Reliable Subnetworks for Distributed Computing

by

David H. Ritter

BSc. Simon Fraser University, 1988

A THESIS SUBMITTED IN PARTIAL FULFILLMENT OF

THE REQUIREMENTS FOR THE DEGREE OF

MASTER OF SCIENCE

in the School

of

Computing Science

SIMON FRASER UNIVERSITY

September 1994

ISBN 0-612-06788-2

Approval

Name: David H. Ritter

Degree: Master of Science

Title of Thesis: Reliable Subnetworks for Distributed Computing

Examining Committee:

Chairman: Dr. David Fracchia

Dr. Joseph Peters
Senior Supervisor

Dr. Tiko Kameda
Supervisor

Dr. Arthur Liestman
External Examiner

Date Approved: Sept 2, 1994

Abstract

Standard distributed computing models make the simplifying assumption that all components of the network are perfectly reliable. Unfortunately this is not realistic as processors and communication edges fail, messages are lost and networks become disconnected. Even when the model assumes random edge failures, there is no indication as to which edges are likely to fail. *Fault-tolerant* algorithms, designed to run under such models, become very complex as they must not only contend with lost messages, but a dynamically changing network topology. In this thesis we introduce a new distributed computing model in which each processor has *a priori* knowledge of the reliability of each incident edge. Not only does this allow us to avoid using unreliable edges it enables us to derive a *reliable subnetwork* which is less vulnerable to failure than the original network.

Unfortunately, computing the reliability of a network is a difficult task as the problem of computing any of the traditional probabilistic reliability measures is *NP-hard*. Even approximation and bounding algorithms, which have polynomial time complexity, tend to be quite complex. For these reasons, we introduce a new probabilistic measure of reliability. The *Average All Pairs* reliability measure is defined as the average, over all pairs of processors, of the most reliable path between a pair of processors. This new reliability measure can be computed in $O(n^3)$ time.

The reliable subnetwork, based on the previous definition of reliability, can be computed exactly with $O(n^3)$ messages. Unfortunately, this computation results in a large number of messages. An approximate reliable subnetwork algorithm is introduced, in which each processor selects its K most reliable incident edges to be members of the reliable subnetwork. The approximate algorithm has a message complexity of $O(e)$. We show by theoretical and experimental analysis that for $K > 2$ the approximation technique produces solutions which are almost as reliable as the exact solution and very likely to be connected. Finally, we augment the approximation algorithm to ensure connectedness. This algorithm has a message complexity of $O(n\log_2 n + e)$.

Acknowledgements

A thesis is never a solitary endeavor and as such there are a number of people to whom I owe much gratitude.

Foremost, to my parents. Somewhere along the line you instilled upon me the importance of education. That lesson was not lost.

Special thanks to my senior supervisor Joseph Peters, your patience, understanding and encouragement were much appreciated.

Thanks also to Art Liestman for supplying me with numerous books, papers and endless advice.

Most importantly I thank my wife Valerie, whose constant love and support gave me the strength to continue and succeed. You gave up the best summer of our lives so I could concentrate on writing. This thesis would certainly not exist if it were not for you.

Table of Contents

List of Figures

List of Tables

List of Algorithms

Chapter 1

Introduction

In recent years traditional computer systems, based on large mainframe computers with a number of connected terminals, have largely been superseded by networks of autonomous processors. As the physical structures of the computing systems have changed so have the manners in which computers are used. Computer networks enable what were once localized functions, such as database modifications, resource sharing and so on, to be distributed over a number of physically independent sites. Networks have also made possible *distributed computations* in which processors with local knowledge cooperate to solve a common problem. As computer networks become commonplace in society and we become more dependent on them, we place a greater demand on the services of the network. It therefore becomes increasingly important that networks remain reliable. Although the reliability of individual network components has improved over time, as networks become more and more complex they become increasingly vulnerable to unpredictable failure (*e.g.*, edge reactivation, power failure, hardware failure, etc.), or hostile attack. Consequently, no computer network is completely reliable.

In networks, processors communicate and share information by sending messages along incident edges. When the edges are reliable, the task is simple because messages are guaranteed to arrive at the intended receiver in finite time and correct order. When edges are unreliable, transmission is disrupted and messages sent over the edge are lost. A *distributed algorithm* executing on an unreliable network must be *fault-tolerant*. Fault-tolerant algorithms operate under the assumption that any edge may fail at any time. Unfortunately, fault-tolerant mechanisms increase the complexity of the algorithm and do not ensure successful completion or even termination in all circumstances.

The complexity of a fault-tolerant algorithm could be significantly improved if we had information as to what types of failures will take place, what components will fail and when the failures will occur. Unfortunately, this information is generally impossible to obtain. However, if the network includes reliability information, then we are able to predict failures and avoid using components which are very likely to fail. In this thesis we examine ways of using reliability information to ensure successful transmission of messages in distributed algorithms.

This thesis is organized in the following manner. In chapter 2 we examine the basic distributed model and the model which assumes edge faults. We then propose a new formal distributed computing model in which processors are reliable but edges are not. Each edge has associated with it a probability of failure which is known to both adjacent processors.

In chapter 3 we examine the concept of network reliability. We start with a brief survey of network reliability measures and show that traditional measures of reliability are all computationally intractable in general. We propose a new measure of reliability based on the average, over all pairs of processors, of the most reliable path between a pair of processors. The *Average All Pairs* measure is shown to be computable in polynomial time.

Chapter 4 examines the issue of reliable transmission between processors. We propose constructing the *Reliable Subnetwork*, which is a structure based on the new reliability measure. The reliable subnetwork is shown to be as reliable as the original network but has far fewer edges and therefore is less prone to individual edge failure. The algorithm to construct the exact reliable subnetwork is shown to have high message complexity and consequently we develop an approximate reliable subnetwork which can be constructed using considerably fewer messages. The algorithm constructs the reliable subnetwork by selecting the K most reliable edges of each processor. Unfortunately, the approximate reliable subnetwork is not guaranteed to be connected. We evaluate the connectivity of the approximate reliable subnetwork and outline an algorithm to repair the disconnection, if it exists.

In chapter 5 we evaluate the approximation algorithm in terms of reliability. The exact reliable subnetwork is identical, in terms of reliability, to the original network. The approximate reliable subnetwork does not have this guarantee. In this chapter we experimentally evaluate the approximate reliable subnetwork and show that in most cases the reliability is only slightly less than the exact solution.

In chapter 6 we present a more detailed summary of the results and conclusions reached. We end the thesis with a discussion of interesting observations and future research directions.

Chapter 2

A Distributed Computing Model with Reliability Information

2.1 Standard Distributed Model Definition:

The standard distributed computing model, as outlined in [S83] and [LP86], is represented by a simple graph $G = (V, E)$, where $|V| = n$ and $|E| = e$. V is the set of vertices in the graph, each representing a processor in the network. In this thesis the terms *node*, *processor* and *vertex* may be used interchangeably, although "processor" is generally applied when discussing networks and "vertex" is generally used when discussing graphs. The *order* of the network refers to the number of nodes. E is a set of edges of the network, each representing a direct bidirectional communication link between two processors, such that $E \subseteq V \times V$. $e_{i,j} \notin$ E if $i = j$. The *size* of the network refers to the number of edges. Two processors, $x,y \in V$, are said to be *neighbours*, or *adjacent*, if $e_{x,y} \in E$. The *neighbourhood* of a processor is then the set of all neighbours of that processor. An edge is *incident* to a processor if it directly connects that processor to a neighbour.

The standard model includes the following basic assumptions concerning processors and messages in the network:

Processor Assumptions:

P1. Processors are identical and indistinguishable except for a distinct label known as the processor *identity*.

P2. The network has no central controller and processors share no common clock.

P3. Each processor has a local non-shared memory of bounded capacity.

P4. The number of processors in the network is fixed but unknown to individual processors.

P5. Each processor knows how many neighbours it has and can distinguish among them. Aside from this no other topological information is known.

P6. Processors may only communicate with each other by passing messages.

P7. Messages received at a processor are processed in the order that they arrive. If more than one message arrives at a processor at the same time (*i.e.*, via a different edge) then they are processed in arbitrary order.

P8. Processing time is negligible compared with transmission delays.

P9. Any subset of $0 < j \leq n$ processors, called *initiators*, may start a distributed computation spontaneously.

P10. Processors are completely reliable and do not fail.

Message Assumptions:

M1. The network is asynchronous, which is to say that distributed computations are message driven.

M2. Messages sent along the same edge are delivered in the same order in which they were sent.

M3. Messages are short, on the order of $O(\log_2 n)$ bits.

M4. Messages are transmitted without error.

M5. Messages are delivered in finite but unbounded time.

In addition to these basic assumptions, we extend the standard model by assuming that each processor has *a priori* knowledge of its neighbourhood, which is to say that a processor knows the identity of each of its neighbours. Notice that this assumption is not essential to the model, as neighbourhood information can be computed by ignorant processors with O(e) messages.

In the standard distributed model the assumption is made that the network is completely reliable, which is to say that all network operations may be performed successfully. The purpose of this simplifying assumption is to enable us to focus on the fundamental aspects of the network rather than specific exceptional characteristics. The reliability of the network is facilitated by certain assumptions concerning the reliability of both processors and communication links. Processors are assumed to be completely reliable (*P10*) which is to say that no processor hardware failures take place for the duration of the computation. As a corollary to this condition, we assume that messages received by a processor are never lost and always dealt with in the appropriate manner. Notice that this says nothing about the correctness of the distributed algorithm, which may contain erroneous statements.

Edges are assumed to be reliable in two respects. First, message transmission (*i.e.*, the physical sending of bits over the communication links) is error free (*M4*). This implies that the message sent by the initiator is identical to the message received at the intended destination. Notice that this assumption does *not* imply that the message is actually received by the intended processor. Consequently, the second edge reliability assumption states that a message sent is eventually delivered to the intended recipient (*M5*). This assertion implies that no messages are ever lost. However, it is a weaker statement than the assertion that the edge is completely reliable since it allows for the possibility of an edge to fail, provided that it does not remain failed forever and it never causes a message to be lost. This assumption implies that the edge must appear to be reliable within the network. For general purposes this is sufficient.

2.1.1 Distributed Complexity Measures:

In non-distributed environments, the efficiency of algorithms is usually measured in terms of processor execution time, given in terms of the size of the input. In distributed situations this measure does not apply since the algorithm is executed among a number of different processors, possibly in parallel, and processing time is negligible when compared to transmission and queuing delays (*P8*). As a result, the efficiency of a distributed computation will be measured entirely in terms of communication activities. A distributed algorithm is subject to the following complexity measures:

Message complexity: This is the total number of messages sent during the execution of a particular distributed algorithm. This is usually given in terms of the number of vertices and/or edges of the network.

Bit complexity: This is the average number of bits transmitted over the network for the duration of the distributed algorithm. As justification of this measure, consider that the transmission of a 1K bit message, such as a file transfer, uses considerably more network resources than the transmission of a 1 bit message used as an acknowledgment.

Total Execution time: This is the time elapsed between the time the first processor starts the execution of a computation and the time the last processor terminates its execution. For the purposes of analyzing the execution time complexity we must contradict some of the standard model assumptions. Specifically, because the network is asynchronous and therefore message transmission times are unbounded, the total execution time is measured assuming that the network is synchronous and that the sending of a message requires one time unit. Since processors have different local clocks, we assume that execution time is measured in global time units with respect to an external observer.

2.2 A Distributed Model With Edge Faults:

The standard distributed computing model makes the assumption that the network is completely reliable and all operations performed on it are done so successfully. But realistically, this is not the case; individual components may fail, messages are lost or duplicated and networks become disconnected. In general, network failures may be modeled in a variety of different ways. Network failures can range from the simple *fail-stop* failures to the more malicious *Byzantine* failures. Fail-stop failures represent the situation in which a network component may fail but never recover for the duration of the distributed computation. With Byzantine failures, a processor, or group of processors, communicate in a chaotic or hostile manner. Byzantine failures are difficult to predict and harder to prevent. These type of failures are relatively uncommon, and we will not consider them further. Fail-stop failures, on the other hand, are too restrictive as it is not uncommon for a network component to fail for a short period of time and then resume normal operation. For these reasons, we consider *intermittent* (also known as *transient*) failures in which a network component can fail and subsequently recover infinitely many times during the distributed computation. More specifically, for the duration of this thesis we consider a distributed computing model in which processors are completely reliable[1] but the edges are subject to intermittent failures.

The distributed computing model with edge faults can be derived from the standard model by withdrawing the message assumption *M5* which states that messages are delivered in finite but unbounded time. Instead, we place no restrictions on the length of time for which a message is in transit. A failed edge is represented as having an infinite transmission delay. This model was first introduced by Afek and Gafni [AG88] and is referred to as the *∞-delay model.* The ∞-delay model is a weaker model than the standard model (the standard model is a special case of it).

Although the ∞-delay model can model any edge failure which may occur in the network, it has an undesirable property; messages sent over non-failed edges experience a

[1] Although the assumption of completely reliable processors is unrealistic, processor failure can be approximated in this model by the failure of all adjacent communication links.

finite transmission delay, but because the delay is unbounded[2] it becomes impossible to distinguish between it and the infinite delay that results when a message is sent over failed edge. More simply put, with the ∞-delay model it is impossible to determine whether or not the edge is functional. This problem is solved by adopting the *dynamic* distributed network model [AAG87]. The dynamic model has the same basic assumptions as the ∞-delay model but includes the additional assumption that each failure or recovery of an edge is eventually reported to both adjacent processors by some underlying edge protocol. The dynamic model is a special case of the ∞-delay model.

Based on the previous assumptions of the dynamic model, the following implications concerning the properties of edges within a non-reliable network are apparent [AE86]:

E1. The network is dynamic and individual edges may fail and subsequently recover. The number of times an edge may fail and recover during the course of an algorithm is unbounded.

E2. Edges have the property of being either in an *operational* state, in which messages arrive error-free in finite time, or *non-operational* state, in which messages are not received or take infinite time. The state of a particular edge is known by both adjacent processors which furthermore can detect changes in the state of the edge within a finite time of the change. We make no assumption as to the particular medium of the edge itself or of the actual mechanism by which a processor can determine functionality of an adjacent edge, only that such is possible.

E3. A message can be sent and received over a edge only during an operational state which lasts at least the duration of the particular message transmission.

E4. When an edge recovers no message may be in transit through it and any messages in transit on the edge at the time of the failure are lost.

[2] The actual transmission delay of a message is influenced by such static factors as the throughput of the edge, transmission protocol and by time dependent factors such as queuing delay, size of the message and transmission error rate. For these reasons the time to transmit a message is unbounded.

We make no assumption concerning the duration of failure of any particular edge nor do we place any upper bound on the number of times any edge can fail and subsequently recover. In general, temporarily failed edges have a finite but unbounded delay due to failure. An edge which has permanently failed (*i.e.*, will not again become operational for the duration of the algorithm) is said to have infinite delay. Nevertheless, we adopt the assumption of *infinitely frequent stability* [AAG87]. Under this condition we assume that the network configuration stabilizes for a long enough period of time to allow some constant number of communication activities. Notice that this assumption is not unreasonable as most communication failures are infrequent [G82].

In addition, we allow both arbitrary topologies and arbitrary edge failures (topology changes) within the network, however we assume the network to be *eventually connected* [AE86]. A network is eventually connected if there exists no edge-cut in which all edges are permanently failed (*i.e.*, no edge-cut persists forever). This assumption is necessary since any permanent edge-cut disconnects the network. Notice that the property of eventual connectivity does not imply that the network is ever connected as a whole.

2.3 A Distributed Model with Reliability Knowledge:

In the previously discussed distributed model, edges were assumed to fail randomly and at completely unexpected times. Distributed algorithms designed to run under such a model are complex because they must not only contend with asynchronous message transmission, but also with lost messages due to a dynamically changing topology. The natural question arises as to what difference, if any, the explicit knowledge of the probability that the edge is operational will have on the functionality and complexity of distributed computations. To the best of our knowledge, this question has never been addressed within a distributed computing context.

In this thesis we introduce a new distributed computing model based on the dynamic distributed model, with the additional assumption that individual processors have explicit knowledge of the reliability of all incident edges. More formally, each edge $e_{i,j} \in E$ has associated with it a given probability $r_{i,j}$, $0 \leq r_{i,j} \leq 1$, which is known *a priori* to both processors $i,j \in V$. The quantity $r_{i,j}$ is the reliability of the edge $e_{i,j}$ and is defined to be the probability that the edge is in an operational state. Conversely, the probability that the edge $e_{i,j}$ is in a failed or non-operational state is equal to $1 - r_{i,j}$. An edge $e_{i,j}$ is completely reliable (*i.e.*, is always in an operational state) if $r_{i,j} = 1$. An edge $e_{i,j}$ is completely unreliable (*i.e.*, is never in an operational state) if $r_{i,j} = 0$. We make the following assumptions regarding the reliability of edges in the network:

R1. The reliability of an edge is independent of the reliability of any other edge in the network. The assumption is that edges fail due to random factors which affect them individually. This assumption is not valid in every circumstance since events, such as natural disasters, tend to incapacitate topologically local sections of the network. In the absence of this assumption, the reliability calculations would be substantially complicated as all applicable conditional probabilities must be known. Also, as pointed out in [H83], the knowledge of what the conditional probabilities should be are generally not available.

R2. The reliability of an edge is fixed for the duration of the distributed computation. This assumption reflects the idea that an edge is a combination of physical components. The reliability of an edge is therefore greatly dependent on the individual reliability of its components. Although physical systems are never completely reliable, the reliability generally stays fixed or depreciates very slowly over a fixed period of time. This is especially true of electronic components, such as those found in communication hardware.

The reliability of an edge is a quantified indication of the probability that the edge is in an operational state. The definition is purposely vague so as to encompass more restrictive notions of reliability. As an example, the reliability of an edge may represent the *availability* of the edge over a given period of time. The *availability* of an edge refers to the percentage of the period of which the edge is in an operational state. If the period is sufficiently large, we would expect the reliability of an edge to be quite high. In contrast, the reliability of an edge may be the probability that the edge does not fail (*i.e.*, always in an operational state) for a given period of time. Given this definition of reliability, as the period increases we expect the reliability of an edge to decrease. Although both definitions of edge reliability are quite different, this model may be used with either one. Regardless of how edge reliability is defined, the reliability attributed to each edge is not an exact value, such as edge length could be regarded as exact, but rather an estimation of the true reliability.

In this thesis we do not specify how the edge reliabilities are determined, nor by what mechanism processors are informed of the reliabilities of incident edges. This is done for two reasons. First, as previously suggested, the reliability of an edge may be defined in different ways. Consequently, the mechanism by which reliability information is gathered may differ depending on the definition used. Second, the reliability of an individual edge may be determined in different ways. The reliability of an edge may be determined experimentally, by monitoring the edge over a given period of time. By keeping statistical information, such as the number of lost messages or the amount of time the edge is in an non-operational state, the reliability of the edge may be calculated. Alternatively, the reliability of an edge may be calculated from an inherent property of the transmission media. For example, fiber optic cable tends to be extremely reliable and therefore would have a high reliability value. In contrast, a twisted pair cable tends to be much more susceptible to electromagnetic interference [S85] and as a result, may have a low reliability value.

In the remaining chapters of this thesis we use the distributed model with reliability knowledge as a foundation on which we build our results.

Chapter 3

Network Reliability

3.1 Introduction:

In the new distributed computing model, the network is represented by a *probabilistic graph* $G = (V,E)$ where V is the set of vertices or processors representing communication sites and E is the set of edges representing communication links between pairs of vertices; let $|V| = n$ and $|E| = e$. Associated with each edge $e_{ij} \in E$ is the reliability r_{ij} which is the probability that the edge is operational. Assumptions concerning the reliability of edges, as given in section 2.3, also hold. An example of a network represented by a probabilistic graph is shown in figure 3.1.

The knowledge of individual edge reliabilities enables us to take advantage of the idea of *network reliability*. In this thesis, we use network reliability in order to construct subnetworks, from the original network, which do not decrease in reliability. This is discussed further in chapter 4.

A network is considered to be *operational*, in the presence of edge failures, provided that each processor in the network can communicate with any other. The reliability of the network is then a quantitative indicator of how operational the network is. This definition of network reliability is too vague to be workable and consequently requires further clarification. Given the network configuration of figure 3.1 we may ask a number of questions: How reliable is this network? Which edges may be removed from the network without reducing its reliability? and so on.

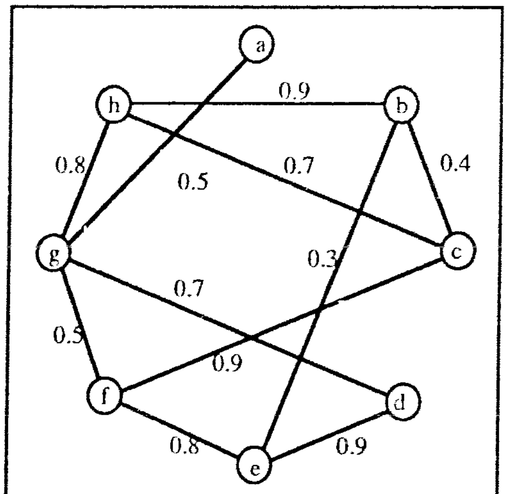


Figure 3.1: Typical Network with Edge Reliabilities

The answers to the above questions are unfortunately not obvious since network reliability, in the broadest sense of the term, strongly depends on a number of interdependent network characteristics such as edge and node reliability, types of expected network failures, ratio of edges to nodes, network topology and (possibly) expected message traffic. For these reasons there exists no universally accepted formal definition of reliability. Instead, existing measures are defined in relatively narrow terms and seem to capture only certain aspects of reliability. In the following section we examine a number of existing reliability measures.

3.2 Measures of Reliability:

A network, *G*, is considered to be "reliable" if certain fundamental communication activities can be performed successfully. Colbourn [C86] identifies a number of basic network operations, but in general the most important criterion for a reliable network is the ability of all processors to communicate with all other processors. A reliability measure based on this criterion is a quantification of the connectedness of the network and is known as the *all-terminal reliability* measure. Frank & Frisch [FF70] and Wilcox [W72] identify a number of different approaches for evaluating all-terminal reliability and related reliability measures.

Reliability measures can be naturally divided into *deterministic* and *probabilistic* measures. With deterministic reliability measures, the individual reliabilities of network components, such as edges and vertices, are generally not known. Instead, reliability is defined in terms of discrete measures such as the number of components that must fail in order to disrupt network operation. The simplest and most widely used deterministic reliability measure is the *edge connectivity* (or simply connectivity) of the network which corresponds to the minimum number of edges which must fail in order to disconnect the network. Deterministic measures are usually applicable in situations where network components fail for non random reasons such as hostile action of an intelligent adversary [H83]. Probabilistic reliability measures, on the other hand, assume specific knowledge of the reliability of applicable network components (in our situation only the edge reliabilities are known) and are applicable when component failures occur randomly. Probabilistic measures define reliability to be the probability that the network remains operative. Because the reliability of the network is given as a probability of success/failure, probabilistic measures are more meaningful than their deterministic counterparts. Unfortunately, probabilistic measures are also much more difficult to compute [W72]. In this thesis we concentrate on probabilistic measures.

The most widely used probabilistic reliability measure is the *probabilistic connectedness* of the network. This reliability measure is closely related to edge connectivity and is the probability that the network remains connected in the presence of failures. Intuitively, both connectivity and probabilistic connectedness are measures of the *survivability* of the network. Survivability refers to the ability of the network to remain operational in the presence of edge failures. Both measures are most meaningful if all vertices are of equal importance.

In theory, computing the probabilistic connectedness, *Con*, of a network *G* is relatively simple. Since the reliability $r_{i,j}$ corresponding to edge $e_{i,j} \in E$ in the network is statistically independent of the reliability of all other edges, computing the probability of the network being in any specific state, defined by the set of operational edges E_1 and failed edges $E_2 = E/E_1$, is simply:

$$\mathrm{Con}(G) = \prod_{e_{i,j} \in E_1} r_{i,j} \times \prod_{e_{i,j} \in E_2} (1 - r_{i,j})$$

Evaluating the probabilistic connectedness is then a matter of summing the resulting probabilities over all operational states (*i.e.*, states in which the network remains connected). Since there are *e* edges, there exists 2^e possible states which the network can be in, all of which must be enumerated. The algorithm has an execution time which is exponential in the size of the network, making this method impractical for all but small networks. While it is possible to improve upon the complete state enumeration algorithm by computing minpaths or mincuts, in the general case all known methods which compute exact solutions take exponential time in the worst case. In fact, it has been shown that computing the probabilistic connectedness, as a network reliability measure, for general networks is #P-complete [C87]. This has led researchers to develop efficient algorithms to compute bounds on the reliability as well as to examine restricted classes of networks in which exact algorithms have been found to run in polynomial time. For a more complete treatment of the subject of connectivity and probabilistic connectedness as measures of reliability see [C89], [H92],[T81] and [W72].

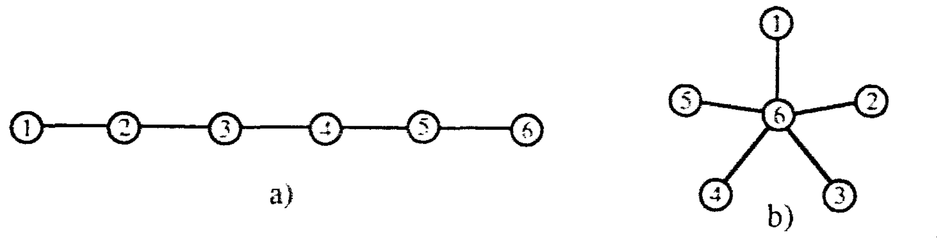


Figure 3.2: The effect of configuration on connectivity

Although the use of probabilistic connectedness as a reliability measure is widespread, it can be shown to be inappropriate under certain circumstances. Consider the two networks shown in figure 3.2. Intuitively, one would expect the network shown in figure 3.2*a* to be less reliable than the network in figure 3.2*b* since any edge failure, particularly one near the center of the network (*i.e.*, the edge connecting nodes 3 and 4), will disconnect a larger proportion of pairs of nodes in the first network than in the second. It is often more serious for a edge failure to isolate one half the processors in the network from the other half than a single processor from the rest. However, in computing reliability based on network connectivity both networks in figure 3.2 will be equally reliable since connectivity reflects the minimum number of edges that must fail to disconnect the network. Consequently, connectivity and probabilistic connectedness are not sensitive to the topology of a network in a way that we would intuitively expect.

The inherent problems of connectivity based reliability measures are partially remedied by *generalized cohesion*. Cohesion is a more general form of connectivity, although less widely used, in which reliability is defined as the minimum number of edges that must be removed from the network in order to isolate any subset of m processors from the rest of the network [W72]. When $m = 1$ cohesion is equivalent to edge connectivity. The analogous probabilistic measure is *probabilistic cohesion* which is defined as the probability that any subset of m processors remains connected in the presence of edge failures. The benefit of this reliability measure is that it enables us to discriminate between different types of failures resulting in a disconnection of the network. When m is small this measure captures the network's ability to withstand minor failures which only disconnect a proportionately small number of processors. When m is large (*i.e.*, $m = n/2$) it captures the ability of the network to withstand major failures which disconnect a large proportion of the processors. While this measure is clearly more versatile than probabilistic connectedness, in practice it is often difficult to determine the appropriate value of m. As an added drawback, probabilistic cohesion also suffers from exponential time complexity making it unattractive as a reliability measure.

It was suggested in the example of figure 3.2 that not all network edge-cuts are equally critical. While connectivity is a particularly good reliability measure for analyzing the probability that all processors are able to communicate, it fails to convey any information regarding the degree of disconnection or the seriousness of the disconnection when edge failures occur. Often, one is more concerned that most communicating pairs of vertices remain connected rather than the whole network remaining connected. Colbourn [C87], has suggested that in many applications a more appropriate measure of reliability is the expected number of processor pairs which can communicate. He has termed this measure the *resilience* of the network since it captures the network's capacity to withstand failures. The resilience, *Res*, of a network, G, can be formally defined as:

$$\text{Res}(G) = \sum_{G_1 \subseteq G} \text{Prob}(G_1) \times \text{Pairs}(G_1)$$

where G_1 is a subnetwork G. Prob(G_1) is the probability that all edges in G_1 are operational and all edges in G_1/G fail. Pairs(G_1) is the count of all the communicating processor pairs in G_1. As with connectivity, computing the resilience of a network is conceptually simple. The resilience of a network is determined by examining all possible subnetworks of the network. For each subnetwork, both the number of communicating processors within the subnetwork and the probability that the subnetwork exists are computed. Both of these values may be computed in polynomial time. Unfortunately, there exists an exponential number of subnetworks in the original network which makes any algorithm using this approach impractical for any reasonably sized network. Colbourn in [C87] developed a polynomial time algorithm to compute resilience on a specific class of networks known as series-parallel, however he also showed that for planar networks the problem is #P-complete.

In general, a number of different strategies for computing exact probabilistic network reliability exist. Unfortunately, exact computation is often prohibitively time-consuming and there is sufficient evidence to suggest that traditional probabilistic reliability problems are inherently intractable. Nevertheless, some restricted classes of graphs, such as trees, series-parallel graphs and complete graphs, enable exact computation of certain measures of network reliability in polynomial time. Such classes of graphs, however, are usually too restrictive and therefore not applicable to most networks.

The intractability of computing exact reliability measures has motivated the development of *approximation* algorithms, which compute estimated reliability values with a specific confidence level. Intractability has also motivated the development of efficient algorithms to compute upper and lower *bounds* on the reliability[CH88]. While approximation and bounding techniques achieve polynomial time complexity, the algorithms still tend to be quite complex, as many compute edge connectivity or enumerate edge cut sets. As an added deterrent, most approximation and bounding algorithms apply only when all edge probabilities are equal [BC86]. For these reasons the majority of approximation and bounding algorithms are less than perfect solutions to the complexity problem.

In the following section we introduce a new measure of reliability which can be exactly computed in polynomial time.

3.3 A New Measure of Reliability:

The previously discussed methods of computing network reliability were shown to be too computationally complex to compute the reliability of even moderate sized networks. For this reason we introduce an efficiently computable measure of reliability based on the reliability of simple paths, rather than edges, between pairs of processors. In essence, this scheme defines the reliability of the network to be the average of the reliabilities of the most reliable path between each pair of processors within the network.

A *path* in a network is a sequence of edges, $e_1, e_2, ..., e_j$ such that e_{i-1} and e_i are adjacent. For the purposes of this thesis we assume that all paths are simple, which is to say that all vertices on the path are distinct. We denote by $P_{x,y}$, a path from vertex x to vertex y. Then, the reliability, $R_{x,y}$, of any path $P_{x,y}$ is the probability that the path is operational. This in turn is the probability that all edges on the path are themselves operational and is computed as

$$R_{x,y} = \prod_{e_{i,j} \in P_{x,y}} r_{i,j}$$

This is the product law of reliabilities which is applicable to any series of independent components. We assume that communication between different processes within the same processor is completely reliable and therefore $R_{x,x} = r_{x,x} = 1.0$. If no path exits between two vertices x and y (*i.e.*, the network is disconnected) then $R_{x,y} = 0$.

We define the *maximum reliable path*, $MP_{x,y}$, between processors x and y, as the path which is the most reliable of any path between the two processors. The Reliability, $MR_{x,y}$, of the maximum reliable path is defined as:

$$MR_{x,y} = \prod_{e_{i,j} \in MP_{x,y}} r_{i,j}$$

Definition 3.1: *Average All Pairs Reliability.* The Average All Pairs reliability *AR* of the network *G* is :

$$AR(G) = \begin{cases} \dfrac{\sum_{i,j \in V} MR_{i,j}}{\binom{n}{2}} & ; \text{if G is connected} \\ 0 & ; \text{otherwise} \end{cases}$$

The *Average All Pairs* measure of network reliability is a probabilistic quantification, in the range [0...1.0], of the reliability of the network. A network G_1 is more reliable and thus more likely to remain operational in the presence of failures than network G_2, if $AR(G_1) > AR(G_2)$. A disconnected network is assumed to have a reliability of 0 to reflect the belief that communication from one vertex to any other vertex within the network is the most fundamental requirement of a reliable network.

Because the reliability of a network is dependent upon such factors as topology and individual component reliability, we can not expect a true numerical representation of reliability. Instead, all reliability measures compute an imprecise value based on some limited criteria. For example, *probabilistic connectedness* measures the probability that the network remains connected, while *resilience* measures the expected number of connected vertices. The *Average All Pairs* reliability measure, on the other hand, measures the average probability that the most reliable path between each pairs of vertices is operational. Because the most reliable path is used in the calculation, this measure is essentially an upper bound on the probability that all pairs of processors can communicate. When reliability information is known in a distributed environment, this measure is particularly appropriate since the concept of paths closely models the passing of messages throughout the network.

We make two observations concerning the *Average All Pairs* reliability measure. First, the measure is *topology sensitive*. By this we mean, the measured reliability of a network changes as the topology changes. Consider again the example in figure 3.2. When connectivity is used as a reliability measure, the reliabilities of networks *a* and *b* are identical, despite having very different topologies. The *Average All Pairs* measure, on the other hand, favors networks with short paths between vertices. This is inherent in the reliability definition since the reliability of a path degrades quickly as its length increases. For example, if we assume all edges in networks *a* and *b* have a reliability of 0.9, then network *a* has a reliability of 0.78 while network *b* has a reliability of 0.84. Clearly, this measure favors short path lengths and networks of small diameter.

Second, with the *Average All Pairs* measure, the influence which an individual edge has on the overall reliability of the network is directly affected by its number of occurrences in the most reliable paths between vertices. Assuming that a message is always sent over the most reliable path between sender and receiver, a very unreliable edge will likely never be used and therefore will have no effect on the overall reliability of the network. Clearly this line of reasoning is valid since, if the edge is unreliable and thus not used, its failure in the network would be of little significance. In contrast, an unreliable edge which is a *cut edge* of the network, will have a large negative influence on the reliability of the network. This fact alone, makes *Average All Pairs* a particularly good measure for the distributed computing environment.

3.3.1 Computing Network Reliability:

The major advantage of the *Average All Pairs* measure of reliability over more traditional measures is found in the computational complexity of computing the exact reliability of a given network. While the previously discussed methods have exponential running time, computing *Average All Pairs* reliability is relatively easy and can be accomplished in polynomial time. The algorithm essentially consists of computing the most reliable path between $\binom{n}{2}$ pairs of vertices in the network. The problem of computing the most reliable path between two vertices is analogous to the problem of finding the shortest path between two vertices. In the shortest path problem, each edge has a (positive) weight or cost associated with it rather than a reliability. The problem is then to find a path from the source vertex to a target vertex which has minimum weight. In the following theorem we show that the most reliable path between two vertices can be found by the shortest path algorithm.

Theorem 3.1: The most reliable path problem is computationally equivalent to the shortest path problem.

Proof: We show that the most reliable path problem, by a simple algebraic transformation, can be solved by any shortest path algorithm and is therefore computationally equivalent. The reliability matrix, *r*, can be transformed to a non-negative edge weight matrix, *d* , as follows:

$$d_{i,j} = \begin{cases} -\log r_{i,j} & ; \text{if } e_{i,j} \in E \text{ and } r_{i,j} \neq 0 \\ 0 & ; \text{otherwise} \end{cases}$$

Let $\phi_{\xi,y} \subseteq E$ be the set of all edges in the shortest path from vertex x to vertex y. The reliability, $MR_{x,y}$, of the most reliable path connecting vertex x to vertex y in G can be computed as:

$$-\log\left(MR_{x,y}\right) = \sum_{e_{i,j} \in \phi_{x,y}} d_{i,j} \qquad \blacksquare$$

The shortest path problem has been studied extensively and a number of efficient algorithms exist for different variations of the problem (see [AHU74]). Although conceptually more simple, the basic problem of finding the shortest path between two specific vertices is no easier than finding the shortest path between a specific vertex and all others (single source shortest path problem). This can be computed in $O(n^2)$ time, using an adjacency matrix, by Dijkstra's algorithm. The problem that most closely resembles what we are trying to compute is the all-pairs shortest path problem in which the shortest paths between all pairs of vertices are computed. This is evidently a generalization of the single source problem and can be computed by n-1 applications of Dijkstra's algorithm. Clearly, the all-pairs shortest path problem can be computed in $O(n^3)$ time. If $e >> n$ the time complexity of the algorithm can be improved by using Fibonacci heaps rather than adjacency matrices.

As indicated in theorem 3.1, a shortest path algorithm can be easily transformed into a most reliable path algorithm. The algorithm to compute *Average All Pairs* network reliability, given in Algorithm 3.1, is an adaptation of the all-pairs shortest path algorithm of Floyd [AHU74].

```
        for i = 1 to n do
(1)         for j = 1 to n do
                PR[i,j] = r[i,j]

        for k = 1 to n do
(2)         for i = 1 to n do
                for j = 1 to n do
                    PR[i,j] = max (PR[i,j], PR[i,k] × PR[k,j])

        AR = 0
        for i = 2 to n do
(3)         for j = 1 to i-1 do
                AR = AR + PR[i,j]

        AR = $\frac{AR}{\binom{n}{2}}$
```

Algorithm 3.1: An Algorithm to Compute the *Average All Pairs* Reliability

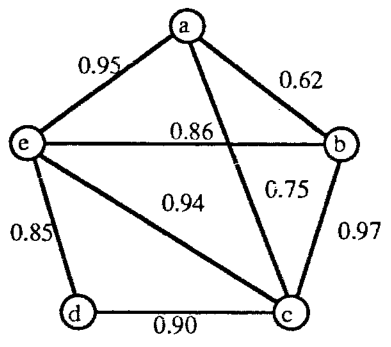
The algorithm is based on dynamic programming techniques in which locally optimum solutions are also globally optimal. The algorithm works by first initializing the adjacency matrix (1), then computing the transitive closure of the adjacency matrix (2). In computing the transitive closure, a path from vertex *i* to *j* is replaced if there exists a more reliable path through vertex *k*. The final step consists of summing the path reliabilities over all pairs of vertices and dividing by the number of pairs to produce the network reliability (3). Notice that after (2) the adjacency matrix contains the reliability of the most reliable path between all vertices.

Theorem 3.2: The algorithm of figure 3.3 to compute the *Average All Pairs* network reliability has a worst case complexity of $O(n^3)$.

Proof: The running time of the algorithm is clearly dominated by the second step, consisting of three nested for loops, and takes $O(n^3)$ time. ■

3.3.2 A Numerical Example:

To end off the discussion of reliability, we show an example of how the *Average All Pairs* reliability is computed. Consider the five vertex network shown in figure 3.3*a*. Reliability is computed by determining the most reliable path between each pair of vertices as is shown in figure 3.3*b*. Notice that because the reliability of a path decreases rapidly as its length increases, paths of length one (single edges) are favored over longer paths. This assertion is verified by the example, as half of the paths between vertices consist of single edges. On the other hand, when local edges are unreliable, such as between vertex *a* and *b*, the most reliable path may be significantly longer. Finally, the network reliability is calculated by averaging the reliabilities of the individual paths.



a) original network

vertex pair	most reliable path	path reliability
a, b	a — e — c — b	0.87
a, c	a — e — c	0.89
a, d	a — e — c — d	0.80
a, e	a — e	0.95
b, c	b — c	0.97
b, d	b — c — d	0.87
b, e	b — c — e	0.91
c, d	c — d	0.90
c, e	c — e	0.94
d, e	d — e	0.85
	network reliability = 8.95/10	0.895

b) reliability calculations

Figure 3.3: A Numerical Example: Computing *Average All Pairs* Reliability.

Chapter 4

Distributed Reliable Subnetworks

4.1 Introduction and Motivation:

In this thesis we are primarily concerned with ensuring reliable communications. Within a network, processors communicate and share information by sending messages along incident edges. When the edges are reliable this is a simple exercise since messages always are correctly delivered to the intended receiver in finite time. When edges fail randomly, the task of sending a message becomes more complex. The sender must not only contend with a dynamically changing network topology (due to the failure and subsequent recovery of edges) but also with lost messages and the possibility of network disconnection. For a distributed algorithm, any one of these problems may result in an incorrect solution, or worse, the failure to terminate. As a result, any distributed algorithm which executes within an unreliable network must be equipped to contend with unreliable communication.

A simple, but extreme, technique to ensure the successful transmission of a message in an unreliable network is *flooding*. With flooding, a message is sent from the source processor to the receiver through every one of its neighbours. Each neighbour, which is not the intended receiver, then broadcasts the message to all neighbours. Barring a network failure which disconnects the sender and the receiver, flooding guarantees that the message will reach the intended receiver in minimum time (since the message is sent over all possible paths). Flooding is simple to implement and requires only local information, however it is clear that the method has serious drawbacks. The amount of message traffic required to send a single message is directly proportional to the degree of each processor of the network. Since numerous messages are generally sent during the execution of a distributed algorithm, an increase in traffic load may result in unacceptable queuing delays at some or all of the processors in the network. Consequently, flooding is only practical in networks in which message traffic is very sparse or when it is imperative that the message reach its destination in the minimum amount of time. We assume neither of these criteria hold.

A more practical method of handling unreliable communication in the network is to make the distributed algorithm *fault-tolerant*. A fault-tolerant distributed algorithm is a distributed algorithm which is able to achieve correct results in the presence of specified communication failures. This is usually accomplished by sending redundant messages to compensate for messages which are lost or delayed due to failed edges. Although fault-tolerant algorithms successfully handle the problem of unreliable communication, most have one or more of the following shortcomings:

- Fault-tolerant algorithms are not robust in the presence of all possible failures. Fault-tolerant algorithms usually restrict the type and/or the number of communication failures which can be successfully handled during the execution of the distributed computation. As an example, some fault-tolerant algorithms assume that processors never fail [AE84], [AG88], or that processors never recover when failed [CCK89], or assume a fixed number of edge failures [IR84]. When these conditions are not met the fault-tolerant algorithm does not guarantee the correctness nor the termination of the computation.

- Fault-tolerant algorithms tend to be much more complex than non-resilient versions and therefore that much harder to develop. For example, in certain algorithms, processors make use of *token* messages which they send to other processors. In unreliable networks a token may be lost due to a failed edge and this may lead to a deadlock situation. A general method of compensating for lost tokens is to send multiple tokens in parallel to ensure that at least one token remains active at any given time [KWZ86]. Whereas a process had only to contend with a single token in the non-resilient algorithm, a process in the multi-token algorithm must receive, recognize and eliminate redundant tokens.

• Because of the added redundancy of messages and extra error checking involved in sending a message, fault-tolerant algorithms tend to have increased message and total execution time complexity as compared to the non-resilient versions. Because edges may fail and recover infinitely many times, algorithms which repeatedly retransmit lost messages over such edges have unbounded message complexity in the worst case [V83].

While fault-tolerant distributed algorithms offer a substantial improvement over flooding techniques, the complexity of the computation and the restriction of network failures limit the practicality of such algorithms.

4.1.1 Adding Reliability Knowledge:

Communication failures complicate and may impede distributed computations. In an unreliable network, the topology changes dynamically due to failed or recovered edges. Processors become aware of neighbourhood topology changes, in finite time, only after edge failures or recoveries occur. As a result, distributed algorithms must incorporate some fault-tolerant mechanisms to manage lost messages. If, however, a processor knew the location and duration of a failure of an incident edge before the failure occurred, the processor would avoid using the edge while it was failed. Clearly, communication in an unreliable network could be completely reliable if the period of failure of each edge were known. Unfortunately, edges fail randomly.

As an alternative to complete failure knowledge, a processor may know the probability of failure, or *reliability*, of an incident edge. Given an edge which is very likely to fail, a processor will avoid using the edge if another, more reliable, path exists. Unfortunately, reliability information alone does not provide enough information to guarantee successful message transmission over an edge, unless the edge is perfectly reliable. Nevertheless, edge reliability gives us a useful indication of how likely the failure of any particular edge is. We use this to our advantage. By sending a message over the most reliable path between processors in the network, we avoid sending a message blindly over a path, which is perhaps inclined to fail.

In this thesis we adopt the distributed model with reliability knowledge, as discussed in section 2.3. Reliability knowledge is particularly advantageous in networks where large variations in edge reliability exist. For example, consider the network of figure 4.1, in which processor *a* sends a message to processor *b*. In the absence of reliability information, the inclination is to send the message over the most direct (*i.e.*, minimum length) path. When reliability information is known however, this turns out to be a particularly poor choice as the path *ab* is extremely unreliable. A much more reliable path, albeit longer, is the path $ac_1c_2c_3c_4b$ which has a reliability of approximately 0.95.

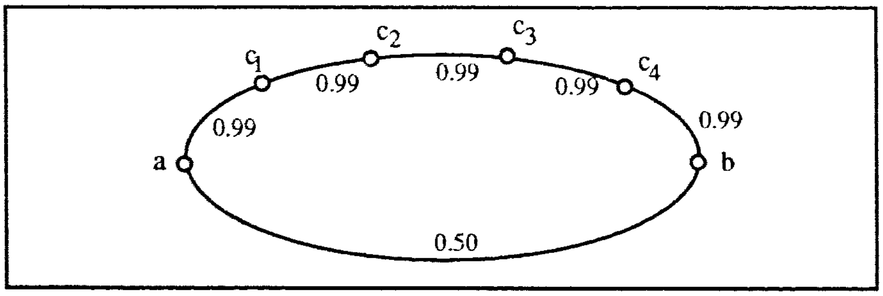


Figure 4.1: Example Network with Variation in Edge Reliability

In this thesis, we adopt the distributed computing model, as outlined in section 2.3, in which completely reliable processors are connected by unreliable edges. The reliability of each individual edge in the network is known by both adjacent processors.

4.1.2 Reliable Distributed Communications:

In this thesis we are primarily concerned with the successful transmission of messages between pairs of processors throughout the network. However, because the network is unreliable and complete failure information is not known, we must incorporate some type of fault-tolerant mechanism to ensure the correctness of a distributed computation.

As has been previously suggested, fault-tolerant mechanisms are restrictive and computationally burdensome. For this reason, we avoid incorporating them into distributed algorithms. Instead, we look for other means of providing reliable message transmission over an unreliable network. Ideally, we desire a physical network which is robust and never fails or impedes message transmission. Unfortunately, we can not change the physical characteristics of the network (of which edge reliability is one). However, we can abstract the network so it gives the illusion of reliability. For this purpose, we partition the network services into three logical layers of functionality, as shown in figure 4.2.

The first layer is the *physical network*. The physical network is unreliable and is characterized by failing edges and unreliable transmission of messages between processors. We make the assumption that the physical network conforms to the new distributed model outlined in chapter 2, and that edge reliability information is known.

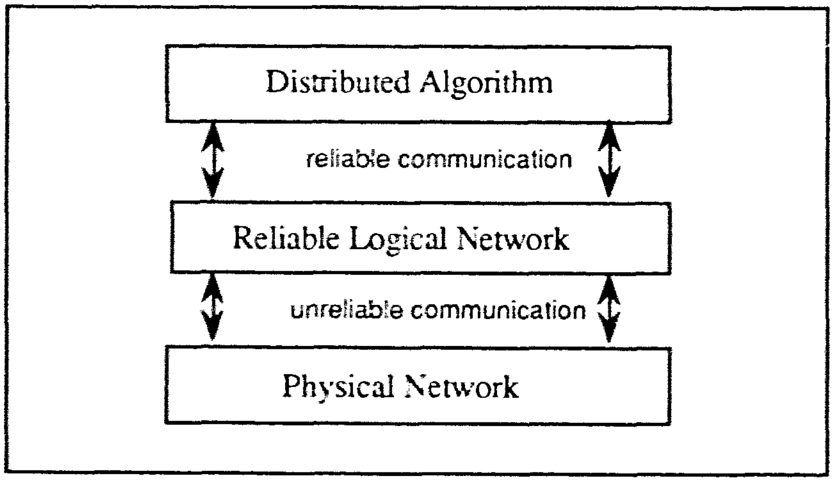


Figure 4.2: Layers of Reliable Distributed Communication

The second layer is the *reliable logical network*. This layer is characterized by the completely reliable transmission of messages. Messages sent over this layer are guaranteed to arrive in finite, but unbounded, time at the intended recipient. The reliable logical layer conforms to the standard distributed computing model.

The final layer is the distributed algorithm itself. Despite the unreliability of the physical network, message transmission takes place over the reliable logical layer which provides reliable transmission services.

The layered reliable communication approach offers two significant advantages over fault-tolerant methods. The chief advantage is the separation of the responsibility for reliable communication away from the distributed algorithm to an independent layer. Not only does this make distributed algorithms easier to develop, as fault-tolerant mechanisms are not incorporated, it makes the algorithms portable to other networks with different failure characteristics. Second, the physical topology of the network can be hidden from the distributed algorithm. Often it is easier to develop a distributed algorithm if the topology is of a symmetrical configuration, such as a ring, a binary tree or a complete network. The reliable logical layer can provide virtual edges between processors, which appear as a single edge to the distributed algorithm, but which are a path constructed from multiple edges. By providing virtual edges, the reliable logical layer can provide any topology configuration given any physical network. This has the added benefit of making distributed algorithms topology independent and therefore more portable.

The layered approach of reliable communication does not resolve the issue of lost messages due to edge failures, but it does remove the responsibility away from the distributed algorithm designer. The designer of the reliable logical layer must still contend with reliability issues. The natural question is how is the reliable logical layer constructed?

In most networks the reliable logical network layer exists as a specific routing process or in the form of routing tables, which are continuously updated as the topology and network conditions change. A *routing* defines a path between every pair of nodes in the network. When a processor wants to send a message, it either passes the message to the routing process or transmits the message over the appropriate edge given in the routing table. When an edge fails and a message is lost, the routing is reconstructed and the message is retransmitted. This strategy is an *adaptive* approach since message losses are permitted to happen and subsequently corrected. In contrast, fault-tolerant methods use a *preventative* approach in which message losses are handled by redundancy.

When reliability information is not available, a *minimum length routing* is generally computed. With this method of routing, the path between two processors is the one with the minimum length.[1] The disadvantage with this scheme is that the shortest length path between two processors may also be the most unreliable. As a result, the routing may frequently be reconstructed as edges in the routing fail. When reliability information is known, a *maximum reliable routing* may be computed. With this method of routing, the path between two processors is the one which is most reliable. The maximum reliable routing has the advantage that network failures are less likely to occur, since the most reliable edges are used. However, the most reliable path between processors may also be the path with the maximum length. In distributed computations, path length is an extremely important criterion, as processing time is negligible as compared to transmission time.

The shortcomings of the previous routing strategies are straightforward. The minimum length routing does not take into consideration the reliability of edges, while the maximum reliable routing does not take into account the length of edges and paths. We require a routing strategy which maximizes reliability while minimizing path length. Unfortunately, because minimum path length and maximum path reliability are conflicting constraints, such an optimal routing may not exist. As a result, a compromise between the minimum length and maximum reliability of a path must be used. A related problem is the *shortest weight-constrained path* problem. For this problem, a nonnegative length and weight is associated with each edge. The problem is to find a simple path between two specified processors such that the length of the path is no longer than L and the weight of the path is no greater than W, where both L and W

[1] In the standard distributed computing model all edges are of length 1. However, the model may be easily extended to include variance in edge length.

are nonnegative values. If all edge lengths or all edge weights are equal, the problem is solvable in polynomial time. In the general case, the problem has been shown to be NP-complete [GJ79]. The reliability of an edge can be transformed to a nonnegative weight, as shown in theorem 3.1, and therefore there exists a one-to-one correspondence between the shortest weight-constrained path problem and the problem of computing a path which minimizes path length and maximizes reliability.

We propose to solve the two constraint routing problem in the following way. First, we reduce the network by eliminating unreliable edges which are redundant. If the edges are chosen carefully, the reduction can be accomplished with no degradation of network reliability. Specifically, an edge can be eliminated if a more reliable path, connecting the two adjacent processors, can be found. The resulting structure will have fewer edges, but more importantly, the remaining edges will be the most reliable edges available in the network. Once the network reduction is completed, we perform a minimum length routing on the reduced network. Because we have eliminated the most unreliable edges, the resulting routing can not include the most unreliable path between two processors, unless no other path between them exists. The resulting routing is a compromise between minimizing path length and maximizing path reliability, although it favors the latter.

4.2 Reliable Subnetworks:

We introduce a network reduction technique based on the *Average All Pairs* measure of reliability, given in chapter 3. Informally, we construct a *reliable subnetwork (RSN)* by including all processors of the original network, but only those edges which are included in at least one most reliable path between any pair of processors. More formally, a reliable subnetwork is defined as follows:

Definition: *Reliable Subnetwork.* G' = (V,E') is a reliable subnetwork of G = (V,E) if $e_{i,j} \in E'$ iff $e_{i,j} \in E$ and $e_{i,j} \in MP_{x,y}$ for some $x,y \in V$, where $MP_{x,y}$ is the most reliable path from some vertex x to vertex y (see section 3.3).

We make the following observations concerning the *RSN* of a network:

- If the network is connected, the *RSN* is connected.
- The *RSN* includes the most reliable path between each pair of processors within the network.
- The reliability, as defined in chapter 3, of the *RSN* is identical to that of the original network.[2]
- The *RSN* generally contains only a subset of the edges of the original graph.

By computing a *RSN*, we are constructing a reliable backbone or core of the original network. Edges not included in the network are logically discarded and not considered unless another *RSN* is constructed.

Once constructed, the *RSN* can generally be used in place of the original network. An exception to this is a distributed computation which is dependent on a very specific topology, such as a complete network or ring. Since the *RSN* generally does not preserve the topology characteristics of the original network, the distributed computation must be modified. Other topology dependent distributed computations such as median and center finding algorithms will execute correctly, but may produce different results on the *RSN* and the original network. Non-topology dependent distributed computations may be executed on the *RSN* without modification.

The *RSN* offers the following advantages over the original network:

1. The *RSN* is a subnetwork which contains all the vertices but only a subset of the edges of the original network. In a distributed environment, where the performance of an algorithm is measured in terms of message complexity, the reduction of edges may realize a significant computational savings.

[2] This observation stems from the similarity between the definition of network reliability (section 3.3) and the definition of a reliable subnetwork.

2. The *RSN* is less prone to individual edge failure than the original network. This observation stems from the fact that the *RSN* generally has fewer edges than the original network. Let the probability that *no* edge fails within the original network and the *RSN* be R_T and R'_T respectively. Then

$$R_T = \prod r_{i,j} \text{ if } e_{i,j} \in E \quad \textit{and} \quad R_{T'} = \prod r_{i,j} \text{ if } e_{i,j} \in E'$$

   since $E' \subseteq E$, then $R'_T \geq R_T$. In addition to having fewer edges than the original network, the *RSN* also has less unreliable edges, since an edge is eliminated if a more reliable path joining the two processors can be found. By virtue of these two facts, the *RSN* will likely experience fewer edge failures over a given period of time. This fact may be significant, especially if a distributed computation uses edges indiscriminately. Notice that this observation does not imply that the *RSN* is more reliable than the original network. If a distributed computation only uses the most reliable path between processors, then the number of *significant* edge failures in the *RSN* and the original network will be identical.

3. If the network is used to construct a distributed spanning tree or other topological structure, then the structure constructed on the *RSN* may be more reliable than the one constructed on the original network. As an example, consider a distributed minimum spanning tree computation. The spanning tree constructed on the *RSN* will likely be more reliable, because the most unreliable edges have been eliminated, than the tree constructed on the original network. The tradeoff is that the spanning tree, constructed on the *RSN*, may also have a longer total path length, since the shortest edges may also have been the most unreliable.

In the remainder of this thesis we present distributed algorithms to construct an exact *RSN* and an approximate *RSN* and examine them in terms of connectivity and reliability.

4.3 Constructing A Distributed Reliable Network:

The problem of distributively computing the reliable subnetwork is determining, at each processor throughout the network, the incident edges which are contained in the structure. No processor knows the complete structure, but rather the local neighbourhood.

As previously discussed, the reliable subnetwork is constructed exactly by computing the most reliable path between all pairs of processors. The most reliable path problem is similar in principle to the classical shortest path problem. In fact, the reliable path problem is slightly easier than the shortest path problem in that one never has to deal with negative edge weights. In the distributed environment, shortest path algorithms have received considerable attention as they form the basis for many routing algorithms.

A basic distributed algorithm to compute the *RSN* of a distributed network is shown in Algorithm 4.1. The algorithm is an adaptation of the distributed *All-Pairs Shortest Path* algorithm by Chen [C82], which is a distributed implementation of the sequential Ford-Bellman-Moore algorithm. Contrary to our assumptions, this algorithm assumes that n, the number of processors in the network, is known. In the algorithm, two tables are created and maintained locally by each processor. The *RP* table is the *reliable path* table. This table holds the reliability of the current most reliable path between the processor and all other processors known to it. The table *P* is the *path* table. This table holds the incident edge of the current most reliable path between the processor and all other processors known to it. The individual entries in the *RP* and *P* tables of processor i, which specify the most reliable path between nodes i and j, are denoted by RP_{ij} and P_{ij} respectively. The set of neighbours adjacent to a processor i is denoted as $N(i)$.

```
Distributed algorithm to compute a reliable subnetwork.
Executed at each processor i.

/* wake up neighbours and initialize the tables */
INITIAL:
        IF (initiator) THEN
            FOREACH $j \in N(i)$                    /* send wakeup msg to all neighbours */
             $j \leftarrow$send(WAKEUP)
        ELSE
            receive (WAKEUP,,j)                   /* receive a wakeup msg if not initiator */
            FOREACH $k \in N(i), k \neq j$                /* relay msg to all neighbours */
                $k \leftarrow$ send (WAKEUP)
        FOREACH $j \in N(i)$ DO   /* initialize the tables */
            $RP_{i,j} = r_{i,j}$
            $P_{i,j} = e_{i,j}$
        become ACTIVE;

/* exchange RP table with neighbours n-1 times */
ACTIVE:
      FOR n-1 iterations DO
          FOREACH $j \in N(i)$ DO
              $j \leftarrow$send(RP)                  /* sending routing tables to all neighbours */
        #msg = 0;
        WHILE (#msg < |N(i)|) DO
          receive(msg,k)
          IF (msg $\neq$ WAKEUP)                        /* ignore old wake up message */
              #msg = #msg + 1
              FOREACH $RP^k_{i,j} \in RP^k$
              IF ($RP_{i,j} \notin$ RP) THEN                       /*not previously in the table*/
                  $RP_{i,j} = RP^k_{i,j}$
                  $P_{i,j} = e_{i,k}$
              ELSE                                               /* already exists in the table */
                  IF ($RP^k_{k,j} * RP_{i,k} > RP_{i,j}$) THEN        /* more reliable path found */
                      $RP_{i,j} = RP^k_{k,j} * RP_{i,k}$;
                      $P_{i,j} = e_{i,k}$;
      ENDFOR
```

Algorithm 4.1: Distributed Algorithm to Compute a Reliable Subnetwork (*RSN*).

The basic algorithm works in two phases. In the initial phase initiators spontaneously wake up and send WAKEUP messages to all neighbours. Upon receiving a WAKEUP message, a processor enters the *initial* state and forwards the message to all other neighbours; subsequent WAKEUP messages are ignored. After the processor awakes, it creates the initial *RP* and *P* tables from its local knowledge and becomes *active*. In the *active* state, processors exchange *RP* tables. An *RP* table received from some neighbour *k* is denoted as RP^k. Upon receiving the table RP^k, processor *i* computes the new reliability of the path from node *i* to *j* through node *k*, that is $(RP^k_{k,j} \times RP_{i,k})$. The processor compares this value with the current most reliable path in its local *RP* table, that is $RP_{i,j}$. If $(RP^k_{k,j} \times RP_{i,k}) > RP_{i,j}$ or if no previous path between i and j exists, then $RP_{i,j}$ is set to $(RP^k_{k,j} \times RP_{i,k})$ and $P_{i,j}$ is set to $e_{i,k}$. Otherwise, the table entry remains unchanged. After the *RP* tables have been received from each neighbour, a processor sends its updated *RP* table to all neighbours. The algorithm terminates after each processor has received *RP* tables from each of its neighbours n-1 times. At the completion of the algorithm, the *RSN* is composed of those edges which are in the table *P* of any processor.

The basic algorithm has a message complexity of $O(n^2d + e)$, where d, $1 < d < n$, is the maximum degree of any vertex in the network. This follows from the following facts. At most, $2e$ wakeup messages will be required to start the distributed computation. During the computation, each of the n processors send reliability information to a maximum of d neighbours n times. However, the basic algorithm is somewhat inefficient as each processor broadcasts reliability information to all neighbours. Both Toueg [T80] and Lakshmanan et al [LTC89] give distributed algorithms for the *All-Pairs Shortest Path* problem which improve the message complexity to O(ne) by sending path information over a spanning tree rather than the complete network. Frederickson [F85] suggests a simple approach which also has a message complexity of O(ne), in which each node broadcasts its local topology to all other nodes in the network. With the global topology, a processor computes the shortest paths locally.

All of the previously mentioned distributed algorithms are based on well known sequential algorithms for the *All Pairs Shortest Path* problem. In a sequential algorithm it is usual to know n, the number of processors in the network , and in fact most algorithms need this value in order to know when execution can be terminated. However, in the standard distributed model we assume that the number of processors is fixed but unknown to any processor (assumption P4). The number of processors in a network can be computed, but at expense of extra messages.

A more serious problem with all the previous algorithms is high message complexity. The best known distributed *All Pairs Shortest Path* algorithm, and thus the best known *RSN* algorithm, has a complexity of O(ne). Since $(n\text{-}1) \leq e \leq (n\text{-}1)^2$, therefore $O(n^2) \leq O(ne) \leq O(n^3)$. Even for a network of moderate size, the number of messages sent will be significant. Since message transmissions take significantly longer than local computations, the time to compute the *RSN* in a dense network may be prohibitive. However, if we examine the messages themselves, the message complexity becomes even more critical. All of the previous algorithms send complete tables of local information which are counted as a single message. In chapter 2 we made the assumption that messages are small, on the order of $O(\log_2 n)$ bits (assumption M3). Each table consists of O(n) entries which if sent individually would increase the message complexity by a factor of *n*. Toueg [T80] recognizes this fact and sends only the changes, after the initial table has been sent. However, in the worst case the number of changes will still be O(n). As a result, the actual message complexity of an O(ne) *RSN* algorithm becomes $O(n^2e)$ and $O(n^3) \leq O(n^2e) \leq O(n^4)$. Even for small networks, the number of messages required to compute the *RSN* may be prohibitive.

The problem of constructing the *RSN* is intrinsically difficult because we are trying to construct a global structure using only local information. The problem is therefore constrained by how efficiently, in terms of the number of messages, each processor can acquire all global information.

The inefficiency of computing the most reliable paths between all pairs of processors in the network, and therefore constructing the *RSN*, leads us to consider alternative solutions. In the following section we outline and examine an approximation algorithm for computing a reliable subnetwork.

4.4 Approximating Reliable Subnetworks:

Distributively constructing the exact *RSN* of a given network is a laborious task which is prohibitive for most large networks. One could argue that the expense is justified since the

RSN is only constructed once or whenever a network topology change causes the *RSN* to become disconnected. However, we must take into account the heavy use of network resources while the construction is taking place, leaving the network unusable to perform other functions. If failures happen frequently the problem is only compounded.

The question which must be asked is whether the exact solution is required or whether an approximate solution will suffice. First, recall that the reliability information associated with each edge in the network is not an exact value, but rather an approximation of the reliability. Consequently, even the exact solution will likely be less than perfect. Second, the motivating factor for constructing the *RSN* is to provide a structure in which the most unreliable edges are eliminated, making the *RSN* less prone to individual edge failure. If an approximate solution meets this criterion and has reduced message complexity, the solution will be expedient.

Given that the exact *RSN* is impractical to construct, we concentrate on finding a distributed algorithm to construct an *Approximate Reliable Subnetwork (ARSN)*. In preparation, we first examine the nature of the distributed environment. Within the network, topology information (*i.e.*, processor identity, edge reliability, etc.) is distributed among the individual processors with no one processor initially having complete knowledge. Because knowledge is localized, the most message efficient distributed algorithms are those which can combine a number of locally computed solutions, either by a single processor or a small group of neighbouring processors, into a global solution. Notice that this is not possible with the exact *RSN* problem, since the locally most reliable path to a neighbour may not be the globally most reliable path. A good approximation algorithm to construct the *RSN* will be one which localizes the computation as much as possible.

The general idea of the algorithm is that each processor constructs a local solution by selecting the *K* most reliable incident edges, where *K* is a "small" constant number such as 2 or 3. When an incident edge is selected, the selecting processor informs the adjacent processor. The *ARSN* is then constructed by combining the local solutions of each processor. Following the construction, each processor knows which of its incident edges are in the *ARSN* but no single processor knows the entire structure. The exception to this is when all edges in the *ARSN* are incident to a single processor (*i.e.*, such as a star configuration), although in this case the processor is unaware that it knows the global structure. When two or more edges incident to a processor have the same reliability value, the tie is broken by concatenating the IDs of both incident processors, larger first, to the reliability of the edge. By changing the tie resolution scheme a different *ARSN* may be constructed. Given the same tie resolution scheme and value of *K*, the *ARSN* is unique for any given combination of topology, processor labeling and edge reliabilities. Changing any of the previous aspects of a network, may result in a different *ARSN*. As with the *RSN*, the *ARSN* can be used in place of the original network, with the exception of topology dependent computations.

A distributed *ARSN* algorithm is given in Algorithm 4.2. The details of the algorithm are as follows: initiators wake up spontaneously in the INITIAL state and send WAKEUP messages to all neighbours. Upon receiving a WAKEUP message, non-initiators wake up and forward the message to all other neighbours. Subsequent WAKEUP messages are ignored. After a processor has initiated or forwarded WAKEUP messages to all neighbours, it enters the ACTIVE state. In the ACTIVE state, each processor *i* computes a *reliable edge* set, RE_i, containing the *K* most reliable incident edges. For each edge in RE_i, an INCLUDE message is sent to the adjacent processor to indicate that the edge is in the *ARSN*. The remainder of the neighbouring processors are sent a REJECT message. When the appropriate messages have been sent to all neighbours, the processor waits to receive messages from all neighbours. When processor *i* receives an INCLUDE message such that $e_{i,j} \notin RE_i$, it adds that edge to RE_i. When a processor *i* receives an INCLUDE message or REJECT message such that $e_{i,j} \in RE_i$, the message is discarded. The algorithm terminates when messages have been received from all neighbours. The *ARSN* consists of the union of all edges from each local RE_i. The solution is distributed amongst all the processors in the network, with no one processor knowing the complete structure.

Approximate distributed algorithm to compute a reliable subnetwork. Executed at each processor *i* for some value of *K*, $0 < K < n$. N_i is the set of processors adjacent to processor *i*.

```
/* wake up neighbours */
 INITIAL:  IF initiator THEN
                $j \leftarrow$ send(WAKEUP), for each $j \in N_i$
           ELSE
                receive (WAKEUP,i);
                $j \leftarrow$ send (WAKEUP) for each $j \in N_i, j \neq i$
           become ACTIVE


/* select K most reliable edges and broadcast to neighbours*/
 ACTIVE:   RE(i) = $\{e_{i,j} \mid$ if $j \in N_i$ AND $r_{i,j}$ is one of the K most reliable edges$\}$
           FOREACH $j \in N_i$ DO
                IF ($e_{i,j} \in$ RE(i)) then
                     $j \leftarrow$ send(INCLUDE)
                ELSE
                     $j \leftarrow$ send(REJECT)
                ENDIF


           /* receive responses from neighbouring processors */
           #msg = 0
           WHILE #msg < $|N_i|$ DO
                receive(msg,j)
                IF (msg $\neq$ WAKEUP)  /* ignore old wakeup message */
                     #msg = #msg + 1
                     IF (msg = INCLUDE AND $e_{i,j} \notin$ RE(i)) THEN
                          RE(i) = RE(i) $\cup$ $\{e_{i,j}\}$
                     ENDIF
                ENDIF
```

Algorithm 4.2: Distributed Algorithm for the Approximate Reliable Subnetwork Problem.

Algorithm 4.2 constructs an approximation of the reliable subnetwork with a total of $4e$ messages ($2e$ messages to wake up and $2e$ messages to inform neighbours whether or not the edge is in the reliable subnetwork) for a message complexity of O(e), where $O(n) \leq O(e) \leq O(n^2)$. Since messages do not convey information other than the message type, message lengths are short (*i.e.*, a minimum of 2 bits to distinguish between message types). As can be seen, the *ARSN* algorithm is a significant improvement, in terms of message complexity and message length, over the exact *RSN* algorithm.

In the evaluation of the *ARSN* we must consider two issues. First, the reliability of the *RSN* is identical to that of the original network since both are based on maximum reliable paths between all pairs of processors. The *ARSN*, on the other hand, does not guarantee that the reliability is identical to the original network, nor does it guarantee that the reliability is even "good". If the reliability of the *ARSN* is significantly less than the *RSN*, the ease of computation is of little consequence. As we increase the value of *K* we get two conflicting results: the reliability of the *ARSN* increases (or at least does not decrease) and the *size* of the subnetwork increases. Likewise, as we decrease the value of *K*, the reliability and size of the *ARSN* decrease. The goal of the approximation scheme is to minimize the number of edges and maximize the reliability of the subnetwork. The question then arises as to how large *K* must be in order to construct a subnetwork with an acceptable reliability. In chapter 5 we examine this issue and evaluate the reliability of the *ARSN* for various values of *K* experimentally. We leave the discussion of reliability until then.

Secondly, the *RSN* is constructed by taking the union of the most reliable paths between all pairs of processors in the network and therefore is guaranteed to be connected, assuming that the original network is connected. The *ARSN*, on the other hand, is constructed by amalgamating local solutions and therefore the possibility exists that the structure will not be connected. As the value of *K* increases we expect that occurrences of disconnections will decrease but not be eliminated. In the following sections we examine the question of connectedness and evaluate the *ARSN* both analytically and experimentally.

4.4.1 Connectedness of the Approximate Reliable Subnetwork:

Connectedness of a network is a globally defined property. The *RSN* guarantees connectedness because it is constructed with a global algorithm. The *ARSN*, on the other hand, is constructed by piecing together local solutions from each processor. Since no locally based algorithm can ensure connectedness, instances of an *ARSN* may be disconnected. As an example of a disconnected instance of an *ARSN*, consider the six node network shown in figure 4.3. When $K = 1$ each individual processor selects the most reliable of the local edges to construct the subnetwork. Unfortunately, the resulting structure has two disjoint partitions. In this section we investigate how often the *ARSN* algorithm of Algorithm 4.2 produces a disconnected structure.

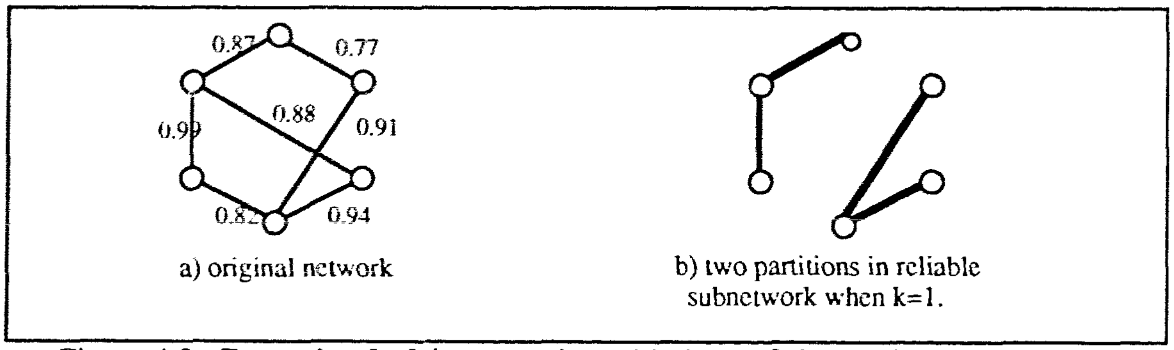


Figure 4.3 : Example of a Disconnection with the *ARSN* Algorithm when K=1.

The basic tool we employ to theoretically analyze the connectivity of the *ARSN* algorithm is *graph enumeration*. We model the network as a graph, ignoring the fact that the algorithm will be a distributed algorithm running on a network. We count two specific classes of graphs constructed by the *ARSN* algorithm given an exact number of nodes and edges: *all* possible graphs constructed by the *ARSN* algorithm and all possible graphs which are connected. Let T_a represent the number of all possible graphs and T_p represent the total number of graphs with the property we are looking for. Because $T_a > T_P$, the value $\frac{T_P}{T_a}$ represents the probability that the algorithm produces a graph with the desired property.

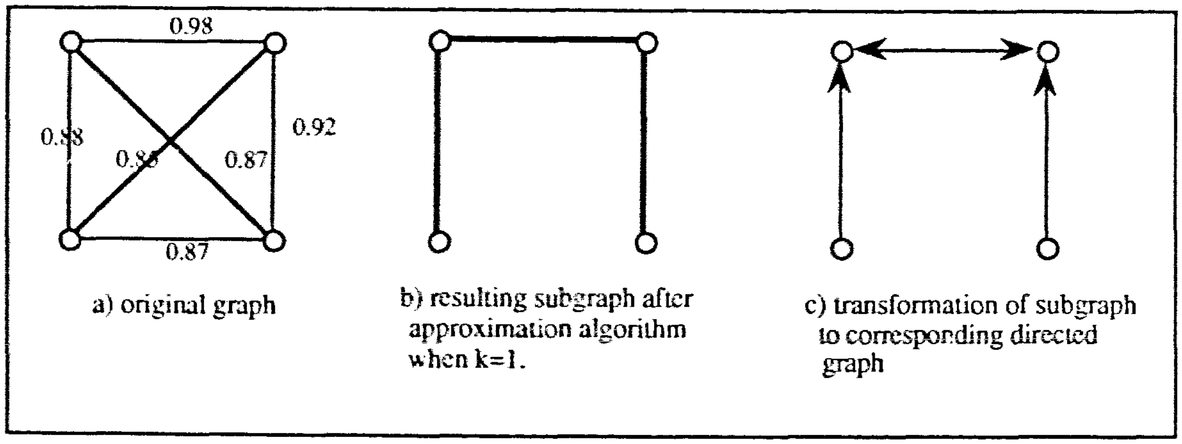

0.98
0.88
0.92
0.88
0.87
0.87
a) original graph
b) resulting subgraph after approximation algorithm when k=1.
c) transformation of subgraph to corresponding directed graph

Figure 4.4: Transformation of an Undirected Graph to a Directed Graph

Enumerating connected general graphs is, unfortunately, a mathematically complex problem. For this reason, we make the simplifying restriction that the original graphs (networks) are complete.

To analyze the connectedness of the resulting structure constructed by the *ARSN* algorithm we transform the structure from an undirected graph to a directed graph in the following way. In the *ARSN* algorithm each vertex (processor) selects the *K* most reliable edges. The edges chosen by each vertex in the graph are represented by a directed *out* edge from that vertex. An edge chosen by both adjacent vertices is represented by two directed edges going in the opposite direction. An example of the transformation is shown in figure 4.4. The total number of possible graphs which can be constructed from any complete graph of *n* vertices is then equivalent to finding the number of labeled[3] directed graphs, $DG_{n,K}$, with *n* vertices such that each vertex has out degree *K*. The formula for this value is derived in theorem 4.1.

Theorem 4.1: The number, $DG_{n,K}$, of labeled directed graphs with *n* vertices such that each vertex has out degree *K* is:

$$DG_{n,K} = \begin{cases} \binom{n-1}{K}^{n} & : \text{if } n > K \\ 0 & : \text{otherwise} \end{cases}$$

Proof. This result follows from the fact that each of the *n* vertices selects an edge from itself to *K* of its *n*-1 neighbours. This selection can be done in $\binom{n-1}{K}$ ways. If $n \leq K$, then no graphs are possible since at least *K*+1 vertexes are required to accommodate exactly *K* outgoing edges from each vertex. ■

The number of connected graphs constructed by the *ARSN* algorithm is then equivalent to the number of weakly connected directed graphs, $DC_{n,K}$, of *n* vertices such that each vertex has out degree *K*. The formula for the value $DC_{n,K}$ is derived in theorem 4.2.

[3] In a labeled graph a vertex is distinguished from all others by a unique label (*i.e.*, processor ID). In an unlabeled graph, one vertex is indistinguishable from another.

Theorem 4.2: The number, $DC_{n,K}$, of labeled weakly connected directed graphs with n vertices in which each vertex has out degree K is:

$$DC_{n,K} = DG_{n,K} - \frac{1}{n}\sum_{j=K+1}^{n-K-1} j \times \binom{n}{j} \times DG_{n-j,K} \times DC_{j,K}$$

Proof. To show that the above recurrence relation holds, consider the number of rooted labeled directed graphs of size n and out degree K. A rooted graph is a graph in which one vertex is distinguished as the root vertex. Clearly there are $nDG_{n,K}$ such graphs since each labeled directed graph can be rooted at any of its vertices. The root vertex must be a member of some component which has j vertices, where $K+1 \leq j \leq n-K-1$ if disconnected or $j=n$ if connected. When the root vertex is a constituent of a component of j vertices there are j ways that the component can be rooted, $\binom{n}{j}$ of choosing the vertices of the group and $DC_{j,K}$ such components. The remaining n-j vertices of the graph form all remaining variations of the directed labeled graph in exactly $DG_{n-j,K}$ ways. Multiplying these factors together and summing over j we obtain:

$$nDG_{n,K} = \left[\sum_{j=K+1}^{n-K-1} j \times \binom{n}{j} \times DG_{n-j,K} \times DC_{j,K}\right] + nDC_{n,K}$$

Rearranging to isolate $DC_{n,K}$ we have

$$DC_{n,K} = DG_{n,K} - \frac{1}{n}\sum_{j=K+1}^{n-K-1} j \times \binom{n}{j} \times DG_{n-j,K} \times DC_{j,K}$$

■

The results of computing[4] the probability $\frac{DC_{n,K}}{DG_{n,K}}$, that a directed graph constructed by the *ARSN* algorithm applied to a complete graph, is connected, is shown in table 4.1 for various values of n and K. When each processor only selects one edge (K=1) the probability that the resulting structure will be disconnected is fairly large and increases as the number of vertices increases. On the other hand, when $K \geq 2$ the probability of a disconnected structure is very small and decreases as the number of vertices increases. When K=3, the probability that

[4] All analytical results in this thesis are computed using the MAPLE mathematics package.

structure is not connected is virtually 0 for even graphs of small order. From these observations, it would seem that the *ARSN* algorithm, when $K>2$, almost always produces a connected structure.

While table 4.1 shows encouraging connectedness results for $K>1$, they are unfortunately not completely representative of the *ARSN* algorithm. The problem is demonstrated by the four vertex graph in figure 4.5 when $K=1$. By the formula of theorem 4.1, all three configurations, and their permutations, are counted in the total number of possible graphs constructed from K_4. However, only the two configurations in part *b* are possible by the *ARSN* algorithm. The problem is that each vertex does not randomly select which *K* edges it will contribute to the *ARSN*, but rather chooses the *K* most reliable incident edges. We assume that a total global ordering of the edges exist, such that edge *i* comes before edge *j* if the reliability of *i* is greater than the reliability of *j*.[5] If such an ordering exists, then there must be an edge in the network

k / n	k = 1 %	k = 2 * %	k = 3 * %
10	69.4	99.90	1.0
20	59.8	99.987	1.0
30	51.1	1.0	1.0
40	45.5	1.0	1.0
50	41.4	1.0	1.0
60	38.3	1.0	1.0
70	35.8	1.0	1.0
80	33.8	1.0	1.0
90	32.1	1.0	1.0
100	30.6	1.0	1.0

* Accurate to at least 3 decimal places.

Table 4.1: Probability of Connection - Labeled Digraphs of Order *n* and Out Degree *K*.

[5] Notice that this assumption does not imply that all edge reliabilities must be unique. When two edges have the same reliability, they can be made unique by appending the IDs of both adjacent processors to the reliability.

which is the global maximum (*i.e.*, most reliable). It is therefore impossible for the two processors adjacent to the global maximum edge to select different edges. As a result, the configuration in part *a* is not legal and the results of table 4.1 are incorrect. Nevertheless, we would expect the connectivity of the *ARSN* to follow a similar pattern as the weakly connected directed graph. This is verified, when K=1, in theorem 4.3.

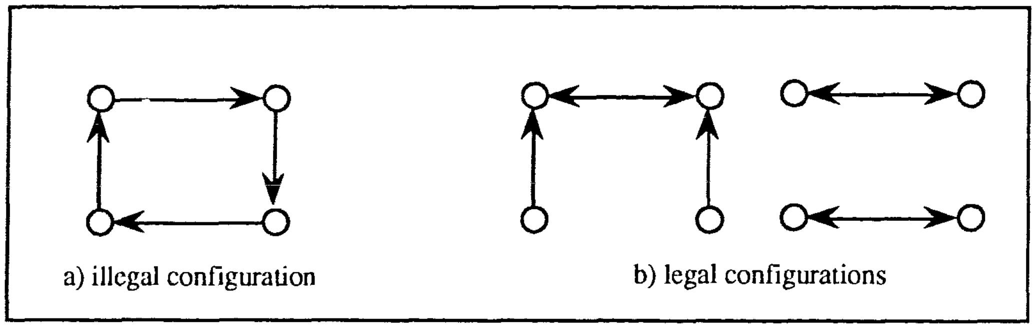


Figure 4.5: Legal and Illegal Configurations for n=4 and K=1.

By considering the ordering of edges, with respect to the edge reliability, an exact formula for the expected connectedness of structures constructed by the *ARSN* algorithm for K=1 is given in theorem 4.3.

Theorem 4.3: The number, $C1_{n,e}$, of edge and vertex labeled graphs with n vertices, n>2, and e edges, e > n-1, for which the *ARSN* algorithm (when K=1) yields a connected graph, is given by the following formula:

$$C1_{n,e} = 6 \times \binom{n}{3} \times SG_{n,3,e},$$

where $SG_{n,i,e,j}$, the number of edge labeled subgraphs constructed by the *ARSN* algorithm with exactly e edges and n vertices, starting with a subgraph containing i nodes and j edges, is given as follows:

$$SG_{n,i,e,j} = \begin{cases} 1, & \text{if } e=j \text{ and } n=i \\ 0, & \text{if } e<j \text{ and } n=i \textit{ or} \\ & \text{if } e=j \text{ and } n \neq i \\ i \times (n-1) \times SG_{n,i+1,e,j+1} + \left[\binom{i}{2} - j\right] \times SG_{n,i,e,j+1}, & \text{otherwise} \end{cases}$$

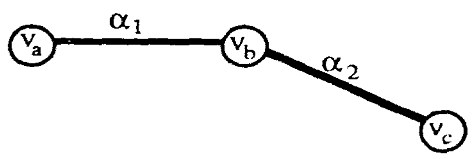
Proof. For $\alpha_i, \alpha_j \in E$ let $\alpha_i > \alpha_j$ iff α_i is more reliable than α_j. There exists a total ordering of the edges $\alpha_1 > \alpha_2 > ... > \alpha_e$. We first show the following lemma.

Lemma 4.1: Each edge $\alpha_i \in E$, $2 \leq i \leq e$, must be incident to a vertex $v_a \in V$ such that v_a is incident to another edge $\alpha_j \in E$ such that $i \neq j$ and $\alpha_j > \alpha_i$, for the graph constructed by the *ARSN* algorithm when $K=1$, to be connected.

Proof. (by contradiction) Assume there exists a connected reliable subgraph constructed by the *ARSN* algorithm, for $K=1$, in which the previous restrictions do not hold. Trivially, both vertices incident to α_1 will choose this edge since it is the globally most reliable edge. Assume there exists an edge $\alpha_i \in E$ that is locally maximum, which is to say that neither of its incident vertices is incident to a more reliable edge. Then by the same token, both vertices incident to α_i must choose this edge since neither is incident to a more reliable edge. Since at least two edges are each chosen by both incident vertices, the total number of edges possible in the graph is at most n-2. Since at least n-1 edges are required to connect n vertices, the resulting graph can not be connected, thus a contradiction exists. ■

The previous lemma is important as it gives us a method to construct and therefore enumerate graphs which will be connected when constructed by the *ARSN* algorithm when K=1. Essentially we construct the graphs by iteratively adding edges, in order of decreasing reliability, and vertices so the restrictions outlined in lemma 4.1 are not violated. The graph is completely constructed when all n vertices and all e edges have been added.

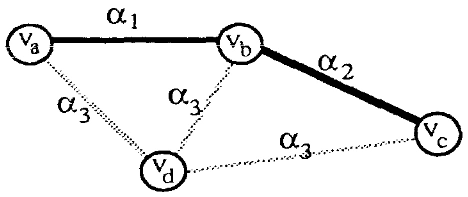
Initially, each valid graph must have the following configuration for some vertices v_a, v_b, $v_c \in V$ and edges α_1 and α_2:



Since the edges are labeled there are exactly six ways they can form unique configurations and $\binom{n}{3}$ ways of selecting the three vertices.

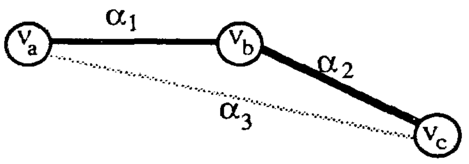
From the initially constructed subgraph the construction continues in either one of the following ways:

1) *Add vertex:* The simplest way of adding a legal edge is to connect a new vertex (one not previously in the graph) to an existing vertex. The configuration may look as follows:



In this case, the vertex v_d is added to the graph by the edge α_3 in one of three ways. In general, if there are i vertices currently in the graph then there are $i*(n-i)$ ways of adding this vertex since the new vertex can be connected to an existing vertex in i ways and the new vertex can be chosen from the remaining in $(n-i)$ ways. This edge will be part of the reliable subgraph computed by the *ARSN* algorithm.

2) *Add cycle*: The second way of adding an edge is to join two previously existing vertices in the subgraph. The configuration may look as follows:



In this case the edge α_3 is added connecting v_a and v_c in exactly one way. In general, if there are currently i vertices and j edges in the subgraph then there are $\binom{i}{2} - j$ ways of adding a cycle. This edge is not in violation of the restriction in lemma 4.1, but will not be part of the reliable subgraph. ■

Corollary 4.3a: The number, $CE1_n$, of edge and vertex labeled configurations of K_n with n vertices, $n>2$, for which the *ARSN* algorithm (with $K=1$) yields a connected graph, is given by the following formula:

$$CE1_n = \sum_{e=n-1}^{\binom{n-1}{2}+1} \left(C1_{n,e} \times \left[\binom{n}{2} - e \right]! \right)$$

Proof. The number of edge and vertex labeled connected graphs of n vertices and e edges which results in a connected *ARSN*, when $K = 1$, is given by the formula $C1_{n,e}$. The total number of edge and vertex labeled complete graphs of n vertices which result in the same configurations as represented by $C1_{n,e}$ can be enumerated by adding the remaining $\binom{n}{2} - e$ edges. These may be added in $\left[\binom{n}{2} - e\right]!$ ways.

The total number of edge and vertex labeled complete graphs with n vertices which results in a connected *ARSN*, when K=1, can be enumerated by summing $c1_{n,e}\times\left[\binom{n}{2}-e\right]!$ over the range (n-1) to $\binom{n-1}{2}+1$. The former represents the smallest number of edges required to connect n vertices. The latter represents the largest number of edges which can be used to connect n vertices in the manner described in theorem 4.3. ■

4.4.2 Connectedness of the Approximate Reliable Subnetwork - Analytical and Experimental Results:

We evaluate the *ARSN* experimentally by simulating a distributed computation on a randomly generated graph (network). Complete random graphs[6] are constructed by uniformly generating a reliability value, in the range [0...1.0], for each of the $\binom{n}{2}$ edges. After each random graph is generated, a spanning tree is constructed to verify connectedness. Graphs found to be disconnected are discarded.

n	Analytical %	Experimental %
5	57.14	57.28
7	24.24	24.21
10	5.265	5.42
13	0.984	0.97
15	0.306	0.31
18	0.050	0.042
20	0.0149	0.028
23	0.00000021	0
25	4.7×10^{-8}	0
30	~0	0

Table 4.2: Percentage of Connected Graphs, Constructed by the *ARSN* Algorithm, when K=1 and G is Complete.

The results of the computations from Corollary 4.3a are shown in table 4.2 along with the corresponding experimental observations of the connectedness of reliable subnetworks constructed by the *ARSN* algorithm. Because of the exponential time required to calculate

[6] Refer to Appendix 1 for a relevant discussion and evaluation of the random number generator.

$CE1_n$, the analytical results could only practically be calculated for networks up to 30 vertices. This is sufficient, however, to show that for even small networks, the probability of constructing a connected network, when $K=1$, is very small and rapidly decreases as the number of vertices is increased. For the experimental results, 500,000 randomly generated complete graphs of *order n* were generated. As can be seen, the experimental observations are virtually identical to the analytical results and therefore confirm the previous observations. From the results of table 4.2 it is obvious that when $K=1$, the *ARSN* is extremely unlikely to be connected, for even moderate sized networks.

As previously mentioned, enumeration of graphs is a mathematically difficult problem. There is no known exact formula for determining the number of connected approximate reliable subnetworks constructed by the *ARSN* algorithm when $K>1$. For this reason, we must rely on experimental results to analyze the connectedness of the structures. The experiments were set up as follows: Networks were randomly generated for values of n = 20, 40, 60, 80 and 100. For each value of n, 100,000 graphs were generated for values of e = 20%, 40%, 60%, 80% and 100%, where 20% means that the graph contained 20% of the possible edges (*i.e.*, a complete graph). For each value of n, a total of 500,000 graphs were generated. For each group of networks, of size e and order n, the *ARSN* algorithm was applied and the number of disconnected subgraphs were counted. The results are tabulated in tables 4.3a, 4.3b and 4.3c.

n\ e	20%	40%	60%	80%	100%
20	0.11	0.02	0.01	0.0	0.0
40	0.01	0.0	0.0	0.0	0.0
60	0.0	0.0	0.0	0.0	0.0
80	0.0	0.0	0.0	0.0	0.0
100	0.0	0.0	0.0	0.0	0.0

Table 4.3a: Percentage of *ARSNs* with $K=1$, for a Given Value of n and e, Which were Connected

n\ e	20%	40%	60%	80%	100%
20	98.22	99.24	99.26	99.29	99.32
40	99.25	99.28	99.47	99.51	99.57
60	99.52	99.54	99.59	99.68	99.70
80	99.52	99.55	99.66	99.71	99.80
100	99.57	99.59	99.68	99.75	99.88

Table 4.3b: Percentage of *ARSNs* with K=2, for a Given Value of n and e, Which were Connected

n\ e	20%	40%	60%	80%	100%
20	99.90	99.92	99.92	99.94	99.93
40	99.95	99.95	99.97	99.98	99.99
60	100.0	100.0	100.0	100.0	100.0
80	100.0	100.0	100.0	100.0	100.0
100	100.0	100.0	100.0	100.0	100.0

Table 4.3c: Percentage of *ARSNs* with K=3, for a Given Value of n and e, Which were Connected

From the results of the previous tables we make the following observations. Confirming the results of table 4.2 , table 4.3a shows that when K=1, the probability that the *ARSN* is connected is extremely small. Notice that the probability of disconnection increases as the number of vertices in the network increases. Since the K=1 *ARSN* is only connected when the selected edges form a spanning tree, this is an expected result. Tables 4.3b and 4.3c show that when K>1 the probability that the *ARSN* algorithm produces a connected structure is extremely large. When K>2 the probability is so large that a disconnected reliable subnetwork was never encountered for values of $n \geq 60$. From these results it can be seen that the *ARSN* algorithm almost always constructs a connected structure when $K \geq 2$.

The connectivity results from table 4.3 show that, for $K \geq 2$, the *ARSN* is practically never disconnected, yet there still remains a slight possibility that this situation occurs. When a disconnected *ARSN* is discovered, we have two options: the *ARSN* may be reconstructed, or the disconnection may be repaired. Reconstructing the *ARSN* generally involves increasing the value of K, since the *ARSN* is unique for a given network and value of K. However, if a number of edges in the network have identical reliability, a different *ARSN* may be constructed

	components $(p_1, p_2, p_3 \ldots p_j)$	k = 2 %	k = 3 %	k = 4 %
2 components	(3,27)	78.82	- *	-
	(4,26)	16.54	90.63	-
	(5,25)	3.42	8.58	92.47
	(6,24)	0.824	0.70	7.12
	(7,23)	0.235	0.069	0.37
	(8,22)	0.079	0.0091	0.023
	(9,21)	0.031	0.0015	0.0021
	(10,20)	0.014	0.00036	0.00025
3 components	(3,3,24)	0.164×10^{-4}	-	-
	(3,4,23)	0.391×10^{-5}	-	-
	(4,4,22)	0.964×10^{-6}	0.281×10^{-12}	-
	(4,5,21)	0.237×10^{-6}	0.377×10^{-13}	-
	(5,5,20)	0.613×10^{-7}	0.552×10^{-14}	0.635×10^{-22}
	(5,6,19)	0.185×10^{-7}	0.715×10^{-15}	0.937×10^{-23}
	(6,6,18)	0.593×10^{-8}	0.104×10^{-15}	0.169×10^{-23}
4 components	(3,3,3,21)	0.140×10^{-10}	-	-
	(3,3,4,20)	0.490×10^{-11}	-	-
	(4,4,4,18)	0.812×10^{-12}	0.251×10^{-25}	-
	(4,4,5,17)	0.355×10^{-12}	0.748×10^{-26}	-
	(4,5,6,15)	0.104×10^{-12}	0.802×10^{-27}	-
	(5,5,5,15)	0.969×10^{-13}	0.745×10^{-27}	0.426×10^{-43}
	(6,6,6,12)	0.470×10^{-13}	0.113×10^{-27}	0.183×10^{-43}

* components which are $\leq K$ are not valid and are therefore shown as '-'

Table 4.4: Probability that a Disconnected *ARSN* is Partitioned Into Separate Components of the Form (p1, p2,...,pj), when n=30.

by changing the way in which ties are broken when choosing edges. Although, a different *ARSN* may be constructed, there is no guarantee that it is connected. If the number of edges in the *ARSN* is of primary concern or it is critically important that the *ARSN* be connected, then repairing the *ARSN* is the obvious choice. However, if the number of partitions is large, the repair algorithm may spend considerable time and messages "piecing" the partitions together. Given this situation, it may be too expensive to repair the disconnection. If the number of partitions is likely to be small, then it may be worth the extra expense to repair the structure.

From the formulas given in theorems 4.1 and 4.2, we can approximate what the partitions will look like when the structure is disconnected. Specifically, the number of directed labeled disconnected graphs, DX, with n vertices, such that two disjoint partitions exist of size p and n-p, and such that each partition is a weakly connected graph with each vertex having out degree K is:

$$DX_{n,p,K} = \binom{n}{p} \times DC_{p,K} \times DC_{n-p,K}$$

This formula follows as a direct result of theorem 4.2. Similar formulas are easily derived for three and four partitions. We use these formulas to determine the characteristics of the partitions by simply enumerating all possible combinations of two, three and four partitions. By adding these values, we can determine the total number of disconnected graphs, with two, three and four partitions. Then, given the number of disconnected graphs for a specific type of partition, we can determine the probability with which the specific partition will occur when the graph is disconnected. Notice that the formulas are based on theorem 4.2, which does not accurately enumerate the number of connected subgraphs, of K_n, constructed by the *ARSN* algorithm. Nevertheless, the results will give an indication as to what size and number of partitions we can expect when a disconnected subgraph is constructed.

The results of the computations are displayed in table 4.4 for K=2, 3 and 4, when n = 30. The results are only partially displayed due to the large number of possible partitions. From the results, it can be seen that when the *ARSN* algorithm constructs a disconnected subnetwork it is very likely to be disconnected into only two partitions. Furthermore, one partition will likely be very small while the other is very large. Because it has fewer processors, the smaller partition can more quickly recognize the disconnection than can a larger partition. Consequently, the smaller partition can also more easily repair the disconnection. Given the results of table 4.4, when a disconnection occurs, it may require comparatively few network resources to repair.

4.4.3 Connectedness of the Approximate Reliable Subnetwork - Summary:

As previously shown, the *ARSN*, for any value of *K*, is not guaranteed to be connected. A disconnected subnetwork is of little benefit, since communication between processors is restricted to members of the same fragment. When *K*=1, the *ARSN* will almost certainly be disconnected, which makes *K*=1 a poor choice for constructing an *ARSN*. Moreover, because the *K*=1 *ARSN* is likely to be partitioned into a number of small fragments, it is expensive to repair. If the user requires a structure with a minimum number of edges, then a maximum reliable spanning tree[7] can be constructed in $O(n\log_2 n + e)$ messages. In terms of connectedness, the previous results show that either *K*=2 or *K*=3 are good choices to construct the *ARSN*. Of these, *K*=2 is likely the better choice, since it has fewer edges than the *K*=3 structure, and is almost as likely to be connected. The *K*>3 *ARSN* offers no advantages over *K*=3, in terms of connectedness, and therefore should not be used.

In the following section we present a distributed algorithm to verify the connectedness of the *ARSN*, and when a disconnection exists, connect the partitions. However, it should be noted that the verification algorithm need not be applied in every circumstance. When *K*=2 ,and especially when *K*>2, the probability of disconnection is virtually nonexistent. In many applications this amount of risk is acceptable. Furthermore, a distributed computation, such as a routing process, can be modified to recognize the disconnection as it performs other tasks. Consequently, it may be possible to assume connectedness of the *ARSN* without catastrophic results if a disconnection occurs. In other circumstances, however, a disconnected subnetwork

[7] The maximum reliable spanning tree is analogous to the minimum weight spanning tree.

may have serious implications. For this reason and for the sake of completeness, an algorithm which constructs a connected *ARSN* is presented.

4.5 Distributed Construction of a Connected Approximate Reliable Subnetwork

In this section we present an improved distributed algorithm to construct an *ARSN* which is guaranteed to be connected regardless of the value of *K*. The connected *ARSN* algorithm consists of three phases:

1. Basic distributed algorithm (Algorithm 4.2)
2. Verification of connectedness.
3. Resolution of the disconnection (if one exists).

4.5.1 Determining the Connectedness of the Approximate Reliable Subnetwork:

We begin the discussion by describing a method for verifying the connectedness of an *ARSN* constructed on a (non-distributed) graph. After the completion of the algorithm, the vertices of the graph are partitioned into one or more groups. The graph is disconnected if at least two non-empty groups exist. Edges incident to each vertex of the graph are labeled in one of two ways. An edge is labeled as *in* if it joins two members of the same group and *out* if it joins members of different groups. If no vertex in the graph has an *out* edge, only one group exists and the *ARSN* is connected. If an *out* edge exists, the graph is partitioned into at least two groups and the *ARSN* is disconnected.

The difficulty of the preceding method is in identifying group members. In order to label incident edges of a vertex, we must be able to identify which of its neighbours belong to the same group. If the *ARSN* contains an edge from vertex *a* to vertex *b*, then clearly both vertices are members of the same group and the edge is labeled as *in*. If an edge connecting vertex *a* to vertex *c* exists but is not in the *ARSN*, then the local knowledge is insufficient to assign a label. A path may exist between vertex *a* to vertex *c* through a different vertex. This problem can be remedied by giving each member of the group the same label. When connecting edges between neighbouring vertices are not in the *ARSN*, we simply compare vertex labels to see if they are members of the same group. Vertex labeling can be accomplished by utilizing a depth-first search algorithm beginning at an arbitrary vertex in the graph. We assume that each vertex has an initial label of 0 and an ID $\neq 0$. As the search procedure visits each vertex in the group, it assigns the ID of the starting vertex as the group label. When the search is completed each vertex, which belongs to the same group as the starting vertex, will have the same label.

In the distributed environment, the labeling procedure is analogous to the distributed *leader election* problem. In general, leader election is the process of identifying, at each processor in the network, the processor with the maximum identity. For the purpose of this thesis, it is not important that the leader be the processor with the maximum identity and therefore we relax this constraint. We verify the connectedness of the *ARSN* by executing a distributed election algorithm over the *ARSN*. When the election is complete, each processor in the network begins the edge labeling process by sending a message, identifying its leader, to each of its neighbours. An edge is labeled as *in*, if both adjacent processors have the same leader. An edge is labeled as *out*, if the adjacent processors have different leaders or if one processor has not participated in an election. This last situation will occur if no processor in a group initiates the election algorithm. When a processor receives an identity message from a neighbour and has not participated in an election itself, it immediately becomes an initiator and starts the election algorithm in its group. Therefore it is not necessary that an initiator exists in each group.

Election, in an arbitrary graph, has been shown to have a message complexity of $\Omega(n\log_2 n + e)$ [LP86]. The general steps of one known algorithm are as follows:

1) Construct a spanning tree - $O(n\log_2 n + e)$ messages.

2) Broadcast the identity of the root of the spanning tree to all other processors in the network - $O(n)$ messages.

Constructing a spanning tree for the election algorithm provides us with an additional benefit. If the *ARSN* proves to be disconnected and we decide to repair the disconnection, we want to preserve the integrity of the *ARSN* by connecting two groups by the most reliable edge between them. Because a spanning tree is already constructed, the most reliable edge can be determined using an echo algorithm [C79] using O(n) messages. An echo algorithm has two stages of execution. The first stage consists of the leader broadcasting a message to all neighbours, which is eventually propagated to the leaves of the spanning tree. In the second stage, the responses from the leaves are propagated back to the leader.

The spanning tree may be constructed using a well known minimum spanning tree algorithm, such as the algorithm of Gallager, Humblet and Spira [GHS83]. Their algorithm has been shown to have optimal message complexity within a constant factor (*i.e.*, $5n\log_2 n + 2e + 4n$ messages). Although optimal, the GHS algorithm constructs a minimum spanning tree when only a spanning tree is required. Consequently, we pay the price in terms of extra messages and extra processing time required. For this reason we present a new spanning tree algorithm, with improved message complexity, in the following section.

4.5.1.1 Distributed Spanning Tree Algorithm For Arbitrary Networks:

To begin the discussion, we briefly outline how to distributively construct a spanning tree for the special case of exactly one initiator. The algorithm is based on a depth-first search procedure outlined previously: initially all processors are marked as *unvisited* and all edges are marked as *unexplored.* The initiator begins the algorithm by marking itself as *visited.* It sends a CONNECT message over an *unexplored* edge, chosen arbitrarily, and marks the edge as *explored.* The sending processor remains passive until a reply is received. Upon receipt of a CONNECT message, a processor marks the edge as *explored* and has two possible responses. If the processor is currently *visited,* the processor sends back a REJECT message to the sender. If the processor is currently *unvisited,* it marks itself as *visited.* The sending processor becomes its parent and the edge over which the message was sent becomes the parent edge. The receiving processor continues the search procedure by sending its own CONNECT message over one of its unexplored edges. When the receiving processor has no more unexplored edges it sends a COMPLETED message to its parent to indicate that it has finished. When the initiator receives a COMPLETED or REJECT message back from its last unexplored neighbour, the algorithm is complete. The spanning tree consists of all parent edges. The parent edge of each processor leads to the root of the spanning tree.

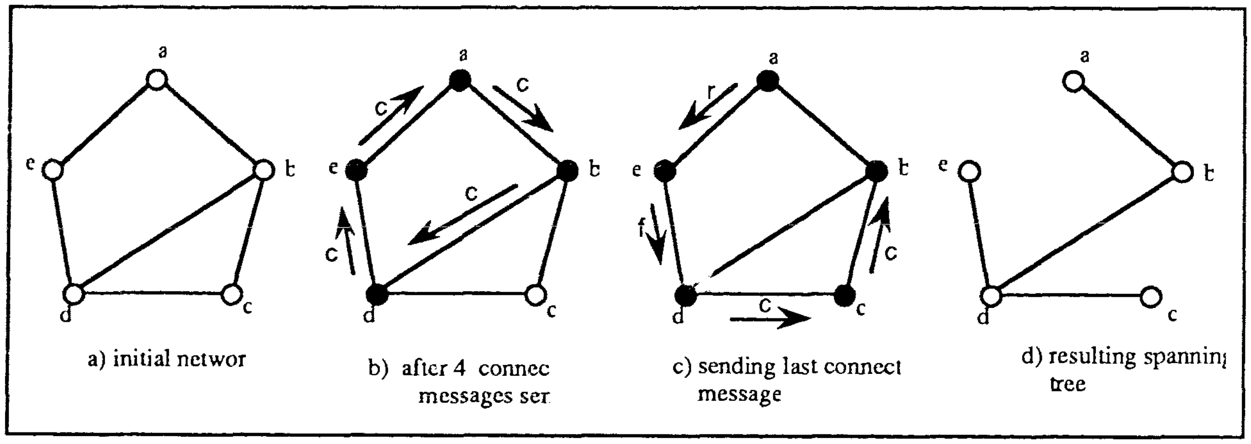

a
e
b
d
c
a) initial networ
b) after 4 connec messages ser
c) sending last connect message
d) resulting spannin; tree
c
f
r

Figure 4.6: Example of general spanning tree algorithm.

Figure 4.6 illustrates how the general spanning tree algorithm works. The network consists of five processors in which a white node represents a processor in an *unvisited* state, while a black node represents a processor in a *visited* state. The messages CONNECT, REJECT and COMPLETE are denoted as *c*, *r* and *f* respectively. Assume that processor *a* begins the algorithm. The sequence of CONNECT messages is as follows: a→b, b→d, d→e and finally e→a (figure 4.6b). The last CONNECT message from e→a is a situation in which the receiving processor is already a member of the spanning tree and consequently it responds with a REJECT message (figure 4.6c). Upon receiving the reply, processor *e* has examined all of its unknown neighbours and therefore has completed its role in building the spanning tree. It sends a COMPLETE message to its parent. The rest of the algorithm follows in the same manner and the complete spanning tree is composed of all parent edges (figure 4.6d).

When multiple initiators are allowed, initiators simultaneously construct fragments of a spanning tree and contend for members. Two spanning tree fragments overlap when a processor from one fragment sends a CONNECT message to a processor of another fragment. In this situation, the sending fragment tries to *annex* the receiving fragment. To handle multiple initiators we make the following changes to the algorithm: Each fragment is identified by the ID of its initiator, which is known to all members of the fragment. When a fragment sends a CONNECT message to a sleeping processor (*i.e.*, a non-initiator which is not currently a member of any fragment), the processor joins the fragment. When a fragment, F1, sends a CONNECT message to a processor which is already a member of a fragment, F2, the message is forwarded to the leader of F2. If ID(F1) < ID(F2) then F1 joins F2. Otherwise, F2 joins F1. Although not discussed here, the algorithm must also include a mechanism to ensure that a cycle is not formed when joining two fragments.

The simple method of joining two fragments, outlined above, leads to higher than optimal message complexity in the worst case. Consider the network in figure 4.7 in which n processors, labeled 0 to n, are arranged as shown. Initially, each processor i is the leader and sole member of fragment i. Fragment 1 begins the algorithm by sending a CONNECT message, denoted as c, to processor 0 (figure 4.7a). Because $0 < 1$, fragment 0 joins fragment 1, which now has two members, by sending a REPLY, denoted as r, message to fragment 1 (figure 4.7b). Processor 2 continues the algorithm by sending a CONNECT message to processor 0. Since processor 0 is not the leader of the fragment, it forwards the message along its parent edge to processor 1. Since $1 < 2$, fragment 1 joins fragment 2 (figure 4.7c). The spanning tree construction continues with each processor i sending a CONNECT message to processor i-2 in order. Eventually all processors are annexed into fragment n (figure 4.7d). It can be seen that for each processor i to become the leader of a fragment containing the processors $i...0$, exactly $2i$ messages are required (one connection request and one reply message over i edges). The total number of messages to create the spanning tree is $2\sum_{i=1}^{n-1} i = O(n^2)$. This is clearly not optimal since even an MST, which is a more tightly constrained spanning tree, can be constructed with $O(n\log_2 n + e)$ messages.

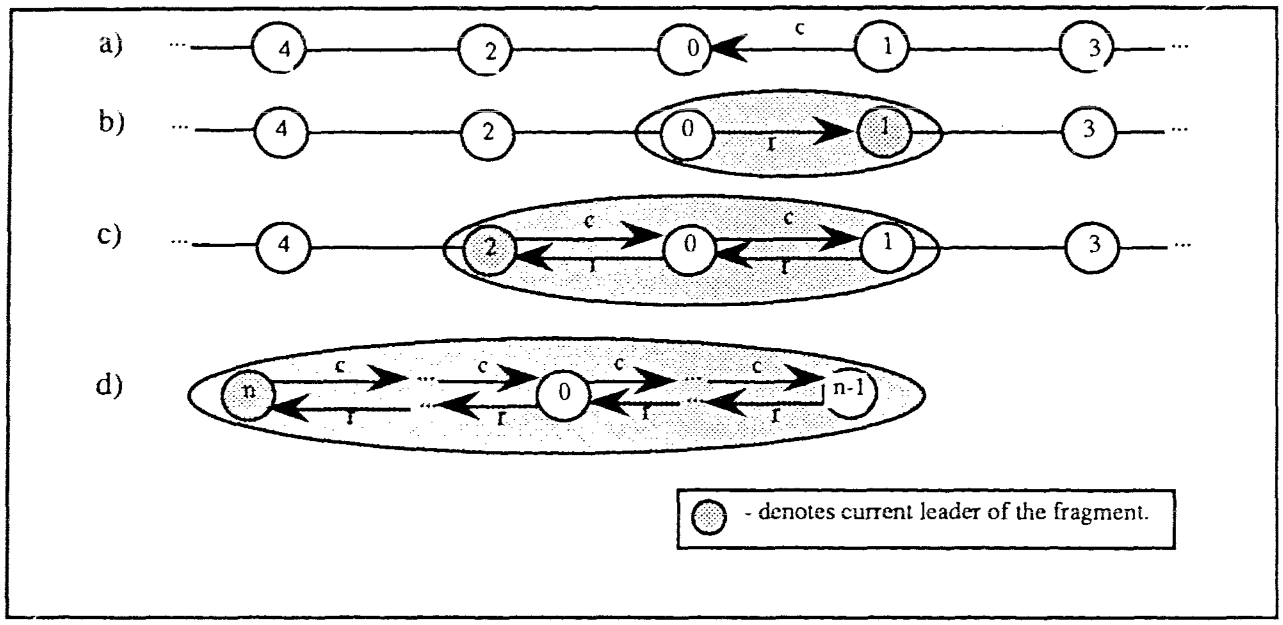


Figure 4.7: Distributed Spanning Tree Algorithm, O(n^2) Complexity in Worst Case.

4.5.1.2 Distributed Spanning Tree Algorithm With Optimal Message Complexity

In order to reduce the message complexity of the spanning tree algorithm we implement the following two changes: First, unlike the MST algorithms of Gallager, et al. [GHS83], and Bar-Yehuda, et al. [BKWZ87], in which each fragment has one processor which acts as leader for the entire existence of the fragment, we introduce the concept of the *emissary*. For the purposes of this algorithm, an emissary is the processor which acts as the current leader of the fragment. At some point in the algorithm, each processor in the network is an emissary of some fragment. By using an emissary, rather than a fixed leader, a connection request takes exactly one message since an emissary can only annex its own neighbour. Second, in the spirit of GHS we implement a *level* based mechanism of joining fragments. The level of a fragment is sent as part of the CONNECT message along with the fragment ID. When a fragment is at level i, the number processors in that fragment is at least 2^{i-1}. By using levels we can prevent a fragment, with a small number of members, from annexing a fragment with a larger number of members. In this situation, it would be more efficient, in terms of the number of messages, for the larger fragment to annex the smaller fragment.

The algorithm begins with all processors in either the *sleeping* or *emissary* states. Sleeping processors have level 0 and do not actively participate in constructing the spanning tree. When a sleeping processor is annexed into a fragment it becomes the current fragment emissary. Initiators, on the other hand, begin the algorithm as an emissary of a single processor fragment. Initially, fragments have level 1. With the exception of propagation delay, all processors in a fragment know the current level and identity of the fragment of which it is a member. The fundamental mechanism by which the algorithm operates is annexation. Specifically, a fragment expands by annexing passive processors or whole other fragments. A fragment which has been annexed by some other fragment ceases to exist. Fragments annex other fragments primarily on the basis of fragment level and in some situations also on the basis of fragment identity (ID). A fragment attempts to annex another fragment, or a sleeping processor, by sending a CONNECT message. The CONNECT message contains the fragment ID and its current level. There are four distinct cases of annexation:

Case 1: *Fragment annexing a sleeping processor (trivial case):* The simplest annexation case is when a fragment, *F*, sends a CONNECT message to a sleeping processor, as illustrated in figure 4.8a. Upon receipt of the connection request, the sleeping processor joins *F* by sending an ACCEPT message, denoted as a, to the emissary of *F* and becomes the new emissary of the enlarged fragment *F'* (figure 4.8b). When the emissary receives an ACCEPT message it knows that a new emissary exists and it enters the *member* state. The level of the fragment does not increase as a result of annexing a sleeping processor.

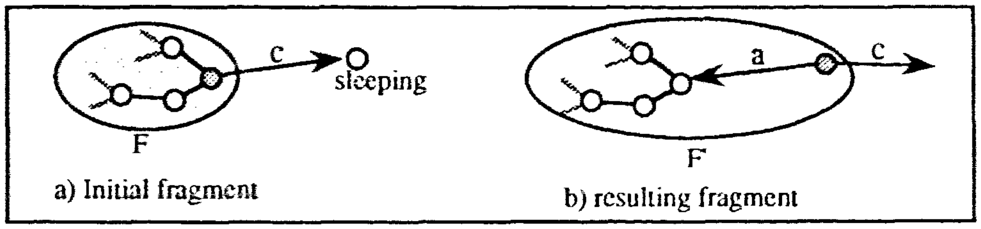


Figure 4.8 : A fragment Annexing a Sleeping Processor

Case 2 : *A smaller leveled fragment trying to annex a fragment with a larger level:* In this situation a fragment *F1* sends a CONNECT message to a processor of fragment *F2*, with *level(F1) < level(F2)* (figure 4.9a). *F1* must join *F2* because its level is smaller. The receiving processor in *F2* recognizes this, since it knows the level and identity of *F2*, and handles the joining of the two fragments without forwarding the message to the current emissary of *F2*. The justification for this is, although *F2* may simultaneously be annexed by yet a larger fragment, the level will never decrease and thus *F1* would join the fragment regardless of the outcome. The receiving processor informs the emissary of *F1* that it is to join *F2* by sending a CHANGE LEVEL/ID message, which includes the current level and ID of *F2*. Upon receiving this message, the emissary of *F1* relinquishes its leadership and broadcasts the change in level and leadership to all members of *F1* (figure 4.9b). After all members are notified *F1* ceases to exist. The current emissary of *F2* remains emissary of the enlarged fragment *F2'* and the level does not increase.

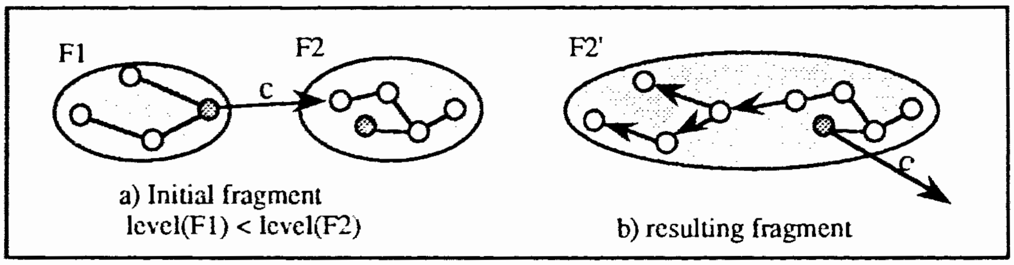


Figure 4.9: A smaller leveled fragment attempting to annex a larger fragment.

Case 3: *A larger leveled fragment annexing a fragment of a smaller level:* In this situation a fragment *F1* sends a CONNECT message to *F2*, where *level(F1) > level(F2)*. Similar to the previous case, when any processor in *F2* receives the CONNECT message it realizes that it can never be leader of the network and must join *F1*. In the distributed MST algorithm of Galleger, Humblet and Spira, the larger leveled fragment is made to wait for the smaller. The main reason for the delay feature is that, after a lower level fragment combines into a higher level fragment, the other processors of the fragment are not informed of the change for an uncertain period of time [GHS83].

In this algorithm we avoid making any fragment wait unless it knows it can not be network leader. Therefore, when a CONNECT message is received from a larger leveled fragment the receiving processor passes a TAKEOVER message to its current emissary. The TAKEOVER message is a more aggressive version of the CONNECT message because it has the added feature that it locks the receiving processor. A locked processor does not process any incoming message, including another TAKEOVER message from a competing fragment. Buffered messages are processed when the processor becomes unlocked. A processor is only unlocked when it receives a CHANGE LEVEL/ID message with a larger level then it currently knows. In the process of annexing *F2* , fragment *F1* will encounter one of three situations:

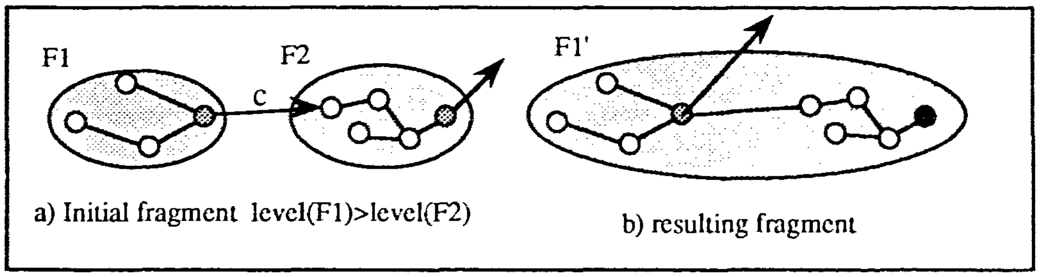


Figure 4.10: A Larger Leveled Fragment Annexing a Smaller Fragment (normal case)

normal situation: The TAKEOVER message is sent up the fragment to the emissary of *F2* (figure 4.10a). When the message arrives at the emissary it is buffered until its current processing is completed. When the message is processed, the emissary immediately relinquishes its leadership and sends a CHANGE LEVEL/ID message, with the level and ID of *F1*, over the spanning tree thereby joining *F1* and unlocking all processors in the process (figure 4.10b). When the emissary of *F1* receives the message with its own level and ID it knows it has captured *F2*.

competing annexation: In this situation another fragment *F3*, with level(*F3*) > level(*F2*) simultaneously tries to annex *F2*. Both *F1* and *F3* send TAKEOVER messages over the fragment F2 to the emissary. Because a TAKEOVER message locks the receiving processor, only one TAKEOVER message will reach the emissary without encountering a locked processor. The locking mechanism allows us to process competing requests in a sequential manner. When the emissary of *F2* receives the TAKEOVER message, say from *F1*, it processes the message as in the first case and sends the CHANGE LEVEL/ID message down the spanning tree. When the message arrives at *v* it is unlocked and immediately processes the TAKEOVER message of *F3*. If level(*F3*) < level(*F1*), the CHANGE LEVEL/ID message is simply sent to *F3*. In this case the both *F2* and *F3* are annexed by *F1* and its level does not increase. If level(*F3*) > level(*F1*), a new CHANGE LEVEL/ID message with the level and ID of *F3* is sent to both *F1* and *F2*. In this case both *F1* and *F2* are annexed by *F3* and its level does not increase. If level(*F3*) = level(*F1*), then a new fragment is created and the level is increased. The leader of the new fragment is the emissary of the fragment with the largest ID. The new CHANGE LEVEL/ID message is sent to *F1*, *F2* and *F3*. When a fragment receives a CHANGE LEVEL/ID message with its own ID but different level, it broadcasts the message to all of its member and its emissary remains in leadership.

increased level: Because of propagation delay, fragment *F2* may have expanded to *F2'* and be in the process of updating the level and ID to its members when *F1* tries to annex it. When this happens, a TAKEOVER and CHANGE LEVEL/ID message will "collide" at some processor of the spanning tree of *F2'* or at the emissary of *F2'*. If level(*F1*) > level(*F2'*) or level(*F1*) = level(*F2'*) *and* ID(*F1*) > ID(*F2'*) then the CHANGE LEVEL/ID message is terminated and the TAKEOVER message proceeds to the emissary of *F2'* as usual. If level(*F1*) < level(*F2'*) then the TAKEOVER message is terminated and *F1* is annexed into *F2'* by propagating the CHANGE LEVEL/ID message from *F2'* to *F1*. If level(*F1*) = level(*F2'*) *and* ID(*F1*) < ID(*F2'*) then F1 is suspended by propagating the CHANGE LEVEL/ID message from *F2'* (see case 4). If level(*F1*) = level(*F2'*) *and* ID(*F1*) = ID(*F2'*) then the TAKEOVER message has encountered its own identity. This occurs when both the larger and smaller simultaneously attempt to annex the other. In this case *F2* sends a CONNECT message, along a different edge, to *F1* and subsequently becomes annexed.

The TAKEOVER message, propagated by the emissary of F1, is halted and a REJECT message is sent back to indicate that the edge creates a cycle.

Case 4: *A fragment tries to annex a fragment of the same level:* In this situation fragment *F1* sends a CONNECT message to fragment *F2* in which level(*F1*) = level(*F2*). Unlike constructing a MST, in which two fragments may only be connected by the minimum-weight edge joining them, two fragments of the same level may simultaneously try to annex each other using two different edges. For this reason, a fragment may only annex another fragment of the same level if it has a larger ID. When a fragment sends a CONNECT message to a fragment of the same level but larger ID, the receiving processor replies with a CHANGE LEVEL/ID message. When an emissary receives a CHANGE LEVEL/ID message with the same level as its own but larger ID, it knows it has tried to annex a fragment with the same level and larger ID and subsequently becomes *suspended.* A suspended fragment is passive and waits to be annexed by a larger or equal leveled fragment. However, a suspended fragment may annex a fragment with a smaller, if the other fragment initiates the annexation. When the annexing fragment has a larger ID, a TAKEOVER message is sent to emissary of *F2* and the situation is analogous to case 3.

Each edge in the network is labeled to distinguish those which have been processed (examined) from those which have not. Initially, every edge is *unexplored.* When an edge is processed it is given a label to identify its status within the network. A *reject* edge connects a fragment to itself (i.e., a cycle) and therefore is not part of the spanning tree. A *parent* edge is a member of the fragment spanning tree and leads to the current emissary of the fragment. Notice that it is always possible to send a message to the current emissary without having to broadcast throughout the fragment. A *child* edge is a member of the spanning tree and leads to a leaf. A *completed* edge is identical to the *child* edge except that all edges in the subtree have been completely explored and no further processing is required.

Because each fragment has a dynamic emissary, rather than a fixed leader, a processor relinquishes control before it has completely examined all adjacent edges. Moreover, when a fragment is annexed, control is seized from the emissary leaving some edges unexamined. For this reason we need a way of passing control back to a processor in our fragment to complete processing. This is accomplished by using the RESUME message. The current emissary passes leadership of the fragment to another processor in the fragment when it has no *unexplored* edges left. In this case it sends the RESUME message over a *child* edge. A leaf in the spanning tree is a processor which has at most one *child* edge and the remainder *rejected.* When a leaf processor has examined all edges then processing at this processor is finished. The leaf processor passes control to its child by sending a COMPLETED message. A COMPLETED message passes on control, like a RESUME message, but signals the receiving processor that processing is finished. A processor which has sent a COMPLETED message will never become emissary again. When a processor receives a COMPLETED message it marks that edge as *completed.* When an interior processor (*i.e.*, a processor with more than one child) receives a COMPLETED message from all but one of its children, it has finished processing and sends its own COMPLETED message to its final child. A processor with only *completed* and *rejected* edges is the leader of the spanning tree and the algorithm terminates.

When the algorithm terminates each processor knows the identity of the fragment which forms the spanning tree. Moreover, each processor can identify the local edges which form the spanning tree (*i.e.*, *completed* edges). The leader of the spanning tree is the last emissary of the fragment. The fragment and the leader of the spanning tree do not necessarily have the same identity.

4.5.1.3 Distributed Spanning Tree Algorithm - Implementation Details

In the preceding algorithm the following processor states are used:

sleeping -> A processor which knows it is not a local maximum, does not actively participate in forming a fragment. Waits passively to be annexed.

emissary -> A processor which is actively seeking out new processors or other fragments to add to the fragment of which it is a part. The current emissary is the fragment leader.

member -> A non-leader and non-emissary processor in a fragment. Plays a passive role in the fragment and the only role it fulfills is to relay message to the leader or emissary.

suspended -> When an emissary sends a CONNECT message to another fragment with the same level but larger ID, the sending fragment is to be annexed by the receiving process. However, the level of the new fragment must increase as a result which requires the cooperation of the emissary of the larger fragment, which may be busy annexing other fragments. Instead the emissary of the smaller fragment enters a suspended state in which it waits to be annexed by another fragment.

The following messages are used in the preceding algorithm:

CONNECT -> A leader or emissary sends this message over a previously unexplored edge as an invitation to join the fragment.

REJECT -> Sent in response to a CONNECT message by a processor which is already a member of the initiator's fragment.

ACCEPT -> Sent in response to a CONNECT message by a *sleeping* processor. In this case the receiving processor joins the fragment and becomes the emissary of the new fragment.

CHANGE LEVEL/ID -> Sent to inform the member of a fragment that the level, and possibly the ID, of the fragment has increased. When two fragments amalgamate it is sent in place of an accept message. Also, this message is sent in response to a CONNECT message from an equal leveled fragment with a smaller ID. In this case the fragment with the smaller ID is suspended.

TAKEOVER-> Sent to an emissary of a smaller leveled fragment or a fragment of equal level but smaller ID. When a processor receives this message it is locked and will not process other messages until it receives a CHANGE LEVEL/ID message with a larger level.

RESUME -> When two fragments amalgamate only one emissary survives to lead the new fragment. As a result a number of edges may be only partially explored (i.e., still in the child state). The current emissary sends this token message to pass leadership to a child when no unexplored edges exist.

COMPLETED -> An emissary sends this message to its parent in the spanning tree to indicate that all possible edges have been examined. When a processor has examined all adjacent edges it enters the *finished_member* state.

4.5.1.4: Distributed Spanning Tree Algorithm:

```
/* sleeping state - a sleeping processor waits to be annexed */
sleeping:
    receive(rec_msg,rec_edge)
    IF msg = CONNECT                              /* an invitation from a fragment to join */
        edge_state[edge] ← child
        rec_edge ← send(ACCEPT)
        become emissary
```

```
/* create fragment and try to annex all other processors */
emissary:
   IF initiator
      level ← 1, frag_id ← my_id;                          /* initialize level and fragment identity */
   FOREACH edge e suchthat (edge_state[e] = unexplored) DO         /* while unexplored edges exist */
    e←send(CONNECT,level,frag_id)                                 /* send connect message on edge e*/
    receive(msg,rec_edge)                                                    /* receive a message */
    WHILE rec_edge ≠ e                                              /* while the reply is outstanding */
       msg_queue ← msg                                   /* save message on queue to process later */
       receive(msg,rec_edge)                                                /* get another message */
    END WHILE

   PROCESS msg OF TYPE:                                /* process the message we were waiting for*/
    1) ACCEPT:                                                     /* annexed a sleeping processor */
       edge_state[rec_edge] ← parent                              /* mark the edge as parent edge  */
       become member                                                      /* relinquish leadership */

    2) REJECT:                                                        /* edge would produce a cycle */
       edge_state[explore_edge] ← reject                                 /* mark edge as rejected */

    3) CHANGE LEVEL/ID:
       IF get_level(msg) = level           /* suspended - tried to annex frag with same level but larger ID */
         edge_state[explore_edge] ← unexplored                               /* reset the edge status */
         become suspended
       ELSEIF get_id(msg)  ≠ frag_id                    /* different ID - annexed by another fragment */
         level ← get_level(msg); frag_id ← get_id(msg)                        /* get new level and ID */
         edge_state[explore_edge] ← parent;                                   /* set new parent edge */
         broadcast(CHANGE LEVEL/ID,level,frag_id)     /* send new level/ID to all members of frag. */
         become member                                                    /* relinquish leadership */
       ELSE                                                       /* we've taken over another fragment */
         IF get_level(msg) > level                                   /* see if the level has changed */
           broadcast(CHANGE LEVEL/ID,level,frag_id)
                                                      /* send new level/ID to all members of frag. */
   END PROCESS msg


  /* after we have received the expected message process the messages in the message queue */
  FOREACH msg in msg_queue DO
     PROCESS msg OF TYPE:

    /* received a connect or takeover message - when received at emissary they have the same effect */
     1) CONNECT:                              /* connect & takeover message process same way */
     2) TAKEOVER:
       IF level > get_level(msg)                                 /* smaller level - we annex them */
         rec_edge←send(CHANGE LEVEL/ID, level, frag_id)         /* send new level/ID to fragment */
         edge_state[rec_edge] ← child                                         /* set edge to child */
       ELSEIF level = get_level(msg)                            /* same level - must increase level */
         level ← level + 1                                           /* increase the level number */
         edge_state[rec_edge] ← child                                    /* set edge state to child */
         rec_edge←send(CHANGE LEVEL/ID,level,frag_id)     /* send new level/ID to other fragment */
         broadcast(CHANGE LEVEL/ID,level,frag_id)            /* inform our frag of new level/ID */
       ELSE                               /* level greater - we are annexed & relinquish control*/
         edge_state[rec_edge] ← parent                               /* set edge state to new parent */
         level ← get_level(msg)                                /* new level is level of larger fragment */
         frag_id ← get_ID(msg)                                  /* new ID is ID of larger fragment */
         rec_edge←send(CHANGE LEVEL/ID,level,frag_id)
                                                       /* this tells larger frag. we have joined it */
```

```
          broadcast(CHANGE LEVEL/ID,level,frag_id)          /* inform our frag of new level/ID */
          become member

      END FOREACH /* no more messages in the queue */
    END FOREACH /* no more unexplored edges */

    FOREACH edge e DO                                 /* if any edge exists which is not completed */
     IF edge_state[e] = child                              /* then pass the leadership on to it */
        e←send (RESUME)
        become member;

    IF edge_state[parent] = child                  /* when only one child edge is left and all others */
        e←send (COMPLETE)                          /* are completed then send a completed message */
        become member;

    /* if no child edges are left then the spanning tree much be complete */
    become finished;

/* a member of the spanning tree - relays messages to the current emissary */
member:
    IF initial = true
        locked ← false

    IF locked = false AND msg_queue NOT empty                 /* process message on the queue first */
      msg ← msg_queue                                                /* get message from the queue */
    ELSE
      receive(msg, rec_edge)                                          /* else receive a new message */

    IF locked = true AND msg CHANGE LEVEL/ID              /* if locked wait until change level/ID msg */
      msg_queue ← msg                                           /* save the message on the queue */

    PROCESS (msg) OF TYPE:                                             /* process the new message */

     1) CHANGE LEVEL/ID:                                     /* change the level and ID of fragment */
        level ← get_level(msg), frag_id ← get_id(msg)                       /* set new level & ID */
        IF locked = true                                                          /* in locked mode */
          locked ← false                                                        /* unlock the node */
          msg ← get_takeover_msg(msg_queue)                  /* check for takeover message buffered */
          IF level > get_level(msg)                     /* if level greater than takeover then ignore */
            edge_state[msg_edge]← child                /* set edge takeover msg arrived on to child */
           broadcast(CHANGE LEVEL/ID,level,frag_id)          /* inform rest of frag of new level/ID */
          ELSEIF level < get_level(msg)                       /* new takeover msg has higher level */
            level ← get_level(msg)                                                   /* get new level */
            frag_id ← get_ID(msg)                                              /* get new fragment ID */
           edge_state[parent] ← child                              /* change old parent edge to child */
           edge_state[msg_edge] ← parent               /* edge that takeover msg came on is now parent */
           broadcast(CHANGE LEVEL/ID,level,frag_id)          /* inform rest of frag of new level/ID */
          ELSE                                        /* takeover msg had same level - create new level */
            level level + 1                                                         /* increase level */
            edge_state[msg_edge] ←child                    /* edge that takeover msg came on is child */
           broadcast(CHANGE LEVEL/ID,level,frag_id)          /* inform rest of frag of new level/ID */

        ELSE                                                                        /* was not locked */
         edge_state[parent] ← child                                /* change old parent edge to child */
         edge_state[rec_edge] ← parent                                         /* set new parent edge */
         broadcast(CHANGE LEVEL/ID,level,frag_id)            /* inform rest of frag of new level/ID */
```

```
  2) CONNECT:
     IF level > get_level(rec_msg)                        /* attacking frag has smaller level */
       rec_edge←send(CHANGE LEVEL/ID, level, frag_id)
                                                          /* send our level/ID to smaller fragment */
       edge_state[rec_edge] ← child                       /* set edge to be a child */
     ELSEIF level = get_level(rec_msg) AND get_id(rec_msg) < frag_id
                                     /* else if same level but smaller ID then suspend the fragment */
       rec_msg←send(SUSPEND)                              /* suspend the attacking fragment */
     ELSE                                                 /* attacking fragment is larger or larger ID */
       locked ← true                                      /* lock the node */
       parent ← send(TAKEOVER)                            /* send TAKEOVER message to parent */

  3) TAKEOVER:                                            /* in the process of being annexed by another node */
     locked ← true                                        /* lock the node */
     parent ← send(TAKEOVER)                              /* send TAKEOVER message to parent */

  4) COMPLETED:                                           /* this branch is completely explored */
     edge_state[rec_edge] ← completed
     become emissary

  5) RESUME:                                              /* the processor becomes emissary of the fragment */
     edge_state[rec_edge] ← child_resme
     become emissary

/* the fragment tried to annex a fragment of same level but greater ID - now waits passively to be annexed
    however can still annex a smaller processor which tries to connect to it*/
suspended:
    receive(msg, rec_edge)                                /* get a new message to process */

    PROCESS msg OF TYPE:

     /* received a connect or takeover message - when received at emissary they have the same effect */
     1) CONNECT:
     2) TAKEOVER:
       IF level > get_level(msg)                          /* smaller level - we annex them */
         rec_edge←send(CHANGE LEVEL/ID, level, frag_id)   /* send new level/ID to fragment */
         edge_state[rec_edge] ← child                     /* set edge to child */
       ELSEIF level = get_level(msg)                      /* same level - must increase level */
         frag_id ← get_ID(msg)                            /* new ID is ID of annexing fragment */
         level ← level + 1                                /* increase the level number */
         edge_state[parent] ← child                       /* set old parent edge to child */
         edge_state[rec_edge] ← parent                    /* set edge state to parent */
         rec_edge←send(CHANGE LEVEL/ID,level,frag_id)     /* send new level/ID to other fragment */
        broadcast(CHANGE LEVEL/ID,level,frag_id)          /* inform our frag of new level/ID */
       ELSE                                  /* level greater - we are annexed & relinquish control*/
         edge_state[rec_edge] ← parent                    /* set edge state to new parent */
         level ← get_level(msg)                           /* new level is level of larger fragment */
         frag_id ← get_ID(msg)                            /* new ID is ID of larger fragment */
         rec_edge←send(CHANGE LEVEL/ID,level,frag_id)     /* tell larger frag. we have joined it */
        broadcast(CHANGE LEVEL/ID,level,frag_id)          /* inform our frag of new level/ID */
         become member
```

Algorithm 4.3: Distributed Spanning Tree Algorithm

4.5.1.5 Distributed Spanning Tree Algorithm - Correctness and Complexity

Theorem 4.4: Algorithm 4.3 eventually terminates with the *completed* edges forming a spanning tree.

Proof. From the detailed discussion of the algorithm, it can be seen that an edge is added to a fragment if, and only if, that edge does not cause a cycle in the fragment. Moreover, every edge in the network is given an opportunity to join a fragment by sending a CONNECT message to one adjacent node. Therefore, if the algorithm terminates, it terminates correctly.

The algorithm terminates when all the edges incident to one node are labeled *reject* or *completed.* If we assume that there is only one initiator, then the spanning tree consists of a single fragment which is constructed entirely by annexing *sleeping* processors. Assuming that the algorithm visits all nodes of the network and considers all edges appropriately, it is obvious that the algorithm terminates. Assume that more than one initiator exists and at least two fragments exist in the network at one time. Therefore, to show that the algorithm terminates, it is sufficient to show that no deadlocks exist when combining fragments. From the discussion in section 4.5.1.3, each time two fragments combine, one emissary relinquishes leadership and the other remains in control of the combined fragment. The exception to this is when a fragment tries to annex another fragment, with the same level but larger ID, and is suspended in the process. The suspended fragment is passive, will never increase its level and will never form the complete spanning tree. A deadlock can only exist if all fragments in the network are suspended. However, this is impossible since a fragment can only be suspended by a fragment with the same level and larger ID. Even if all fragments have the same level, there must be one fragment with the largest ID which can not be suspended. Therefore, at least one processor is an active emissary at all times and no deadlock exists. ■

Theorem 4.5: The total number of messages used to construct a spanning tree by Algorithm 4.3 is $O(n\log_2 n + e)$.

Proof. By definition, the maximum level of any fragment is $\log_2 n$. During each level, at most n-1 TAKEOVER, RESUME and CHANGE LEVEL/ID messages may be sent for a total of $3(n-1)\log_2 n$. The only exception to this is the CHANGE LEVEL/ID message which can also be used to suspend another fragment. In the worst case, n fragments can exist and therefore at most n-1 fragments can be suspended through the entire execution of the algorithm. A CONNECT message and its response (i.e., ACCEPT, REJECT) is sent over each edge in the network for a total of 2e messages. Finally, at most n-1 COMPLETED messages are sent when all processing is completed at a node.

The total number of messages is: $3(n-1)\log_2 n + 2e + 2(n-1)$ which is $O(n\log_2 n + e)$. ■

4.5.1.6 Determining the Connectedness of the Approximate Reliable Subnetwork - Algorithm Details:

Once the spanning tree has been constructed on the *ARSN*, using the previous algorithm, the test for disconnection is straightforward. Each processor in the network checks for *out* edges and then reports the results to the leader. The algorithm may be implemented as follows: the leader of the spanning tree broadcasts a request to search for *out* edges. When a processor receives the broadcast it performs the search by sending an INQUIRE message, containing the spanning tree ID, over each edge which is not in the *ARSN*. When a processor receives an INQUIRE message which contains the same spanning tree ID, the edge is an *in* edge and the processor replies with a CONNECTED message. If a processor receives an ID, different than its spanning tree ID, the edge is an *out* edge and the receiving processor replies with a DISCONNECTED message. A leaf processor in the spanning tree sends a CONNECTED message to its parent, if it has no *out* edges, and a DISCONNECTED message otherwise. An interior processor in the spanning tree sends a CONNECTED message to its parent, if it has no *out* edges and it received a CONNECTED message from each of its children. Otherwise, the interior processor sends a DISCONNECTED message. This communication pattern has been referred to as a *convergecast* in [A85]. The algorithm terminates when the leader receives a reply from each of its children. The *ARSN* is disconnected if the leader receives at least one DISCONNECTED message.

Since the *ARSN* has a reduced size of at most K(n-1) edges, the spanning tree can be constructed with $3n\log_2 n + 2K(n-1) + 2n$ messages. The convergecast requires exactly 2(n-1) messages to begin the check for connectedness and return the results. Each edge which is not in the *ARSN* is queried for a total of $2(e - Kn)$ messages. Therefore, the total number of messages to determine if an *ARSN* is connected is: $3n\log_2 n + 2K(n-1) + 2n + 2(n-1) + 2(e - Kn)$ which is $O(n\log_2 n + e)$.

4.5.2 Repairing A Disconnected Approximate Reliable Subnetwork.

When the *ARSN* is found to be disconnected we must decide whether to rebuild the *ARSN* or repair the existing structure. As previously discussed, if we rebuild the *ARSN* it generally requires that the value of K be increased. The exception to this is if a property of the network can be changed, for example processor labeling, or if the tie breaking scheme can be changed. In these situations it is possible to reconstruct the *ARSN* using the same value of K and possibly produce a different structure which is connected. If we assume that the value of K is increased, the new *ARSN* can be constructed as follows. The leader of the spanning tree broadcasts a message to all members of the fragment which informs them to select an additional edge. Upon receiving the message each edge selects its $K+1$ most reliable edge and includes it in the *ARSN*, if it is not already a member. When a processor adds a new edge to the structure, the processor at the other end of the edge is informed. Beginning with the leader, the spanning tree algorithm is executed which sends a connect message over each new edge. When the new spanning tree is complete, the leader broadcasts the new fragment identity. Upon receiving the new fragment identity, each processor tests previous out edges to see if they still connect to a different fragment. If at least one edge remains *out*, then the *ARSN* remains disconnected and K is increased again and the procedure is repeated.

Although the *ARSN* can easily be rebuilt, the additional edges required may negate any previous advantage which the structure offered. Furthermore, because the *ARSN* is not guaranteed to be connected, it may require that the value of K be incremented numerous times before a connected structure is constructed. In general, it will be better to repair the existing structure. We justify our choice based on the following observations:

1. Based on the observations of section 4.3, when an *ARSN* for $K \geq 2$ is disconnected it is likely to be partitioned into only two groups. The repair algorithm can connect two fragments by finding the most reliable edge between them.
2. We have spent considerable network resources already on building the spanning tree and checking for connectedness. In light of the previous observation and because the spanning tree is already in place, it may be a simple task to connect the fragments.

When the *ARSN* is disconnected, there exists at least two fragments, and possibly more, which must somehow be connected. Each fragment has an active leader and together they must cooperate to join into a single connected *ARSN*. Joining fragments of the spanning tree is very similar to constructing a spanning tree. The spanning tree algorithm, with minor modifications, could be used to join the fragments except that algorithm 4.3 connects two fragments by an arbitrary edge. In order to maintain the reliability of the *ARSN*, fragments must be joined by the most reliable edge between them. In fact, the algorithm to repair the disconnection will be almost identical to a minimum-weight spanning tree algorithm. Because there are a number of distributed MST algorithms available we will not formally define a disconnection repair algorithm. However, we briefly outline the steps of the algorithm which are as follows. The leader of a fragment broadcasts a request for the most reliable *out* edge in the fragment. Upon receiving the request, a processor tests each unused edge to see if it is an *out* edge and sends, by a convergecast, the reliability the most reliable *out* edge in its subtree. When the leader of the fragment knows the most reliable *out* edge, it initiates a connection request between the two fragments. Through an annexation process, similar in concept to that of algorithm 4.3, the two fragments are joined. The leader of the new fragment again broadcasts a request for the most reliable *out* edge. The process terminates and the *ARSN* is connected when no *out* edges are reported.

The number of messages required to repair a disconnected *ARSN* is ultimately dependent on the number of fragments which exist. From the results of table 4.4, when an *ARSN* is disconnected there is likely to be two partitions, one small and one large. In this situation, the smaller fragment will likely discover and initiate the repair, before the larger discovers a disconnection exists. The spanning tree algorithm can be modified to annex a disconnected fragment when a REPAIR message is received. The number of extra messages required to repair the disconnection is minimal. In the worst case however, when $K=1$ and each fragment contains 2 processors, there can be O(n) fragments. Therefore, the number of messages to repair the disconnection in the worst case is $O(n\log_2 n + e)$.

Chapter 5

Empirical Results:

Evaluating the Approximate Reliable Subnetwork

5.1 Motivation:

In chapter 4 we gave a distributed algorithm to compute an *approximate reliable subnetwork* (*ARSN*) for a predetermined value of *K*, of a given general network. The algorithm is based on a graph reduction technique whereby all but the *K* most reliable edges for each vertex are eliminated. The primary advantage of the *reliable subnetwork* is the reduction in the *size* of the network. Since the network is reduced by eliminating the most unreliable edges, we can expect the total number of significant edge failures to be reduced (since the most unreliable edges are not used a failure of one of these is of little consequence). Because of this, we reduce the frequency which local routing tables need to be updated or regenerated. Also as a result of reduced *size*, a distributed computation, which is measured in terms of message complexity, would be expected to execute noticeably faster. Unfortunately, both of these benefits do not come without a price. Because we are reducing the graph by eliminating edges, we unavoidably decrease the reliability of the *ARSN* with respect to the original network (after all we can not increase the reliability of a graph by taking away edges). Given this invariant, the question arises as to how significantly does the *ARSN* reduce the reliability?

In this chapter we empirically evaluate the *ARSN* for various values of *K*, given networks of different order, size and reliability of communication edges. We evaluate the *ARSN* in terms of reliability, path length and total edge count. We conclude with a discussion of appropriate values of *K* and an overall analysis of the *ARSN*.

5.2 Test Philosophy:

Comparing the reliability of an *ARSN* to the original network is a relatively straightforward task, however, making general observations which apply to all *ARSN*s is much more difficult. The difficulty arises due to the fact that there is an infinite number of network topologies to test. Because it is possible to test only a small portion of possible network topologies, our approach is to randomly generate networks to be tested. The intent is to obtain a representative sample of general graphs, which may include any number of different classes of graphs. However, when graphs are completely random it becomes difficult to make general observations as the physical structure of the graph may affect the outcome of the results. For this reason, we restrict the types of topologies we look at by making the following assumptions, concerning the physical properties of the graphs:

Graph Order: The primary advantage of the *ARSN* is in the reduction in edges , or network size, without a significant reduction in network reliability. For a distributed algorithm , reduction in terms of messages sent may be significant if the order of the network is sufficiently large. However, for a small network, say less than 10 vertices as one might find in a local area network, the small number of vertices may not justify the overhead costs of computing the *ARSN*. Moreover, if the network is sufficiently small, the computational savings realized in the reduction of the network size may be insignificant. For this reason we consider only network topologies where the number of vertices is large. In the specific networks which we generate we use values of n = 20, 40 , 60, 80 and 100.

Graph Size: As in the previous case, the primary advantage of the *ARSN* is in the computational savings when the size of the network is reduced. However, even when the size of the network is large, if each vertex has small degree, the reduction in the size of the network will be small even when $K=1$. An example of this type of network is the ARPAnet where most vertices have 2 or 3 edges. For this reason we restrict ourselves to dense networks of large size. In the specific networks which we generate we use values of $e = 20\%, 40\%, 60\%, 80\%$ and 100% of all possible edges.

Edge reliability: It is difficult to accurately specify the reliability of a typical edge in a computer network. Recall that we define the reliability of an edge as the probability that it is in a operational state. The definition was left intentionally vague so as to encompass a number of more restrictive definitions of reliability. For instance, if edge reliability is defined in terms of the *availability* of an edge for a specific period of time, then we would expect the reliability of an individual edge to be quite high. In a study of the ARPAnet, by Frank and Chou [FC74], where availability was the reliability criteria, it was found that the average edge had a reliability of 98%. In contrast, if we define reliability to be the probability that an edge has no failures during a given period of time, then we would expect the reliability of a typical edge to be much less. Even the physical properties of the transmission medium can affect the reliability of an individual edge. For these reasons it is difficult to give typical bounds on the reliability of an edge. We make the assumption that the reliability of individual edges can have significant variance[1]. In our analysis we generate graphs in which edges have the following ranges: [100%...1%], [100%...50%], [100%...75%] and [100%...95%].

The original intent for constructing an *ARSN* was to provide a reliable environment in which to execute distributed algorithms. Consequently, the ideal environment for evaluating an *ARSN* would be to physically create a network, introduce various edge faults, compute the

[1] After all, if all the edges in the network had the same reliability then an algorithm which randomly chose K edges for each vertex would produce the same effect as computing the RSN.

reliable network and run various distributed algorithms. Given this environment, one could then compare the types and quantities of edge failures encountered on the *ARSN* with the identical algorithms running on the original network.

Unfortunately, such a setup is beyond the scope of this thesis. Accordingly, we must resort to evaluating *ARSN*s of randomly generated networks.[2] Moreover, because we have no way of simulating the execution of a distributed algorithm over a network with edge failures and recoveries, we must evaluate the *ARSN* using other criteria. In particular we use the following three measures in evaluating the *ARSN*:

1) *Reliability:* By far, the most important evaluation criteria of the *ARSN* is the reliability, as defined in chapter 3, in comparison to the reliability of the original network. By definition, the reliability of the original network must be greater than or equal to that of the *ARSN*. However, if the reliability of the *ARSN* is not significantly less and the number of edges has been significantly reduced, then the *ARSN* offers certain advantages for distributed execution over the original network.
2) *Reliable Path length*: Ideally, the *ARSN* will be used as a general communication network in which routing tables are computed and messages are potentially passed between all pairs of processors. We assume that a message travels over the most reliable path between two processors. If path lengths differ significantly between the *ARSN* and the original network, then messages must travel farther to reach their destination and the justification for computing the structure is weakened. We examine both average and worst case (maximum) most reliable path length per graph.
3) *ARSN Size:* As previously deliberated, the primary advantage of the *ARSN* is in the computational savings due to the reduction in the size of the network. If the network reduction is significant then the cost overhead for computing the *ARSN* and for determining connectedness can be justified.

[2] Refer to Appendix 1 for a relevant discussion and evaluation of the random number generator.

5.3 Test Setup & Methodology:

The test was organized in the following manner. Random graphs were generated for each of the 100 combinations of order, size and edge reliability (as discussed in the previous section). For each graph combination, a 1000 graphs were created and evaluated, for a total sample of 100,000 graphs. The random graphs were generated using the algorithm of Nijenhuis and Wilf [NW78] which generates random labeled graphs of fixed size and order based on the probability model B for random graphs.[3] The algorithm uniformly generates, at random, a subset of size j from a set of size n, where $j \leq n$. Generating a graph, of order n and size e, is then a matter of conceiving a labeling for the $\binom{n}{2}$ possible edges and randomly choosing e edges. The reliability of each generated edge is uniformly generated from the given range. After each random graph is generated a spanning tree is constructed to verify connectedness. Graphs found to be disconnected are discarded.

Following the generation of a connected graph, the reliability of the graph is computed using algorithm 3.1. In addition, if the graph is complete, the most reliable paths between all pairs of processors is computed. From these paths, we generate the average and maximum path length values.

The *ARSN* is constructed, for values of $K = 1...9$, [4] using algorithm 4.2 to choose the K most reliable edges for each processor. The *ARSN* is then checked for connectedness by constructing a spanning tree. If it is found to be disconnected, the *ARSN* is repaired by sequentially adding the most reliable edge, not currently in the *ARSN*, until the structure is connected. For each *ARSN* constructed, the reliability is computed. The actual reliability values for the *ARSN*s computed are only meaningful as a comparison to the reliability of the original graph. For this reason, we *normalize* the reliability of an instance of the *ARSN* by

[3] For a complete discussion of probability models A, B and C see [P85].

[4] Although approximate reliable subnetworks were computed for values of K=1...9 it was found that for values of K>5 there was no significant difference in reliability or path lengths compared to the original graph. For this reason, all *ARSN*s with K>5 are removed from further consideration

expressing it as a percentage of the reliability of the original graph. A normalized reliability of 95% for a particular *ARSN*, for example, means that the structure is 95% as reliable as the original network.

If the original graph is complete, we compute the most reliable path between all pairs of processors in the *ARSN*. From these path lengths we can compute the maximum most reliable path and the average path length of the most reliable paths. In addition, we count the number of edges in the *ARSN* to determine how much the size of the network has been reduced.

5.3.1 Empirical Results - Reliability:

The results of the normalized reliability comparison are listed in appendix 2, tables A2.1 through A2.25. Each table represents the *ARSN* computations for specific values of *n* and *e*. The reliability of each edge was randomly generated once from each of the four reliability ranges ([100%...0%], [100%...50%], [100%..75%] and [100%...95%]). This results in four individual graphs with the same physical structure. Each table shows the average of the *normalized reliability* for values of *K* from 1 to 5. As well, for each reliability and *K* combination, the standard deviation, minimum reliability and maximum reliability are recorded.

As an initial observation, the reliability *ARSN*, for all values of *e*, *n* and *K* tested, is remarkably high. Even in the worst case, (table A2.1, $n = 20$, $e = 38$) when K=1, the *ARSN* has an average normalized reliability of approximately 82%. When K=2, the worst case average normalized reliability is greater than 91%. In the best case, (table A2.25, $n = 100$, $e = 4950$) with $K = 1$, the average normalized reliability is almost 100%. These results clearly indicate that, on average, the reliability of the *ARSN* is not significantly compromised and therefore is a suitable substitute for the original network. Yet, there are specific instances in which the reliability of the *ARSN* is significantly less then the original (table A2.1, $n = 20$, $e = 38$, K= 1, Range = [100%..0%], Min. Rel. = 51.25). Clearly certain types of networks lend themselves to the construction of a good *ARSN*. We briefly examine some of the contributing factors.

The predominant factor affecting the reliability of the *ARSN* is the value of *K*. As shown in each of the tables of appendix 2, the normalized reliability increases as the value of *K* increases , regardless of order, size or reliability range used. This result is obvious if we consider that increasing *K*, increases the number of edges in the *ARSN*. Choosing an appropriate value of *K*, therefore, is a matter of weighing the benefits of a network of small size versus a more reliable network. At one extreme we choose a value of *K*= 1. This *ARSN* has the least edges and in fact it is a spanning tree (when connected). It has the lowest reliability for all values of *K*, although the reliability is still quite high on average. The major drawback of using *K* = 1 is the high probability that the *ARSN* is disconnected (refer to table 4.2). In all but a few instances, the *ARSN* was disconnected and had to be repaired by the reconnection algorithm (section 4.5.2). Alternatively, when *K*=5, the resulting structure is more reliable and, for all practical purposes, never disconnected. Because the structure is assumed to be connected with high probability, we can forgo the check for connectedness (section 4.5.1) which may save considerable computational resources. The disadvantage of this structure is that it contains significantly more edges than the *K*=1 *ARSN* and will be identical to the original network if each vertex has edge degree 5 or less. A reasonable alternative is the *ARSN* when *K*=2 . This structure is more reliable than the *K*=1 *ARSN* and in many instances is "almost" as reliable as the *K*=5 *ARSN*. It has far fewer edges than the *K*=5 *ARSN* and less than twice the edges of the *K*=1 *ARSN*. The *K*=2 *ARSN* is likely to be connected with a probability > 99% for values of n ≥ 20 (see table 4.4) and as a result the check for connectedness may be unnecessary.

It can be seen from these results that a significant factor in the construction of a "good" *ARSN* is the range used to compute the reliability of the edges. In each table ,as the edge reliability range decreases the overall reliability increases, as we would expect. The *ARSN* and the original network differ in reliability only when at least one edge of the original network is not included in the *ARSN*. In this case, the most reliable path from at least one vertex to another will not be optimum. When the range of edge reliability is small, the alternate path is likely to be almost as reliable as the optimum. As the range increases we would expect the reliability of the non-optimal path to decrease.

Another significant factor is network *size*. From the tables in appendix 2, it can be seen that when the size of the network is increased the normalized reliability increases almost without exception, for smaller values of *K*. From a combinatorial point of view this seems unlikely since as more edges are added the greater the number of paths that exist in the network. Therefore, there is less chance of locally choosing the most reliable paths. We can offer two possible reasons for this phenomenon. First, the complete network is the densest and therefore the most reliable. In this type of network it is often better to use a multiple edge path which uses a few very reliable edges rather than an unreliable local edge. Therefore, as the network approaches completeness it becomes increasingly likely that a few highly reliable edges dominate the most reliable paths between pairs of vertices. Because our algorithm is greedy, it is almost certain that these edges will be chosen. Second, with a network of small size there can be great variance in the physical structure of the network. Some combinations of edges and edge reliability may produce a pathological structure which can not be reduced reliably by a locally based algorithm. As the number of edges in the network increases, the number of physical variations is reduced and consequently this effect is diminished.

The effect of *order* on normalized reliability of an *ARSN* is almost negligible. From the tables in appendix 2, when the percentage of edges is constant there is little disparity in normalized reliability when *n* is varied.

5.3.2 Empirical Results - Path Length:

The results of the *ARSN* most reliable path length comparisons, for values of K = 1...5, are shown in tables 5.1 (maximum path length) and 5.2 (average path length) . For comparison purposes, only complete networks of order n, for n = 20, 40, 60, 80 and 100, are used. The complete network represents the worst case for path length since there are the greatest number of individual paths available. Accordingly, these results will apply to other networks of smaller size. It was found that there was no statistical difference in the results obtained between graphs with different edge reliability ranges and consequently these are not considered.

ARSN	n = 20	n = 40	n = 60	n = 80	n = 100
complete	7.8	11.2	13.8	15.6	18.1
K=1	11.2	17.5	21.7	29.5	38.2
K=2	9.1	13.0	16.0	18.3	21.4
K=3	8.2	11.4	14.2	15.8	18.6
K=4	7.8	11.4	13.8	15.6	18.1
K=5	7.8	11.2	13.8	15.6	18.1

Table 5.1: Maximum Path Length of the Most Reliable Paths of an *ARSN* for values of *K*=1..5, Constructed from a Complete Graph of order *n*.

ARSN	n = 20	n = 40	n = 60	n = 80	n = 100
complete	2.7	3.4	3.9	4.6	5.2
K=1	4.7	6.1	7.2	8.0	9.3
K=2	3.4	4.6	5.5	6.4	7.1
K=3	2.9	3.7	4.4	5.2	5.7
K=4	2.8	3.5	4.1	4.8	5.3
K=5	2.7	3.4	4.0	4.7	5.2

Table 5.2: Average Path Length of the Most Reliable Paths of an *ARSN* for values of *K*=1..5, Constructed from a Complete Graph of order *n*.

From the results in the preceding tables, it can be seen that as the value of *K* increases, both the maximum and average most reliable path length are reduced and become virtually equivalent to the most reliable path lengths of the complete graph, when *K*>4. Notice however, the sharp increase in both maximum and average path lengths when *K* = 1. Given that the *K*=1 *ARSN* structure is a spanning tree, this result is not surprising. For *K*>1, both the maximum and average path lengths are not significantly worse than for the complete graph and are therefore acceptable.

5.4.3 Empirical Results - Graph Size:

The results of the *ARSN* graph size comparisons for values of *K* = 1...5 are shown in tables 5.3. Both the actual number of edges and the percentage of possible edges are shown. As in the pervious case, only complete networks of order n, for *n* = 20, 40, 60, 80 and 100, are used. For this result, the complete network is used because it is the most dense. Networks of smaller size will show virtually identical results for values of *K* less than the smallest edge degree of any vertex in the network. Accordingly, these results will apply to other networks of smaller size. Again, it was found that there was no statistical difference in the results obtained between graphs with different edge reliability ranges and consequently these are not considered.

ARSN	n = 20	n = 40	n = 60	n = 80	n = 100
complete	190 (100%)	780 (100%)	1770 (100%)	3160 (100%)	4950 (100%)
K=1	19.0 (10.0%)	39.0 (5.0%)	59.0 (3.3%)	79.0 (2.5%)	99.0 (2.0%)
K=2	26.3 (13.8%)	55.7 (7.1%)	80.4 (4.6%)	109.9 (3.5%)	137.0 (2.7%)
K=3	38.1 (20.0%)	78.1 (10.0%)	115.0 (6.5%)	157.0 (4.9%)	195.9 (3.9%)
K=4	49.6 (26.1%)	101.4 (13.0%)	150.2 (8.5%)	202.4 (6.4%)	253.3 (5.1%)
K=5	60.8 (32.0%)	123.0 (15.8%)	185.9 (10.5%)	247.8 (7.8%)	309.2 (6.2%)

Table 5.3 : Graph Size for *ARSN* of Various Values of *K*, Constructed from Complete Graphs of Order *n*.

From the results in table 5.3 it can be seen that the *ARSN* significantly reduces the size of a dense network for all values of *n*. The result is more dramatic as *n* increases. Notice that for values of *K*=1 the *ARSN* is a spanning tree and has exactly *n*-1 edges. Hence, the *K*=1 *ARSN* is by far the most economical structure to execute distributed computations on. Surprisingly however, the *K*=2 *ARSN*, averaged over all values of *n*, has only 38% more edges then the *K*=1 structure. Since the *K*=2 *ARSN* is more likely to be connected, (table 4.2) it is a much more robust structure than the *K*=1 *ARSN*.

5.4 Summary

In this chapter we have empirically evaluated the *ARSN* on the basis of reliability, path length and network size. Overall, the results are very encouraging. Given a sufficiently dense network, it is possible to reduce a network by choosing only the *K* most reliable edge for each vertex without substantially reducing the reliability of the network. Although different extremes of *K* produced structures with different advantages and disadvantages the best value seems to be *K*= 2. The *K*=2 *ARSN* is generally very reliable, has small network *size* and path lengths and most important, is virtually always connected. This last fact enables us to create the *ARSN* without having to check for connectedness.

Chapter 6

Conclusion and Future Directions

6.1 Conclusion.

Throughout this thesis our primary focus has been to provide reliable message transmission over an unreliable network. For this purpose, we begin by introducing a new distributed computing model. Unlike previous models, the new model assumes that the reliability of each edge is known prior to execution. Reliability knowledge is the first step in achieving reliable transmission since it allows unreliable edges to be avoided when a more reliable alternative exists.

In chapter 3 we considered network reliability. Classical measures of network reliability, such as connectivity and resilience are known to be intractable in the general case. For this reason, we propose a new definition of network reliability. The *Average All Pairs* measure is based on the reliability of the most reliable path between all processors in the network. The new measure is unique for two reasons. First, unlike its predecessors, it can be computed exactly in $O(n^3)$ time. Second, due to the similarity between the reliability definition and the way messages are passed in a network, the measure is ideally suited to distributed computing environments.

In chapter 4 we examined the issue of message transmission over an unreliable network. Typically, network unreliability is solved by making the distributed algorithms fault-tolerant. Fault-tolerant mechanisms typically make the algorithm more complex and only solve the problem in limited circumstances. As a solution to unreliable transmission, we propose constructing a *Reliable Subnetwork*. The reliable subnetwork is a reduction in the size of the network based on the *Average All Pairs* reliability measure. Aside from retaining the reliability of the original network, the primary advantage of this structure stems from the reduced number of edges. Because we generally only keep the most reliable edges, any distributed algorithm executing on the reliable subnetwork will encounter fewer edge failures and experience a reduced message complexity.

The exact reliable subnetwork algorithm unfortunately incurs a large message complexity. For this reason, we construct an approximation to the reliable subnetwork. The approximate algorithm is based on a greedy technique whereby processors combine their K most reliable local edges to form a global structure. The approximate solution can be constructed with O(e) messages. Unfortunately, the approximate structure does not retain the reliability of the original network nor does it ensure connectedness. By analytical and experimental means, we are able to show, with high probability, when $K>1$, the approximate reliable subnetwork is connected. When $K=1$ it is almost never connected. To ensure that the structure is connected, a new algorithm is outlined which guarantees connectedness. The algorithm checks for connectedness by constructing a spanning tree and checking for *out* edges. A new spanning tree algorithm is proposed, in which larger fragments never wait for smaller fragments. A connected approximate reliable subnetwork can be constructed with $O(n\log_2 n + e)$ messages.

In chapter 5 we again examined the approximate reliable subnetwork. As previously stated, the approximate solution does not have the same reliability as the original network. We evaluate the approximation algorithm experimentally by computing the reliability of approximate reliable subnetworks constructed from randomly generated graphs. In terms of reliability, it was shown that the most significant factors affecting the reliability are the size of K (obviously) and the *size* (number of edges) in the network. Network *order* does not seem to be an important factor. From the results it can be shown, that the reliability of approximate reliable subnetworks is not significantly less than the original network. We also evaluate the approximate structure in terms of maximum path length and average path length. Again, these were shown not to be significantly greater than in the original network.

6.2 Areas of Further Research.

With any type of research there are avenues which remain unexplored for reasons of time, complexity or both. In this section we briefly outline a few areas which may provide interesting avenues of research.

First, the network is unreliable and as such, even typically reliable edges may fail from time to time. We have not discussed the implications of an edge failing either during or after the construction of the reliable subnetwork. If the edge fails during the construction phase (*i.e.*, before the distributed algorithm has begun execution) the solution is obvious. Since the failed edge was only part of the local solution, it can be replaced with the next most reliable incident edge. When the failure occurs after the construction is completed and a distributed algorithm, or even the algorithm to check for connectedness, has begun, the solution is not so simple. A distributed algorithm may locally label certain edges, such as *member* of a spanning tree, and therefore a simple replacement scheme is not possible. Unfortunately in this situation, it is not possible to completely dispose of fault-tolerant mechanisms. One possible solution is to assume an *eventually connected* network, as described by Awerbuck and Even [AE84], in which we basically wait until the edge becomes operational again. If we assume that the reliable subnetwork will rarely fail, then a viable alternative is to start the computation again when a failure is encountered and the structure has been reconstructed.

A related question concerns how the reliable subnetwork should be repaired once a member has failed. A possible solution is to simply ignore the failure, since no restrictions on the number of edges for the reliable subnetwork exist. Eventually, however, every edge in the subnetwork is likely to fail. Another solution is to replace the failed edge. This solution retains the number of edges of the structure, but may cause the reliability of the structure to degrade over time.

Finally, we have introduced a new distributed computing model with assumed edge reliability. We have chosen to use this information by constructing the reliable subnetwork. In reality, this research has only scratched the surface. Many other solutions and applications are possible.

References

[AAG87] Afek,Y., Awerbuch, B. & Gafni, E., "Applying Static Network Protocols to Dynamic Networks", Proc. of the 28th IEEE Ann. Sym. on Foundations of Comp. Sci. , 1987.

[AG88] Afek,Y., & Gafni, E., "End-to-End Communication in Unreliable Networks", Proc. 7th Ann. ACM Symp. Princ. Distr. Comp., 1988, pp. 131-148.

[AHU74] Aho, A., Hopcroft, J. & Ullman, J. " The Design and Analysis of Computer Algorithms", Addison-Wesley, Reading, Mass., 1974.

[A85] Awerbuch, B., "Complexity of Network Synchronization", JACM, Vol. 32, No. 4, Oct 1985.

[AE84] Awerbuch, B. & Even, S., "Efficient and Reliable Broadcast is Achievable in an Eventually Connected Network", Proc. 3th Ann. ACM Symp. Princ. Distr. Comp.,1984, 278-281.

[AE86] Awerbuch, B. & Even, S., "Reliable Broadcast Protocols in Unreliable Networks", Networks, Vol. 16, 1986.

[AMS89] Awerbuch,B., Mansour,Y., & Shavit, N., "Polynomial End-to-End Communication", Tech Report TM-391, MIT, Cambridge, Mass., 1989.

[B79] Ball, Michael, O., "Computing Network Reliability", Operations Research, Vol 27, No. 4, July-August 1979, pp. 823-837.

[BKWZ87] Bar-Yehuda, R., Kutten, S., Wolfstahl, Y. and Zaks, S., "Making Distributed Spanning Tree Algorithms Fault-Resilient", Proc of 4th Symn. Theor. Ascpects of Comp. Sci., 1987, Lecture Notes in Comp. Sci., Springer-Verlag.

[BC86] Brecht, T. & Colbourn, C., "Improving Reliability Bounds in Computer Networks", Networks, Vol 16, 1986.

[BDFS84] Broder,A., Dolev,D., Fischer, M. & Simons, B.,"Efficient Fault-Tolerant Routings in Networks", Jour. of ACM, 1984, pp. 536-541.

[C79] Chang, E., "An Introduction to Echo Algorithms", Proc. 1st Int. Conf. on Distributed Computing. 1979.

[C82] Chen, C., "A Distributed Algorithm for Shortest Paths", IEEE Trans. on Computers C-31, No. 9, Sept 1982.

[C89] Colbourn, Charles.J., "Combinatorial Aspects of Network Reliability", NATO ARN on Topological Network Design, June 1989.

[C85] Colbourn, Charles.J., " Optimum Communication Spanning Trees in Series-Parallel Networks", SIAM J. of Computing, 1985, pp. 915-925.

[C86] Colbourn, Charles.J., " The Combinitorics of Network Reliability", Oxford University Press, Oxford and New York, 1986.

[C87] Colbourn, Charles.J., " Network Resilience", SIAM J. of Discrete Methods, 8, July 1987, 404-409.

[CH88] Colbourn, C. & Harms, D., "Bounding All-Terminal Reliability in Networks", Networks, Vol 18, 1988.

[CR87] Cidon,Israel & Rom, Raphael, "Failsafe End-to-End Protocols in Computer Networks with Changing Topology", IEEE Trans. on Communications, Vol Com-35, No. 4, April 1987, pp 410-413.

[DHSS84] Dolev,D., Halpern,J., Simons,B. & Strong,R., " A New Look at Fault-Tolerant Routings", Proc. 16th Ann. ACM Symp. on Theory of Comp., 1984, pp. 526-535.

[FF70] Frank, F. & Frisch, I., "Analysis and Design of Survivable Networks", IEEE Trans. on Communications, Vol COM-18, 1970.

[FC74] Frank, F. & Chou, W., "Network properties of the ARPA network", Networks, Vol 4, 1974.

[F85] Frederickson, G., "A Distributed Shortest Path Algorithm for a Planar Network", CSD TR527, Dept. of Computer Sciences, Purdue University, 1985.

[GHS83] Gallager,R.,G., Humblet, P., A., and Spira, P.,M., "Distributed algorithms for minimum-weight spanning trees", ACM TOPLAS 5:66-77, 1983.

[GJ79] Garey, M. & Johnson, D., " Computers and Intractability: A Guide to the Theory of NP-Completeness", W.H. Freeman and Company, N.Y. 1979.

[G82] Garcia-Molina, H., "Elections in a Distributed Computing System" IEEE Transactions on Computers, Vol C-31, No. 1, Jan 1982.

[GS84] Goldberg, Zvi & Shacham, Nachum, "A Distributed Network Protocol with limited Span", Report No. SRI/TSC 83-1, SRI International, Menlo Park, Ca., 1983.

[H83] Harms, D., "An Investigation Into Bounds on Network Reliability", MSc Thesis, Dept. of Comp. Sci., Univ. of Sask., 1983.

[JM82] Jaffe, J. & Moss, F., " A Responsive Distributed Routing Algorithm for Computer Networks", IEEE Trans. on Comm. Vol. COM-30 No. 7, 1982.

[JLR78] Johnson, D., Lenstra, J. and Rinnooy Kan, A. "The Complexity of the Network Design Problem", Networks, Vol 8, 1978, pp. 278-285.

[K69] Knuth, D., "The Art of Computer Programing - Vol. 2, Seminumerical Algorithms", Addison-Wesley Press, Reading Mass.,1969.

[KWZ86] S. Kutten, Y. Wolfstahl and S. Zaks, Optimal distributed election in complete networks, Tech Report #430, Computer Sci. Dept. Technion Israel, (Aug 1986).

[LTC89] Lakshmanan, K., Thulasiraman, K. & Comeau, M., "An Efficient Distributed Protocol for Finding Shortest Paths in Networks with Negative Weights". IEEE Trans. on Software Engineering, Vol 18, No. 5. May 1989.

[LP86] Liestman, A. & Peters, J., "Distributed Algorithms", Tech. Rep. TR 86-10, School of Computing Science, SFU, 1986.

[NW78] Nijenhuis, A & Wilf, H. S.. "Combinatorial Algorithms for Computers and Calculators", Academic Press, 1978.

[P78] Palmer, E. "The Enumeration of Graphs", Selected Topics in Graph Theory, Academic Press, 1978.

[P85] Palmer, E. "Graphical Evolution, An Introduction to the Theory of Random Graphs", Wiley, New York, 1985.

[PFTV88] Press,W. Flannery, B. Teukolsky S. & Vetterling, W., "Numerical Recipes in C - The Art of Scientific Computing", Cambridge Univ. Press, NY 1988.

[S83] Segall,A., "Distributed Network Protocols", IEEE Trans. on Information Theory, IT-29(1), January 1983.

[S85] Stallings, W., "Data and Computer Communications", Macmillian Publishing, 1985.

[SA81] Soi, I. & Aggarwal,K., "Reliability Indices for Topological Design of Computer Communication Networks", IEEE Trans. on Reliability, Vol R-30, No. 5, Dec. 1981.

[T81] Tanenbaum, Andrew, S., "Computer Networks", Prentice-Hall, Englewood Cliffs, N.J., 1981.

[T80] Toueg, S., "An All-Pairs Shortest Path Distributed Algorithm", Tech. Report RC 8327, IBM Thomas J. Watson Research Center, 1980.

[V83] Vishkin, V., "An Efficient Distributed Orientation Algorithm", Proc. 9th IEEE Conf. on Distributed Computer Systems. 1984.

[W72] Wilkov, Robert,S., "Analysis and Design of Reliable Computer Networks", IEEE Trans. on Communications, Vol com-20, No. 3, June 1972.

Appendix 1

Evaluation of the Random Number Generator

Many of the result of this thesis are of direct consequence of experimental simulation : on randomly generated graphs. The graph generation program creates the graphs with the use of a random number generator which produces uniform deviates. Values generated by the random number generator are uniform in the sense that every number, in a specified range of the generator, is just as likely as any other to be generated. Randomly generated values are used both to select the edges of a graph, given a specific number of nodes, and to produce the reliability of each edge.

Unfortunately, random number generators do not produce truly random numbers but rather pseudo-random numbers based on a seed value and a mathematical calculation. Successive random numbers, although appearing random, are in fact related and in some cases may cause unwanted correlation between groups of numbers giving rise to inaccurate results[1]. It is therefore imperative that the random number generator behaves sufficiently random and is not a contributing factor in any of the experimental results. In light of this, the random number generator was subjected to the following three specific empirical tests to test its "randomness":

Chi-Square: The *chi-square* test is perhaps the most basic of all random generator tests. The test statistically compares actual generated sequences of random numbers with the

[1] Although correlation between random numbers is a serious problem it does not generally affect values used in a one dimensional order. For a more complete discussion of this see Press et al [Pres88].

predicted outcome based on the probability of generating a specific number or group of numbers. The statistic *V* is used to quantify the randomness of the sequence and is computed as sum of the squares of the differences between the observed number in the sequence and the expected numbers. A relatively high or low value of V generally indicates an insufficiently random sequence. In this test a sequence of 100,000 values was generated with 50 degrees of freedom.

Run test: The *run test* tests a sequence of numbers for the length of its monotone subsequences, either increasing or decreasing. The underlying idea is that for a sequence of a given length the number of monotonic runs of a specific length can be theoretically predicted. A sequence with too few or too many runs of a specific length indicate an insufficiently random sequence. As with the chi-square test, a statistic V is computed to evaluate the sequence. In this test a sequence of 100,000 values was generated and is evaluated as a chi-square test with five degrees of freedom.

Collision test: The *collision test* has been especially designed to detect the deficiencies of poor generators. In this test a small number of random values are generated in comparison to the number of predetermined intervals which cover the entire spectrum of possible random values. When a random number is generated it will usually fall into a previously empty interval, but if it falls into a non-empty interval then a collision has occurred. The number of collisions are counted and compared with the expected number. In this test 2^{20} intervals are used and 2^{14} random numbers are generated.

The previously outlined tests are not meant to be an exhaustive evaluation of a particular random number generator but instead to give a general indication of how random the sequences are which it produces. The reader who wants more information on the specific tests (and a number of others) should consult Knuth Vol 2 [K69].

For comparison purposes three uniform random number generators were tested. Generator *C* is an early unix system random number generator *rand* which is known to be insufficiently random in some cases. Generator *C* has a period of 2^{31}-1 and generates numbers in the range [0...2^{15}-1]. Generator *B* is the unix system random number generator *lrand48* which uses the well known linear congruential algorithm with 48-bit integer arithmetic. Generator *B* has a period of 2^{48}-1 and generates numbers in the range [0...2^{31}-1]. Generator *A*, which is used in all the emperical evaluations, is the system routine *lrand48* augmented with a *shuffle* algorithm, described in [K69], to improve its randomness. The shuffle algorithm essentially is used to break sequential correlations in random number generators.The algorithm works by initially filling a fixed size buffer with random values. On each call of the random number generator, the number generated by the system random number generator acts a hash key into the buffer and the value at that location is returned as the random value. A new random number is generated to refill the location.

Each of the three random number generators were tested fifteen times with different sets of data for each of the three specific tests in order to get a realistic evaluation of the algorithm. The results of all three tests are identified by one of *reject*, *suspect*, *almost suspect*, describing varying degrees of non-randomness in the test sequences or by *pass*, indicated by an empty box and meaning the sequence is sufficiently random. The individual random number generator is evaluated on the basis of the three tests. The generator passes or produces sequences which are sufficiently random if all three of the tests pass. A test, on the other hand, passes if the majority of the sequences pass the test.

The results of the tests, shown in figure A.1, indicate that all three random number generators pass the tests and are sufficiently random. However, it can be seen that generator *C* is somewhat questionable and the worst of the three. Generators *A* and *B* are both good random number generators with *A* being slightly better than *B*. Because all the graphs used in the experiments are created with generator *A*, it can be said that they are sufficiently random and the results obtained by their evaluation are valid.

		Experiment Number														
	Random Test	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
A	Chi-Square Test											○				
	Run Test					◍										
	Collision Test				○											
B	Chi-Square Test											○	◍			
	Run Test									○		◍				◍
	Collision Test				○											
C	Chi-Square Test			◍							○	●				◍
	Run Test	○					◍						○			
	Collision Test			◍			◍						●			

● **Reject** ◍ **Suspect** ○ **Almost Suspect**

Figure A.1: Results of random number generator tests

Appendix 2

Reliability Analysis of the Approximate Reliable Subnetwork

The tables of this section show the reliability analysis of the approximate reliable subnetwork for a number of randomly generated graphs. Graphs were generated by varying the order, size and edge reliability.

Edge Rel. Range		k = 1	k = 2	k = 3	k = 4	k = 5
100% - 0	Reliability	82.62	91.76	99.20	99.64	99.99
	Std. Dev.	15.32	6.87	1.63	0.14	0.01
	Min. Rel.	58.98	67.83	89.91	98..62	99.90
	Max. Rel.	94.74	100.0	100.0	100.0	100.0
100% - 50%	Reliability	89.29	94.70	99.38	99.81	99.99
	Std. Dev.	12.51	5.61	1.14	0.12	0.01
	Min. Rel.	69.62	78.28	93.01	98.99	99.90
	Max. Rel.	99.08	100.0	100.0	100.0	100.0
100% - 75%	Reliability	92.48	96.35	99.45	99.89	99.99
	Std. Dev.	6.51	2.98	0.79	0.12	0.01
	Min. Rel.	78.32	85.04	95.31	99.27	99.95
	Max. Rel.	100.0	100.0	100.0	100.0	100.0
100% - 95%	Reliability	95.36	98.98	99.81	99.97	99.99
	Std. Dev.	2.65	0.81	0.23	0.07	0.01
	Min. Rel.	86.44	94.99	98.34	99.57	99.70
	Max. Rel.	100.0	100.0	100.0	100.0	100.0

Table A2.1: Normalized Network Reliability N = 20, E = 38 (20 % of maximum)

Edge Rel. Range		k = 1	k = 2	k = 3	k = 4	k = 5
100% - 0	Reliability	89.74	94.53	99.35	99.90	99.98
	Std. Dev.	10.25	4.19	0.77	0.18	0.05
	Min. Rel.	68.52	81.52	95.32	98.42	99.62
	Max. Rel.	99.51	99.98	100.0	100.00	100.00
100% - 50%	Reliability	92.90	96.41	99.41	99.88	99.98
	Std. Dev.	5.60	2.96	0.61	0.15	0.05
	Min. Rel.	77.55	84.31	97.03	99.15	99.69
	Max. Rel.	99.88	99.98	100.0	100.0	100.0
100% - 75%	Reliability	95.32	97.79	99.61	99.92	99.99
	Std. Dev.	2.65	1.75	0.36	0.09	0.02
	Min. Rel.	84.08	88.86	98.16	99.54	99.86
	Max. Rel.	99.46	99.68	100.00	100.0	100.0
100% - 95%	Reliability	97.36	99.15	99.79	99.93	99.99
	Std. Dev.	1.56	0.42	0.13	0.05	0.01
	Min. Rel.	91.42	97.40	99.42	99.77	99.93
	Max. Rel.	99.71	99.81	99.97	100.0	100.0

Table A2.2: Normalized Network Reliability N = 20, E = 76 (40 % of maximum)

Edge Rel. Range		k = 1	k = 2	k = 3	k = 4	k = 5
100% - 0	Reliability	91.61	95.34	99.38	99.91	99.98
	Std. Dev.	10.53	5.13	0.70	0.19	0.08
	Min. Rel.	67.16	75.50	96.65	98.75	99.32
	Max. Rel.	99.81	100.0	100.0	100.0	100.0
100% - 50%	Reliability	93.89	97.54	99.57	99.89	99.97
	Std. Dev.	4.84	1.84	0.45	0.12	0.07
	Min. Rel.	79.21	91.31	97.49	99.34	99.49
	Max. Rel.	99.98	99.95	100.0	100.0	100.0
100% - 75%	Reliability	95.17	98.40	99.75	99.94	99.98
	Std. Dev.	1.47	1.45	0.21	0.07	0.03
	Min. Rel.	90.92	93.15	98.93	99.58	99.87
	Max. Rel.	99.16	99.91	100.0	100.0	100.0
100% - 95%	Reliability	96.32	98.74	99.77	99.94	99.97
	Std. Dev.	0.92	0.35	0.18	0.05	0.02
	Min. Rel.	95.30	97.82	99.20	99.71	99.85
	Max. Rel.	99.67	99.81	99.96	100.0	100.0

Table A2.3: Normalized Network Reliability N = 20, E = 114 (60 % of maximum)

Edge Rel. Range		k = 1	k = 2	k = 3	k = 4	k = 5
100% - 0	Reliability	93.14	96.07	99.39	99.90	99.97
	Std. Dev.	7.66	3.52	0.70	0.26	0.01
	Min. Rel.	77.51	86.96	95.62	98.60	99.29
	Max. Rel.	99.45	99.98	100.0	100.0	100.0
100% - 50%	Reliability	95.40	97.58	99.58	99.91	99.98
	Std. Dev.	3.25	1.87	0.40	0.09	0.03
	Min. Rel.	86.14	90.29	97.60	99.57	99.81
	Max. Rel.	99.32	99.81	100.0	100.0	100.0
100% - 75%	Reliability	96.52	98.81	99.77	99.94	99.98
	Std. Dev.	1.43	0.77	0.17	0.07	0.03
	Min. Rel.	91.46	96.63	98.86	99.44	99.70
	Max. Rel.	99.37	99.95	99.96	100.0	100.0
100% - 95%	Reliability	98.03	99.33	99.78	99.95	99.99
	Std. Dev.	0.72	0.26	0.09	0.05	0.02
	Min. Rel.	95.14	98.15	99.46	99.70	99.84
	Max. Rel.	98.59	99.74	99.93	99.99	100.0

Table A2.4: Normalized Network Reliability N = 20, E = 152 (80 % of maximum)

Edge Rel. Range		k = 1	k = 2	k = 3	k = 4	k = 5
100% - 0	Reliability	93.95	97.35	99.59	99.93	99.99
	Std. Dev.	2.93	1.81	0.48	0.13	0.03
	Min. Rel.	83.54	91.43	97.59	99.26	99.81
	Max. Rel.	99.29	99.96	100.0	100.0	100.0
100% - 50%	Reliability	95.67	98.71	99.79	99.96	99.99
	Std. Dev.	1.48	0.91	0.19	0.06	0.02
	Min. Rel.	85.12	95.32	99.18	99.72	99.82
	Max. Rel.	99.33	99.95	100.0	100.0	100.0
100% - 75%	Reliability	97.14	99.23	99.86	99.95	99.99
	Std. Dev.	0.85	0.51	0.14	0.06	0.02
	Min. Rel.	92.41	96.87	99.34	99.72	99.83
	Max. Rel.	99.56	99.95	100.0	100.0	100.0
100% - 95%	Reliability	98.68	99.86	99.98	99.99	99.99
	Std. Dev.	0.41	0.11	0.02	0.01	0.01
	Min. Rel.	96.92	99.38	99.90	99.96	99.98
	Max. Rel.	99.59	99.99	100.0	100.0	100.0

Table A2.5: Normalized Network Reliability N = 20, E = 190 (100 % of maximum)

Edge Rel. Range		k = 1	k = 2	k = 3	k = 4	k = 5
100% - 0	Reliability	83.21	91.45	99.22	99.65	99.94
	Std. Dev.	12.52	5.11	1.13	0.54	0.18
	Min. Rel.	65.31	73.93	93.87	96.18	98.74
	Max. Rel.	95.48	98.41	99.90	100.0	100.0
100% - 50%	Reliability	88.53	93.99	98.88	99.75	99.96
	Std. Dev.	6.44	2.93	0.76	0.25	0.05
	Min. Rel.	76.78	86.50	95.85	98.86	99.68
	Max. Rel.	97.28	99.06	99.90	100.0	100.0
100% - 75%	Reliability	92.06	96.85	99.38	99.85	99.97
	Std. Dev.	2.83	1.40	0.35	0.16	0.03
	Min. Rel.	92.46	92.44	98.09	99.29	99.75
	Max. Rel.	97.85	99.33	99.09	99.29	100.0
100% - 95%	Reliability	95.37	98.02	99.61	99.87	99.96
	Std. Dev.	1.29	0.45	0.15	0.05	0.03
	Min. Rel.	93.08	97.06	99.06	99.62	99.76
	Max. Rel.	97.29	99.36	99.87	99.96	99.99

Table A2.6: Normalized Network Reliability N = 40, E = 156 (20 % of maximum)

Edge Rel. Range		k = 1	k = 2	k = 3	k = 4	k = 5
100% - 0	Reliability	88.34	93.17	98.91	99.73	99.94
	Std. Dev.	6.81	2.67	0.81	0.35	0.10
	Min. Rel.	68.26	84.05	95.89	97.21	99.47
	Max. Rel.	95.67	98.85	99.87	100.0	100.0
100% - 50%	Reliability	92.63	96.44	99.30	99.83	99.96
	Std. Dev.	3.72	1.75	0.44	0.13	0.05
	Min. Rel.	84.00	88.17	97.66	99.33	99.72
	Max. Rel.	96.83	99.10	99.91	99.99	100.0
100% - 75%	Reliability	94.34	97.20	99.52	99.87	99.96
	Std. Dev.	1.88	0.77	0.25	0.07	0.03
	Min. Rel.	90.22	95.46	99.62	99.68	999.82
	Max. Rel.	97.26	99.18	99.94	99.98	100.0
100% - 95%	Reliability	95.76	98.05	99.65	99.88	99.95
	Std. Dev.	0.94	0.31	0.12	0.05	0.03
	Min. Rel.	93.61	97.93	99.21	99.70	99.85
	Max. Rel.	97.54	99.51	99.85	99.95	99.98

Table A2.7: Normalized Network Reliability N = 40, E = 312 (40 % of maximum)

Edge Rel. Range		k = 1	k = 2	k = 3	k = 4	k = 5
100% - 0	Reliability	90.68	95.51	99.16	99.80	99.95
	Std. Dev.	6.10	2.26	0.51	0.19	0.07
	Min. Rel.	78.01	86.21	97.35	98.80	99.63
	Max. Rel.	96.89	99.37	99.91	99.99	100.0
100% - 50%	Reliability	93.56	97.33	99.41	99.82	99.96
	Std. Dev.	3.64	1.15	0.30	0.11	0.04
	Min. Rel.	87.29	91.97	98.29	99.46	99.75
	Max. Rel.	96.71	99.29	99.93	99.99	100.0
100% - 75%	Reliability	94.99	98.31	99.58	99.87	99.95
	Std. Dev.	1.82	0.62	0.18	0.08	0.03
	Min. Rel.	92.27	96.25	99.15	99.61	99.81
	Max. Rel.	97.66	99.37	99.87	99.99	100.0
100% - 95%	Reliability	95.77	99.11	99.65	99.84	99.92
	Std. Dev.	0.77	0.21	0.08	0.05	0.03
	Min. Rel.	93.75	98.32	99.34	99.66	99.83
	Max. Rel.	97.73	99.46	99.81	99.91	99.97

Table A2.8: Normalized Network Reliability N = 40, E = 468 (60 % of maximum)

Edge Rel. Range		k = 1	k = 2	k = 3	k = 4	k = 5
100% - 0	Reliability	92.06	95.88	99.31	99.84	99.97
	Std. Dev.	3.37	1.90	0.37	0.13	0.04
	Min. Rel.	83.19	89.85	98.13	99.33	99.74
	Max. Rel.	96.98	99.15	99.94	99.99	100.0
100% - 50%	Reliability	94.28	97.75	99.55	99.88	99.96
	Std. Dev.	1.93	1.05	0.23	0.07	0.03
	Min. Rel.	90.41	93.71	98.88	99.60	99.86
	Max. Rel.	97.76	99.44	99.94	99.99	100.0
100% - 75%	Reliability	95.27	98.57	99.63	99.87	99.59
	Std. Dev.	1.24	0.57	0.16	0.03	0.01
	Min. Rel.	92.32	97.08	99.18	99.71	99.87
	Max. Rel.	97.44	99.48	99.93	99.97	99.99
100% - 95%	Reliability	96.86	99.01	99.68	99.86	99.95
	Std. Dev.	0.75	0.30	0.09	0.05	0.02
	Min. Rel.	93.36	97.59	99.44	99.73	99.84
	Max. Rel.	97.98	99.66	99.85	99.95	99.98

Table A2.9: Normalized Network Reliability N = 40, E = 624 (80 % of maximum)

Edge Rel. Range		k = 1	k = 2	k = 3	k = 4	k = 5
100% - 0	Reliability	92.85	97.75	99.62	99.91	99.98
	Std. Dev.	3.24	1.18	0.23	0.07	0.03
	Min. Rel.	87.91	93.75	98.57	99.61	99.81
	Max. Rel.	97.30	99.54	99.99	100.0	100.0
100% - 50%	Reliability	95.63	98.72	99.75	99.93	99.98
	Std. Dev.	1.89	0.62	0.17	0.05	0.02
	Min. Rel.	91.13	96.78	99.08	99.72	99.87
	Max. Rel.	97.68	99.79	99.99	100.0	100.0
100% - 75%	Reliability	96.41	99.41	99.88	99.97	99.97
	Std. Dev.	0.83	0.25	0.07	0.02	0.01
	Min. Rel.	91.53	98.71	99.67	99.88	99.97
	Max. Rel.	97.89	99.89	99.99	100.0	100.0
100% - 95%	Reliability	97.61	99.90	99.98	99.99	99.99
	Std. Dev.	0.44	0.06	0.02	0.01	0.00
	Min. Rel.	93.65	99.99	99.92	99.98	99.98
	Max. Rel.	98.23	99.99	100.0	100.0	100.0

Table A2.10: Normalized Network Reliability N = 40, E = 780 (100% of maximum)

Edge Rel. Range		k = 1	k = 2	k = 3	k = 4	k = 5
100% - 0	Reliability	82.81	91.05	98.17	99.64	99.93
	Std. Dev.	6.95	3.64	0.99	0.25	0.10
	Min. Rel.	69.87	77.58	95.21	98.64	98.80
	Max. Rel.	94.27	97.30	100.0	100.0	100.0
100% - 50%	Reliability	88.79	94.71	98.97	99.73	99.93
	Std. Dev.	4.02	1.87	0.46	0.18	0.06
	Min. Rel.	77.32	89.99	97.7	99.16	99.95
	Max. Rel.	95.84	98.85	99.78	99.99	100.0
100% - 75%	Reliability	91.88	96.71	99.31	99.80	99.95
	Std. Dev.	1.96	1.03	0.25	0.09	0.03
	Min. Rel.	83.94	93.77	98.73	99.54	99.85
	Max. Rel.	96.67	98.81	99.81	99.95	100.0
100% - 95%	Reliability	94.47	98.46	99.53	99.82	99.92
	Std. Dev.	1.12	0.43	0.12	0.05	0.03
	Min. Rel.	89.66	97.07	99.07	99.66	99.85
	Max. Rel.	96.78	99.26	99.72	99.91	99.85

Table A2.11: Normalized Network Reliability N = 60, E = 354 (20 % of maximum)

Edge Rel. Range		k = 1	k = 2	k = 3	k = 4	k = 5
100% - 0	Reliability	87.46	94.35	98.85	99.68	99.92
	Std. Dev.	4.57	2.12	0.54	0.23	0.09
	Min. Rel.	71.80	88.02	97.09	98.56	99.44
	Max. Rel.	94.25	98.71	99.77	99.97	99.99
100% - 50%	Reliability	91.22	96.48	99.31	99.80	99.94
	Std. Dev.	2.46	1.39	0.23	0.09	0.04
	Min. Rel.	83.47	88.40	98.61	99.55	99.78
	Max. Rel.	95.38	99.26	99.84	99.97	99.99
100% - 75%	Reliability	94.13	97.94	99.52	99.85	99.95
	Std. Dev.	1.49	0.69	0.15	0.06	0.02
	Min. Rel.	88.38	95.30	99.12	99.66	99.95
	Max. Rel.	96.30	99.02	99.86	99.97	99.99
100% - 95%	Reliability	94.94	98.91	99.57	99.79	99.89
	Std. Dev.	0.74	0.20	0.07	0.03	0.02
	Min. Rel.	91.06	98.06	99.33	99.70	99.83
	Max. Rel.	97.68	99.25	99.69	99.86	99.93

Table A2.12: Normalized Network Reliability N = 60, E = 708 (40 % of maximum)

Edge Rel. Range		k = 1	k = 2	k = 3	k = 4	k = 5
100% - 0	Reliability	90.11	95.95	99.18	99.78	99.94
	Std. Dev.	3.02	1.45	0.40	0.13	0.05
	Min. Rel.	81.49	91.67	97.50	99.44	99.78
	Max. Rel.	96.06	98.87	99.78	99.98	100.0
100% - 50%	Reliability	92.75	97.45	99.41	99.81	99.94
	Std. Dev.	1.49	0.84	0.23	0.08	0.03
	Min. Rel.	88.77	94.95	98.62	99.52	99.83
	Max. Rel.	96.03	98.87	99.82	99.95	99.99
100% - 75%	Reliability	93.99	98.32	99.54	99.83	99.93
	Std. Dev.	0.91	0.48	0.13	0.06	0.03
	Min. Rel.	90.44	96.73	99.07	99.60	99.85
	Max. Rel.	97.05	99.15	99.82	99.92	99.98
100% - 95%	Reliability	94.45	99.00	99.54	99.76	99.86
	Std. Dev.	0.55	0.12	0.05	0.04	0.02
	Min. Rel.	91.62	98.61	99.65	99.81	99.88
	Max. Rel.	95.96	99.23	99.85	99.91	99.96

Table A2.13: Normalized Network Reliability N = 60, E = 1062 (60 % of maximum)

Edge Rel. Range		k = 1	k = 2	k = 3	k = 4	k = 5
100% - 0	Reliability	91.64	96.76	99.30	99.81	99.94
	Std. Dev.	2.51	1.07	0.30	0.10	0.04
	Min. Rel.	85.12	93.75	98.21	99.51	99.83
	Max. Rel.	95.95	98.78	99.85	99.97	99.99
100% - 50%	Reliability	93.15	97.78	99.45	99.83	99.95
	Std. Dev.	1.44	0.67	0.15	0.06	0.03
	Min. Rel.	88.61	96.26	99.04	99.65	99.85
	Max. Rel.	96.51	99.14	99.75	99.94	99.99
100% - 75%	Reliability	94.26	98.53	99.57	99.83	99.93
	Std. Dev.	0.85	0.30	0.09	0.04	0.02
	Min. Rel.	90.76	97.78	99.36	99.71	99.87
	Max. Rel.	96.54	99.17	99.79	99.91	99.97
100% - 95%	Reliability	96.02	99.02	99.49	99.71	99.82
	Std. Dev.	0.39	0.08	0.04	0.03	0.02
	Min. Rel.	94.55	98.83	99.35	99.60	99.75
	Max. Rel.	98.14	99.19	99.61	99.78	99.89

Table A2.14: Normalized Network Reliability N = 60, E = 1416 (80 % of maximum)

Edge Rel. Range		k = 1	k = 2	k = 3	k = 4	k = 5
100% - 0	Reliability	92.41	98.09	99.66	99.15	99.98
	Std. Dev.	1.68	0.75	0.19	0.06	0.02
	Min. Rel.	86.26	95.85	98.68	99.68	99.98
	Max. Rel.	95.39	99.15	99.94	99.98	100.0
100% - 50%	Reliability	95.61	99.01	99.82	99.95	99.98
	Std. Dev.	0.87	0.44	0.11	0.04	0.02
	Min. Rel.	90.79	97.60	99.40	99.72	99.87
	Max. Rel.	97.42	99.79	99.96	99.99	100.0
100% - 75%	Reliability	96.23	99.57	99.92	99.98	99.99
	Std. Dev.	0.51	0.19	0.05	0.02	0.01
	Min. Rel.	90.92	99.07	99.78	99.90	99.97
	Max. Rel.	98.69	99.95	99.97	99.99	100.0
100% - 95%	Reliability	97.21	99.92	99.98	99.99	100.0
	Std. Dev.	0.32	0.03	0.01	0.01	0.0
	Min. Rel.	92.35	99.80	99.95	99.98	99.99
	Max. Rel.	99.14	99.99	99.99	100.0	100.0

Table A2.15: Normalized Network Reliability N = 60, E = 1770 (100 % of maximum)

Edge Rel. Range		k = 1	k = 2	k = 3	k = 4	k = 5
100% - 0	Reliability	82.51	91.95	98.41	99.62	99.91
	Std. Dev.	5.88	2.79	0.71	0.20	0.07
	Min. Rel.	71.66	83.23	96.23	98.91	99.69
	Max. Rel.	95.31	97.57	99.54	99.95	99.99
100% - 50%	Reliability	89.02	95.17	98.86	99.69	99.91
	Std. Dev.	3.19	1.53	0.42	0.13	0.06
	Min. Rel.	80.06	89.07	96.86	99.22	99.63
	Max. Rel.	94.49	97.32	99.52	99.93	99.99
100% - 75%	Reliability	92.82	96.91	99.32	99.80	99.93
	Std. Dev.	1.79	0.89	0.19	0.07	0.04
	Min. Rel.	85.19	94.16	98.78	99.59	99.82
	Max. Rel.	96.87	98.42	99.75	99.98	99.99
100% - 95%	Reliability	94.78	98.56	99.51	99.81	99.93
	Std. Dev.	1.03	0.31	0.09	0.04	0.02
	Min. Rel.	91.32	97.11	99.26	99.65	99.85
	Max. Rel.	97.57	99.26	99.71	99.86	99.95

Table A2.16: Normalized Network Reliability N = 80, E = 632 (20 % of maximum)

Edge Rel. Range		k = 1	k = 2	k = 3	k = 4	k = 5
100% - 0	Reliability	88.63	94.95	99.01	99.73	99.91
	Std. Dev.	3.12	1.42	0.37	0.13	0.06
	Min. Rel.	73.51	91.45	97.72	99.21	99.67
	Max. Rel.	95.26	97.89	99.59	99.95	99.99
100% - 50%	Reliability	92.23	96.87	99.32	99.78	99.93
	Std. Dev.	1.77	0.90	0.20	0.08	0.03
	Min. Rel.	85.43	94.58	98.45	99.41	99.83
	Max. Rel.	95.95	98.57	99.73	99.92	99.99
100% - 75%	Reliability	94.35	97.99	99.48	99.82	99.93
	Std. Dev.	1.08	0.47	0.11	0.05	0.03
	Min. Rel.	89.43	96.37	99.18	99.64	99.82
	Max. Rel.	97.06	98.91	99.73	99.91	99.99
100% - 95%	Reliability	95.34	98.87	99.49	99.85	99.96
	Std. Dev.	0.58	0.13	0.06	0.04	0.02
	Min. Rel.	92.35	98.39	99.30	99.61	99.87
	Max. Rel.	97.54	99.17	99.75	99.92	99.99

Table A2.17: Normalized Network Reliability N = 80, E = 1264 (40 % of maximum)

Edge Rel. Range		k = 1	k = 2	k = 3	k = 4	k = 5
100% - 0	Reliability	91.15	95.88	99.14	99.75	99.92
	Std. Dev.	2.58	1.13	0.25	0.09	0.05
	Min. Rel.	83.53	92.92	98.58	99.45	99.73
	Max. Rel.	97.15	98.72	99.70	99.91	99.99
100% - 50%	Reliability	93.68	97.64	99.41	99.81	99.93
	Std. Dev.	1.42	0.59	0.16	0.06	0.03
	Min. Rel.	89.93	95.77	98.95	99.55	99.84
	Max. Rel.	97.36	98.78	99.79	99.93	99.99
100% - 75%	Reliability	94.91	98.35	99.50	99.80	99.94
	Std. Dev.	0.76	0.38	0.11	0.05	0.03
	Min. Rel.	91.32	97.23	99.22	99.61	99.87
	Max. Rel.	97.07	99.03	99.81	99.94	99.99
100% - 95%	Reliability	95.31	98.89	99.52	99.84	99.95
	Std. Dev.	0.58	0.10	0.05	0.03	0.02
	Min. Rel.	92.37	98.58	99.26	99.65	99.89
	Max. Rel.	96.94	99.09	99.83	99.95	99.99

Table A2.18: Normalized Network Reliability N = 80, E = 1896 (60 % of maximum)

Edge Rel. Range		k = 1	k = 2	k = 3	k = 4	k = 5
100% - 0	Reliability	92.63	96.75	99.27	99.78	99.93
	Std. Dev.	2.65	1.09	0.22	0.09	0.03
	Min. Rel.	87.42	92.55	98.61	99.48	99.83
	Max. Rel.	96.71	98.55	99.75	99.95	99.99
100% - 50%	Reliability	94.56	97.93	99.46	99.80	99.93
	Std. Dev.	1.21	0.51	0.12	0.05	0.03
	Min. Rel.	89.39	96.26	99.08	99.64	99.85
	Max. Rel.	97.13	98.89	99.70	99.92	99.97
100% - 75%	Reliability	95.36	98.60	99.52	99.80	99.93
	Std. Dev.	0.73	0.30	0.10	0.05	0.02
	Min. Rel.	91.53	97.66	99.23	99.66	99.85
	Max. Rel.	97.51	99.22	99.73	99.88	99.96
100% - 95%	Reliability	97.59	98.99	99.53	99.81	99.95
	Std. Dev.	0.50	0.07	0.04	0.03	0.02
	Min. Rel.	96.20	98.71	99.23	99.68	99.86
	Max. Rel.	97.97	99.41	99.75	99.88	99.95

Table A2.19: Normalized Network Reliability N = 80, E = 2528 (80 % of maximum)

Edge Rel. Range		k = 1	k = 2	k = 3	k = 4	k = 5
100% - 0	Reliability	93.16	98.73	99.72	99.92	99.98
	Std. Dev.	1.35	0.53	0.13	0.05	0.02
	Min. Rel.	87.26	96.80	99.29	99.70	99.91
	Max. Rel.	96.36	99.51	99.94	100.0	100.0
100% - 50%	Reliability	94.75	99.20	99.85	99.96	99.99
	Std. Dev.	0.92	0.33	0.07	0.03	0.01
	Min. Rel.	91.76	97.69	99.64	99.87	99.92
	Max. Rel.	96.97	99.85	99.98	99.99	100.0
100% - 75%	Reliability	95.33	99.67	99.94	99.98	99.99
	Std. Dev.	0.56	0.15	0.03	0.01	0.01
	Min. Rel.	91.55	98.92	99.86	99.96	99.97
	Max. Rel.	97.69	99.92	99.99	100.0	100.0
100% - 95%	Reliability	95.54	99.85	99.97	99.99	99.99
	Std. Dev.	0.24	0.06	0.02	0.01	0.00
	Min. Rel.	93.54	99.66	99.93	99.96	99.99
	Max. Rel.	97.69	99.97	99.99	100.0	100.0

Table A2.20: Normalized Network Reliability N = 80, E = 3160 (100 % of maximum)

Edge Rel. Range		k = 1	k = 2	k = 3	k = 4	k = 5
100% - 0	Reliability	83.67	91.55	98.75	99.65	99.94
	Std. Dev.	5.63	2.31	0.62	0.14	0.06
	Min. Rel.	72.54	85.35	97.84	99.04	99.74
	Max. Rel.	95.92	97.59	99.61	99.96	99.99
100% - 50%	Reliability	89.20	95.61	98.94	99.72	99.94
	Std. Dev.	2.73	1.40	0.38	0.11	0.06
	Min. Rel.	79.61	89.68	97.15	99.35	99.65
	Max. Rel.	95.56	97.61	99.57	99.94	99.99
100% - 75%	Reliability	93.38	97.21	99.41	99.84	99.95
	Std. Dev.	1.53	0.73	0.15	0.05	0.03
	Min. Rel.	85.69	94.16	98.78	99.59	99.82
	Max. Rel.	97.37	98.42	99.75	99.98	99.99
100% - 95%	Reliability	96.37	98.69	99.67	99.84	99.96
	Std. Dev.	0.86	0.25	0.07	0.04	0.02
	Min. Rel.	91.25	97.11	99.26	99.65	99.85
	Max. Rel.	97.79	99.26	99.71	99.86	99.95

Table A2.21: Normalized Network Reliability N = 100, E = 990 (20 % of maximum)

Edge Rel. Range		k = 1	k = 2	k = 3	k = 4	k = 5
100% - 0	Reliability	89.47	95.14	99.21	99.84	99.92
	Std. Dev.	2.98	1.20	0.24	0.10	0.06
	Min. Rel.	73.84	93.84	97.89	99.31	99.68
	Max. Rel.	96.22	97.89	99.59	99.95	99.99
100% - 50%	Reliability	93.22	97.51	99.32	99.79	99.94
	Std. Dev.	1.54	0.82	0.12	0.07	0.03
	Min. Rel.	85.31	95.64	98.84	99.52	99.81
	Max. Rel.	97.25	98.93	99.73	99.93	99.99
100% - 75%	Reliability	95.03	98.28	99.55	99.83	99.95
	Std. Dev.	0.97	0.39	0.08	0.04	0.03
	Min. Rel.	90.30	97.01	99.28	99.71	99.84
	Max. Rel.	98.02	98.99	99.78	99.92	99.99
100% - 95%	Reliability	96.56	98.87	99.53	99.87	99.97
	Std. Dev.	0.53	0.11	0.05	0.03	0.02
	Min. Rel.	93.66	98.57	99.47	99.64	99.89
	Max. Rel.	97.54	99.25	99.87	99.95	99.99

Table A2.22: Normalized Network Reliability N = 100 E = 1980 (40 % of maximum)

Edge Rel. Range		k = 1	k = 2	k = 3	k = 4	k = 5
100% - 0	Reliability	92.05	96.71	99.24	99.75	99.92
	Std. Dev.	2.24	0.94	0.22	0.08	0.04
	Min. Rel.	83.60	94.87	98.92	99.54	99.81
	Max. Rel.	97.81	98.89	99.78	99.94	99.99
100% - 50%	Reliability	93.89	97.90	99.54	99.84	99.93
	Std. Dev.	1.42	0.47	0.13	0.05	0.03
	Min. Rel.	90.79	96.81	99.00	99.61	99.87
	Max. Rel.	98.02	98.87	99.80	99.94	99.99
100% - 75%	Reliability	95.06	98.63	99.61	99.88	99.95
	Std. Dev.	0.78	0.28	0.07	0.04	0.03
	Min. Rel.	92.46	98.54	99.34	99.84	99.89
	Max. Rel.	97.93	99.14	99.85	99.95	99.99
100% - 95%	Reliability	96.86	99.03	99.57	99.88	99.96
	Std. Dev.	0.43	0.09	0.05	0.03	0.02
	Min. Rel.	93.51	98.59	99.33	99.79	99.92
	Max. Rel.	97.97	99.09	99.83	99.95	99.99

Table A2.23: Normalized Network Reliability N = 100 E = 2970 (60 % of maximum)

Edge Rel. Range		k = 1	k = 2	k = 3	k = 4	k = 5
100% - 0	Reliability	93.62	97.19	99.33	99.84	99.93
	Std. Dev.	2.01	0.89	0.21	0.08	0.03
	Min. Rel.	85.12	94.51	98.84	99.50	99.85
	Max. Rel.	95.95	98.64	99.78	99.96	99.99
100% - 50%	Reliability	95.11	98.17	99.52	99.85	99.93
	Std. Dev.	0.92	0.45	0.10	0.05	0.03
	Min. Rel.	89.52	97.04	99.18	99.67	99.88
	Max. Rel.	97.29	98.97	99.74	99.96	99.99
100% - 75%	Reliability	96.16	98.68	99.66	99.86	99.94
	Std. Dev.	0.56	0.24	0.08	0.04	0.02
	Min. Rel.	92.74	98.06	99.29	99.74	99.90
	Max. Rel.	96.14	99.31	99.78	99.96	99.99
100% - 95%	Reliability	97.86	99.15	99.57	99.86	99.97
	Std. Dev.	0.39	0.07	0.04	0.03	0.01
	Min. Rel.	97.12	98.89	99.34	99.73	99.90
	Max. Rel.	98.07	99.48	99.81	99.99	99.99

Table A2.24: Normalized Network Reliability N = 100 , E = 3960 (80 % of maximum)

Edge Rel. Range		k = 1	k = 2	k = 3	k = 4	k = 5
100% - 0	Reliability	94.69	98.55	99.76	99.94	99.98
	Std. Dev.	1.33	0.49	0.11	0.04	0.02
	Min. Rel.	92.74	96.69	99.26	99.78	99.89
	Max. Rel.	97.54	99.46	99.94	99.99	100.0
100% - 50%	Reliability	95.85	99.35	99.89	99.97	99.99
	Std. Dev.	0.81	0.25	0.05	0.02	0.01
	Min. Rel.	92.48	98.34	99.71	99.92	99.97
	Max. Rel.	98.19	99.87	99.98	100.0	100.0
100% - 75%	Reliability	98.22	99.73	99.96	99.99	99.99
	Std. Dev.	0.40	0.11	0.02	0.01	0.00
	Min. Rel.	94.53	99.23	99.91	99.97	99.98
	Max. Rel.	99.11	99.93	100.0	100.0	100.0
100% - 95%	Reliability	99.49	99.95	99.99	99.99	100.0
	Std. Dev.	0.26	0.02	0.01	0.0	0.0
	Min. Rel.	98.38	99.87	99.98	99.99	99.99
	Max. Rel.	99.82	99.99	100.0	100.0	100.0

Table A2.25: Normalized Network Reliability $N = 100$, $E = 4950$ (100 % of maximum)