U.S.N.A.---Trident Scholar project report: no. 286 (2001)

Advanced Shipboard Control Systems

by

Midshipman Jonathan J. Vanecko, Class of 2001
United States Naval Academy
Annapolis, Maryland

____

Assistant Professor Edwin L. Zivi
Weapons and Systems Engineering Department

____

____

CDR Bradley D. Taylor, USNR
Weapons and Systems Engineering Department

____

____

Acceptance for the Trident Scholar Committee

Professor Joyce E. Shade
Chair, Trident Scholar Committee

____

____

Form SF298 Citation Data

Report Date *("DD MON YYYY")* 07052001	**Report Type** N/A	**Dates Covered (from... to)** *("DD MON YYYY")*

Title and Subtitle Advanced shipboard control systems	**Contract or Grant Number**
	Program Element Number
Authors Vanecko, Jonathan J.	**Project Number**
	Task Number
	Work Unit Number
Performing Organization Name(s) and Address(es) US Naval Academy Annapolis, MD 21402	**Performing Organization Number(s)**
Sponsoring/Monitoring Agency Name(s) and Address(es)	**Monitoring Agency Acronym**
	Monitoring Agency Report Number(s)
Distribution/Availability Statement Approved for public release, distribution unlimited	
Supplementary Notes	
Abstract	
Subject Terms	
Document Classification unclassified	**Classification of SF298** unclassified
Classification of Abstract unclassified	**Limitation of Abstract** unlimited
Number of Pages 90	

REPORT DOCUMENTATION PAGE

Form Approved
OMB No. 074-0188

1. AGENCY USE ONLY (Leave blank)	2. REPORT DATE 7 May 2001	3. REPORT TYPE AND DATE COVERED

4. TITLE AND SUBTITLE Advanced shipboard control systems	5. FUNDING NUMBERS
6. AUTHOR(S) Vanecko, Jonathan J.	
7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)	8. PERFORMING ORGANIZATION REPORT NUMBER
9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES) US Naval Academy Annapolis, MD 21402	10. SPONSORING/MONITORING AGENCY REPORT NUMBER Trident Scholar project report no. 286 (2001)

11. SUPPLEMENTARY NOTES

12a. DISTRIBUTION/AVAILABILITY STATEMENT This document has been approved for public release; its distribution is UNLIMITED.	12b. DISTRIBUTION CODE

13. ABSTRACT:

For the new millennium, the U.S. Navy has made a fundamental commitment to drastically reduced crew sizes. Automated systems can reduce crew size significantly, improving the efficiency and effectiveness of a ship. Automation must be continuously available and dependable under all conditions, including combat battle damage. To continue to work during the worst casualty conditions, command and control networks must automatically reconfigure around battle damage. By autonomously routing data around damaged components in an intelligent manner, network fragment healing dramatically improves distributed control system survivability. Dynamic reconfiguration, using network fragment healing, can provide the continuity of communications service that is required aboard U.S. Naval vessels during combat operations. To achieve survivable, distributed communication, an industry proven networking standard, ANSI-709.1 Lon-Works, is extended to military applications. Specifically, the topology of a semi-mesh connection of rings is investigated through availability analysis. Hypercube and semi-mesh topologies are scalable. Furthermore, enhanced network fragment healing algorithms that route message traffic around damaged network components in a more efficient manner are investigated. In the future, these routing algorithms will be evaluated through network simulation, and validated with data obtained from network fragment healing tests performed aboard YP679, an Office of Naval Research test craft that is representative of a small scale combatant.

14. SUBJECT TERMS network fragment healing, network topology, fault tolerant control systems, network modeling	15. NUMBER OF PAGES 90
	16. PRICE CODE

17. SECURITY CLASSIFICATION OF REPORT	18. SECURITY CLASSIFICATION OF THIS PAGE	19. SECURITY CLASSIFICATION OF ABSTRACT	20. LIMITATION OF ABSTRACT

Abstract

For the new millennium, the U.S. Navy has made a fundamental commitment to drastically reduced crew sizes. Automated systems can reduce crew size significantly, improving the efficiency and effectiveness of a ship. Automation must be continuously available and dependable under all conditions, including combat battle damage.   To continue to work during the worst casualty conditions, command and control networks must automatically reconfigure around battle damage.

By autonomously routing data around damaged components in an intelligent manner, network fragment healing dramatically improves distributed control system survivability. Dynamic reconfiguration, using network fragment healing, can provide the continuity of communications service that is required aboard U.S. Naval vessels during combat operations.

To achieve survivable, distributed communication, an industry proven networking standard, ANSI-709.1 Lon-Works, is extended to military applications. Specifically, the topology of a semi-mesh connection of rings is investigated through availability analysis. Hypercube and semi-mesh topologies are also investigated using Monto Carlo analysis to determine if these topologies are scalable. Furthermore, enhanced network fragment healing algorithms that route message traffic around damaged network components in a more efficient manner are investigated. In the future these routing algorithms will be evaluated through network simulation, and validated with data obtained from network fragment healing tests performed aboard YP679, an Office of Naval Research test craft that is representative of a small scale combatant.

Keywords

Network Fragment Healing, Network Topology, Fault Tolerant Control Systems, Network Modeling

Acknowledgments

LCDR Frank Novak, ONR 334, and the United States Naval Academy Trident Scholar program provided significant support that allowed this research to be undertaken. Adept Systems Inc. also provided significant guidance in overcoming the learning curve associated with this project, specifically network fragment healing and availability analysis fundamentals.

Nomenclature

ANSI 709.1-A: LONTalk networking protocol specification

Beta1: A time period directly following a packet transmission

Beta2: The time period of a randomizing slot. A node that has a packet to send waits an integer number of Beta2 slots before beginning packet transmission.

Collision Avoidance: An optional feature of a carrier sense multiple access protocol where a scheme is employed to avoid collisions before they occur.

Collision Detection: An optional feature of a carrier sense multiple access protocol that allows collisions to be detected, and network specific actions to be taken as a result.

CRC (Cyclic Redundancy Check): A type of error checking used to ensure that the message is received in its entirety.

CSMA (Carrier Sense Multiple Access): A specific type of medium access control where the node wishing to transmit first senses the network to see if the carrier is idle. This allows more than one node to transmit on a common medium without centralized arbitration.

CSMA / CD (Collision Detection): A modification to the basic CSMA protocol that allows collisions on the network to be quickly detected. Once a collision is detected the colliding nodes send signals to ensure that all nodes on the network are aware of the collision. The nodes then remain silent for a random amount of time (backoff) before attempting to transmit again.

Differential Manchester Encoding: A way to encode data in binary form that has several advantages. With DM encoding, a clock signal is passed along with the information so that receiving nodes can easily synchronize on the message. DM encoding has no DC component and can be transformer coupled.

Framing: A method of packaging data whereby each packet is broken down into several fields containing specific information.

ISO: International Standards Organization.

Lon-Works: A networking technology developed by Echelon Corporation that is very robust and ideal for distributed control.

MAC (Media Access Control): The method by which a message is placed on the physical network.

Network Fragment Healing: Process by which messages can be routed around physical network damage.

Node: A connection point on the network where the processing of data occurs.

OSI (Open Systems Interconnection) Model: A framework created by the International Standards Organization to ease the exchange of network information.

P-Persistent CSMA: The type of collision avoidance used by the Lon-Works network. A prediction of future message traffic is generated to dynamically adjust the number of Beta2 slots that nodes randomize over in order to decrease collisions.

Packet: The information that is placed on the network. A packet contains data that is sent along with other information the receiving nodes require.

Priority Slot: A slot of Beta2 duration reserved for the transmission of priority packets. These slots precede the randomizing slots and ensure that urgent messages are placed on the network before messages of normal priority.

Protocol Data Unit: A format for transmitting data between software entities. A protocol data unit is an element that consists of control information and data. Each layer of the OSI model has its own specific protocol data unit (PDU) such as a Network PDU, Physical PDU, etc.

Router: A device that transfers data between source and destination subnetworks. In this context, a router consists of two nodes placed back to back.

Slot: A time period during which a node may attempt to transmit a packet.

TCP/IP: A very robust protocol that provides communication between pairs of processes on a network and allows IP addresses, or network connections, to be declared. This allows messages to be transmitted between multiple networks and is commonly used as the protocol for the Internet.

Token Ring: A type of protocol used in control applications. Each station on the network is allowed to transmit only while holding a token, which ensures reliable communications because only one token is used for the ring. After the token is held by a station for a designated amount of time it relinquishes possession of the token to another station, and that station is then allowed access to the network.

Technical Nomenclature

Beta1: > [*1bit time + (2 ** $t_p + t_m$*)*]

Beta2: > [*2 ** $t_p + t_m$]

BL: An estimate of current message backlog. $1 \leq BL \leq 63$

CRC: For Lon-Works the cyclic redundancy check is generated by the polynomial $X^{16} + X^{12} + X^5 + 1$.

CT: A time base used to generate Beta1, Beta2, and the preamble. Depending on media supported, each node will support one of the following values for CT: {600ns, 1.2us, 4.8us, 9.6us}

D_{mean}: The average randomizing delay between packets. Because T_d is uniformly distributed $W_{base} = \frac{(W_{base})}{2}$.

T_d: Random delay generated on the interval *(0...(BL**W_{base}*)-1)*. T_d is an integer number of Beta2 randomizing slots.

t_m: Time period between an idle channel detection and the first transition of the outgoing data packet.

t_p: Physical propagation delay defined by media length.

w: The size of the randomizing window. $w=(BL*W_{base})-1$

W_{base}: Number of randomizing slots in the base randomizing window.

Table of Contents

Introduction

The reliance of the U.S. Navy upon systems automation will steadily increase as technology advances and crew sizes are reduced. To support a smooth transition of crew functions to dependable automation, a method to ensure reliable data communications must be developed.

The implementation of dependable automation aboard U.S. Navy ships will directly support the removal of sailors from repetitive tasks, and allow them to be placed into decision-making positions. Furthermore, many safety benefits are possible with the automation of engineering and damage control. Specifically, an automated sensor network gives sailors a clearer picture of what is happening throughout the ship during casualty conditions and allows them to employ damage control teams more effectively. Although these systems can greatly improve the efficiency of a ship, they must provide continuous real time availability during the worst casualty conditions.

Network fragment healing dramatically improves distributed control system survivability by routing data around damaged network components. Moreover, network fragment healing has the intelligence to autonomously sense damage, and choose an alternate path. By doing so, network fragment healing is able to provide the continuity of communications service that is required aboard a U.S. Naval vessel during combat operations.

Recently, the Office of Naval Research (ONR) has shown interest in this technology because of the possibilities it could have aboard naval vessels. ONR has implemented a small scale Lon-Works network aboard YP 679, a dedicated test craft, and is currently conducting preliminary research. The YP craft are used at the United States Naval Academy for Midshipmen training and are representative of a small-scale combatant. One of the ONR team members, Adept Systems Inc, has pioneered the use of network fragment healing with Lon-Works networks. This research extends network fragment healing for application on a full-scale combatant through topology and algorithm investigation.

Background

With the decision to shift manpower intensive functions to automation aboard U.S. Navy ships, the realization that these functions require reliable data communications surfaced. Formerly, reliable data communications came in the form of redundancy. For example, DDG-51 flight I ships use the data multiplex system (DMS), which consists of five redundant backbone cables [1]. This system provides only limited survivability by means of redundancy and is very expensive. Furthermore, because the network is implemented in a hierarchical fashion, the system is vulnerable to damage.

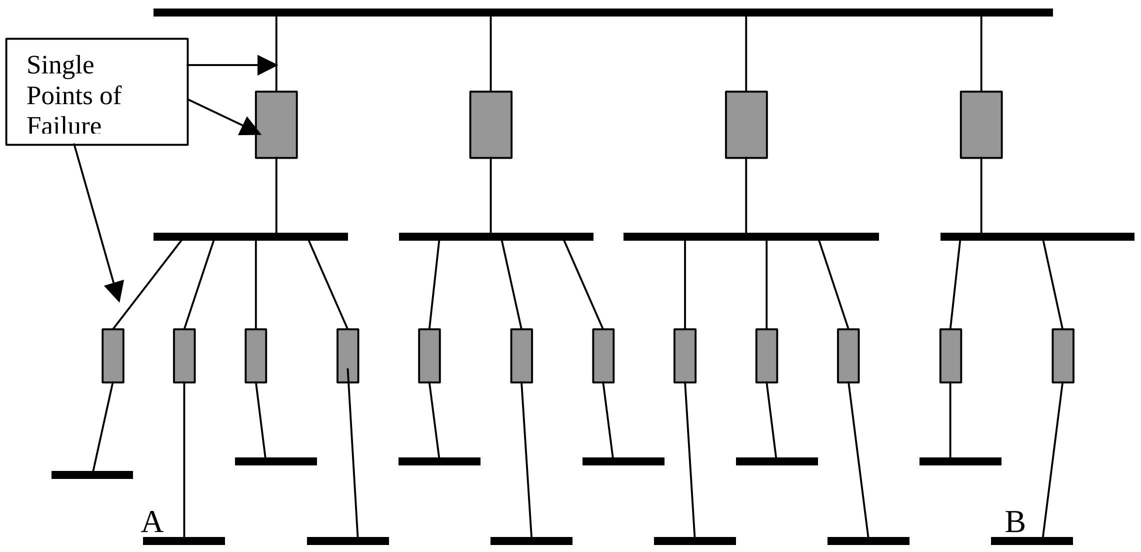


Figure 1 – Hierarchical Control System [7]

If any of the controllers are lost, then the performance of the system is greatly degraded or completely lost. Figure 1 is a typical hierarchical control system and readily shows the problems associated with this type of topology in survivable control applications. If any damage is sustained, and a breakage occurs, then network components have no way of regaining communications and are segregated, or fragmented, from other parts of the network. Figure 1 shows that nodes A and B have only one path by which to communicate. If this path is damaged, there is no other route messages can travel between the nodes. This lack of survivability is characteristic of hierarchical networks. By using a more distributed approach greater reliability can be obtained without massive redundancy.

One method to avoid the problem of a single point of failure is to implement a Component Level Intelligent Distributed Controls System or CLIDCS. In a CLIDCS, nodes are organized in a peer fashion that ensures minimum degradation to network performance due to node or link failure. Furthermore, a CLIDCS has imbedded intelligence at each node that allows it to cooperate with other nodes, as well as operate in their absence. This nodal level intelligence gives a CLIDCS a distinct advantage over other types of control networks [2].

Another way to improve communications reliability is through a process called network fragment healing. Network fragment healing is, as the name implies, a way to reconfigure or heal the network when physical damage is sustained and a communication path is lost. Using a topology that supports redundant paths, properly configured routers can create secondary routes to reconnect nodes that have been isolated by faults. This greatly improves data communication reliability since many different paths are between any two points on the network.

In order for the sensors, actuators, and other commercial equipment to work together, a control algorithm for the network must be developed. In keeping with the Navy's decision to use Commercial Off The Shelf (COTS) technology, a cost effective, industry proven, distributed control technology, Lon-Works, was chosen for network fragment healing and is the foundation of this research. Lon-Works is presently used in thousands of applications in the industrial and transportation sectors and is a recognized open standard. Although several other networking packages are currently available, Florida Atlantic University has demonstrated the capability of Lon-Works to support distributed machinery control. Lon-Works is an industrial strength distributed control system development environment that uses Neuron C, ANSI C with three network specific extensions. The challenge that users face is that Lon-Works does not support redundant network paths that would be needed if the semi-mesh of rings topology is to be implemented. Recently, Florida Atlantic University and Adept Systems tackled this problem by writing control algorithms that mask the redundant paths from the LONTalk Protocol. Should a fault occur, this redundancy is selectively unmasked allowing the system to function properly.

Florida Atlantic University and its commercial partner, Adept Systems, have been investigating the use of intelligent self-healing networks for several years and have made many profound discoveries. Recently, great success has been achieved by applying this knowledge to autonomous underwater vehicles. Using Lon-Works industrial control technology, Florida Atlantic has created network fragment healing algorithms for small-scale networks. The objective of this research is to extend these results to more complex military systems, supporting the next generation of surface combatants. This is made feasible by studying possible topologies that will optimize network fragment healing with regards to time for reconfiguration and path redundancy. By creating a computer simulation that models the Lon-Works network aboard YP 679, a network model that can help extend Lon-Works to higher order systems can be created. Furthermore, the data generated by the computer simulation can be correlated to actual Lon-Works network data. This data was collected by Adept Systems during tests aboard YP 679 and consists of network message traffic during times of normal network operation as well as during network reconfiguration. By correlating this data to the network simulation, results can be validated, making the research more meaningful.

Basic Network Calculations

Some of the basic calculations needed for any network include the latencies due to the propagation of a signal through the medium. The speed of light is 299,792,458 m/s or about 300 million m/s. For a distance of 10m, which is half the circumference of a ring on the YP network, this corresponds to a propagation delay of about $3.3*10^{-8}$ s or 33ns. Again using the YP as a base model we see that the transmission rate is 78kbits/s and the period is $1.28x10^{-5}$s. Dividing the propagation delay by the transmission rate gives us a value of .0025, which is the fraction of the transmission that will be lost at the beginning and end of each high or low of the transmitted signal. Because the signal that is being transmitted is a binary signal, the loss of ¼% of a signal does not affect reception because the high or low signal will still be present for 99.75% of the signal.

Determining the percentage lost on a 1.5Mbit/s network, which is about 20 (19.23) times faster, shows that there will be a 5% loss on each transition of the encoded signal. This still is not enough signal attenuation to cause data loss since the system only needs to determine a high or low and can do so with only part of the total signal.

OSI ISO MODEL

In order to facilitate a meaningful exchange of information among different users, the International Standards Organization (ISO) created an Open System Interconnection (OSI) model with seven primary layers, as shown in Figure 2. Each layer defines a subsystem that can logically be separated from the others.

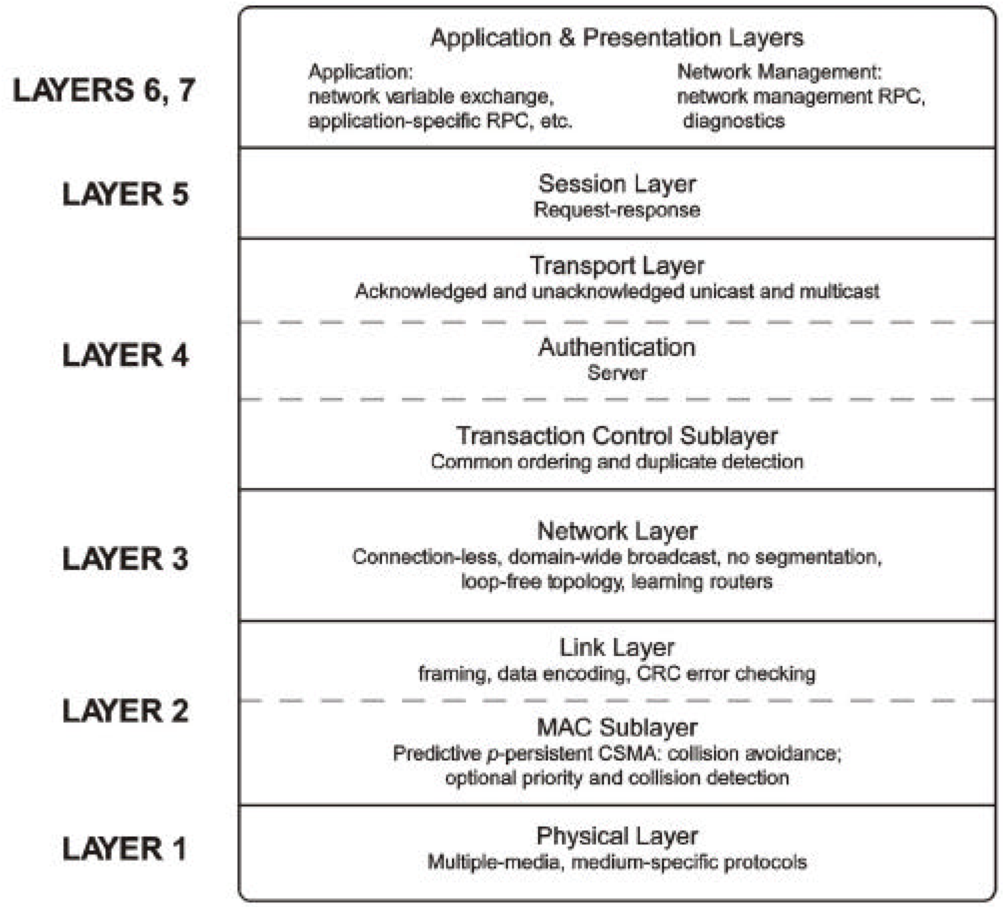

LAYERS 6, 7
Application & Presentation Layers
Application: network variable exchange, application-specific RPC, etc.
Network Management: network management RPC, diagnostics
LAYER 5
Session Layer
Request-response
LAYER 4
Transport Layer
Acknowledged and unacknowledged unicast and multicast
Authentication
Server
Transaction Control Sublayer
Common ordering and duplicate detection
LAYER 3
Network Layer
Connection-less, domain-wide broadcast, no segmentation, loop-free topology, learning routers
LAYER 2
Link Layer
framing, data encoding, CRC error checking
MAC Sublayer
Predictive p-persistent CSMA: collision avoidance; optional priority and collision detection
LAYER 1
Physical Layer
Multiple-media, medium-specific protocols

Figure 2 – ISO OSI Model [3]

One distinct advantage to an OSI model is that each level of the network is clearly defined, allowing different users to understand how a specific layer of the protocol stack is implemented. Furthermore, a layered framework has the ability to be upgraded one layer at a time, saving costs, and allowing single layers to be replaced by hardware and software from an outside network vendor [8]. This research is concerned primarily with the lower three layers because of their involvement in placing messages on, and receiving messages from the physical media.

Layer 1 – Physical Layer

The first layer, the physical layer, provides the means to place messages on the network based on the type of media being used. One important aspect of the physical layer is the actual connection of points, or nodes, along the network. When a network is to be used for control, the topology, which governs how the network is wired, plays an important role. Figure 3 shows the most basic type of network configuration, a linear bus.

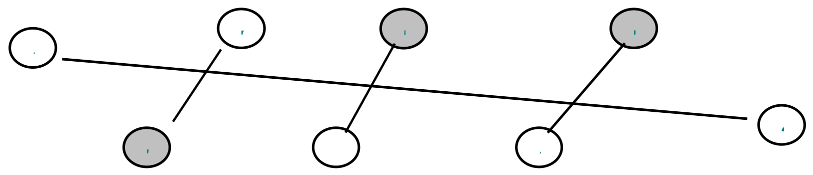


Figure 3 – Linear Bus Topology [7]

A linear bus is a single communication path that nodes can be attached to. Although this topology is very cost effective, it is not very robust. A single break along the path at any point divides the bus and isolates one half from the other, causing a loss of communications. A simple way to increase survivability is to bend the bus around and form a ring, as in Figure 4.

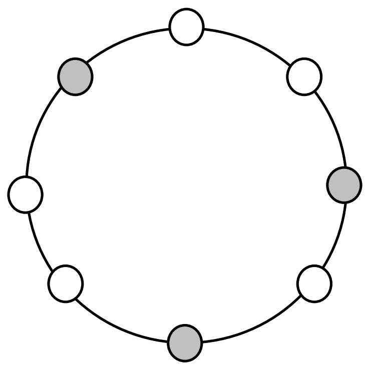


Figure 4 – Ring Topology [7]

With very little extra cost, the network has now doubled its availability by allowing data to flow in either direction. When a single break occurs, nodes will not be isolated from the rest of the system.

To further improve survivability more complex topologies can be implemented.

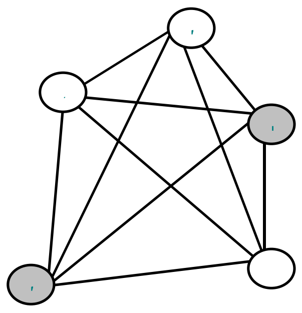


Figure 5 – Mesh Topology [7]

In a mesh topology, shown in Figure 5, the availability of the network actually increases when nodes are added. With each additional node, the number of paths between any two points increases at nearly an exponential rate. Although this topology is extremely robust, it is very complex and can be extremely expensive to implement due to wiring costs.

One way to achieve a higher level of availability with a low network cost is by implementing a topology such as a semi mesh of rings, as seen in Figure 6.

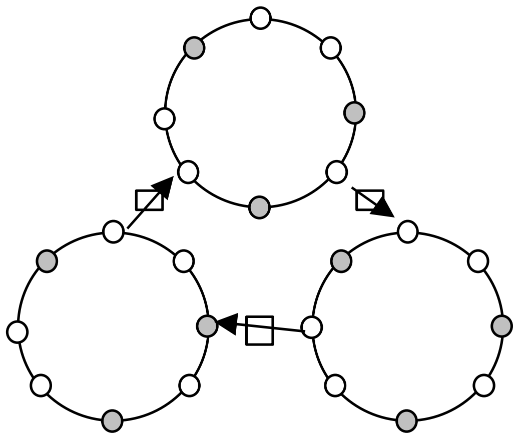


Figure 6 – Mesh of Rings Topology [7]

A semi mesh of rings is the configuration currently used on the Office of Naval Research's YP 679 test bed for network fragment healing, and allows for relatively inexpensive implementation of a very robust network topology. This topology allows for a greater number of nodes on the network because the number of mesh paths is now only dependant on the number of rings placed on the network, and not the number of nodes. By placing nodes onto rings and then implementing a very robust topology with the rings, cost is kept down without any serious sacrifice in performance. Furthermore, the loss in performance is small compared to the decrease in cost and complexity. Therefore, this topology is well suited to networks with many nodes.

A hypercube of rings is very similar to that of a semi mesh but differs in the way that the rings are interconnected. With a semi mesh of rings, there is no set pattern for connecting the rings together where as, a hypercube of rings would mirror a standard hypercube, but instead of nodes at each of the vertices there would be a ring of nodes. Hypercubes are used in industry because of their robust design and low cost relative to a full mesh topology. To implement a full mesh topology the number of connections increases in an exponential fashion while a hypercube increases linearly. Furthermore, hypercubes have multiple paths between points on the network that make it a good choice for network fragment healing. The implementation of a hypercube of rings must be investigated and compared with a semi-mesh of rings topology to ensure that the best topology is chosen for large-scale systems.

Layer 2 – MAC / Network Layer

MAC

The second layer of the OSI model is the data link layer. In the ANSI 709.1 Lon-Works protocol, this layer is divided into two sections, the media access control layer (MAC), and the link layer. The MAC layer plays a very important role and is needed because many nodes must vie for access to the physical media in an efficient manner. There are many different types of algorithms that can be used to place messages on networks but the goal of all of them is to place data on the network as quickly, and reliably as possible without collision. The ANSI 709.1 protocol uses an algorithm called predictive p-persistent carrier sensed medium access / collision avoidance (CSMA/CA) with optional collision detection to ensure reliable data communications [3,7].

Framing

A second function of the data link layer is to ensure that outgoing messages are correctly framed. By framing messages, each important section of the message, including the priority, address, and backlog, is always placed in the same location, allowing a receiving node to quickly find the location of each piece of information.

Data Encoding

The data link layer also provides the means by which information is encoded and prepared for transmission. The YP 679 Lon Works network uses a communications transceiver that makes use of Differential Manchester encoding, one of several types of binary encoding [3]. The binary system, which is used almost exclusively in computer software and hardware applications, is characterized by a signal that is either high or low. This makes recovery of the signal from the media easier than other forms of transmission, especially in the presence of noise. One important advantage of Differential Manchester is that the transformed binary signal has no direct current (DC) component, and can be coupled. Also, a Differential Manchester signal has a mid bit transmission that can be used to convey timing information along with the data being transmitted [8]. By encoding timing information along with the transmitted signal, the synchronization of clocks and decoding of the message are simplified. A logic one is produced when the polarity of the leading signal is opposite the polarity of the trailing signal, and a logic zero is produced when the polarity of the leading and trailing signals is the same.

CRC

Finally, the data link layer provides cyclic redundancy checking (CRC) to ensure that the message is free of errors. The sending node computes the CRC on each outgoing packet with the generating polynomial $X^{16} + X^{12} + X^{5} + 1$ and places the remainder from the calculation in the header of the outgoing packet. The receiving node then calculates the CRC on the incoming packet and compares its remainder to the remainder in the header. If the two remainders match, then the incoming message is error free. The CRC used in the Lon-Works network provides detection of error bursts up to 16 bits in length and 99.9% of error bursts greater than 16 bits can be detected as well [8].

Layer 3 – Network Layer

Routing

The final layer that this project concerns itself with is the network layer. The main function of this layer is to allow messages to traverse the network by way of routing. Routers are either configured or can learn where different addresses on the network are by observation. When a packet is received the router consults its routing table, a list that contains address locations relative to the router, and forwards the message along in the direction of the desired address. In the case of the ANSI 709.1 Lon-Works protocol, a router also has the ability to optionally forward priority messages ahead of messages of normal priority. Moreover, Lon-Works routers allow messages sent from nodes without access to priority slots, but with the priority bit set, to be forwarded as priority messages if the router has a priority slot that is currently empty. If the message has the priority bit set but there are no available priority slots, the message will be forwarded in the normal manner. Another function of the network layer is to support unicast, multicast, and broadcast messages. By sending a message to multiple nodes simultaneously the time required to send multiple redundant messages is eliminated.

Protocol Operation

Background Information

Before Lon-Works can be extended to higher order systems with more nodes, the lower layer implementation of the Lon-Works protocol must be understood. This preliminary network analysis is compiled based on the information found through literature searches of white papers written by the Echelon Corporation, analysis of the LONTalk protocol specification ANSI 709.1-A, Electronic Industries Alliance specification, and other information obtained through Echelon's web site, http://www.echelon.com/ [3,5,6,7].

With the release of several white papers on the LONTalk networking protocol and a reference implementation by Adept Systems Inc, the LONTalk Protocol has transitioned from a proprietary specification to an open networking standard available as ANSI 709.1-A. Much of the background research contained in this paper seeks to give a concise picture of the lower level operation of ANSI 709.1-A. This information is not contained in any one source, which makes the comprehension of ANSI 709.1-A difficult without in depth study. This report conveys the information contained in ANSI 709.1-A and other previously proprietary specifications in a single source that allows the reader an in depth look at the LONTalk Protocol.

In order for the Lon-Works network to be fully understood, several key definitions and diagrams must first be explained. This analysis concerns itself only with the lower levels of the protocol stack because these levels describe how a node places a packet of data onto the physical media.

Echelon implements this in the MAC (Media Access Control) layer with a collision avoidance protocol known as predictive p-persistent CSMA, (Carrier Sense Multiple Access). CSMA is preferred over other types of MAC because it allows a truly distributed real time control system to be implemented. In other types of MAC, such as a token passing ring, a central control node or circulating token grants other nodes access in a fair manner. For example, token passing rings work well for networks that do not require continuity of service. Token passing rings require complex procedures to adapt to changes in network topology that may occur when nodes are added or damage is sustained. In these cases, a token passing ring must restart, wait for each node to shake hands with its neighbors, and then create a new token before allowing access to the medium. Moreover, a token ring network has the luxury of failing to a predetermined state to diagnose any problems. The time associated with this reconfiguration is unacceptable for a shipboard real time control system.

Carrier sense multiple access allows topology changes because there is no centralized control for network access. By allowing each node to control its access to the medium, a truly distributed control architecture can be implemented. Furthermore, CSMA allows many nodes to share a common medium with a low probability of a collision. On a Lon-Works network there is only a 4% collision rate at 98% channel capacity [2]. Echelon has implemented a variation of the traditional CSMA protocol in Lon-Works that contains several advantages. First, Lon-Works dynamically adjusts the number of Beta2 slots over which the nodes contend in order to decrease the likelihood of a collision. By dynamically adjusting the number of Beta2 slots a normal distribution of network delay times is obtainable. Some CSMA protocols, such as Ethernet, have the problem of large increases in delay time as more and more users try to contend for the media. This can easily be seen on a home pc that is connected to the Internet during times of high network usage. Its speed compared to the same Internet connection during off peak hours is greatly reduced. Lon-Works, on the other hand, minimizes degradation in network performance as network traffic increases. This makes Lon-Works especially well suited for distributed real time control.

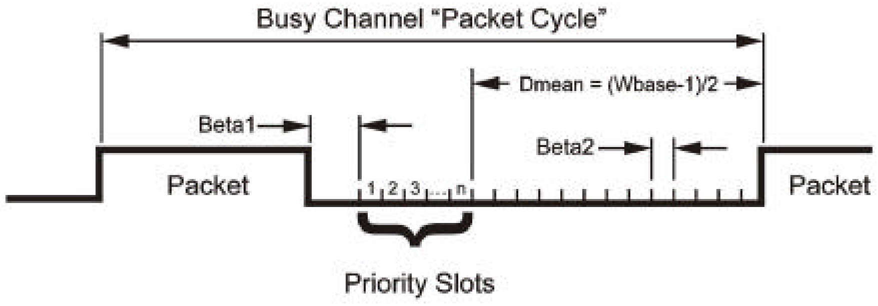


Figure7-Busy Channel Packet Cycle [3]

In order to understand the predictive p-persistent CSMA algorithm key components of a packet cycle must be introduced. Figure 7 is important because it shows graphically the parts of a typical busy channel packet cycle. Beta1 is the period immediately following the end of a packet and is used to ensure that communications have ended before attempting to place another packet on the network. Although Beta1 can take on many different values based on media type and other tuning parameters, it will always be constrained by *Beta1>1bit time+(2 ** t_p + t_m*)* where t_p is the physical propagation delay and t_m is the detection and turn around delay in the MAC sub layer. Beta2 is a randomizing or contention slot whose value is equal to *CT * (40 + 20 * v)* where CT is a media specific time base between 600ns and 9.6us and v is a tuning parameter on the interval 0…255. Beta2 is also defined as always being greater than *(2**t_p+t_m). Priority slots, which each have a width of Beta2, are the final component of the packet cycle. Nodes with important functions may be given access to priority slots to speed up delivery of time critical messages. Nodes may be assigned distinct or shared priority slots, but the maximum number of priority slots will not exceed 127. In most cases, each node will be assigned its own unique priority slot because contention for a priority slot results in a collision. If two nodes were assigned the same priority value and tried to transmit a packet there would be no way to avoid this collision since priority slots are not contended for. One of the few cases where nodes would be assigned the same priority value is when a network management tool is being used. Since the network management tool would never ask two nodes to transmit at once, there is no chance of more than one node attempting to transmit in the same priority slot. [3,5,6,7]

One point that must be understood about the division of the busy packet cycle into slots is that each slot, although visualized as having a specific width, is actually defined by a time value. Lon Works CSMA is implemented so that there are an integer number of time delay slots over which the nodes randomize. This means that if a node wishes to transmit in the 3rd priority slot it must first wait a Beta1 time in seconds and then also wait for 2*Beta2 time slot seconds to pass [3].

Now that the basic definitions have been covered, the operation of the MAC layer can be discussed in greater detail. Media access procedures depend on the state of the network. In order for layer two of the protocol stack to know the network state, however, there must be communication with the physical layer. Lon-Works does this through the use of several requests as show in Figure 8. Data is shared among layers of the network by simply requesting it. The link layer, for instance, after verifying that the message is complete, would call the Frame_OK primitive, or function, in order to pass the back log information from the received message to the MAC sublayer. By simply calling a function, layers of the protocol stack can pass information quickly and easily.

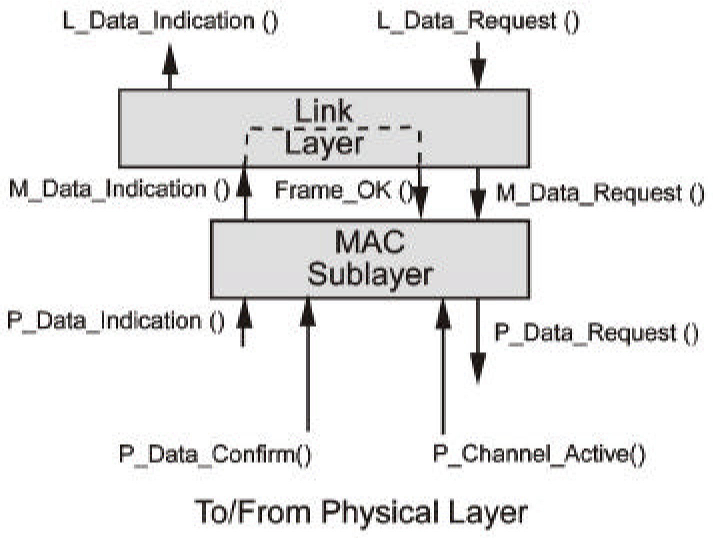


Figure 8 – Physical Layer Interaction [3]

One advantage of a CSMA based MAC is that it may be implemented using a full duplex method, as Lon-Works is, so that listening and sending are separate, concurrent actions.

Node Receiving

Each node on the Lon-Works network has the ability to monitor the network at all times for transmissions. Figure 9 shows that upon detection of a transmission, the node will synchronize on the pre-amble of the packet and determine who the packet is for. If the packet is for the node, then it will process the packet and deliver the message to the appropriate destination within the node. If the packet is not for the node, synchronization will be maintained for at least the remainder of the message as well as Beta1, all of the priority slots, and W_{base} randomizing slots [3].

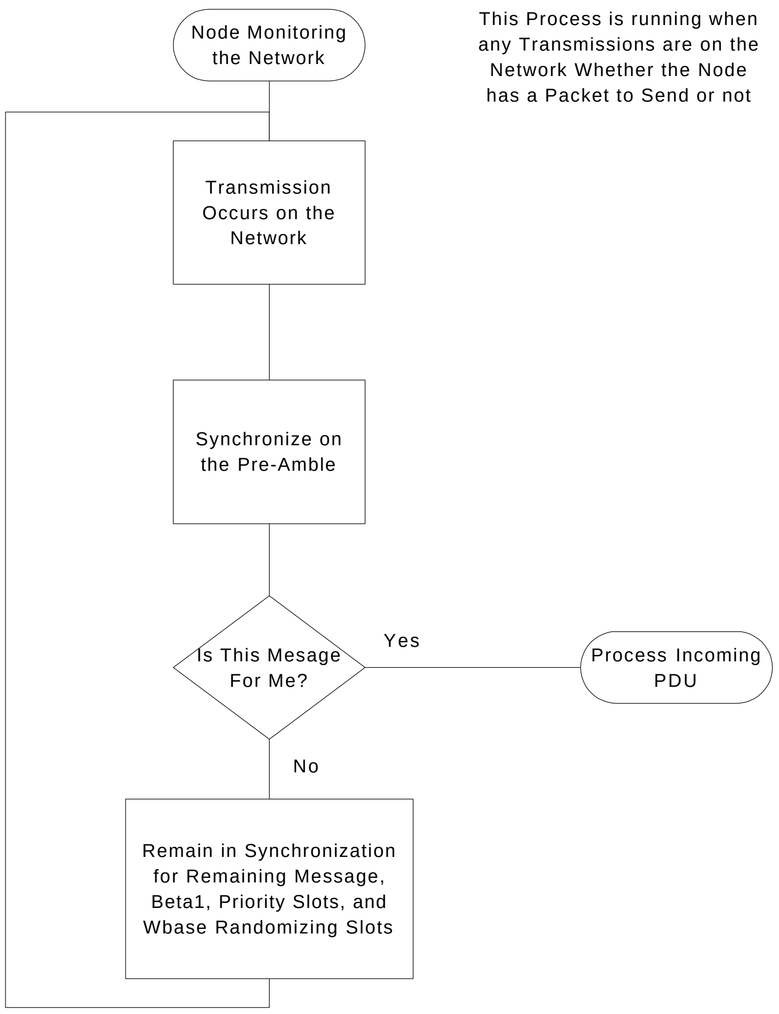

Node Monitoring the Network
This Process is running when any Transmissions are on the Network Whether the Node has a Packet to Send or not
Transmission Occurs on the Network
Synchronize on the Pre-Amble
Is This Mesage For Me?
Yes
Process Incoming PDU
No
Remain in Synchronization for Remaining Message, Beta1, Priority Slots, and Wbase Randomizing Slots

Figure 9 – Node Receiving

Network Busy With Message To Send

Figure 11 depicts the logical process that each node performs to send a message when the network is busy. In this case, the node will first wait for the end of the current message transmission and continue to wait for one Beta1 slot to ensure that the network is idle. Since all nodes are synchronized on the previous packet cycle, each node will end its Beta1 slot at essentially the same time. This synchronization among nodes helps to eliminate the chance of a collision. Throughout the Beta1 period, the node must be listening to the network to see if any other transmissions are occurring on the network. If a transmission is detected, the node will try to receive the incoming packet and will hold its packet until the new transmission has terminated. After sensing an idle Beta1 period, the node will attempt to send the packet as either a priority or normal packet. This is determined by checking the L2Hdr, Figure 10, which contains a 1 bit priority (Pri) field that tells the layer how this message is to be sent. The L2Hdr is an 8 bit section of the Link Protocol Data Unit/MAC Protocol Data Unit (LPDU/MPDU) frame that contains information necessary for the complete transmission of a packet. The L2Hdr includes the delta back log (BL) which contains information on how much message traffic will be generated by the transmission. The cyclic redundancy check which is computed over the entire network protocol data unit (NPDU) and is generated using the International Consultative Committee for Telephone and Telegraph (CCITT) CRC-16 standard polynomial of $X^{16}+X^{12}+X^5+1$ is also contained in the L2Hdr [3,8].

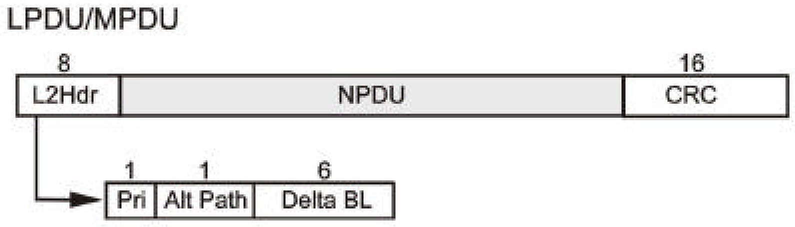


Figure 10 - LPDU/MPDU [3]

The advantage to using a protocol data unit structure is that each node knows specifically where to locate information that is needed to process the packet. Appendix C contains the complete protocol data unit summary for each layer of the OSI model and shows the location of each piece of information located within a Lon-Works packet.

If the packet to be sent is a priority message, the node must first determine if the node sent a priority packet during the previous packet cycle. This is done to ensure that other nodes will not be barred from transmission by a node with several high priority packets queued to send. If a node did not send a priority packet during the previous packet cycle it will simply wait for the correct priority slot, and, if the network is still idle, send. In the case where a priority packet was sent, the node must carry out a fairness algorithm. The node will wait for all priority and randomizing slots to expire and if the medium is still idle send out its packet. If a packet is detected on the network while waiting for the priority and randomizing slots to expire, then the node will try to receive the transmission. When the transmission ceases, the node will carry out the normal Beta1 period, wait, and then normally attempt to access the appropriate priority slots.

For a packet of normal priority to be sent, a different route is taken to place the message on the network. First, the node must wait for all of the priority slots to expire. The node will continue to wait a random delay T_d that is on the interval *(0…(BL*Wbase)-1).* BL is the integer backlog estimate between 1 and 63, and Wbase is the number of randomizing slots in the base randomizing window. The BL is calculated by each node through the examination of the headers that pass through the network. Each L2Hdr field contains information about the amount of message traffic that will be generated as a result of packet delivery. Because most Lon-Works messages require acknowledgements, this is an effective way to predict future message traffic. By dynamically adjusting the BL value, each node decreases its likelihood of a collision upon transmission. The value T_d is a delay time in seconds that corresponds to a randomizing slot. If when T_d has expired, the network is still idle, then the node may send out a packet. Figure 11 shows the flow chart for a node that wishes to send a packet while the network is busy and allows the decision process to be visualized [3].

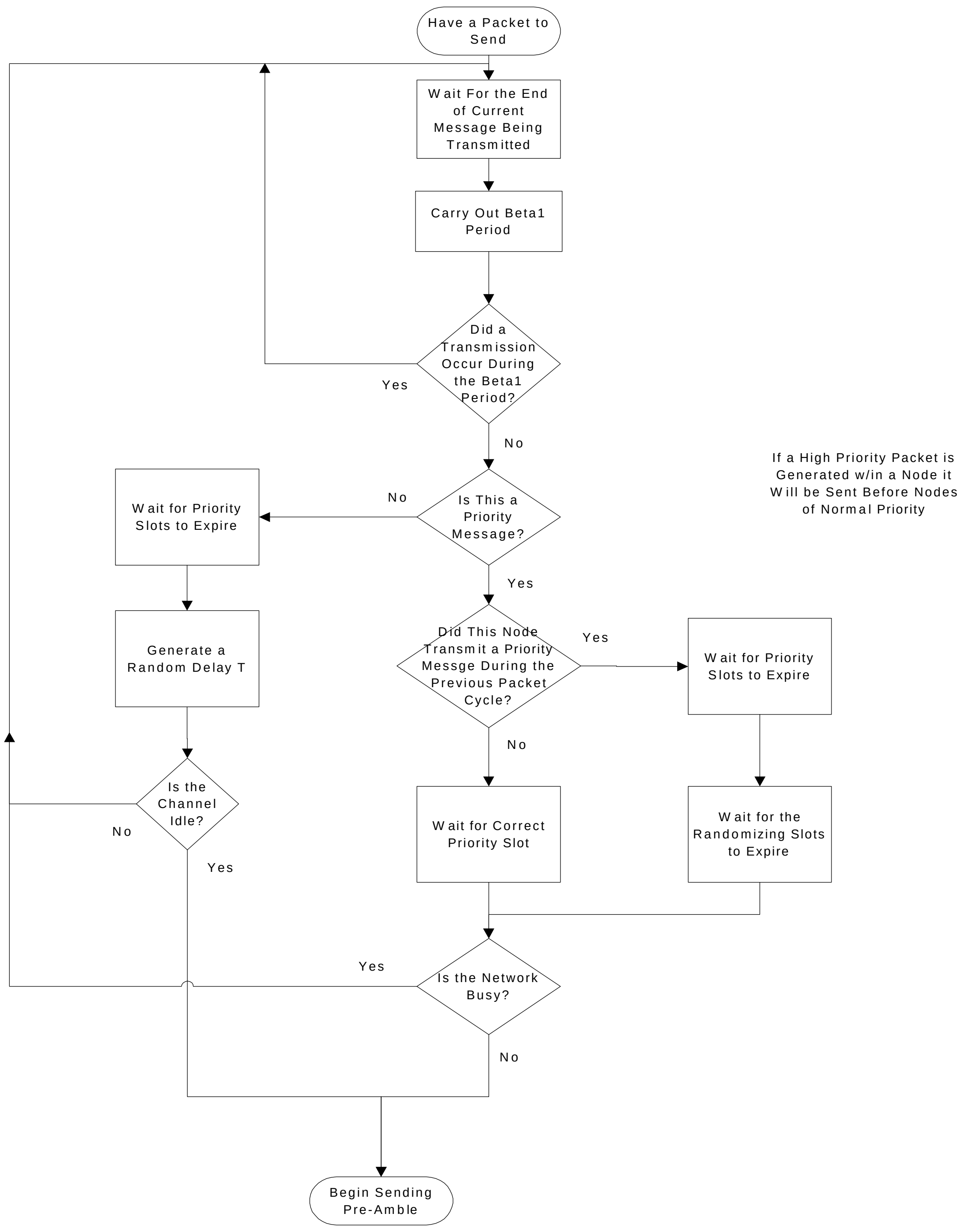

Have a Packet to Send
Wait For the End of Current Message Being Transmitted
Carry Out Beta1 Period
Did a Transmission Occur During the Beta1 Period?
Yes
No
Is This a Priority Message?
No
Yes
Wait for Priority Slots to Expire
Generate a Random Delay T
Is the Channel Idle?
No
Yes
Did This Node Transmit a Priority Messge During the Previous Packet Cycle?
Yes
No
Wait for Priority Slots to Expire
Wait for Correct Priority Slot
Wait for the Randomizing Slots to Expire
Is the Network Busy?
Yes
No
Begin Sending Pre-Amble
If a High Priority Packet is Generated w/in a Node it Will be Sent Before Nodes of Normal Priority

Figure 11 – Node With Message to Send – Busy State

Network Idle With Packet To Send

Another media access case occurs when a node is trying to send out a packet and the network has been idle for a long time. This case is shown in figure 12. Because Lon-Works obtains synchronization information from packets that are placed on the network, synchronization cannot be maintained if the network has been idle for longer than [*Beta1+priority slots+Dmean*]. Thus a different access method must be used. In this case the node will carry out the Beta1 period and if no transmissions are detected send out its message immediately. If a transmission is detected during the Beta1 period the node will send the message using the busy network case [3].

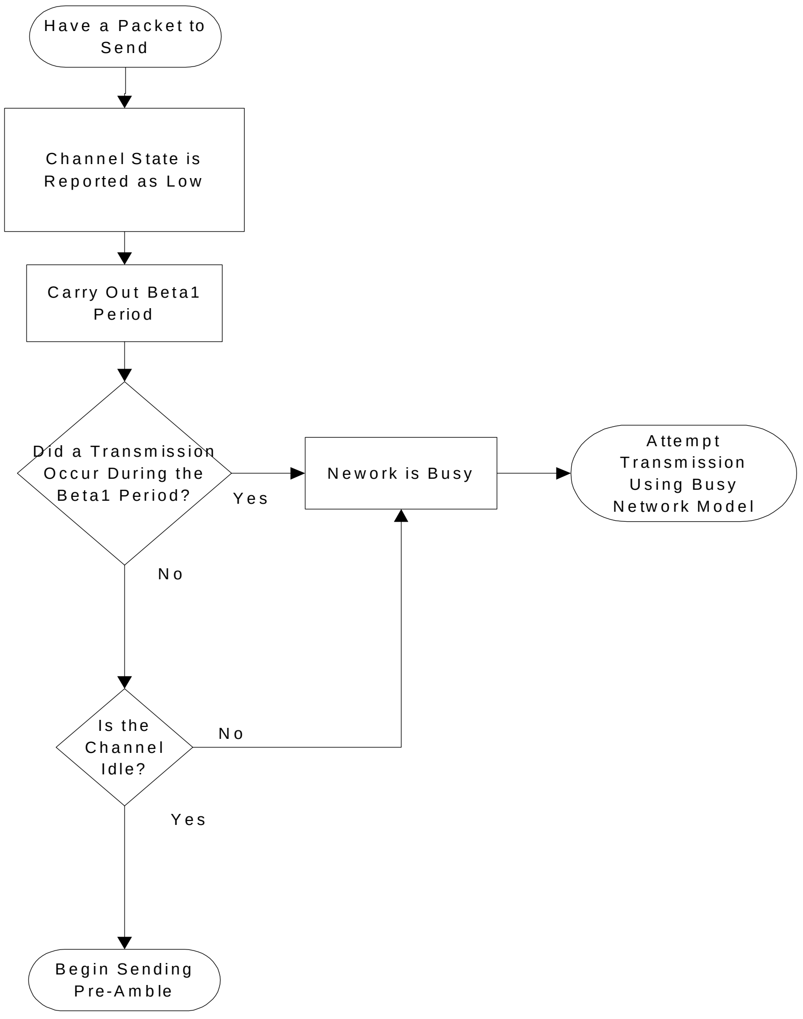

Have a Packet to Send
Channel State is Reported as Low
Carry Out Beta1 Period
Did a Transmission Occur During the Beta1 Period?
Yes
Nework is Busy
Attempt Transmission Using Busy Network Model
No
Is the Channel Idle?
No
Yes
Begin Sending Pre-Amble

Figure 12 – Node With Message to Send - Idle State

Reliability Analysis

In order for a topology to be suitable for use aboard a naval combatant it must be cost effective, extremely survivable, and also highly scalable. A topology that is scalable is one that may be implemented without network degradation on both networks with few nodes and on networks with thousands of nodes. By assigning reliability values to nodes, routers, and links, each network reliability calculation can give insight into the effectiveness of a particular network topology.

Linear bus reliability was determined by assigning a reliability of .5 to each node and physical link. Figure 13 shows the rapidly decreasing reliability of a linear bus as the number of nodes increases. This is highly intuitive because as you increase the number of nodes, the probability of a failure in some part of the network increases, thus decreasing your overall reliability.

A ring topology is merely a bus topology that is looped around to allow transmission to occur in both directions. This is a cost effective improvement to a linear topology because you are only increasing your cost by one connection, while doubling your survivability. This increase in survivability is readily observable in Figure 13, which contains a plot of both a linear bus and ring topology. It is apparent that the ring topology has nearly twice the reliability of a bus topology for the same number of nodes

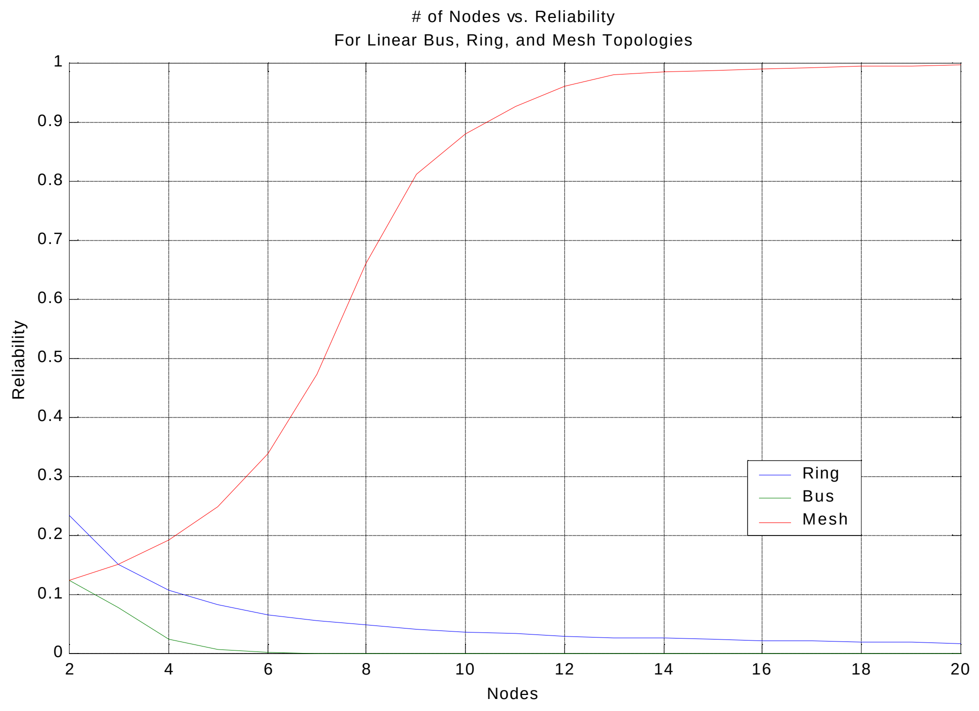


Figure 13 – Reliability vs. Number of Nodes

In a mesh network, each node is connected to every other node in the network and the overall reliability of a mesh topology increases as nodes are added to the network. Although meshes are extremely survivable and reliability increases with additional nodes, they are not scalable since the number of mesh connections is $(nodes - 1)!$. This is not a problem for networks with few nodes, but when a network with twenty nodes is considered the problem becomes obvious. For example, to add the 20th node to a system, you must add nineteen additional paths. Furthermore, with twenty nodes there are 121645100408832000 connections, which is unreasonable for implementation.

One way to obtain a practical, robust, network topology is to place nodes onto rings and then create a partial mesh of these rings. This allows a network to support many nodes without adding nearly exponentially increasing numbers of connections. Furthermore, it can be seen from Figure 14 that you can place several nodes on each ring without a substantial degradation in network performance. This is evidenced by comparing the reliability value of a full mesh topology with five nodes to the reliability of a mesh of six rings having five nodes on each ring. The full mesh topology only has a reliability of 0.158 greater, even though it has twenty-five fewer nodes. The mesh topology does, however, increase in reliability at a much faster rate, although the small number of nodes possible on the mesh network overshadows this advantage.

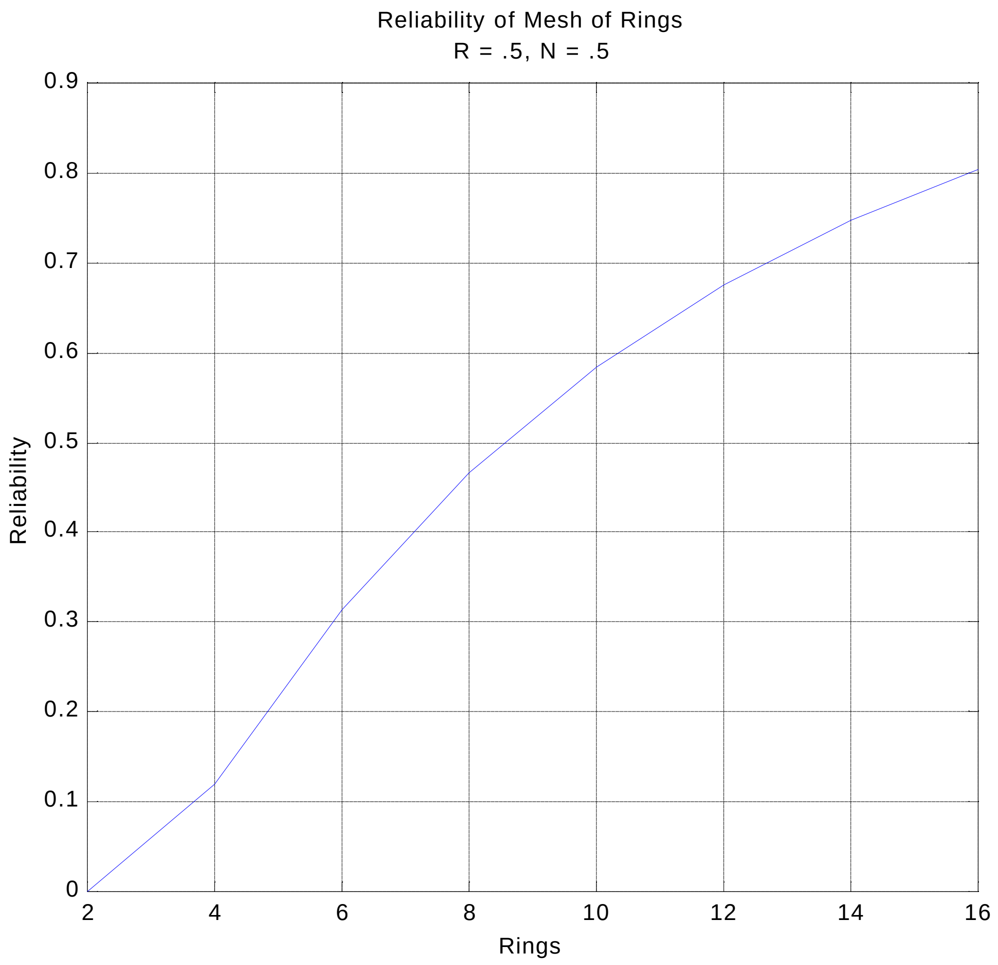


Figure 14 – Reliability vs. Number of Rings

Hypercube

In small networks with very few nodes the topology of the network has little effect on the reliability of message transmission. Since there are so few points of failure, due to the small number of physical connections, almost any topology will provide the redundancy and structure needed for reliable communication. The opposite is true, however, in large-scale systems. Since the number of physical connections dictates the physical transmission reliability, the more refined the topology and the fewer hops, or segments, a message must traverse, the more likely the message is to get to be transmitted without error.

While analyzing the YP 679 topology of a semi-mesh of rings an interesting topology surfaced that is more effective for large-scale systems. The semi-mesh is extremely well suited for small-scale systems, but it is not as effective as a hypercube for networks with large numbers of nodes. Although the semi-mesh topology is scalable, in order to maintain an average of three links per ring as initially proposed, the expansion of the network looks rather chaotic. The current YP 679 configuration, where each ring has been shrunk down to a node for clarity, is shown in figure 15.

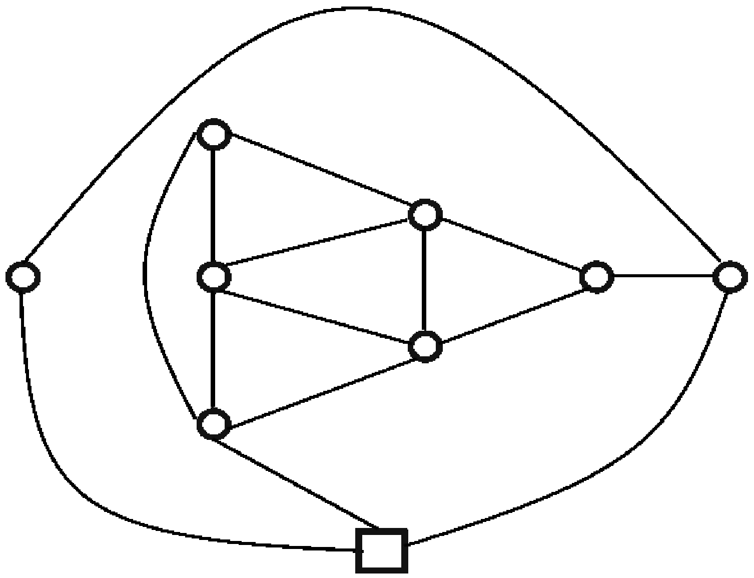


Figure 15 – Existing YP 679 Semi-Mesh Topology

Although there is some order in the network the semi-mesh topology guidelines are not followed as some rings have more than three links. Furthermore, with only eight rings, there are several node combinations that result in long message transmission paths. Since it is desired that each ring have on average no more than three connections, the lack of order in the network forces it to expand. The hypercube of rings, as shown in figure 16, is an alternative to the semi-mesh of rings topology and allows a more structured topology to be implemented.

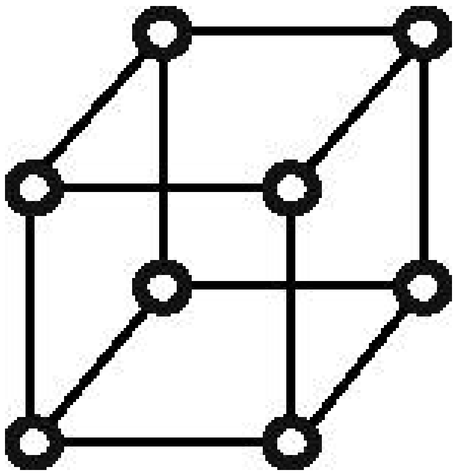
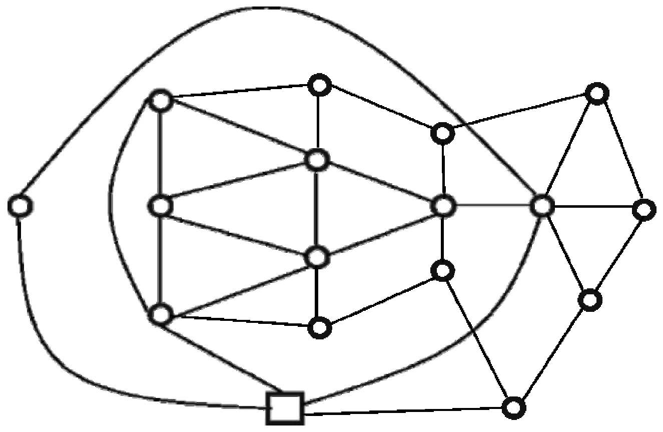
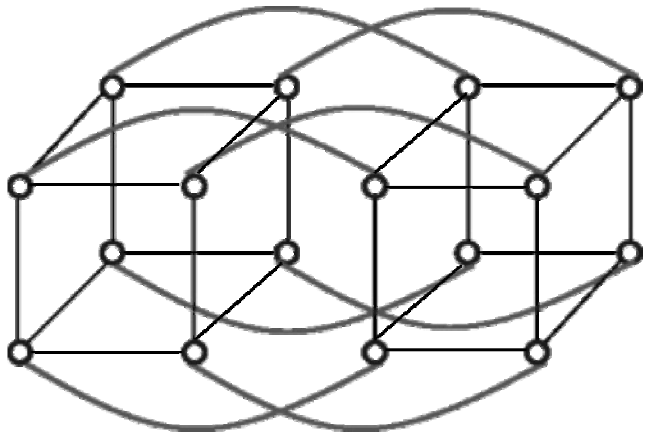


Figure 16 – YP 679 Topology Implemented as a Hypercube

The advantage of this topology is that as the number of nodes increases, the hypercube's structure maintains a low number of hop counts. This can be seen in Figure 17 where the semi-mesh and hypercube topologies are shown with twice as many rings as on YP 679. While the semi-mesh of rings topology expands because of the connectivity restraint of 3 connections per node, the hypercube of rings topology stays compact and is able to add several short cut paths that allow messages to quickly traverse the network.



YP 679 Semi-Mesh Topology With Twice as Many Nodes



Hypercube Topology With
Twice as Many Nodes

Figure 17 – Expanded Semi-Mesh and Hypercube Topologies

Preliminary research was conducted on three topologies, the full mesh, semi mesh, and hypercube. Data was obtained by using Monto Carlo Analysis which is ideal for systems where measuring data is nearly impossible. Monto Carlo Analysis is based on the central limit theorem. If you take a large number of test cases, the values will begin to converge upon the correct answer due to the central limit theorem. In the case of the full mesh, semi-mesh, and hypercube a Monto Carlo analysis was done for networks containing between one and twenty nodes. Ten starting and terminating nodes were chosen for each case and the average number of hops computed. Multiplying the number of nodes in the current network by a uniformly distributed pseudo random number that is rounded to the nearest whole number chose starting and terminating nodes. These values were computed using MATLAB and thus cannot be considered entirely random since some pattern does exist albeit only after millions of cases have been evaluated. The data in Appendix E shows several important trends. Figure 18 shows that the number of physical connections in the network remains relatively close for both the semi-mesh and hypercube topologies. For a given number of nodes the number of physical connections in the network is nearly the same for each topology with the hypercube having on average one or more fewer paths than a semi-mesh.

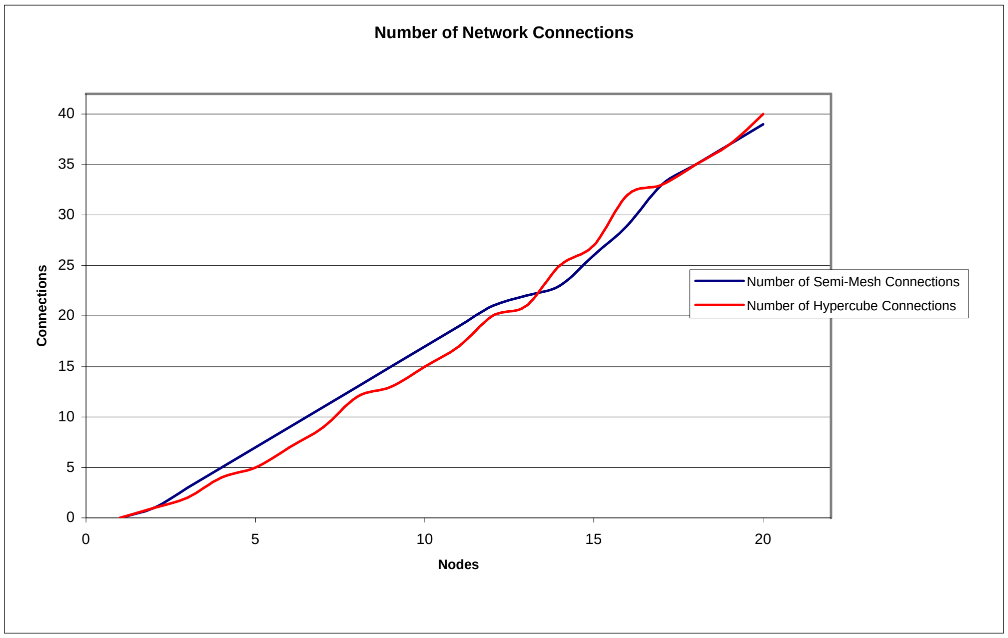


Figure 18 – Number of Network Connections

If each of the nodes in the Monto Carlo analysis is considered to be a ring with ten nodes then nearly two hundred nodes could be contained in a system of twenty rings. If this system is to be implemented for a large-scale system with thousands of nodes, however, the divergence can be seen in Figure 19. This divergence becomes greater as nodes are added to the systems, showing that the hypercube of rings topology increases reliability for large scale systems by keeping the average hop count low.

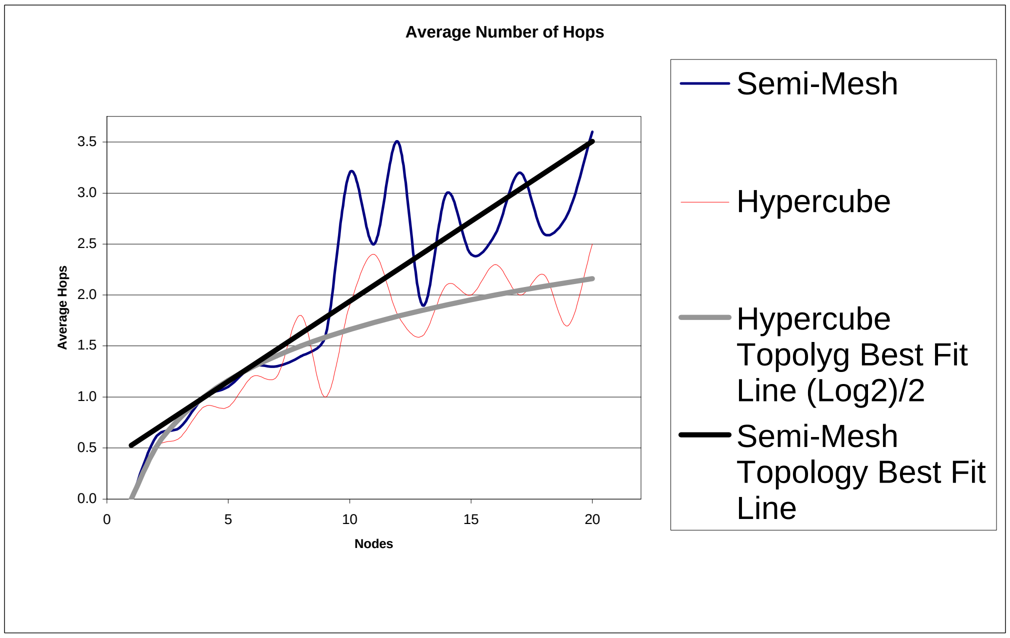


Figure 19 – Average Hop Count

By projecting the best fit lines for each topology out to the case where one hundred nodes are present, as seen in Figure 20, the divergence of the average hop count becomes very large. For the semi-mesh and hypercube, straight and (Log_2 / 2) best fit lines approximate this projection most accurately. If, as would be implemented on a large-scale system, each node in the analysis is considered to be a ring with ten nodes in the real system, only 1000 nodes are present in the projection. This is nearly 10^2 fewer nodes than would be found on a standard U.S. Navy ship showing that for large-scale systems a hypercube of rings is a good topology selection.

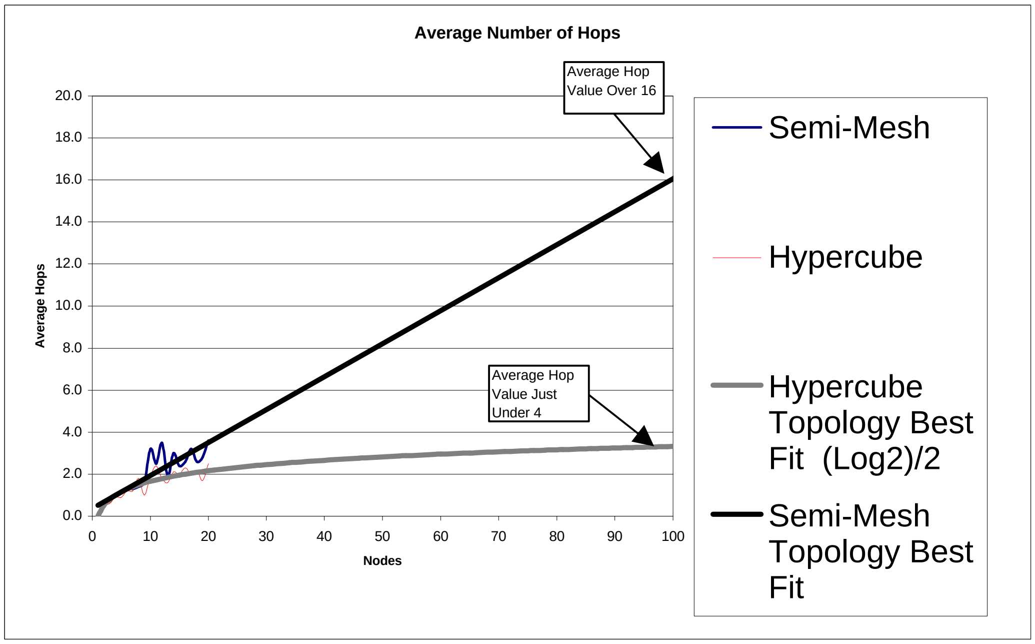


Figure 20 – Projection of Average Hop Count

Lon-Works Network Fragment Healing Components

Nodes

There are three main components found in LONTalk networks that implement network fragment healing: nodes, routers, and sentinels. The node, which is the element where data transmission originates and terminates, is the most common element found in LONTalk networks. Specifically, a LONTalk node combines local and network processing into a single micro-processor. A typical node would be connected to one or more sensors or actuators and performs functions coded in Echelon Neuron C. Nodes play a very important role, as they are responsible for generating, receiving, and processing messages in the network.

What makes LONTalk nodes different than nodes in other control networks is that embedded intelligence has been added in a low cost fashion. Because of the embedded intelligence, nodes are able to evaluate their performance and correct problems when they occur. For example, a node might automatically recalibrate attached sensors. Embedded intelligence also allows nodes to continue operation when they become separated from the network by communications failures. Nodes on a LONTalk network are responsible for sending messages without the use of a token or other signaling mechanism.

Routers

A router is the component responsible for passing messages between network rings. In a LONTalk system with network fragment healing, a router is a node connected back to back with another node through their I/O ports. A router can be thought of as a means by which messages pass between different sections of the network. Routers inspect the destination address of messages and then consult a routing table to determine if these messages need to be forwarded to the adjacent ring. A routing table contains information on which messages a router should forward, and which messages should be not be forwarded. This routing information is important because the LONTalk protocol does not support multiple paths between source and destination nodes. Each router has a specific routing table which reveals exactly one path between any two nodes. By manipulating the information in the routing table, sentinels control the network configuration. Since sentinels know the physical topology of the network and have the ability to change routing tables, multiple paths can be hidden from the protocol and unmasked in order to reroute messages. Although the physical topology of the network is statically defined, the sentinels can dynamically perform network fragment healing by enabling one of several alternate paths.

If after consulting a routing table, a message is flagged for forwarding, the router forwards the message to the adjacent ring router. On the adjacent ring, the forwarded message contends with any other pending messages for a transmission slot. The YP 679 network uses LONTalk configured routers. Configured routers are different than other types of routers in that the routing table is explicitly configured to enforce a specific network topology. Configured routers are well suited for network fragment healing because they will not actively search for the network topology which would lead to the discovery of latent multiple paths. Sentinel network management messages maintain each routing table so that the physical topology redundancy is hidden from the LONTalk protocol.

LONTalk priority message compliant routers allow nodes that are not given access to priority slots the ability to request that messages be forwarded as priority messages. When these messages reach routers and are queued they are transmitted before any normal priority messages. This allows nodes of normal priority to ensure that time critical messages are forwarded quickly.

Sentinels

Sentinels are another important component of a Lon-Works network. Sentinels, like routers, are specialized nodes that are responsible for maintaining the physical topology seen by the Lon-Talk Protocol. Sentinels monitor the network both actively and passively in order to ensure that no faults are present. If faults are detected the sentinel takes action to remedy the problem in order to ensure reliable data transmission. Because sentinels maintain the physical connections seen by the protocol, fragmented sections must have access to a sentinel in order to reconfigure the routers and change message paths. Thus, each segment has a sentinel and each sentinel on a ring is given a priority in order to assure that only one sentinel, the highest priority, is active at any given moment.

When a sentinel determines that a fault may have occurred it takes actions to explicitly determine if and where this break has occurred. By using an imbedded “intelligent switch” the sentinel can break the network on one of its sides, turning the ring into a bus. By sending a message requiring acknowledgment to the farthest node on the bus the sentinel can see if all nodes are intact. If the farthest node does not respond the sentinel works backward and sends messages until a node responds. The sentinel now knows that this is where the break has occurred. In order to ensure that the ring has not become segmented the sentinel must also perform the same test on the ring from the opposite direction. This is possible by breaking the network on its opposite side using the “intelligent switch.” The sentinel then performs the same actions to determine if and where another break has occurred.

Sentinels also have the capability to detect faults passively. Since actively breaking the network after a fault has occurred would further fragment the network, sentinels listen for status heartbeats from the nodes on the ring. Heartbeats sent from nodes are used similar to a human heartbeat in determining node status. If several adjacent nodes do not transmit heartbeats, then a fault may have occurred on the network and will be investigated, much like a pulse is a good indication of human status.

Sentinels not only check for faults on the network, but also communicate to other sentinels about what state the network is in. By sharing information on node status and network breaks sentinels can choose healing paths more intelligently. Also, if sentinels did not communicate with each other then there is a possibility that multiple sentinels could break and ping the network at once. By passively sending heartbeats on the network, sentinels can determine the state of other sentinels on the ring and act accordingly. If there are sentinels of higher priority sending heartbeats, a lower priority sentinel stays passive. If a higher priority sentinel quits sending heartbeats, however, the next lowest priority sentinel assumes the responsibilities of network reconfiguration. This method of passing information allows the sentinels to know what other sentinels are active while allowing the receipt of heartbeat messages.

Although sentinels are responsible for examining the physical status of the ring they also ensure that there are no faults between rings by sending messages to active sentinels on adjacent rings. This allows both the ring path integrity and inter-ring integrity to be known.

Sentinels are the backbone of network fragment healing because they know the physical topology of the network. When faults occur, sentinels locate the faults and send information to the routers so that messages can be rerouted around damage. Any changes to the topology are passed between sentinels so that router tables can be updated and continuous communications can be achieved.

Lon-Works Enhancement

Lon-Works is a feature rich, low cost, industry proven, commercial networking standard for distributed control systems. However, Lon-Works has one fundamental limitation for mission critical military systems. Since the LONTalk protocol does not allow multiple communication paths, Lon-Works is not survivable. Network fragment healing adds essential survivability by providing redundant communication paths that are hidden from the LONTalk protocol.

As can be seen in many of the topologies introduced, multiple paths are essential if network fragment healing is to take place. If, as Lon-Works in its commercial form requires, only one node to node path exists then there would be no way to route messages around damage sustained by the network. Thus, the redundancy that must physically exist in a distributed environment is required to be masked from the LONTalk protocol. Having routers maintain only one specific path between any two points on the network can achieve this. By declaring the path that messages between two nodes must take, the LONTalk protocol can function normally. When damage occurs, the path between the two nodes declared in the routing table can be changed, effectively hiding the multiple paths from the Lon-Talk protocol. Figure 21 shows that there are many paths between nodes A and B. Since the Lon-Talk protocol does not support these multiple paths a router statically declares a path that messages between the two nodes will use. In this case router 3 will forward all messages between nodes A and B. If network damage is sustained and router 3 is no longer available to pass messages, a new path is declared through routers 1 and 2 for messages traveling between nodes A and B. This example shows that although multiple paths can exist between two nodes on a Lon-Works network, these paths may be made unavailable in order to comply with the LONTalk protocol.

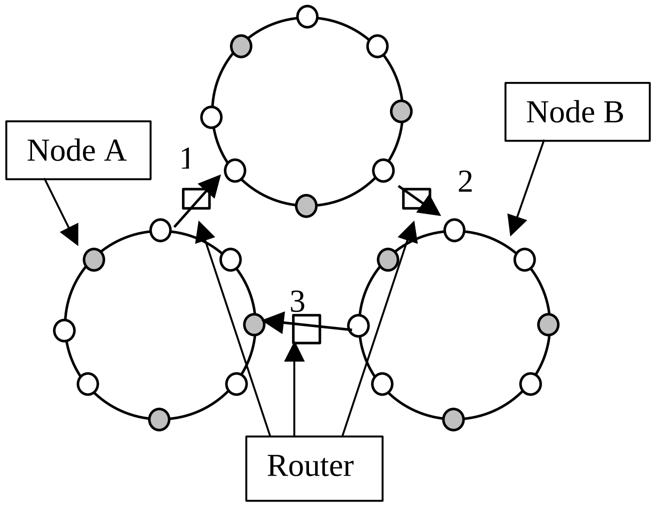


Figure 21 – Small Network With Multiple Paths

Reasons For Modeling

Although network fragment healing has been installed and tested on small-scale systems, there is a need to advance these findings to support larger networks with thousands of nodes. Physical testing is powerful tool in evaluating small-scale systems because it allows the network to be evaluated and the problems inherent with any system to surface. Unfortunately, building large-scale research systems with thousands of nodes is impractical because of the overhead associated with construction and troubleshooting. The use of simulation software, however, allows higher order systems to be implemented quickly at a much lower cost.

There are several advantages to using computer models instead of large-scale systems. Models allow data flow to be studied much more closely since the network simulation can be stopped and evaluated at any instant in time. Furthermore, by creating a model of the communication protocol’s inner workings, limitations and strengths of a system come to light since the protocol must be analyzed in great depth. Computer models also allow the user more complete control of the network and its actions since parameters can be changed with a few keystrokes. This notion of changing system parameters quickly also allows the user to scale the computer model and resize the simulation by including more nodes. The user can also control message traffic generation, allowing more accurate simulation of specific network scenarios such as network fragmenting, and high volumes of network traffic to be studied. Finally, modeling offers simulation results that can be made independent of protocol stack processing time, transmission time, and message queuing times.

Network fragment healing currently performs well in ensuring reliable network communications but several improvements could be made. By advancing the algorithms used to control network fragment healing, system reconfiguration time can be improved. Currently, when the network reconfigures, the shortest physical path between two nodes is chosen as the healing path. This, however, may not be the path with the shortest message delivery time. If the path chosen has very heavy traffic associated with it, communication can actually be slower. Also, the chosen path may not be the most reliable path due to its physical location. Some messages are more important than others and it is desirable for them to traverse a path that has less chance of being damaged. Because of this, a new algorithm for network fragment healing that includes message traffic calculation, shortest path distance, and reliability must be used to ensure the highest level of communications continuity.

Software Design Considerations

Computer modeling is an effective way to simulate and evaluate network communication protocols. There are several design considerations that must be evaluated before a simulation platform can be chosen. First, LONTalk is a highly distributed control system where no one node or group of nodes controls the flow of information. In LONTalk, processing capabilities have been pushed down to the lowest level to avoid single points of failure. This distribution of processing also enables continuous intelligent operation at the nodal level during casualty conditions. Distributed control offers greater reliability, but is more difficult to develop and simulate. An early simulation design consideration was the use of software packages that support distributed environments.

Furthermore, LONTalk nodes, the lowest level of the control system, have embedded, highly intelligent processors that control all facets of message generation, transmission, and processing. This means that there is no central access scheme, such as a token being passed, which grants nodes access to the network. Instead, each node autonomously attempts to send messages while avoiding collisions with other transmissions. The ability of a software package to simulate this asynchronous nature of LONTalk messaging is critical in creating a realistic model of the YP 679 communications.

Finally, a major consideration when choosing a software package is the learning curve associated with it. There are literally hundreds of programming languages available today, many of which are appropriate for modeling network communications. The language chosen, however, must be one that provides modeling flexibility while minimizing the time associated with learning a new programming language. If the learning curve is too great, the time available for coding the model will be dramatically reduced.

Commercial Package

Commercially available network simulation software packages are one alternative to developing a new network simulation program. Commercially available packages allow networks to be quickly configured and evaluated, but are typically only written for common protocols such as TCP/IP, token rings, and CSMA/CD. Since LONTalk is a very specialized network protocol, there is no simulation package available to quickly create and test LONTalk Network Fragment Healing networks. Furthermore, existing simulation software packages would require extensive modification to incorporate LONTalk Network Fragment Healing features. Such extensive modification would be very time consuming considering the learning curve associated with understanding the inner workings of an existing software simulation package. Furthermore, commercial network simulation packages are relatively expensive and often cannot be modified by the end user.

MATLAB

Another promising software package evaluated was MATLAB, from MathWorks, Inc. This platform was a natural choice because it is widely used in industry and has many built in functions for the interpretation of data. It does not, however, offer any features to model the asynchronous nature of Lon-Works. MATLAB “m” or script files are implemented within a single source code file. Furthermore, MATLAB provides limited messaging facilities and does not support the use of header files. This means that the LONTalk asynchronous message passing features would have to be developed from scratch. This could be accomplished by creating functions to emulate message passing and asynchronous processing but this would have to be done in addition to writing the network model code, taking away research time.

Python

Since several software packages met the design criteria, a significant research effort involved the determination of which one to use. The first programming language that was considered in depth was Python, a scripting language written in C that is available as freeware at www.python.org. Python offered several advantages for modeling distributed control including threads and micro-threads. Python threads share the processing power of the computer, allowing multiple programs, or sections of code, to be run at the same time. Python threads allow the asynchronous nature of the Lon-Works network to be programmed in a single piece of Python source code. Python was recommended by Dr. Sam Smith of Adept Systems Inc., since threads and micro-threads could be used to implement nodes, routers, and sentinels. This model of the network was very attractive but the use of the Python language offered several disadvantages for this project.

Python is a very specialized new language that is distributed as freeware and is commercially unsupported. As freeware, the license to use this software is obtained free of charge at the Python web site. A freeware software package raises significant questions concerning up to date software patches, documentation, and support. With comparatively little language documentation available, the Python learning curve was expected to be higher than other better documented programming languages. Therefore, the advantages of Python were judged to be outweighed by the risk of using developmental software with an unknown learning curve. Furthermore, Python is not a widely used programming language so future applications of Python are uncertain. In conclusion, Python would have offered a great challenge to a beginning programmer trying to simultaneously learn a new language and model a Lon-Works network.

C++

Another language that offered several advantages is C++. This language is very popular and is well documented. Moreover, as a general purpose language, C++ imposes few design constraints on the simulation of the LONTalk networking protocol.

C++ is an object oriented programming language meaning that it has built in capabilities to model real world objects. Furthermore, C++ classes allow new objects to be defined as extensions of existing object classes. Classes can communicate with each other through the use of messages in a manner that is analogous to network communication.

C++ is also a multitasking environment that allows asynchronous processing to be implemented as independent tasks or threads. Threading allows the asynchronous nature of nodes to be realistically modeled within a single task. In addition, several source files and their respective header files can comprise a project and be compiled into one executable.

The main reason that C++ was chosen as the base programming language is its relation to the C programming language, which had already been studied in previous work. This decreased the learning curve to an acceptable level. Furthermore, C++ is widely used and help can be obtained from many sources.

Borland C++ Builder vs. Microsoft Visual C++

Since C++ is a very widely used programming language there are many different software packages available for writing and compiling C++. Microsoft Visual C++ and Borland C++ Builder are both fully capable C++ compilers but offer several advantages beyond C++ compilation and debugging. Both packages offer advanced visual shortcuts to implement user interfaces and other graphical tools associated with the Windows environment. This allows the network model to be controlled by physical components like in other Windows based applications.

Both software packages are very similar in debugging, support, and other basic characteristics and are consistently some of the best C++ compilers on the market. Microsoft Visual C++, however, has an enormous learning curve associated with it. The programmer must explicitly code all graphical user interfaces and the overhead required to learn these functions is large. Borland C++ Builder, on the other hand, offers a visual component library that holds pre-made functions and shortcuts that make the creation of user interfaces and graphical displays easy. Borland Builder contains a special window, called a form, where objects can be placed. Borland Builder then in turn creates the code required to implement this object in the source file and the object is quickly added to the application. This drag and drop capability speeds software development and removes the problems associated with visual object creation. Moreover, Borland C++ Builder has very good support on both the web and via phone. Finally, Borland C++ Builder is licensed by the United States Naval Academy, allowing its usage by students and faculty members. This eliminates the need to purchase this software with research funds.

Procedure

A LONTalk network is a distributed control network that can contain hundreds or thousands of asynchronous nodes. This asynchronous behavior must be modeled to accurately reflect distributed communication and control. Borland C++ Builder was chosen because of its capability to support distributed asynchronous processes and threads. Threads are independent, schedulable software entities that coexist within a single process and address space. Discussions with Adept Systems Inc. and literature searches led to a decision to use threads. Initially, it was believed that threads could be used to readily emulate the asynchronous behavior of LONTalk nodes. Specifically, node threads would send messages to ring threads, which would in turn pass the messages to the appropriate destination node and ring. Although this is a simplification of the LONTalk protocol, it accurately models the asynchronous nature of nodes and messages that travel along one or more rings before arriving at the intended destination. Messages would be passed between these threads using message posting methods, or functions, built into the C++ language. Nodes could then simply designate a message number, fill a message structure with information, and then post their message transmission. By invoking a complementary retrieval method, destination rings or nodes would receive the posted message. Since threads share a common address space, messages could be passed by reference, and would not need to be physically copied at each intervening step. Unfortunately, the Windows threading model is limited to a maximum of sixteen threads, severely limiting the scalability of this network mode. This constraint prevented nodes from being implemented with threads in the network simulation.

Using the knowledge learned in the first iteration, a new model was created based on multiple threads within multiple processes. In this model each ring would be a separate process, with nodes implemented as threads within each of the applications. This model built upon the knowledge initially learned and included an interactive graphical user interface which takes advantage of the Borland C++ Builder Visual C Library. This model allowed each process to have its own graphical interface containing start and stop buttons for the ring, and a memo box to log incoming message information. The message statistics could then be analyzed to find message latency between network nodes. In a manner analogous to the first simulation model, this method transmits messages by invoking inter-process message passing methods. After some research, it was determined that the Borland inter-process message handling was quite complex and beyond the scope of this investigation.

In order to get away from requirement to use Windows messaging methods, another simplification of the model was made. Without losing any functionality, threads could be used to represent rings in the network. Although threading constraints kept the maximum number of rings at sixteen, this is enough to adequately model the network and still allow manipulation of the algorithms and topology to be studied. Since Borland C++ Builder does not support microthreads, each node on the ring would be modeled as a case.

All threads are created within a single source code and messages can be passed by using global variables that are accessed with critical sections. By using critical sections, threads are able to lock out other threads from a section of code that is currently in use. If multiple nodes attempt to access a variable without the use of critical sections, there is no way of ensuring that variable data will not be overwritten while in use. Critical sections fix this problem by allowing threads to manipulate data safely. When a thread is done using a section of code, it will leave the critical section and allow other threads access. If critical sections are to be used, however, they must be used by all functions that have access to that section of code. If functions are not forced to use critical sections, they maintain the ability to manipulate the data at all times. This effectively negates the advantages of using critical sections.

This model worked fine in theory, but the overhead required to learn threading was large. Furthermore, a preliminary investigation into network models that used only basic C++ tools showed that a model of the Lon-Works network does not require complex implementation.

After researching ways to further simplify the model, a final design iteration was created for the Lon-Works network model. In order to simplify the simulation coding and take advantage of the object oriented nature of C++, nodes and rings were to be implemented as classes. Each ring would be a specific instance of the ring class, and be created with a constructor. A constructor is a method associated with all classes that is passed information in order to create an instance of a class. The information passed to the constructor allows each instance of a class to contain information specific to that class instance. Once an instance of each ring was created, the node constructor could be called to create the number of nodes required on that particular ring.

This methodology took advantage of the fact that each router is just a special case of a node. By changing the parameters of nodes, routers can be easily created and given the information necessary to pass messages.

This design was also very streamlined in how messages were passed between different nodes. A pool of message structures was created with two fields. The first declared whether the particular structure was in use, and the second contained a pointer to the message being transmitted. This allowed a method to be called to find an open message structure, and then flag that structure for use. When a terminating node receives a message, the message structure is reset to show it is not in use, and then placed back into the pool to be used by another node with a message to send. Also, by using the queuing capabilities available in the Standard Template Library (STL), the structure containing the message could be placed in the queue and sorted according to its priority. By using the priority queue defined in the STL, messages can be easily sorted according the their priority.

This model also allows the asynchronous nature of Lon-Works messaging to be simulated. The average delay for nodes trying to send during a busy network state is a normally distributed number of Beta2 slots. If non-priority messages are given priority values between 1000 and 2000 and this value is then multiplied by a normalized, normally distributed, random number, the messages placed in the queue will have a normal distribution. This directly correlates to the LONTalk protocol, as nodes in a Lon-Works network have no guarantee that their messages will be sent before any other node of the same priority. Since the priority queue is set up to process the highest priority message, this normally distributed insertion pattern would model the delays seen in a real Lon-Works network.

Another advantage of this model is that data is not only written to the screen but each message is also logged into a text file. This allows the message to be located at creation and destruction and the latency to be calculated. This model also has a dramatically decreased learning curve associated with it. By adequately compromising between accurate simulation and minimum learning overhead the implementation of a Lon-Works network simulation using C++ classes is very feasible.

Results

Currently, the simulation network does not work, however, great strides have been taken toward completing a working model in the near future. With each day C++ comprehension improves and the simulation can be advanced towards the end goal of data generation. With the completion of the simulation, the computer model data can be correlated to YP 679 test data to verify that the simulation works properly. With a simple model verified with actual data, the results obtained when the simulation is changed to model more complex large scale systems are more accurate. This research continues to be undertaken with the intent to complete the simulation in the near future.

Synopsis

With the Navy's commitment to reduced shipboard manning the effectiveness of automated systems has become paramount. If sailors are to be replaced with automation it must be dependable during the worst casualty conditions and provide the continuity of communications service necessary to maintain operational readiness.

By implementing topologies that are inherently survivable, the number of messages lost due to the physical medium will be dramatically reduced. These topologies cannot be chosen based solely on their undamaged configuration, but instead must be evaluated by their ability to provide the most effective structure for distributed control during casualty conditions. A topology analysis was done on the Office of Naval Research test craft YP 679 in order to determine if the semi-mesh of rings topology is scalable to higher order systems.

Algorithms that support network fragment healing must also be continuously evaluated and advanced so that message latency and transmission reliability can be improved. By allowing the network fragment healing algorithms to more intelligently determine how messages are passed, the network becomes more and more distributed as decision making is pushed down to the lowest level. In order to study large-scale network fragment healing, a network simulation was written in C++ in order to model the asynchronous messaging of a Lon-Works network.

Conclusions

The LONTalk protocol is well suited for distributed control applications.

Flow charts on a nodes send and receive processes were created after in depth technical analysis of the ANSI 709.1A LONTalk networking protocol.

The semi-mesh of rings topology provides effective network fragment healing on YP 679. The proposed hypercube of rings topology improves network performance and survivability for larger systems. This exciting discovery may have important implications for the next generation of surface combatants.

Recommendations

In order to apply the results obtained in this research to large-scale systems further research should be conducted in these areas:

- More in depth Monto Carlo analysis of the hypercube and semi-mesh topologies.
- Conduct a reliability analysis of a hypercube of rings topology.

- Build on the C++ code written to complete network fragment healing model.
- Analyze Lon-Works network fragment healing message data to determine network latencies, healing paths chosen after a break, and network congestion.
- Investigate new healing algorithms for the network using the Lon-Works model.

References

[1] Blackwell, Luther M. "Data Multiplex System (DMS)-Aspects of Fleet Introduction." Naval Engineers Journal. Nov 1989: 41-47.

[2] Smith, Samuel. Survivable Shipboard CLIDCS Automation Infrastructure Development. Feb 13, 1999.

[3] EIA-709.1-A Control Network Protocol Specification. Arlington, VA: Electronic Industries Alliance: 1999.

[4] Echelon Corporation. LonWorks Marketing Bulletin "LonWorks 78kbps Self-Healing Ring Architecture." Aug 1993.

[5] Echelon Corporation. LonWorks Engineering Bulletin "LonTalk Protocol." Apr 1993.

[6] Echelon Corporation. LonWorks Engineering Bulletin "Optimizing LonTalk Response Time." Aug 1991.

[7] Smith, Samuel, Kevin White, et. al. Final Report. "Dependable network Topologies With Network Fragment Healing For C.O.I.D.C.S For Naval Shiboard Automation." Department of Ocean Engineering, Florida Atlantic University: 1999.

[8] Freeman, Roger L. Practical Data Communications. New York: John Wiley & Sons, Inc, 1995.

Bibliography

Adept Systems Inc. “A C Reference Implementation of the LonTalk Protocol on the MC68360.” Jul 1998.

Anderson, T., et al., “Dependability: Basic Concepts and Terminology.” Dependable Computing and Fault Tolerance. Ed. J. Laprie. Dec 1990.

Blackwell, Luther M. “Data Multiplex System (DMS)-Aspects of Fleet Introduction.” Naval Engineers Journal. Nov 1989: 41-47.

Carnivale, J. A. DD-21 Presentation. Jan 1999.

Deitel H.M. and P. J. Deitel. C++ How To Program. Upper Saddle River: Prentic-Hall Inc., 1998.

Dunn, Stan et al. Dependable Network Topologies With Network Fragment Healing For C.L.I.D.C.S. For Naval Shipboard Automation. Feb 18, 1999.

Echelon Corporation “Introduction to the LonWorks System.” 1999.

Echelon Corporation. LonWorks Engineering Bulletin “Enhanced Media Access Control with LonTalk Protocol.” Jan 1995.

Echelon Corporation. Echelon White Paper. “Determinism in Industrial Computer Control Network Applications.” Jan 1995.

Echelon Corporation. “Determinism in Audio Computer Control Network Applications.” Jan 1995.

Echelon Corporation. LonWorks Marketing Bulletin “LonWorks 78kbps Self-Healing Ring Architecture.” Aug 1993.

Echelon Corporation. LonWorks Engineering Bulletin “LonTalk Protocol.” Apr 1993.

Echelon Corporation. LonWorks Engineering Bulletin “Optimizing LonTalk Response Time.” Aug 1991.

Echelon Corporation. Product Specification. “Twisted Pair and Generic Control Modules.”

EIA-709.1-A Control Network Protocol Specification. Arlington, VA: Electronic Industries Alliance: 1999.

Fault Tolerant Systems Practitioner's Workshop. System Fault Tolerance. Carnegie Mellon University, 1991.

Freeman, Roger L. Practical Data Communications. New York: John Wiley & Sons, Inc, 1995.

Hollingworth, Jarrod et. al. C++ Builder 5 Developer's Guide. Indianapolis: Sams Publishing, 2001.

Hsu, John Y. Computer Networks: Architecture, Protocols, and Software. Boston: Artech House, Inc, 1996.

Lively, Kenneth et. al. Advanced Control Concepts for an Integrated Power System (IPS) Warship. 1999.

Miano, John et. al. Borland C++ Builder How-To. Corte Madera: Waite Group Press, 1997.

Preisel, John H. "The Evolution of Machinery control Systems Aboard U.S. Navy Gas Turbine Ships." Naval Engineers Journal. May 1989: 102-113.

Madan, Pradip. "LonWorks Technology for Intelligent Distributed Interoperable Control Networks."

Madan, Pradip. "Overview of Control Networking Technology."

Montgomery, Gloria. "In the Spotlight: Chief Engineer for the Navy RADM Kathleen Paige." Surface Warfare. Nov/Dec 1999: 2-5.

Raji, Reza S. "Smart Networks for Control." IEEE Spectrum. Jun 1994. pp49-55.

Smith, Samuel, Kevin White, et. al. Final Report. "Dependable network Topologies With Network Fragment Healing For C.O.I.D.C.S For naval Shiboard Automation." Department of Ocean Engineering, Florida Atlantic University: 1999.

Smith, Samuel. Survivable Shipboard CLIDCS Automation Infrastructure Development. Feb 13, 1999.

Swick, Mark, James White, Michael Masters. "A Summary of Communication Middleware Requirement for Advanced Shipboard Computing Systems." NSWC, Dahlgren Division Combat Systems Technologies Branch.

Tennefoss, Michael R. "Technology Comparison: LonWorks Systems versus DeviceNet." 1999.

Wang, Paul S. C++ With Object Oriented Programming. Boston: PWS Publishing Company, 1994.

Zivi, Ed and Tim McCoy. “Control of a Shipboard Integrated Power System.” Proceedings of the 33rd Conference on Information Sciences and Control. Mar 17-19, 1999.

Appendix A – MATLAB Reliability Plots

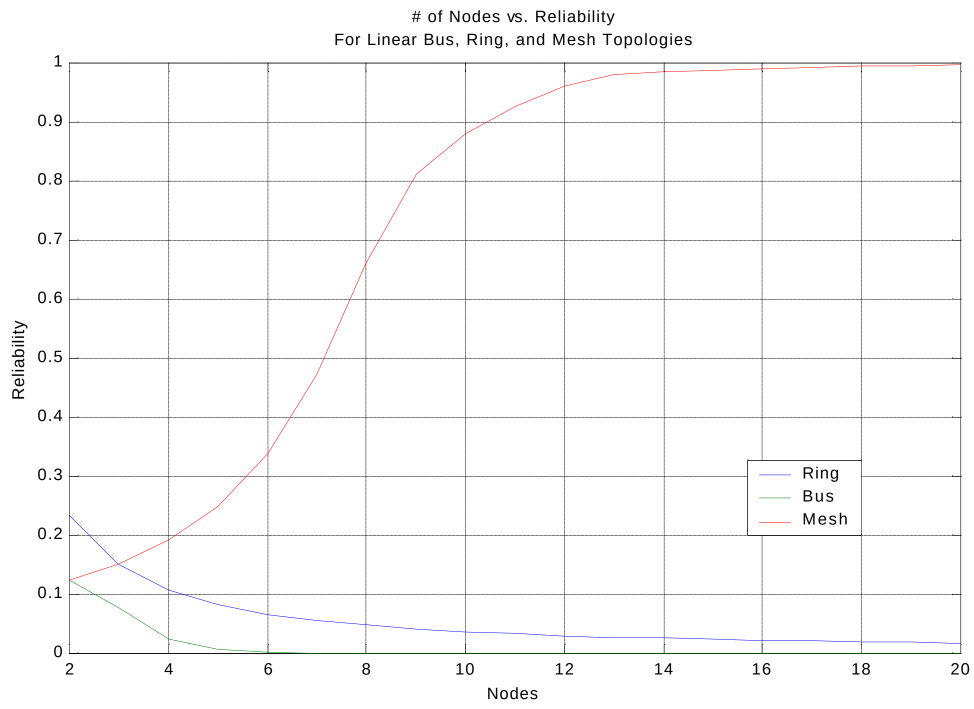


Figure A.1 – Ring, Bus, & Mesh Reliabilities vs. Number of Nodes

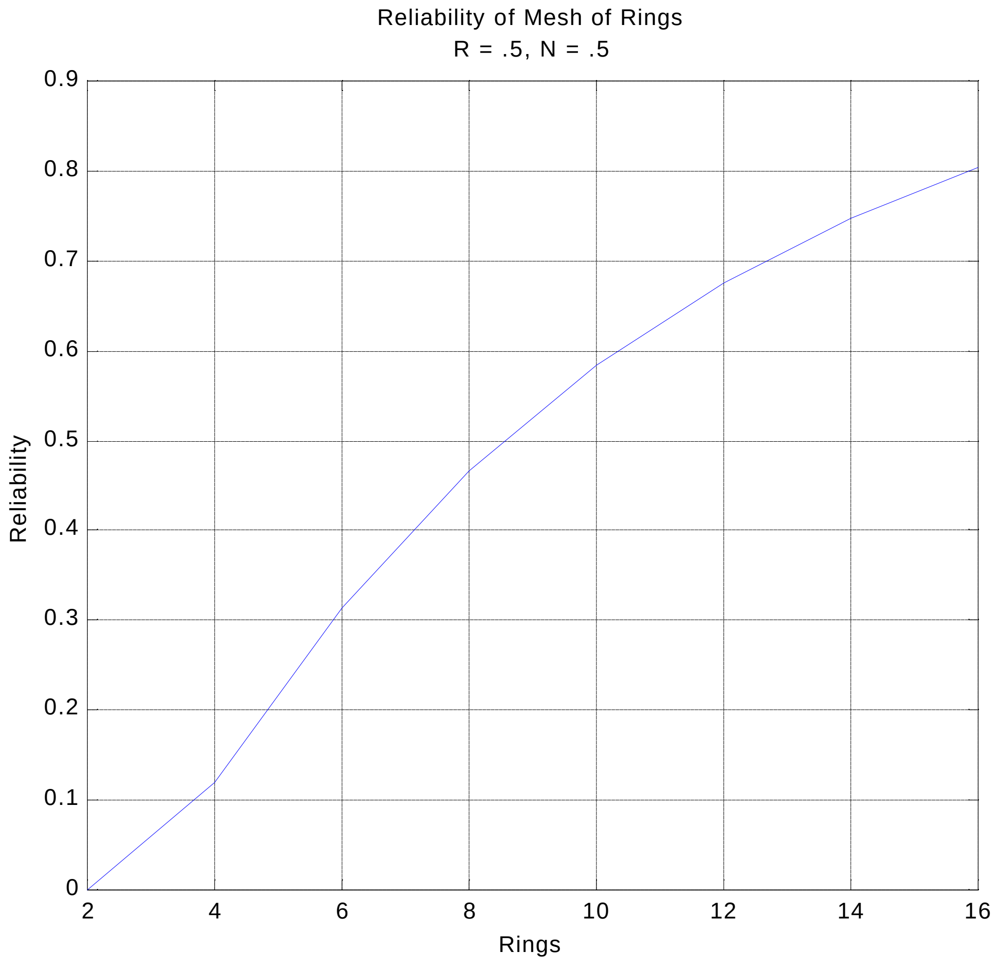


Figure A.2 – Mesh of Rings Reliability

Appendix B — Matlab Code

B.1 – Series Reliability Calculation

```
%1/c Jonathan Vanecko
%Trident
%6 December 2000

%Adapted from the work of Kevin White, Adept Systems Inc.

%Assumption:  A node on the path has the same reliability
%as a node that is transmiting or receiving

%Function to calculate the reliability of a series combination of elements
function [value] = CalcRSeries( vector )
%Initialzing the value variable with the first element of the list
value = vector(1);
%Iterations for each of the elements in series
for i = 2:length(vector)
   %the value is just the previous value times the current value
   value = value * vector(i);
end
```

B.2 – Parallel Reliability Calculation

```
%1/c Jonathan Vanecko
%Trident
%6 December 2000

%Function that calculates the parallel reliability of a vector

function [value] = CalcRParallel( tab )
%Initializes value to 1 minus the first list value to prepare
%for parallel calculation
reliability = 1 - tab(1);
%loop to finish calculation reliability
for i = 2: length(tab)
   %Value equals the previous value * 1-current value
   reliability = reliability * ( 1 - tab(i) );
end
%calculating the parallel reliability value by subtracting the
%previous value from 1
%reliability = 1 - reliability;
```

B.3 – Bus Topology Reliability Calculation

```
%1/c Jonathan Vanecko
%Trident
%6 December 2000

%Adapted from the work of Kevin White, Adept Systems Inc.

%Assumption:  A node on the path has the same reliability
%as a node that is transmitting or receiving

%Function to calculate the availability of a path
%Path is a bus topology

%The function operates by first calculating the availability
%of all paths from the shortest path possible (between 2 nodes)
%up to the longest path possible (N nodes)

%Rn=Reliability of a node
%Rl=Reliability of the physical media
%N=Number of nodes in the network
%value=Used to store the hi, low, and average availabilities
%It will pass these values to the screen
%CalcRSeries=function to calculate a series reliability


function [value] = CalcRBus2( Rn, Rl)

%If there is 1 node present then the Ao=Rn
list=0;
nodes=20;
for i=2:nodes
  for n = 2:i
    %Forms the vector for the reliability of the nodes
    for j = 1:n
      list1(j) = Rn;
    end

    %Forms the vector for the reliability of the physical media
    for j=1:(n-1)
      list2(j)= Rl;
    end

    %Passing the merged reliability vector to the series calculation
    Ri(n-1) = CalcRSeries( [list1,list2] );
```

```
 end

 %Creating the output matrix of reliabilities for each number of nodes
 value(i-1) = mean(Ri);

end
%Plotting the node vs reliability
y=2:1:nodes;
plot(y,value)
```

B.4 – Mesh Topology Reliability Calculation

```
%1/c Jonathan Vanecko
%Trident
%8 November 2000

%Adapted from the work of Kevin White, Adept Systems Inc.

%Assumption:  A node on the path has the same reliability
%as a node that is transmitting or receiving

%Function to calculate the availability of a path
%Path is a mesh topology with a reliability for a node and link of
%R and contains N nodes

function [value] = CalcRMesh( Rn, Rl)

  Ri = [];
  for count=2:20
    N=count;
    for i = 2:N

            %creates a list of reliability values R in a matrix 1x(2*current node# -1)
            %this list is the reliability matrix for all of the nodes and links
      %in a given path where N nodes are traversed

        %Forms the vector for the reliability of the nodes
        for j = 1:i
        list1(j) = Rn;
        end

        %Forms the vector for the reliability of the physical media
      for j=1:(i-1)
          list2(j)= Rl;
        end

      %combine the reliabilities for nodes and links into one vector
        list=[list1 list2];

        %calculating the series reliability of the nodes and links
      r = CalcRSeries( list );

        %calculation of the number of ways to traverse i nodes in a
        %network having a total of N nodes
      P = NumOfSegPath( i, N );
```

```
          if (P > 1000)
                P = 1000;
          end

       %creates a vector of the reliabilities for each of the
          %possible paths of the network.  with each pass network
          %path reliabilities are added
          Ri = [ Ri r*ones(1,P) ];

      end

    %calculates the parallel reliability of all the
    %possible paths
    value(count-1) = CalcRParallel(Ri);

    %setting list paramaters to 0 so that old data does not get
    %added to then end of vectors that have less
    %data inserted than the previous amount
    list=0;
    list1=0;
    list2=0;
    Ri=0
 end
 t=2:1:20;
plot(t,value);
```

B.5 – Ring Topology Reliability Calculation

```
%1/c Jonathan Vanecko
%Trident
%8 November 2000

%Adapted from the work of Kevin White, Adept Systems Inc.

%Assumption:  A node on the path has the same reliability
%as a node that is transmitting or receiving

%This function is used to calculate the overall availability
%of a ring of nodes

%The function will first calculate the availability of the
%smallest path possible on the top (2 nodes) and then calculate
%the longest path possible on the bottom (N nodes).  The
%function will then calclate the availabilities of all possible
%combinations of path lengths from a sender node to a receiver
%node on both the top path and bottom path of the ring
%A series availability is given for both the top and bottom
%paths and then a parallel availability can be calculated
%The max, min, and average Ao is then outputted

%Rn=Reliability of a node
%Rl=Reliability of the physical media
%N=Number of nodes on the ring
%CalcRBus2=Function to calculate the reliability of a bus path
%CalcRParallel=Function to calculate parallel reliability

function [value] = CalcRRing2( Rn, Rl)

  value = [];
  nodes=20;
  N=nodes;
    for i=2:N
    N=i;
    for n = 2:N
        %n1=path length along one route to the node
        n1=n;
      %n2=path length along the other route to the node
        n2=(N+2)-n1;
        %Calculates the reliability from start node to finish node along upper path
      Rtop=CalcRbus2(Rn,Rl,n1);
```

```
        Rtop=min(Rtop);
        %Calculates the reliability from start node to finish node along lower path
      Rbottom=CalcRBus2(Rn,Rl,n2);
        Rbottom=min(Rbottom);
        %Passing the reliabilities so that a parallel reliability can be found
      Ri(n-1)=CalcRParallel([Rtop,Rbottom]);
    end
    %Creating a value vector for output
    value(i-1) = mean(Ri);
    Ri=[];
  end

%Plotting the node vs reliability
y=2:1:nodes;
plot(y,value)
```

B.6 – Ring Mesh Topology Reliability Calculation

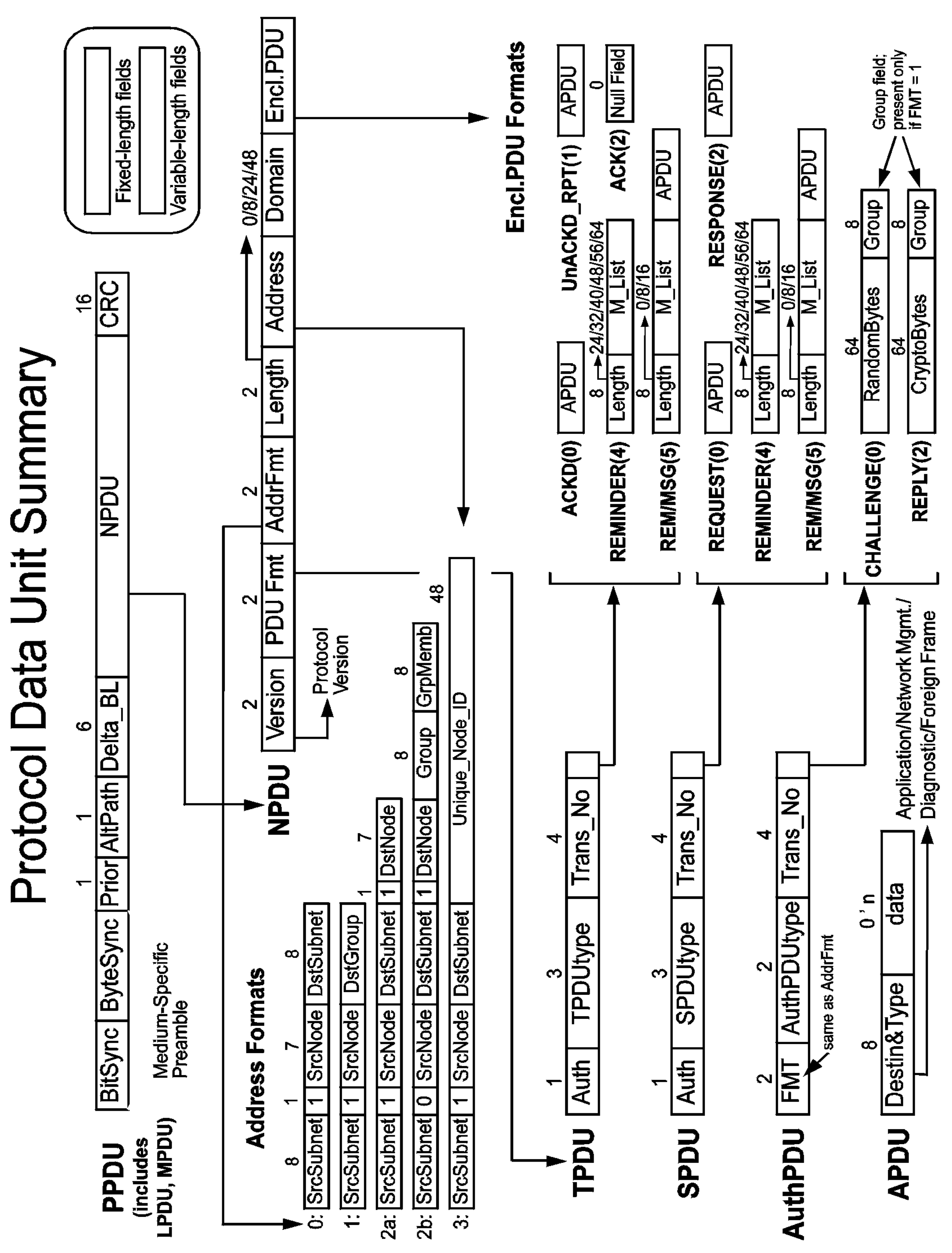
```
%1/c Jonathan Vanecko
%Trident
%6 December 2000

%Assumption:  A node on the path has the same reliability
%as a node that is transmitting or receiving

%Calculation of the semi-mesh of rings topology
%Reliability of router = reliability of node because it is just a node too

%N=Number of nodes on each ring
%R=number of rings but in this case the value is changed by the program in the for loop
%By deleting the for loop you can input different numbers of rings
%Rl=reliability of a link
%Rn=reliability of a node

function [value] = SemiMeshRing(N, R, Rn, Rl)

if R==1
  CalcRRing2( Rn, Rl, N )
end
noderel=[0];
for z = 2:2:14
  %Assigning the number of rings from the z value
  R=z;
  %The number of routers on a ring is equal to the number of rings-1
  routers=R-1;
    %The number of nodes per ring is equal to the number of routers + number of nodes
  nodesperring=N+routers;
  %Nodes and routers have the same reliability because they are both nodes
  %Calculating the number of nodes&routers on one side of the ring
  half=floor(nodesperring/2);
    %loop to go through the iterations of possible number of rings hopped out of R rings
    total
  for i = 2:R
        %Function call that returns the possible number of paths between 2 nodes
        %that pass through i rings out of the R total rings in the mesh of rings
    numberofways=NumOfSegPath(i,R);
    %The number of rings in the current iteration
    rings=i;
        %Forms a vector with the reliabilities of the nodes on 1/2 of the ring
    for j = 1:(half)
      ringvector=[ringvector Rn];
```

```
        end
            %Add to the ringvector the reliabilities of the links between nodes on teh ring
        %There are N-1 links between nodes
        for m = 1:(half-1)
          ringvector=[ringvector Rl];
        end
        %Calculate the series reliability of one ring
        onering=CalcRSeries(ringvector);
        %Mulitiply the reliability by the total number of rings
        partial=rings*onering;
        %Reliability calculation to take into account links between rings
        for k = 1:(rings-1)
          onepath=Rl*partial;
        end
            %Form the vector of reliabilities for 2^n paths due to the ring structure
        for l = 1:(2^rings)
          parallelpass=[parallelpass onepath];
        end
        %Reliability of all ways through i nodes out of the R total nodes
        oneway(i)=CalcRParallel(parallelpass);
        %Initializing vectors to 0
        ringvector=[];
        parallelpass=[];
      end
        %Calculates the parallel reliability for all of the different ways betwen 2
      %Points on a network with R rings
      systemrel=CalcRParallel(oneway);
        %Matrix that stores the reliability data for each pass with R total rings
      noderel=[noderel systemrel]
    end
    %Plots
    t=2:2:16;
    plot(t,noderel)
```

B.7 – Ring Path Calculation

```
%1/c Jonathan Vanecko
%Trident
%8 November 2000

%Adapted from the work of Kevin White, Adept Systems Inc.

%Assumption:  A node on the path has the same reliability
%as a node that is transmitting or receiving

%this function returns the number of ways to traverse the network given
%S=the number of nodes that you are planning on traversing
%N=the number of nodes in the mesh
%This means that if you want to know the number of ways to travel from
%one node to another touching only S nodes out of the N total nodes then
%call the NumOfSegPath function.

%Each node may only be transversed once for any path between nodes

function [value] = NumOfSegPath( S, N )

    value = 1;
  for i = 2:S-1

    %i will go between 2 and the number of nodes that
    %the path will traverse.  Thus, there can be no
    %greater S than the number of nodes in the network

        value = value * (N-i);
    %value = the previous value * (the number of nodes -1)

  end

return
```

Appendix C – Protocol Data Unit Summary (PDU)



PDU Summary [3]

Appendix D – Monto Carlo Analysis Plots

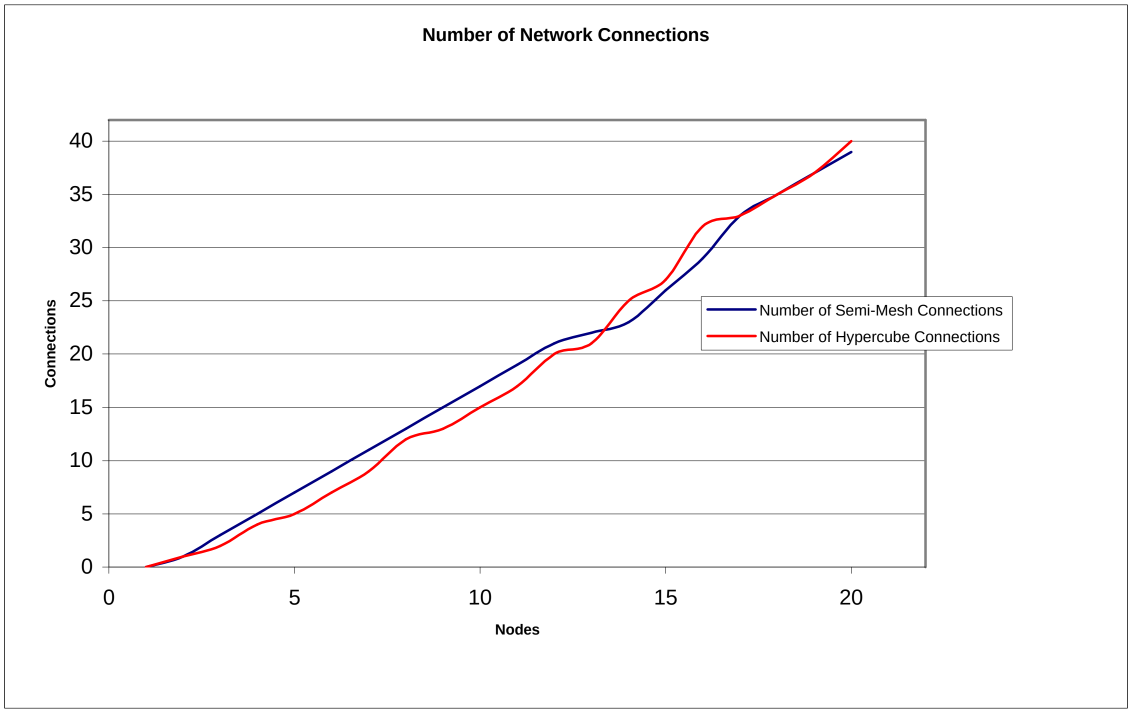


Figure D.1 – Plot of Physical Network Connections vs. Number of Nodes

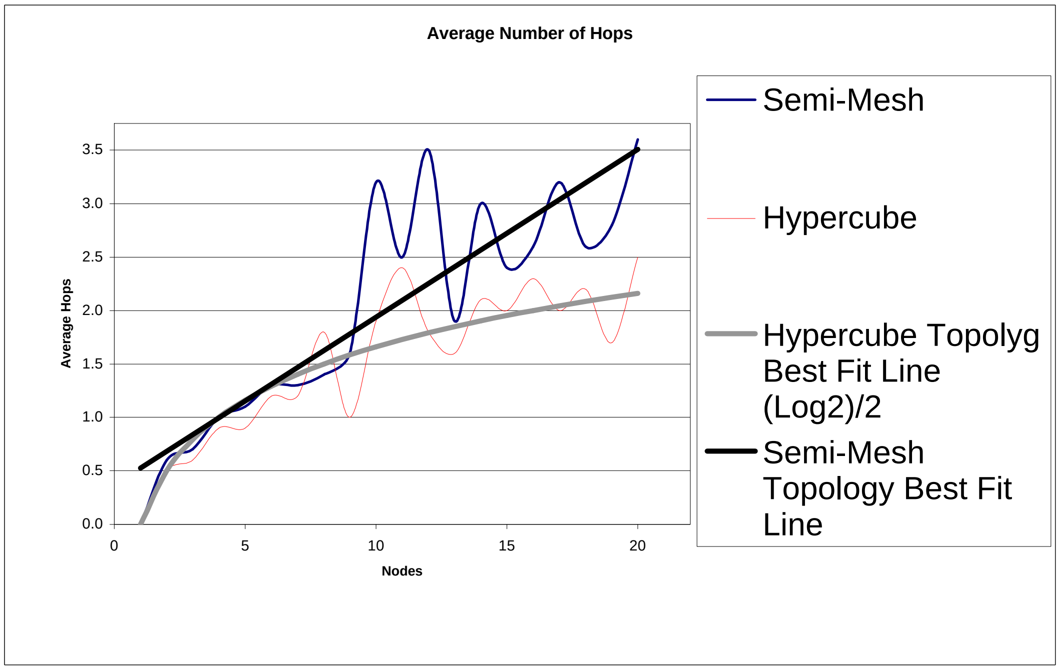


Figure D.2 – Plot of Average Hops vs. Nodes (Semi-Mesh & Hypercube)

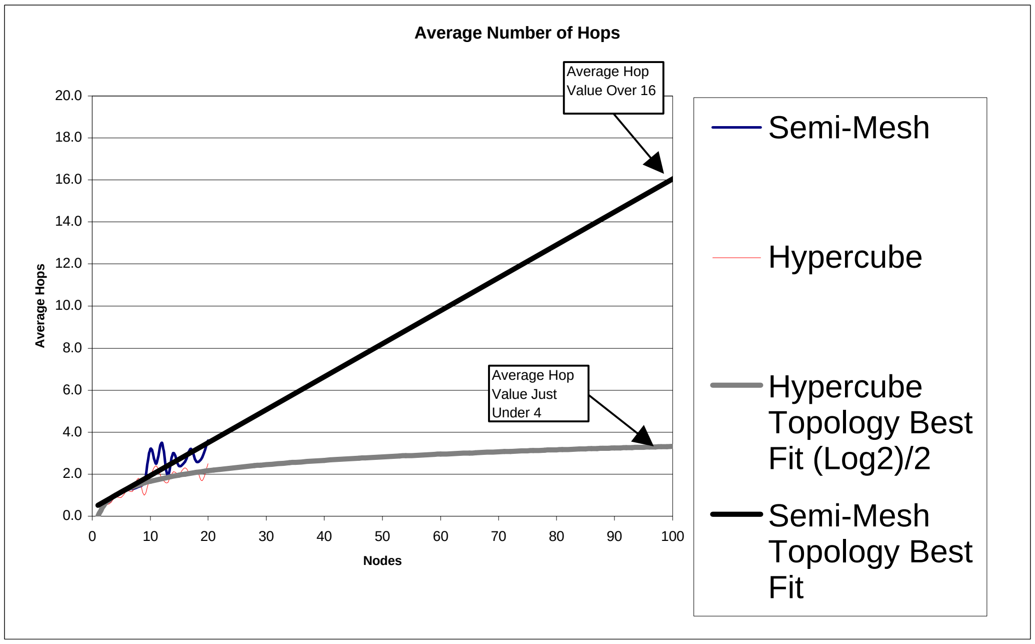


Figure D.3 – Projection of Average Hops vs. Number of Nodes

Appendix E—Monto Carlo Analysis Data

E.1 – Full Mesh Topology – 10 Trials with 20 Node Pairs

	Trial 1			Trial 2			Trial 3		
Nodes	Start Node	Finish Node	Hops	Start Node	Finish Node	Hops	Start Node	Finish Node	Hops
1	0	0	0	0	0	0	0	0	0
2	0	0	0	1	1	0	0	1	1
3	0	0	0	1	1	0	0	2	1
4	1	1	0	2	3	1	1	3	1
5	3	0	1	1	3	1	2	2	0
6	5	1	1	5	0	1	2	2	0
7	1	3	1	2	1	1	2	2	0
8	0	1	1	5	2	1	7	6	1
9	5	3	1	2	4	1	8	8	1
10	9	0	1	7	3	1	6	9	1
11	4	9	1	1	8	1	1	1	0
12	9	9	1	1	9	1	6	4	1
13	3	6	1	11	2	1	9	7	1
14	1	10	1	3	6	1	5	2	1
15	7	6	1	12	8	1	12	14	1
16	7	11	1	12	0	1	7	1	1
17	9	8	1	6	13	1	8	15	1
18	12	4	1	5	17	1	4	15	1
19	15	1	1	3	16	1	4	9	1
20	8	11	1	7	7	1	12	15	1

Trial 4			Trial 5			Trial 6		
Start Node	Finish Node	Hops	Start Node	Finish Node	Hops	Start Node	Finish Node	Hops
0	0	0	0	0	0	0	0	0
1	0	1	0	0	0	0	1	1
1	2	1	1	2	1	1	1	0
3	0	1	1	2	1	1	1	0
1	2	1	1	3	1	2	3	1
0	2	1	1	4	1	1	2	1
5	4	1	6	2	1	6	5	1
0	2	1	6	0	1	0	0	0
2	5	1	1	8	1	3	3	1
3	0	1	5	6	1	6	1	1
2	10	1	9	1	1	9	6	1
11	3	1	2	6	1	9	3	1
6	3	1	5	5	0	3	8	1
11	7	1	10	6	1	0	5	1
7	7	1	11	11	0	4	8	1
12	11	1	13	15	1	3	3	0
0	8	1	5	2	1	16	4	1
14	16	1	11	16	1	17	9	1
17	15	1	12	18	1	15	2	1
2	3	1	18	14	1	9	2	1

Trial 7			Trial 8			Trial 9		
Start Node	Finish Node	Hops	Start Node	Finish Node	Hops	Start Node	Finish Node	Hops
0	0	0	0	0	0	0	0	0
1	0	1	0	0	0	1	1	0
1	1	0	2	0	1	2	2	0
2	2	0	3	1	1	1	0	1
4	1	1	4	3	1	4	1	1
3	1	1	4	4	0	0	0	0
1	1	0	5	2	1	3	1	1
4	1	1	7	2	1	3	6	1
7	0	1	3	2	1	0	2	1
4	4	1	2	6	1	3	0	1
8	5	1	3	2	1	8	4	1
7	11	1	9	0	1	5	10	1
2	8	1	2	8	1	2	5	1
3	6	1	4	11	1	11	8	1
12	10	1	8	4	1	2	5	1
4	6	1	8	6	1	5	4	1
13	10	1	1	8	1	7	1	1
2	4	1	9	13	1	6	12	1
7	12	1	2	9	1	0	5	1
1	14	1	3	10	1	13	14	1

Trial 10

Start Node	Finish Node	Hops
0	0	0
0	0	0
1	0	1
2	3	1
3	3	0
4	3	1
3	2	1
2	2	0
3	7	1
1	4	1
2	8	1
3	3	0
4	5	1
3	6	1
7	9	1
2	14	1
0	6	1
11	11	1
0	9	1
6	8	1

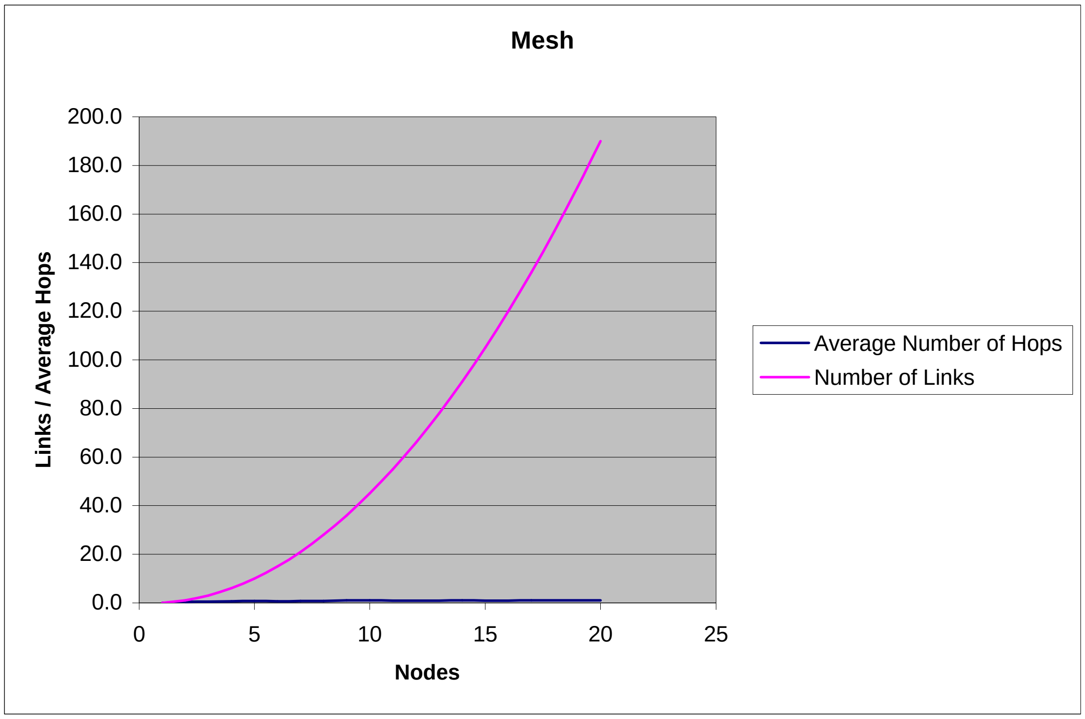


Figure E.1.1 – Average Hops and Number of Links

E.2 – Semi-Mesh Topology – 10 Trials with 20 Node Pairs

	Trial 1			Trial 2			Trial 3		
Nodes	Start Node	Finish Node	Hops	Start Node	Finish Node	Hops	Start Node	Finish Node	Hops
1	0	0	0	0	0	0	0	0	0
2	0	1	1	0	1	1	1	1	0
3	0	1	1	2	0	1	2	0	1
4	1	1	0	2	0	1	2	1	1
5	4	0	2	1	4	1	2	0	1
6	4	4	0	4	3	1	3	4	1
7	5	5	0	0	6	3	0	2	1
8	6	6	0	4	3	1	7	4	3
9	4	1	1	4	0	2	2	5	1
10	3	5	2	5	0	2	2	5	1
11	6	4	2	1	9	5	6	0	3
12	2	3	2	2	3	2	1	7	5
13	3	8	4	5	9	1	12	9	1
14	12	3	5	0	2	1	0	9	6
15	2	2	0	1	3	1	7	4	3
16	14	13	4	15	0	4	4	2	1
17	14	16	3	11	0	5	15	5	2
18	6	14	2	12	15	3	3	5	4
19	7	17	3	10	14	3	1	5	2
20	5	9	4	17	7	3	5	8	2

Trial 4			Trial 5			Trial 6		
Start Node	Finish Node	Hops	Start Node	Finish Node	Hops	Start Node	Finish Node	Hops
0	0	0	0	0	0	0	0	0
0	1	1	0	1	1	0	0	0
1	1	0	0	0	0	1	2	1
2	3	2	2	1	1	0	0	0
1	1	0	1	1	0	3	2	2
1	4	1	3	4	1	3	2	2
4	2	1	3	3	0	4	0	2
5	2	1	1	5	2	0	3	1
1	6	3	5	3	2	6	2	2
4	7	4	7	6	2	1	9	6
7	3	5	8	7	1	10	6	2
3	7	5	3	6	3	10	1	4
9	11	1	9	7	1	7	2	3
2	11	4	2	10	3	1	1	0
3	0	1	14	4	4	2	13	4
9	6	3	11	3	5	0	2	1
13	8	3	8	3	4	11	13	4
15	16	1	6	7	2	1	0	1
15	8	3	12	4	4	6	8	1
6	9	4	7	11	2	16	7	3

Trial 7			Trial 8			Trial 9		
Start Node	Finish Node	Hops	Start Node	Finish Node	Hops	Start Node	Finish Node	Hops
0	0	0	0	0	0	0	0	0
0	1	1	0	1	1	1	1	0
0	1	1	0	1	1	1	1	0
2	0	1	1	3	1	2	3	2
2	3	2	2	2	0	2	1	1
2	3	2	3	2	2	0	4	2
0	4	2	0	1	1	1	3	1
5	4	1	5	6	1	3	2	2
2	4	1	2	4	1	1	2	1
2	9	4	8	2	3	2	8	3
1	3	1	8	9	2	3	4	1
1	8	4	2	9	5	2	3	2
7	11	2	11	11	0	3	0	1
5	4	1	13	8	3	0	9	6
2	10	3	1	3	1	5	12	3
10	12	1	15	4	2	5	1	2
15	4	3	3	11	5	10	8	2
11	1	5	2	10	3	12	7	2
16	0	4	6	13	2	11	15	3
15	9	4	10	4	3	17	3	4

Trial 10

Start Node	Finish Node	Hops
0	0	0
0	0	0
0	1	1
2	1	1
2	3	2
0	2	1
5	3	2
5	7	2
5	8	2
7	3	5
0	6	3
10	4	3
4	9	5
0	1	1
12	4	4
4	10	3
12	10	1
12	8	3
2	16	3
18	3	7

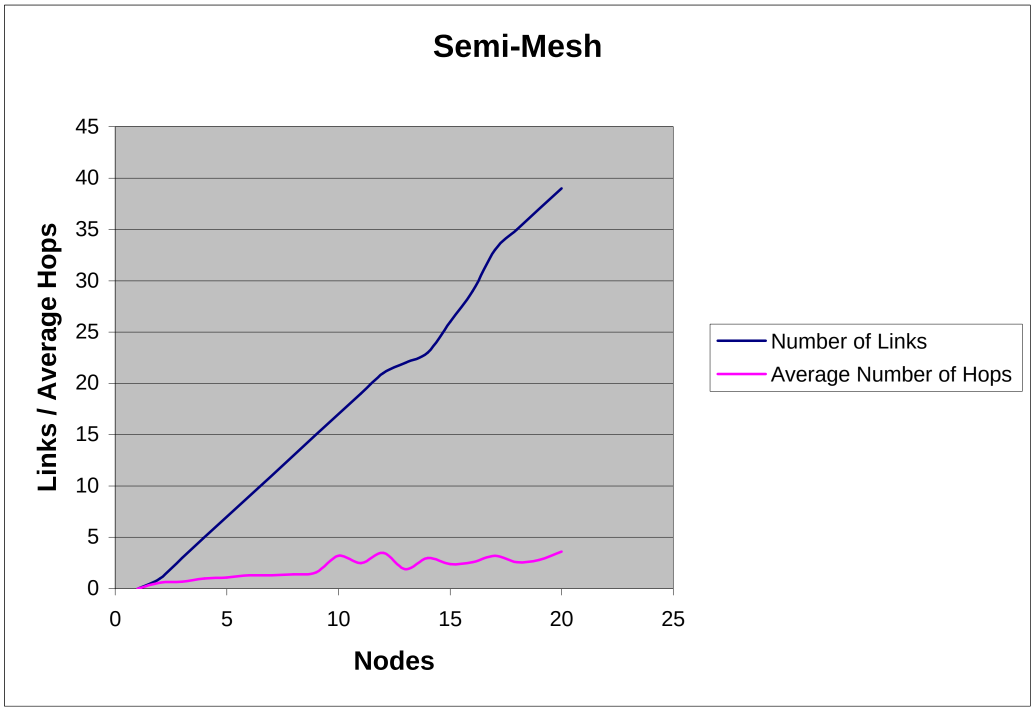


Figure E.2.1 – Average Hops and Number of Links

E.3 – Hypercube Topology – 10 Trials with 20 Node Pairs

	Trial 1			Trial 2			Trial 3		
Nodes	Start Node	Finish Node	Hops	Start Node	Finish Node	Hops	Start Node	Finish Node	Hops
1	0	0	0	0	0	0	0	0	0
2	0	0	0	1	0	1	1	1	0
3	1	1	0	1	1	0	0	1	1
4	0	3	1	2	3	1	0	2	2
5	0	3	1	2	3	1	0	2	2
6	1	4	3	2	2	0	1	3	2
7	3	2	1	1	2	1	4	1	3
8	0	4	2	5	1	2	0	4	2
9	1	1	0	6	1	1	3	2	1
10	4	8	1	6	2	2	5	4	1
11	10	2	1	5	4	1	9	7	3
12	1	9	3	4	5	1	10	3	2
13	10	9	1	7	4	1	9	8	1
14	6	12	3	6	10	3	0	1	1
15	5	9	2	2	9	3	14	7	2
16	4	1	3	10	11	1	13	11	3
17	10	8	2	3	8	3	13	12	1
18	3	8	3	4	8	2	3	3	0
19	9	5	2	9	7	4	2	15	2
20	8	2	4	16	3	3	18	16	1

Trial 4			Trial 5			Trial 6		
Start Node	Finish Node	Hops	Start Node	Finish Node	Hops	Start Node	Finish Node	Hops
0	0	0	0	0	0	0	0	0
0	1	1	0	0	0	0	1	1
1	2	1	0	0	0	0	2	2
2	2	0	1	1	0	2	2	0
2	2	0	1	1	0	2	2	0
1	1	0	4	3	1	3	1	2
0	0	0	1	5	2	2	3	1
4	5	1	5	2	3	6	0	2
5	6	1	1	2	1	3	6	3
7	7	0	0	8	3	9	6	4
3	9	1	6	10	3	6	8	3
1	6	1	5	5	0	7	9	3
4	1	3	0	2	2	4	2	2
12	2	3	11	11	0	13	8	2
14	8	3	13	1	3	1	7	2
11	8	1	6	0	2	4	2	2
7	4	1	15	6	2	12	10	3
3	14	2	16	9	3	11	5	2
15	8	2	7	3	2	4	12	1
4	14	3	10	0	2	15	17	3

Trial 7			Trial 8			Trial 9		
Start Node	Finish Node	Hops	Start Node	Finish Node	Hops	Start Node	Finish Node	Hops
0	0	0	0	0	0	0	0	0
0	0	0	1	0	1	1	1	0
1	2	1	2	1	1	2	2	0
2	3	1	2	0	2	2	3	1
2	3	1	2	0	2	2	3	1
2	2	0	2	3	1	3	1	2
2	3	1	4	3	1	3	4	1
4	2	2	1	0	1	5	6	1
7	2	1	4	6	2	4	4	0
0	3	2	3	6	3	3	1	2
8	1	4	0	8	3	6	3	3
3	2	1	4	11	2	3	10	2
5	5	0	7	0	3	5	4	1
8	4	3	9	0	1	9	7	3
3	0	1	1	13	3	6	6	0
3	6	3	14	1	2	7	10	3
5	6	1	12	13	1	1	13	3
12	13	1	9	2	3	2	12	3
6	1	1	2	7	1	16	16	0
2	18	1	16	0	2	3	16	3

Trial 10

Start Node	Finish Node	Hops
0	0	0
1	0	1
1	1	0
1	2	1
2	1	1
0	3	1
3	2	1
3	1	2
1	1	0
7	2	1
3	1	2
11	0	3
8	6	2
8	10	2
11	10	1
6	9	3
10	5	3
7	8	3
2	10	2
8	17	3

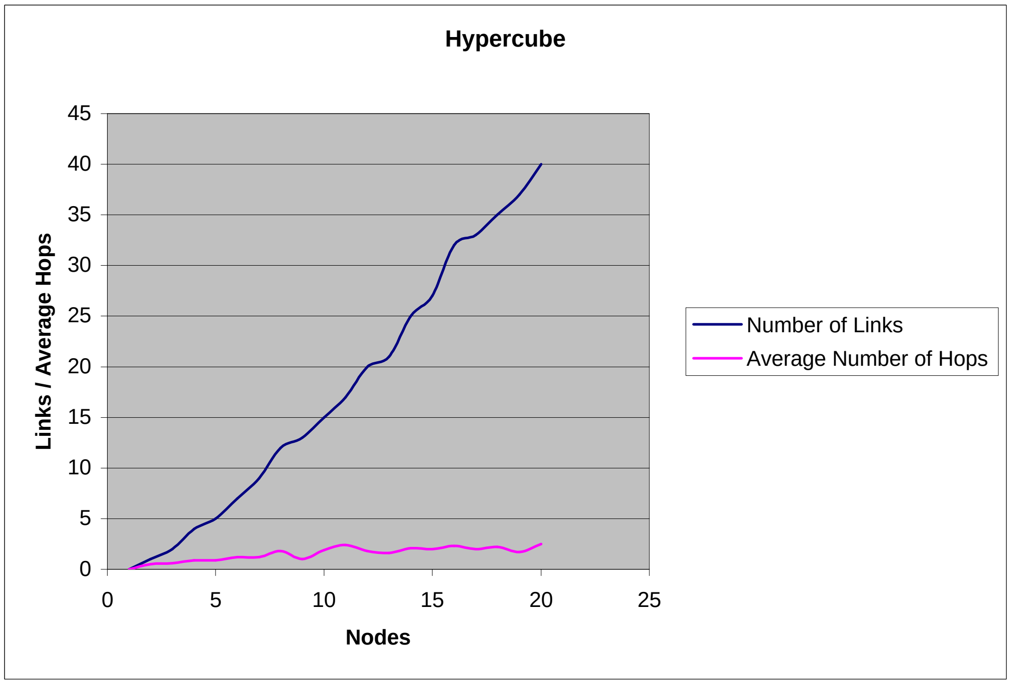


Figure E.3.1 – Average Hops and Number of Links

Appendix F – Network Fragment Healing Model

F.1 – Current Network Fragment Healing Model State

Currently the network fragment healing algorithm does not work properly. Because of the many in depth design iterations the time left for study of the current model was drastically decreased. Several sections of the model do work however, and a framework has been created. Although the network simulation does not currently work, it is feasible that the model may be completed in the near future. A structure has been decided upon and now only the code to complete the design is required. Recently, several accomplishments have been made as the C++ language has become more familiar. The algorithm to active check for faults on the network is written and functions properly. Also, a message queuing method has been decided upon to hold messages that desire to be sent. Through hard work every attempt will be made to complete the network model and generate data.

F.2 – Network Simulation Source Code

```
//---------------------------------------------------------------------------

#include <vcl.h>
#pragma hdrstop

#include "Network2.h"

//---------------------------------------------------------------------------
#pragma package(smart_init)
#pragma resource "*.dfm"
TForm1 *Form1;

//message structure to emulate the types of parameters passed in a LONTalk message
struct MessageStructure{
        int PriorityNumber;
        int BeginningRing;
        int BeginningNode;
        int EndingRing;
        int EndingNode;
        int TheMessage;
        int DeltaBacklog;
        void *nxtmessage;
};

//message structure that contains a flag to show usage and a pointer to a message being
sent
struct MessagePool{
        int used;
        void *msg;
};

//creating a pool of  message structures to be used
MessagePool Pool[MAX_MESSAGES];

//declaration of ring routing tables statically (configured initially in Lon-Works)
int TABLE_0_2[MAX_RINGS][MAX_NODES] =
{{1,1,1,1,1,1},{1,1,1,1,1,1},{0,0,0,0,0,0}};
int TABLE_0_5[MAX_RINGS][MAX_NODES] =
{{1,1,1,1,1,1},{0,0,0,0,0,0},{1,1,1,1,1,1}};
int TABLE_1_2[MAX_RINGS][MAX_NODES] =
{{1,1,1,1,1,1},{1,1,1,1,1,1},{0,0,0,0,0,0}};
int TABLE_1_5[MAX_RINGS][MAX_NODES] =
{{0,0,0,0,0,0},{1,1,1,1,1,1},{1,1,1,1,1,1}};
```

```
int TABLE_2_2[MAX_RINGS][MAX_NODES] =
{{0,0,0,0,0,0},{1,1,1,1,1,1},{1,1,1,1,1,1}};
int TABLE_2_5[MAX_RINGS][MAX_NODES] =
{{1,1,1,1,1,1},{0,0,0,0,0,0},{1,1,1,1,1,1}};

//------------------------------------------------------------------------
class Ring
{
public:
        Ring(int NumberOfNodes, int RingNumber);
        void PassMessageToNode(void);
        void InsertMessageInQueue(void);
        void CheckForFragments(void);
        void SendMessage(void);
        void ReceiveMessage(void);
        int nodes;
        int RingNumber;
private:


};
//------------------------------------------------------------------------
class Node
{
public:
        Node(int NoNum, int RiNum);
        void OutgoingMessage(void);
        int NodeNumber;
        int RingNumber;
        int RouterOrTerminal;
        int MyRouterTable[3][6];
        void BuildRouterTable(int RouterOrTerminal, int
Table[MAX_RINGS][MAX_NODES]);
        void BuildMessage(void);
        //put router table in if necessary
private:
        void PleaseQueueThis(void);
        void WriteToFile(void);
};
//------------------------------------------------------------------------
Node* RingPointer0[6];
Node* RingPointer1[6];
Node* RingPointer2[6];
//------------------------------------------------------------------------
//Initialization of the form
```

```
__fastcall TForm1::TForm1(TComponent* Owner)
        : TForm(Owner)
{
        //user defined
}
//---------------------------------------------------------------------------
//section to determine what happens when start network button is clicked
void __fastcall TForm1::StartNetworkClick(TObject *Sender)
{
        //clear the message boxes upon start up
        mRing1->Lines->Clear();
        mRing2->Lines->Clear();
        mRing3->Lines->Clear();

        //turn off the start button
        StartNetwork->Enabled = FALSE;
        //turn on the stop button
        StopNetwork->Enabled  = TRUE;
}
//---------------------------------------------------------------------------

//section to determine what happens when the stop network button is clicked
void __fastcall TForm1::StopNetworkClick(TObject *Sender)
{
        //terminate the threads for each ring


        //turn on the start network button
        StartNetwork->Enabled = TRUE;
        //turn off the stop network button
        StopNetwork->Enabled = FALSE;
}
//---------------------------------------------------------------------------
//ring constructor
Ring::Ring(int NumberOfNodes, int RingNumber)

{
        switch(RingNumber){

        case 0 : {

                //crete an array of nodes for the ring invoking the constructor method
                for (int i = 0; i<=5; i++){
                        RingPointer0[i] = new Node(i, 0);
                }
        break;
        }
        case 1 : {
                for (int i = 0; i<=5; i++){
                        RingPointer1[i] = new Node(i, 1);
                }
        break;
        }
        case 2 : {
                for (int i = 0; i<=5; i++){
                        RingPointer2[i] = new Node(i, 2);
                }
        break;
        }
        }
}
//------------------------------------------------------------------------
//node constructor
Node::Node(int NoNum, int RiNum)
{
       //declaring initial values in the node
       NodeNumber = NoNum;
        RingNumber = RiNum;
        RouterOrTerminal = 0;
}
//------------------------------------------------------------------------
//method that will build a router table for each router on a ring
void Node::BuildRouterTable(int, int Table[MAX_RINGS][MAX_NODES])
{







}


//------------------------------------------------------------------------
//function called on start up to begin the creation of rings and nodes
void InitializeNetwork(int MessageNumber[MAX_RINGS][MAX_NODES])
{
        //initializing three rings
        //first paramater is the number of rings
        //second is the ring number
        Ring *One = new Ring(MAX_NODES, 0);
        Ring *Two = new Ring(MAX_NODES, 1);
        Ring *Three = new Ring(MAX_NODES, 2);


        int a = 0;
        int b = 0;
        while (a < MAX_RINGS)
        {
                MessageNumber[a][b] = 0;

                while (b < MAX_NODES)
                {
                    MessageNumber[a][b] = 0;
                    b++;
                }
        a++;
        }

}
//-------------------------------------------------------------------------
//method called to inform a node to build and send a message
void MakeMessageNotification(int CurrentRing, int CurrentNode)
{
        switch(CurrentRing){

        case 0 : {
                RingPointer0[CurrentNode]->BuildMessage();
        }

        case 1 : {
                RingPointer1[CurrentNode]->BuildMessage();
        }

        case 2 : {
                RingPointer2[CurrentNode]->BuildMessage();
        }
        }
}
//-------------------------------------------------------------------------
//main program loop routine
//after initialization program will loop until simulation time ends
int main(void)
{
        int MessageNumber[MAX_RINGS][MAX_NODES];
```

```
//function to initialize the network with the number of rings
//and the number of nodes associated with each ring
InitializeNetwork(MessageNumber);

for (int i = 0; i < MAX_MESSAGES; i++)
{
      Pool[i].used = 0;
}

//Filling the Routing table of specific nodes on each ring
//by calling the BuildRouterTable method
RingPointer0[2]->BuildRouterTable(1, TABLE_0_2);
RingPointer0[5]->BuildRouterTable(1, TABLE_0_5);
RingPointer1[2]->BuildRouterTable(1, TABLE_1_2);
RingPointer1[5]->BuildRouterTable(1, TABLE_1_5);
RingPointer2[2]->BuildRouterTable(1, TABLE_2_2);
RingPointer2[5]->BuildRouterTable(1, TABLE_2_5);

//Initialize the counter variables
int CurrentRing = 0;
int CurrentNode = 0;

//Main program while loop to control simulation time
while (NETWORK_TIME < NETWORK_STOP_TIME)
{

      //Function to notify each ring when it should generate
      //a message to send
      MakeMessageNotification(CurrentRing, CurrentNode);

      CurrentNode++;

      //If you are on a node that is greater than the
      //number of nodes on the ring go to the next ring
      if(CurrentNode > MaxNodes[CurrentRing])
      {
            CurrentRing++;

            //If the current ring is larger than the number of
            //rings then go back to ring 0 node 0
            if(CurrentRing > (MAX_RINGS - 1))
            {
                  CurrentRing = 0;
```

```
                }
            CurrentNode = 0;
            }
        }
    return 0;
```

F.3 – Network Simulation Header File

```
//-----------------------------------------------------------------------

#ifndef Network2H
#define Network2H
//-----------------------------------------------------------------------
#include <Classes.hpp>
#include <Controls.hpp>
#include <StdCtrls.hpp>
#include <Forms.hpp>

//-----------------------------------------------------------------------
//global constants—allow for easy manipulation of the network
//only need to change these values
#define MAX_NODES  6
#define MAX_RINGS  3
#define NODE_COUNT  0
#define NETWORK_TIME  0
#define NETWORK_STOP_TIME 10
#define MAX_MESSAGES 100

int MaxNodes[3] = {5,5,5};
//-----------------------------------------------------------------------
//class information for the form
class TForm1 : public TForm
{
__published:   // IDE-managed Components
        TButton *StartNetwork;
        TButton *StopNetwork;
        TMemo *mRing1;
        TMemo *mRing2;
        TMemo *mRing3;
        void __fastcall StartNetworkClick(TObject *Sender);
        void __fastcall StopNetworkClick(TObject *Sender);
private:        // User declarations

public:         // User declarations
        __fastcall TForm1(TComponent* Owner);
};
//-----------------------------------------------------------------------
extern PACKAGE TForm1 *Form1;
//-----------------------------------------------------------------------
#endif
```

F.4 – Active Network Fault Detection

```
/*
1/c Jonathan Vanecko
17 April 2001
This source code could be added to the network simulation as a function or
method that is used to check the physical state of the network on a ring.  It
could be called by the sentinels when they actively break the network and
attempt to find faults.  Because all the nodes are ordered sequentially
any fault found can be isolated and the exact location determined.

The fault check works by starting at a given node (could be passed to the
function) and checking each path.  This path information is contained in
the NetworkStateArray which represents a routing table in the
simulation.  If a node reaches the highest (or lowest) value in the array
then it must loop back to the lowest (or highest) node to accurately model
the ring topology.
*/

//---------------------------------------------------------------------------
#include <iostream.h>
#include <conio.h>
#include <vcl.h>
#pragma hdrstop

//---------------------------------------------------------------------------

#pragma argsused
int main(int argc, char* argv[])
{
        //there is 1 ring
        //there are 6 nodes
        //there are 6 links
        //0-1,1-2,2-3,3-4,4-5,5-0

        //default state of links is 1 == continuity
        //array of network states starting at 0-1
        //and finishing with 5-0
        int NetworkStateArray[6] = {1,  0,  1,  1,  0,  1};
                          //  01  12  23  34  45  50

        //declaring the start and stop node for the physical
        //continuity test
        int nodestart = 5;
        int nodestop = 2;
```

```
//initializing both the clockwise and counter clockwise directions
//to a value of 1
int transverse1 = 1;
int transverse2 = 1;
//initialize loop counter variables
int i,j;

//flag for when the path check crosses 0
bool Rollover = FALSE;

//numbers used to determine if a rollover has occured
int max = 6;
int min = -1;

//the counterclockwise direction continuity check
i = (nodestart -1);
while (i !=(nodestop -1))
{
	//if the current node is equal to the min value
	//they you need to go to the highest node on the ring
	if (i == min)
	{
		Rollover = TRUE;
	}

	//case 1 when the destination node is the sender node
	else if ((nodestart-1)==(nodestop-1))
	{
		transverse1 = 1;
		break;
	}
	//case 2 when the stop node is lower than the start node
	else if ((nodestop-1)<(nodestart-1))
	{
		transverse1 *=NetworkStateArray[i];
	}
	//case 3 where the start node is lower than the stop node
	//and thus the check must go across the 0-5 boundry
	else
	{
		//while the connection is to the right of zero
		if (Rollover == FALSE)
		{
			transverse1 *=NetworkStateArray[i];
```

```
                }
                //but if the connection crosses zero
                else
                {
                        i = i + 8;
                        Rollover = FALSE;
                }
        }
i--;
}
//if communication can be established counterclockwise
//then there is no reason to check the other direction
if (transverse1 == 1)
{
        //let the user know that the path works
        cout<<"The first path works just fine"<<endl;
        getch();
}
//if communication can not be established we must try the
//clockwise direction
else
{

        j = (nodestart);
        while (j !=(nodestop))
        {
                if (j == max)
                {
                        Rollover = TRUE;
                }

                //case 1 when the stop node is greater
                //than the start node
                else if ((nodestop)>(nodestart))
                {
                        transverse2 *=NetworkStateArray[j];
                }
                //case 2 where the stop node is lower than that
                //start node and the check must cross the zero
                //boundry
                else
                {
                        //while the connection is to the right of zero
                        if (j<=5)
                        {
```

```
                    transverse2 *=NetworkStateArray[j];
                }
                //but if the connection crosses zero
                else
                {
                    Rollover = TRUE;
                    j = j - 7;
                }
            }
        j++;
        }
    }
    if (transverse2 == 1)
    {
        //let the user know that the first path was faulted
        //but data can be sent to the node in the other direction
        cout<<"The first path didn't work so we went the other way"<<endl;
        getch();
    }
    else
    {
        //both directions contain faults
        //the only way to access the destination node is by using
        //routers connected to other rings
        cout<<"Neither direction works.  Try using a router"<<endl;
        getch();
    }



    return 0;
}
```