements')
plt.xlabel('Round Trip Time in ms')
plt.title('Tactical Network Server to Server Latency CDF')
plt.xlim(xmax=2000)
plt.xlim(xmin=0)
plt.legend(loc='lower right')
plt.show()
Listing A.6: Kullbak-Liebler Example

```python
#!/opt/local/bin/python2.7

import random
import scipy.stats as stat
import numpy as np
from matplotlib import pyplot as plt
import sys

def kl(p, q):
    internalsum = 0.0
    p = np.array(p, dtype=np.float)
    q = np.array(q, dtype=np.float)
    for i in range(len(p)):
        if p[i] == 0.0:
            # covers also case of qi=0 and pi=0
            next
        elif q[i] == 0.0 and p[i] != 0.0:
            print 'error case'
        else:
            # print 'qi=', q[i], 'pi=', p[i]
            internalsum = internalsum + p[i] * np.log2(p[i] / q[i])
    return internalsum

d1 = [0, 1, 2, 3]
p = np.sum(d1)

f1 = np.random.normal(200, 10, 5000)
f2 = np.random.normal(200, 60, 5000)
f3 = np.random.normal(200, 30, 5000)
f4 = np.random.normal(200, 11, 5000)

hist2, bins = np.histogram(f2, bins = 100)
total = np.sum(hist2)
normalized2 = np.true_divide(hist2, total)
hist1, bins = np.histogram(f1, bins)
normalized1 = np.true_divide(hist1, total)
hist3, bins = np.histogram(f3, bins)
normalized3 = np.true_divide(hist3, total)
hist4, bins = np.histogram(f4, bins)
normalized4 = np.true_divide(hist4, total)

print 'stat.entropy(normalized1)', stat.entropy(normalized1)
print 'stat.entropy(normalized2)', stat.entropy(normalized2)
print 'stat.entropy(normalized3)', stat.entropy(normalized3)
print 'stat.entropy(normalized4)', stat.entropy(normalized4)
print 'my_kl = kl(f2, f1)', kl(normalized1, normalized2)
print 'my_kl = kl(f3, f1)', kl(normalized1, normalized3)
print 'my_kl = kl(f4, f1)', kl(normalized1, normalized4)

fig, ax = plt.subplots(1)
```

148
\begin{verbatim}
width = 0.7*(bins[1]-bins[0])
center = (bins[-1]+bins[1:])/2
plt.bar(center, normalized2, align = 'center', width = width, color='y', label='Q1,SD=60')
plt.bar(center, normalized3, align = 'center', width = width, color='g', label='Q2,SD=30')
plt.bar(center, normalized4, align = 'center', width = width, label='Q3,SD=11')
plt.bar(center, normalized1, align = 'center', width = width, alpha=.5, color='r', label='P,SD=10')
textr = '$D_{KL} (P || Q1) =1.85$ $D_{KL} (P || Q2) = .9$ $D_{KL} (P || Q3) = .008$'
props = dict(boxstyle='square', facecolor='none')
ax.text(0.05, 0.95, textr, transform=ax.transAxes, verticalalignment='top', fontsize = 14, bbox=props)
plt.xlim(xmin =0, xmax =400)
plt.title('Kullback-Leibler Divergence of Q1, Q2, and Q3 from P')
plt.legend()
plt.show()
\end{verbatim}
import random
import scipy.stats as stat
import numpy as np
from matplotlib import pyplot as plt
import mixture
import csv
import sys

if len(sys.argv) != 2:
    print('usage:', str(sys.argv[0]), 'filename')
sys.exit()
else:
    filename = str(sys.argv[1])

allhosts = {}
mylist1=[]
justhosts=[]

for lines in open(filename, 'r'):
    thisProbe = []
a=lines.split('"')
first=a[2].strip()
firstb = first.split(""")
first = firstb[1]
delaya=a[3].strip()
delayb = delaya.split(".")
delay = delayb[0]
seconda=a[4].strip()
secondb = seconda.split(""")
second = secondb[1]

if first in allhosts:
    thisProbe = allhosts[first]
else:
    justhosts.append(first)
    thisProbe[first] = 0
    thisProbe[second] = delay
    allhosts[first] = thisProbe

servers = random.sample(justhosts, 10)
hosts = random.sample(justhosts, 20)
best = {}
lowestserver = ""

for h in hosts:
    lowestdelay = 1000000
    for s in servers:
        if s == h:
            lowestdelay = 0
            lowestserver = s
        elif int(allhosts[h][s]) < lowestdelay:
            lowestdelay = int(allhosts[h][s])
            lowestserver = s
    #print h, s, allhosts[h][s], lowestdelay, lowestserver
    best[h] = lowestserver
```
# print servers
# print hosts

myfile=open(filename+'-best.js','w')
print >> myfile , 'var bestserver={};'
for i in best:
    print >> myfile, 'bestserver["%s"]=%s="%s"; %i, best[i], allhosts[i][best[i]]
myfile.close()

myfile=open(filename+'-servers.js','w')
print >> myfile , 'var servers=new Array();'
inc = 0
for i in servers:
    for j in servers:
        if i == j:
            next
        else:
            print >> myfile, 'servers["%s"]=%s="%s",%s="%s"; %inc, i, allhosts[i][j], j)
            inc = inc + 1
myfile.close()

myfile=open(filename+'-hosts.js','w')
print >> myfile , 'var hosts=new Array();'
inc = 0
for j in servers:
    for i in hosts:
        print >> myfile, 'hosts["%s"]=%s="%s",%s="%s"; %inc, i, allhosts[i][j], j)
        inc = inc + 1
myfile.close()

myfile=open(filename+'-redirs.js','w')
print >> myfile , 'var redirects={};'
for i in hosts:
    print >> myfile, 'redirects["%s"]=%s; %i),
    inc = 1
    random.shuffle(servers)
for j in servers:
    if inc == len(servers):
        print >> myfile, "%s":"%s" % (j, allhosts[i][j]).
    else:
        print >> myfile, "%s":"%s" % (j, allhosts[i][j]),
        inc = inc + 1
print >> myfile, '};'
myfile.close()
```
Listing A.8: Satellite Connector

```csharp
using System;
using System.Collections.Generic;
using System.Text;
using GraphManager;
using GraphManager.GraphGenerators;

[Serializable]
public class SatelliteHubSettings : IGraphGeneratorSettings
{
    // Core A is directly connected to how many other cores...
    public int CoreDegree = 3;

    // IP Addresses
    public int SubA = 10;
    public int SubB = 200;

    // Leaf B is directly connected to how many cores?
    // public int LeafDegree = 1;
    // We will use the num links on the leaf router to determine this value for each leaf
    // So, if you want the default case, set min and max to 1 for links on the leaf routers

    public class SatSubnet
    {
        public Guid Id;
        public int SubA;
        public int SubB;
        public int SubC;
        public int SubD;

        public SatSubnet(Guid id, int a, int b)
        {
            Id = id;
            SubA = a;
            SubB = b;
            SubC = 0;
            SubD = 1;
        }

        public string GetCoreIp()
        {
            return string.Format("{0}.{1}.{2}/{16}", SubA, SubB, 0, 1);
        }

        public string GetSubnet()
        {
            return string.Format("{0}.{1}.{2}/{16}", SubA, SubB, 0, 0);
        }

        public string GetNextLeafIp()
        {
            SubD++;
            if (SubD >= 255)
            {
            }
        }
    }
```
SubD = 1;
SubC++;
// we don’t worry about going past 2^16 leaves.
}
return string.Format("{0}.{1}.{2}.{3}/16", SubA, SubB, SubC, SubD);
}

public class SatelliteHubConnector : IGraphGenerator
{
RouterType CoreType;
RouterType LeafType;
string ErrString = "Satellite Hub Connector MUST have 2 router types, 'Sat Core' and 'Sat Leaf'. There must be at least one router of each type."

#region IGraphGenerator Members

public void GenerateRouterGraph(ref Graph g, IGraphSettings settings, RouterTypeAssociation rta, params List<Router>[] vertexSets)
{
Dictionary<Guid, SatSubnet> coreSubnets = new Dictionary<Guid, SatSubnet>();
SatelliteHubSettings s = settings as SatelliteHubSettings;

if (rta.Count != 2 || vertexSets.Length != 2 || vertexSets[0].Count == 0 || vertexSets[1].Count == 0)
throw new Exception(ErrString);
CoreType = LeafType = null;
foreach (RouterType rt in rta)
{
    if (rt.Name == "Sat Core")
        CoreType = rt;
    else if (rt.Name == "Sat Leaf")
        LeafType = rt;
}
if (CoreType == null || LeafType == null)
    throw new Exception(ErrString);

List<Router> cores;
List<Router> leafs;
if (vertexSets[0][0].RouterType.Name == "Sat Core")
{
    cores = vertexSets[0];
    leafs = vertexSets[1];
}
else
{
    cores = vertexSets[1];
    leafs = vertexSets[0];
}

ConnectCores(g, s.CoreDegree, cores);

Random rand = new Random();
List<Router> notSaturated = new List<Router>();
int subb = s.SubB;
foreach (Router r in cores)
coreSubnets[r.ID] = new SatSubnet(r.ID, s.SubA, subb++);
if(r.MaxLinks > r.EdgeNumber)
    notSaturated.Add(r);
}

foreach (Router r in leaves)
{
    for (int i = r.EdgeNumber; i < r.MaxLinks; i = r.EdgeNumber)
    {
        if (notSaturated.Count == 0)
            return; //nothing else to do here...

        //connect each leaf link to a random core
        int x = rand.Next(notSaturated.Count);
        int start = x;
        bool failed = false;
        while (notSaturated[x].incidentEdges.ContainsKey(r))
        {
            x++;
            if (x >= notSaturated.Count)
                x = 0;
            if (x == start)
            {
                failed = true;
                break;
            }
        }
        if (!failed)
        {
            Edge ed = g.AddEdge(r, notSaturated[x], 1, Router.GetEdgeDelay(r, notSaturated[x]));
            ed.SrcIp = coreSubnets[notSaturated[x].ID].GetNextLeafIp();
            ed.DestIp = coreSubnets[notSaturated[x].ID].GetCoreIp();
            ed.Subnet = coreSubnets[notSaturated[x].ID].GetSubnet();
            if (notSaturated[x].EdgeNumber >= notSaturated[x].MaxLinks)
                notSaturated.Remove(notSaturated[x]);
        }
        else
        {
            //all remaining core routers are already connected to us.
            break;
        }
    }
}

#endregion

//using the ConnectorRandomMesh code for the cores
public void ConnectCores(Graph g, int degree, List<Router> cores)
{
    Random rand = new Random();
    List<Router> notSaturated = new List<Router>();
    notSaturated.AddRange(cores);

    //connect each router to "degree" other random routers
foreach (Router r in cores)
{
    // We will saturate this now
    notSaturated.Remove(r);

    // create remaining edges to random unsaturated routers
    for (int i = r.EdgeNumber; i < degree; i = r.EdgeNumber)
    {
        if (notSaturated.Count == 0)
            break;
        int x = rand.Next(notSaturated.Count);
        int start = x;
        bool failed = false;
        while(notSaturated[x].incidentEdges.ContainsKey(r))
        {
            x++;
            if (x >= notSaturated.Count)
                x = 0;
            if (x == start)
            {
                failed = true;
                break;
            }
        }
        if (!failed)
        {
            g.AddEdge(r, notSaturated[x], 1, Router.GetEdgeDelay(r, notSaturated[x]));
            if (notSaturated[x].EdgeNumber >= degree)
                notSaturated.Remove(notSaturated[x]);
        }
        else
        {
            // all remaining core routers are already connected to us.
            break;
        }
    }
}

Listing A.9: WIN-T Connector

```csharp
using System;
using System.Collections.Generic;
using System.Text;
using GraphManager;
using GraphManager.GraphGenerators;

public class WinTSettings : IGraphGeneratorSettings
{
    public int NumBattalionsPerBrigade = 4;
    public int NumCompaniesPerBattalion = 6;
    //public int BrigadeCoreCount = 1; // per brigade — fixed
    //public int BrigadeTcnCount = 1; // per brigade — fixed
    public int BattalionRouterCount = 3; // per battalion
    public int CmdTruckCount = 2; // per company
    public int TruckCount = 8; // per company

    // IP Addresses
    public int SubA = 10;
    public int SubB = 150;

    // Brigade Core Kbit/s & Delay used for delay between brigades
    // Brigade TCN Kbit/s & Delay used for connecting to battalion routers (T1)
    // Battalion Router Kbit/s & Delay used for ALL connections within Battalion net (1MB)
    // Cmd Truck Kbit/s & Delay used for connections to battalion net (64kbit).
    // Truck Kbit/s & Delay — used for all connections on company net
    // All link counts are ignored.
}

public class WinTConnector : IGraphGenerator
{
    public int NumBattalionsPerBrigade = 4;
    public int NumCompaniesPerBattalion = 6;
    //public int BrigadeCoreCount = 1; // per brigade — fixed
    //public int BrigadeTcnCount = 1; // per brigade — fixed
    public int BattalionRouterCount = 3; // per battalion
    public int CmdTruckCount = 2; // per company
    public int TruckCount = 8; // per company

    // IP Addresses
    public int SubA = 10;
    public int SubB = 150;

    // Brigade Core Kbit/s & Delay used for delay between brigades
    // Brigade TCN Kbit/s & Delay used for connecting to battalion routers (T1)
    // Battalion Router Kbit/s & Delay used for ALL connections within Battalion net (1MB)
    // Cmd Truck Kbit/s & Delay used for connections to battalion net (64kbit).
    // Truck Kbit/s & Delay — used for all connections on company net
    // All link counts are ignored.
}

public void GenerateRouterGraph(ref Graph g, IGraphGeneratorSettings settings, RouterTypeAssociation rta, params List<Router>[] vertexSets)
{
    WinTSettings settings = settings as WinTSettings;
    Edge ed;
```

156
int brigEdge = 0;
int batEdge = 0;
int compEdge = 0;
int last = 0;

InitSubnets(settings_);

// first, remove all of the routers created for this connector by the parent
foreach (List<Router> lr in vertexSets)
{
    foreach (Router r in lr)
    {
        if (g.Vertices.ContainsKey(r.ID))
            g.Vertices.Remove(r.ID);
        lr.Clear();
    }
}

// make sure we have all of the correct router types
if (rt.Count != 5)
    throw new Exception(ErrString);
BrigadeCoreType = BrigadeTcnType = BattalionType = CmdTruckType = TruckType = null;
foreach (RouterType rt in rta)
{
    if (rt.Name == "Brigade Core")
        BrigadeCoreType = rt;
    else if (rt.Name == "Brigade TCN")
        BrigadeTcnType = rt;
    else if (rt.Name == "Battalion TCN")
        BattalionType = rt;
    else if (rt.Name == "Command Truck")
        CmdTruckType = rt;
    else if (rt.Name == "Win-T Truck")
        TruckType = rt;
}
if (BrigadeCoreType == null || BrigadeTcnType == null || BattalionType == null || CmdTruckType == null || TruckType == null)
    throw new Exception(ErrString);

// create Brigade Core routers and full mesh them
BrigadeCore = new Router(BrigadeCoreType);
g.Vertices.Add(BrigadeCore.ID, BrigadeCore);

// create Brigade TCNs and connect them to Brigade Cores
BrigadeTcn = new Router(BrigadeTcnType);
g.Vertices.Add(BrigadeTcn.ID, BrigadeTcn);
g.AddEdge(BrigadeTcn, BrigadeCore, 1, Router.GetEdgeDelay(BrigadeTcn, BrigadeCore), BrigadeCoreType.BandwidthMean); brigEdge++;

BattalionTcns = new List<List<Router>>();
CmdTrucks = new List<List<Router>>();
Trucks = new List<List<Router>>();
foreach (int batt = 0; batt < settings_.NumBattalionsPerBrigade; batt++)
{
    BattalionTcns.Add(new List<Router>());
    CmdTrucks.Add(new List<Router>());
    Trucks.Add(new List<Router>());
    }
// create Battalion TCNs

for (int i = 0; i < settings_.BattalionRouterCount; i++)
{
    BattalionTcns[batt].Add(new Router(BattalionType));
    g.Vertices.Add(BattalionTcns[batt][i].ID, BattalionTcns[batt][i]);

    // Connect To Brigade TCN
    last = g.Edges.Count;
    g.AddEdge(BattalionTcns[batt][i], BrigadeTcn, 1, Router.GetEdgeDelay(BattalionTcns[batt][i], BrigadeTcn),
               BrigadeTcnType.BandwidthMean);
    brigEdge++;
    if (g.Edges.Count == last) { Console.WriteLine("Err1"); }

    // Full Mesh to selves on Battalion Net
    for (int j = i; j > 0; j--)
    {
        last = g.Edges.Count;
        ed = g.AddEdge(BattalionTcns[batt][i], BattalionTcns[batt][j], 1, Router.GetEdgeDelay(BattalionTcns[batt][i], BattalionTcns[batt][j]),
                       BattalionType.BandwidthMean);
        ed.Subnet = GetSubnet(batt);
        ed.SrcIp = GetIp(BattalionTcns[batt][i].ID, batt);
        ed.DestIp = GetIp(BattalionTcns[batt][j].ID, batt);
        batEdge++;
        if (g.Edges.Count == last) { Console.WriteLine("Err2"); }
    }
}

for (int cmp = 0; cmp < settings_.NumCompaniesPerBattalion; cmp++)
{
    CmdTrucks[batt].Add(new List<Router>());
    Trucks[batt].Add(new List<Router>());

    // create Command Trucks
    for (int i = 0; i < settings_.CmdTruckCount; i++)
    {
        CmdTrucks[batt][cmp].Add(new Router(CmdTruckType));
        g.Vertices.Add(CmdTrucks[batt][cmp][i].ID, CmdTrucks[batt][cmp][i]);

        // Connect To Brigade TCN
        last = g.Edges.Count;
        g.AddEdge(CmdTrucks[batt][cmp][i], BrigadeTcn, 1, Router.GetEdgeDelay(CmdTrucks[batt][cmp][i], BrigadeTcn),
                   CmdTruckType.BandwidthMean);
        brigEdge++;
        if (g.Edges.Count == last) { Console.WriteLine("Err3"); }

        // Full Mesh To Battalion Net
        for (int j = 0; j < BattalionTcns[batt].Count; j++)
        {
            last = g.Edges.Count;
            ed = g.AddEdge(CmdTrucks[batt][cmp][i], BattalionTcns[batt][j], 1, Router.GetEdgeDelay(CmdTrucks[batt][cmp][i], BattalionTcns[batt][j]),
                           BattalionType.BandwidthMean);
            ed.Subnet = GetSubnet(batt);
            ed.SrcIp = GetIp(CmdTrucks[batt][cmp][i].ID, batt);
            ed.DestIp = GetIp(BattalionTcns[batt][j].ID, batt);
            batEdge++;
            if (g.Edges.Count == last) { Console.WriteLine("Err4"); }
        }
    }
}
for (int j = cmp - 1; j >= 0; j--)
{
    last = g.Edges.Count;
    ed = g.AddEdge(CmdTrucks[batt][cmp][i], CmdTrucks[batt][j][k], 1,
                   Router.GetEdgeDelay(CmdTrucks[batt][cmp][i], CmdTrucks[batt][j][k]), BattalionType.BandwidthMean);
    ed.Subnet = GetSubnet(batt);
    ed.SrcIp = GetIp(CmdTrucks[batt][cmp][i].ID, batt);
    ed.DestIp = GetIp(CmdTrucks[batt][j][k].ID, batt);
    batEdge++;
    if (g.Edges.Count == last) { Console.WriteLine("Err 5"); }
}

// these can't be created because we can't have multiple connections between a single router pair
// and these routers will be connected on the company net later
/*
for (int j = i - 1; j >= 0; j--)
{
    last = g.Edges.Count;
    g.AddEdge(CmdTrucks[batt][cmp][i], CmdTrucks[batt][cmp][j], 1,
              Router.GetEdgeDelay(CmdTrucks[batt][cmp][i], CmdTrucks[batt][cmp][j]), BattalionType.BandwidthMean);
    batEdge++;
    if (g.Edges.Count == last) { Console.WriteLine("Err 6"); }
}*/

// Full Mesh To Company Net
for (int j = i - 1; j >= 0; j--)
{
    last = g.Edges.Count;
    g.AddEdge(CmdTrucks[batt][cmp][i], CmdTrucks[batt][cmp][j], 1,
              Router.GetEdgeDelay(CmdTrucks[batt][cmp][i], CmdTrucks[batt][cmp][j]), TruckType.BandwidthMean);
    ed.Subnet = GetSubnet(batt, cmp);
    ed.SrcIp = GetIp(CmdTrucks[batt][cmp][i].ID, batt, cmp);
    ed.DestIp = GetIp(CmdTrucks[batt][cmp][j].ID, batt, cmp);
    compEdge++;
    if (g.Edges.Count == last) { Console.WriteLine("Err 7"); }
}

// create Win-T Trucks
for(int i=0;i<settings_.TruckCount;i++)
{
    Trucks[batt][cmp].Add(new Router(TruckType));
    g.Vertices.Add(Trucks[batt][cmp][i].ID, Trucks[batt][cmp][i]);
// Full Mesh To Company Net
for (int j = i - 1; j >= 0; j--)
{
    last = g.Edges.Count;
    ed = g.AddEdge(Trucks[batt][cmp][i], Trucks[batt][cmp][j], 1, Router.GetEdgeDelay(Trucks[batt][cmp][i],
                                                                                      Trucks[batt][cmp][j]), TruckType.BandwidthMean);
    ed.Subnet = GetSubnet(batt, cmp);
    ed.SrcIp = GetIp(Trucks[batt][cmp][i].ID, batt, cmp);
    ed.DestIp = GetIp(Trucks[batt][cmp][j].ID, batt, cmp);
    compEdge++;
    if (g.Edges.Count == last) { Console.WriteLine("Err 8"); }
}
for (int j = 0; j < CmdTrucks[batt][cmp].Count; j++)
{
    last = g.Edges.Count;
    ed = g.AddEdge(Trucks[batt][cmp][i], CmdTrucks[batt][cmp][j], 1,
        Router.GetEdgeDelay(Trucks[batt][cmp][i], CmdTrucks[batt][cmp][j]), TruckType.BandwidthMean);
    ed.Subnet = GetSubnet(batt, cmp);
    ed.SrcIp = GetIp(Trucks[batt][cmp][i].ID, batt, cmp);
    ed.DestIp = GetIp(CmdTrucks[batt][cmp][j].ID, batt, cmp);
    compEdge ++;
    if (g.Edges.Count == last) { Console.WriteLine("Err"); }
}
}
Concole.WriteLine("Brig={0} Bat={1} Cmp={2} " , brigEdge, batEdge, compEdge);

#endregion

private int SubA;
private int SubB;
Dictionary<string, Dictionary<Guid, string>> Ips;
Dictionary<string, string> Subnets;
Dictionary<string, int> NextC;
Dictionary<string, int> NextD;

private void InitSubnets(WinTSettings s)
{
    SubA = s.SubA;
    SubB = s.SubB;
    int subc = 0;
    int subd = 0;

    Ips = new Dictionary<string, Dictionary<Guid, string>>() ;
    Subnets = new Dictionary<string, string>();
    NextC = new Dictionary<string, int>();
    NextD = new Dictionary<string, int>();

    for (int btn = 0; btn < s.NumBattalionsPerBrigade; btn++)
    {
        string name = string.Format("BTN_{0}\" , btn);
        // btn networks are /26 (64 ips)
        Subnets[name] = string.Format("{0}.{1}.{2}.{3}/26" , SubA, SubB, subc, subd);
        Ips[name] = new Dictionary<Guid, string>();
        NextC[name] = subc;
        NextD[name] = subd + 1;
        subd += 64;
        if (subd > 192)
        {
            subd = 0;
            subc++;
        }
    }
}

for (int btn = 0; btn < s.NumBattalionsPerBrigade; btn++)
for (int srw = 0; srw < s.NumCompaniesPerBattalion; srw++)
{
    string name = string.Format("SRW_{0}_\{1\}", btn, srw);
    //srw networks are /27 (32 ips)
    Subnets[name] = string.Format("{0}.\{1}.\{2}.\{3}/27\", SubA, SubB, subc, subd);
    Ips[name] = new Dictionary<Guid, string>();
    NextC[name] = subc;
    NextD[name] = subd + 1;
    subd += 32;
    if (subd > 224)
    {
        subd = 0;
        subc++;
    }
}

private string GetSubnet(int btn, int srw = -1)
{
    string net;
    if (srw == -1)
    {
        net = string.Format("BTN_{0}\", btn);
    }
    else
    {
        net = string.Format("SRW_{0}_\{1}\", btn, srw);
    }
    return Subnets[net];
}

private string GetIp(Guid id, int btn, int srw = -1)
{
    string net;
    if (srw == -1)
    {
        net = string.Format("BTN_{0}\", btn);
    }
    else
    {
        net = string.Format("SRW_{0}_\{1}\", btn, srw);
    }
    if (!Ips[net].ContainsKey(id))
    {
        Ips[net][id] = string.Format("{0}.\{1}.\{2}.\{3}/2{4}\", SubA, SubB, NextC[net], NextD[net], srw == -1 ? 6 : 7);
        NextD[net] = NextD[net] + 1;
    }
    return Ips[net][id];
}
Listing A.10: Tactical Core Connector

using System;
using System.Collections.Generic;
using System.Text;
using GraphManager;
using GraphManager.GraphGenerators;

[Serializable]
public class TacticalCoreSettings : IGraphGeneratorSettings
{
    public int M0 = 1;
    public int LeafM0 = 1;
    public int SeedNetworkSize = 2;

    public int Sat2Pct = 14;
    public int Sat4Pct = 43;

    //this is probably the most correct way to do things. Look down at the code to see the differences.
    public bool CoreStrictProb = true;
    public bool LeafStrictProb = true;

    //The following 2 parameters are only meaningful if UseStrictProbability is false:

    //modifies probabilities by this amount when performing the BarabasiAlbert algorithm
    //so the likelihood of each connection will still be the same relative to its peers
    //but it is less likely that a given router will have m0 connections. It will most likely
    //have m0*XXXFactor connections.
    public double LeafFactor = 1.0;

    //normally I check oldest routers first to see if a new connection should be added
    //if this is 'true', then I check newest routers first
    public bool NewRoutersHavePriority = true;
}

public class TacticalCoreConnector : IGraphGenerator
{
    RouterType CoreType;
    RouterType LeafType;
    RouterType SatType;

    List<Router> Cores;
    List<Router> Leafs;
    List<Router> Sats;
    string ErrString = "TacticalCoreConnectorMUSThave3 router types named TC_Core, TC_Leaf, TC_Sat";

    #region IGraphGenerator Members
    public void GenerateRouterGraph(ref Graph g, IGraphGeneratorSettings settings, RouterTypeAssociation rta, params List<Router>[] vertexSets)
    {
        TacticalCoreSettings s = settings as TacticalCoreSettings;
        Random r = new Random();

        //leafConnCount is for adding new leafs to existing leafs & cores (and also cores to cores)
```csharp
int CntLeaf = 0;
//satConnCount is for adding new sat to existing sat & cores
int CntCore = 0;

//these are routers already in the network that can be connected to
List<Router> CoreAndLeafs = new List<Router>();
List<Router> ConnectedCores = new List<Router>();

//make sure we have all of the correct router types
if (rta.Count != 3)
    throw new Exception(ErrString);
CoreType = LeafType = SatType = null;
for (RouterType rt in rta)
{
    if (rt.Name == "TC_Core")
        CoreType = rt;
    else if (rt.Name == "TC_Leaf")
        LeafType = rt;
    else if (rt.Name == "TC_Sat")
        SatType = rt;
}
if (CoreType == null || LeafType == null || SatType == null)
    throw new Exception(ErrString);
for (int i = 0; i < vertexSets.Length; i++)
{
    if (vertexSets[i][0].RouterType.Name == "TC_Core")
        Cores = vertexSets[i];
    else if (vertexSets[i][0].RouterType.Name == "TC_Leaf")
        Leafs = vertexSets[i];
    else
        Sats = vertexSets[i];
}
if (Cores.Count < s.SeedNetworkSize)
    throw new Exception(string.Format("You must have at least {0} Core routers.", s.M0));

//seed the network with a full mesh of cores
for (int i = 0; i < s.SeedNetworkSize; i++)
{
    CoreAndLeafs.Add(Cores[i]);
    Cores.Add(Cores[i]);
    for (int j = i + 1; j < s.SeedNetworkSize; j++)
    {
        g.addEdge(Cores[i], Cores[j], 1, Math.Max(Cores[i].Delay, Cores[j].Delay));
        CntLeaf++;
        CntCore++;
    }
}

//now attach the rest of the cores
//this algorithm is a version of BarabasiAlbert.
//each new node that is added to the network must have a minimum of one connection created
//and a maximum of M0 connections.
//NOTE: I do not grab a single random number and then match it to an existing router
//but instead grab a random number for each existing router. This is how I allow for random
//chance to dictate the number of connections that a particular new router will have
//I could have done this by just calculating the number of new links first, but I decided to
```
// go this route for whatever reason.
for (int i = s.Min; i < Cores.Count; i++)
{
    Dictionary<Router, bool> connected = new Dictionary<Router, bool>();
    if (s.CoreStrictProb)
    {
        for (int cnt = 0; cnt < s.Min; cnt++)
        {
            double loc = r.NextDouble();
            double p = 0;
            for (int j = 0; j < CoresAndLeafs.Count; j++)
            {
                double prob = (double)CoresAndLeafs[j].EdgeNumber / (double)CntLeaf;
                if (prob <= p)
                {
                    break;
                }
            }
    }

    else
    {
        while (connected.Count == 0)
        {
            for (int j = 0; j < CoresAndLeafs.Count & & connected.Count < s.Min; j++)
            {
                double prob = (double)CoresAndLeafs[j].EdgeNumber / (double)CntLeaf;
                if (prob <= p)
                {
                    break;
                }
            }
    }

    if (s.NewRoutersHavePriority)
    {
        CoresAndLeafs.Insert(0, Cores[i]);
        ConnectedCores.Insert(0, Cores[i]);
    }

    else
    {
        CoresAndLeafs.Add(Cores[i]);
        ConnectedCores.Add(Cores[i]);
    }
CntLeaf += connected.Count;
CntCore += connected.Count;

// now attach the sat routers
// we connect them in pairs for now.
// this also uses the Barbata- Albert algorithm
int sat1pct = 100 - s.Sat2Pct - s.Sat4Pct;
int sat2pct = sat1pct + s.Sat2Pct;
int sat1cnt = (int)Math.Round((double)sats.Count * (double)sat1pct / 100.0, 0);
int sat2cnt = (int)Math.Round((double)sats.Count * (double)sat2pct / 100.0, 0);
for (int i = 0; i < sats.Count; i++)
{
    double loc = r.NextDouble();
    double p = 0;
    for (int j = 0; j < ConnectedCores.Count; j++)
    {
        p += (double)ConnectedCores[j].EdgeNumber / (double)CntCore;
        if (loc < p || j == (ConnectedCores.Count - 1))
        {
            int dd;
            if (i < sat1cnt)
            {
                dd = Sats[i].Delay;
            } 
            else if (i < sat2cnt)
            {
                dd = Sats[i].Delay + 2;
            }
            else
            {
                dd = Sats[i].Delay + 4;
            }
            g.AddEdge(Sats[i], ConnectedCores[j], 1, Math.Max(dd, ConnectedCores[j].Delay));
            CntCore++;
            break;
        }
    }
}

// finally attach the leaf routers
// this also uses the Barbata- Albert algorithm
for (int i = 0; i < Leafs.Count; i++)
{
    Dictionary<Router, bool> connected = new Dictionary<Router, bool>();
    if (s.LeafStrictProb)
    {
        for (int cnt = 0; cnt < s.LeafM0; cnt++)
        {
            double loc = r.NextDouble();
            double p = 0;
            for (int j = 0; j < CoresAndLeaves.Count; j++)
            {
                p += (double)CoresAndLeaves[j].EdgeNumber / (double)CntLeaf;
                if (loc < p || j == (CoresAndLeaves.Count - 1))
                {
                    // its ok if this is a dup edge, since we want 1 to M0 edges.
                    if (!connected.ContainsKey(CoresAndLeaves[j]))
                    {
                        g.AddEdge(Leafs[i], CoresAndLeaves[j], 1, Math.Max(Leafs[i].Delay, CoresAndLeaves[j].Delay));
                        CntLeaf++;
                    }
                }
            }
        }
    }
}
connected[CoresAndLeaves[j]] = true;

break;

}
}

else
{
while (connected.Count == 0)
{
for (int j = 0; j < CoresAndLeaves.Count && connected.Count < s.LeafM0; j++)
{
    double p = s.LeafFactor * (double)CoresAndLeaves[j].EdgeNumber / (double)CntLeaf;
    if (r.NextDouble() <= p)
    {
        g.AddEdge(Leaves[i], CoresAndLeaves[j], 1, Math.Max(Leaves[i].Delay, CoresAndLeaves[j].Delay));
        CntLeaf++;
        connected[CoresAndLeaves[j]] = true;
    }
}
}
if (s.NewRoutersHavePriority)
{
    CoresAndLeaves.Insert(0, Leaves[i]);
}
else
{
    CoresAndLeaves.Add(Leaves[i]);
}
CntLeaf += connected.Count;
}

#endregion
Appendix B

Network Studies


Study 2: Afghanistan TIGR Overlay All Pairs ICMP survey December 24, 2012

Study 3: Iraq TIGR Overlay All Pairs Traceroute survey October 8, 2008

Study 4: Afghanistan TIGR Overlay Traceroute survey September 4-21, 2013

Study 5: Afghanistan TIGR Overlay User to Server Round Trip AJAX survey August 17, 2013