Network Layer Protocols
With Byzantine Robustness

by

Radia Perlman

S.B., Massachusetts Institute of Technology (1973)
S.M., Massachusetts Institute of Technology (1976)

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

Doctor of Philosophy

at the

Massachusetts Institute of Technology

August 1988

Signature of Author Signature redacted
Department of Electrical Engineering and Computer Science
24 August 1988

Signature redacted

Certified by
Dave Clark
Senior Research Scientist
Thesis Supervisor

Signature redacted

Accepted by
Arthur C. Smith
Chairman, Departmental Committee on Graduate Students

Network Layer Protocols With Byzantine Robustness

by

Radia Perlman

Submitted to the Department of
Electrical Engineering and Computer Science on August 24, 1988
in partial fulfillment of the requirements for the Degree of
Doctor of Philosophy in Computer Science

Abstract

The Network Layer of a network architecture is a distributed protocol that facilitates packet delivery across multiple hops. One of its chief functions is the calculation of routes throughout the network. Traditional Network Layer protocols have addressed robustness in the face of simple failures, i.e. nodes or links becoming inoperative. This thesis examines Network Layer protocol designs that are robust in the presence of Byzantine failures, i.e., nodes that through malice or malfunction exhibit arbitrary behavior such as corrupting, forging, or delaying routing protocol messages.

Acknowledgments

I'd like to thank my advisor, Dr. Dave Clark. He encouraged me to return to graduate school in the computer science department, and guided me through all the requirements, especially, of course, this thesis.

Prof. Robert Gallager's papers have been quite an inspiration for me through the years. His technical advise and feedback have been quite valuable, and I am grateful to have had him on my thesis committee.

I'd also like to thank the third member of my thesis committee, Prof. Nancy Lynch. I was introduced to the more theoretical side of the distributed algorithm field in her course, and her input into my thesis has helped in its clarity.

One member of Prof. Lynch's group, Alan Fekete, has been particularly helpful. He has read more drafts of my thesis than anyone else (possibly even me!), found bugs, made suggestions, and provided moral support and encouragement.

I'd especially like to thank Whit Diffie. He happened to be on the East Coast at just the right time to stop by, discuss my thesis, and suggest a very nice simplification, which allowed the complete removal of a clumsy mechanism.

Other people from whom I've received valuable technical feedback are Michael Speciner, George Varghese, and Lixia Zhang. Lixia deserves special thanks for putting up with me as an office mate for two years.

I'd also like to thank Digital Equipment Corporation for providing my financial support during graduate school. The members of the Network Architecture group at Digital created a stimulating and enjoyable work environment. In particular, I'd like to thank Tony Lauck, our group's leader and chief inspiration. I have learned almost everything I know about networks from him, and this thesis topic was his suggestion. I really appreciate his thoughtful and thorough review of this thesis.

And of course, I wish to thank my family, especially my children, Dawn and Ray. Without them life just wouldn't be as wonderful, or as much fun.

Contents

Chapter 1

Introduction

1.1 Overview

The purpose of this thesis is to present the design of a computer network that will be resistant to malfunctions due to such causes as hardware faults, software bugs, and network misconfiguration. A tradeoff is made between absolute robustness and practicality. The focus of this thesis is on one specific "layer" of a computer network architecture, the "Network Layer", which is responsible for delivery of packets across multiple hops. This is a particularly interesting layer because it is a protocol in which all nodes participate. Most network protocols are 2-party protocols, offering neighbor to neighbor communication, or end system to end system communication. The Network Layer, in contrast, requires cooperation of all nodes.

In this chapter, we introduce some of the basic concepts necessary for an appreciation of this problem.

1. We introduce the "Byzantine Generals Problem", because that was where the term "Byzantine fault" was first defined and used.

2. We explain the model of Network Layer used in this thesis.

3. We discuss the types of faults against which we wish to defend.

1.2 Byzantine Generals Problem

The term "Byzantine failure" was first defined in connection with the consensus problem, which is commonly known as the "Byzantine Generals Problem". Briefly, in the Byzantine Generals Problem, there are some number, n, of processors, known as "generals". They are fully connected, in the sense that any pair of generals can communicate over a private channel.

One distinguished general transmits a binary value to the other $n - 1$ generals; "attack", or "retreat".

Some number of generals may be faulty (traitorous), possibly including the distinguished general. A faulty general may behave in an arbitrary manner, such as sometimes failing to send messages, sending malformed messages, sending well formed messages with incorrect information, or sending contradictory messages to different nodes. Given that, the problem is for all the nonfaulty generals to agree on the same value, with the following correctness conditions.

termination All nonfaulty generals eventually reach a decision.

agreement All nonfaulty generals reach the same decision.

validity If the distinguished general is nonfaulty, the decision reached is the value transmitted by the distinguished general.

This problem has generated considerable interest in the computer science community, resulting in over a hundred published articles and several theses. A good overview of the state of the problem as of 1983 is given in [Fis]. Some of the major results in the subject are:

- No solution exists if more than 1/3 of the processors are faulty. [PSL]
- If t faults are to be tolerated, at least t rounds are required for any deterministic algorithm. [FL]
- In an asynchronous system, the problem is unsolvable [FLP].

On the face of it, designing a Network Layer to be robust in the face of Byzantine failures would seem far more complex than the Byzantine Generals' Problem, because:

- The network is not fully connected.
- The network is asynchronous.
- The decision a nonfaulty node must make is far more complex than a binary value — it involves a conclusion as to the topology of the network, decisions about allocations of finite resources, and route calculations.

This thesis will show that a Network Layer robust in the face of Byzantine faults can be built, and although it will be more expensive than traditional Network Layers, it will actually be practical to implement, at least in modest sized networks (thousands of nodes).

The differences between the Byzantine Generals problem and the Byzantine Network Layer problem that cause the Byzantine Network Layer problem to be tractable (despite the difficulties of the Byzantine Generals problem) are:

simultaneity We will show that the Network Layer can be designed to work even if the nonfaulty nodes do not simultaneously agree on the topology.

agreement We will show that the Network Layer can be designed so that agreement between nonfaulty processors is not necessary – a nonfaulty processor just needs to have an approximate idea of the correct topology in order for routing to work.

termination The Network Layer never terminates, or *commits* to a decision – decisions change over time.

cryptography We resort to cryptographic means to solve the problem. The Byzantine Generals Problem also becomes more tractable with the use of cryptography [LSP].

1.3 Public Key Cryptography

We make use of public key cryptography throughout this thesis. The concept of public key cryptography was introduced by Diffie and Hellman in [DH]. Briefly, it assumes the capability of a node X to choose a pair of functions, E and D, such that E and D are inverses, and that knowledge of E does not yield knowledge of D. The function E becomes X's "public key", whereas D is X's "private key".

Encryption of a packet destined for X can be accomplished by any node with knowledge of E. Assuming E is widely known (because it is a "public key"), any node can encrypt packets destined for X, but because E is assumed hard to invert, no node other than X can decrypt the packet (though X can easily decrypt by using D, its private key).

Digital "signatures" are possible with a public key scheme as well. The assumption is that X can construct a signature for some set of data, which is a function of the data and D. Any node, using E, can verify a signature, but cannot construct one.

1.4 OSI Reference Model

The ISO (International Standards Organization) has defined the OSI Reference Model as a framework for network architectures [ZIM]. Although most networks do not strictly conform to

the model, the OSI Reference Model is useful as a basis for discussion.

To make the network architecture problem tractable, functionality is broken into "layers", much like a large computer program being partitioned into subroutines. The layers are numbered from 1 to n. Layer k uses the services provided by layer $k-1$, and in turn adds functionality and provides services to layer $k+1$. Communication is logically between peer layers (i.e., layer k communicates with layer k in another node). However, except for the lowest layer, layer k must communicate with a peer layer k by presenting packets of information to layer $k-1$ for delivery, with the peer layer k receiving the packets directly from the layer $k-1$ in the destination node.

See [Zim] or [Tan] for a discussion of why ISO chose to layer networks, and how they chose where to place the layers. The networking community tends to agree with the functionality of the lower 4 layers, but disagrees about the function of higher layers. Layers above 4 are irrelevant to this thesis.

1 – **Physical** This layer delivers bits across one hop. It deals with such problems as modulation, clocking, voltage levels, and connector shapes.

2 – **Data Link** This layer is responsible for delivering packets of information across one hop. It deals with framing of packets, error detection, and link usage coordination on shared media.

3 – **Network** This layer is responsible for storing and forwarding packets across many hops, so that a mesh network (arbitrary multi-hop topology) will be logically fully connected. It deals with issues such as route calculation, allocation of finite resources, and packet fragmentation and reassembly.

4 – **Transport** This layer is responsible for adding end to end reliability. It deals with conversation setup, packet numbering for reordering and retransmission of lost packets, and addressing of processes within a node.

5-7 – **Session, Presentation, User** These are the layers that actually use the network. Typical protocols defined are file transfer, packet voice, mail, and remote terminals.

1.5 Network Layer Protocols

The purpose of the Network Layer is to extend the functionality in a mesh network so that the network appears to be fully connected. The Network Layer's job is to forward packets across

multiple hops so that any pair of nodes connected to the network may communicate with each other.

There are differing views of the Network Layer within the networking community. Some of the important areas of controversy are:

- Type of Service.

  The "connection-oriented" model of a Network Layer requires the Network Layer to reliably deliver a stream of packets from source to destination, without loss, duplication, corruption, or misordering. In this type of service, before packets can be transmitted between nodes A and B, one of A or B must first initiate a connection. Following connection completion, a stream of packets in each direction can be sent. And after the conversation is completed, a disconnection is executed. The Codex network implements this type of Network Layer [BG], and the X.25 standard [Ryb] specifies this type of Network Layer.

  The "datagram" model of a Network Layer requires only a "best-effort" service. Packets entrusted to the Network Layer should have a high probability of being delivered, but no guarantees are made against possibly lost, duplicated, or corrupted packets, and order of receipt is not guaranteed to be the same as order of transmittal. In this type of service, each packet is independently addressed, and self-contained. No "connect" or "disconnect" operations are performed. It is up to the Transport Layer, using the Network Layer datagram service, to provide connection-oriented services, if they are required for higher layer protocols. The ISO "Connectionless" Network Layer specifies this type of Network Layer.

- Degree of Self-Configuration.

  Some Network Layers rely heavily on manual maintenance of routes. The SNA Network Layer [SNA] is of this type. Complete routes to each destination are calculated and manually maintained at each source node.

  An alternate approach is to make the network as self-configuring as possible. The ideal network of this type would automatically assimilate new nodes as they are hooked in, and automatically reconfigure around failed components. The ARPANET, DNA, ISO connectionless Network Layer, and Codex designs all favor automatic operation.

In this thesis we will assume a datagram form of Network Layer, and will strive for minimal manual maintenance. The Network Layer model we use is that of a distributed protocol,

implemented on all the nodes in the network, that accepts as input (from the Transport Layer) packets with a Destination address. The Network Layer must then, with reasonable probability, deliver the packet to that Destination node. The Network Layer is just a datagram service, and as such is allowed to lose, misorder, duplicate, misdeliver, and even corrupt data packets, just so long as when a physical path exists between source and destination nodes, each packet independently has a "reasonable" probability (for instance, $> 1/2$) of successful delivery.

A Network Layer protocol fitting the above model often acts as follows.

- It discovers and disseminates topological information about the network.
- It calculates routes, based on the disseminated topological information.
- It forwards data packets, based on the calculated routes.

The two most popular classes of distributed routing algorithms that are widely used in networks are:

Distance Vector In this form of routing algorithm, the routing computation is done in a distributed fashion. Each node is responsible for maintaining a vector consisting of its distance to each destination. Each entry in the vector is computed either by default (the destination which is the node itself is zero distance away), or by minimizing based on neighbors' distances to that destination.

The original ARPANET algorithm [McQ74] was a Distance Vector scheme.

Link State In this form of routing algorithm, local topological information is ascertained by each node, and broadcast throughout the network, so that each node has a database giving complete topological details of the network.

The "New ARPANET algorithm" [MRR] is a Link State scheme.

In this thesis we use a modified link state scheme to achieve robustness. The distance vector form of algorithm is less promising as a candidate because in that form of routing there is much more dependence on cooperation. By its very definition, raw topological information is not disseminated – only the results of computation. There seems to be no way to confirm the validity of information, when the only information is the completed computation.

1.6 Levels of Robustness

Failures in a network are caused by faults involving nodes or links. A **"simple failure"** consists of a node or link becoming inoperative, and ceasing to function at all. **"Byzantine failures"**, on the other hand, are caused by nodes or links which continue to operate, but incorrectly. A node with a Byzantine failure may corrupt messages, forge messages, delay messages, or send conflicting messages to different nodes.

There are various levels of robustness that can be achieved:

Simple Robustness Traditional Network Layer algorithms achieve robustness in the face of "simple failures", i.e. nodes or links becoming inoperative. Some currently operating Network Layer protocols were designed assuming zero probability of incorrect behavior such as corrupted control messages or partially functioning nodes. In such a network, complex and tedious manual intervention may be required to restore the network to proper operation after such an event. [Ros].

Self Stabilization The next stage of robustness is an algorithm that is "self-stabilizing", which guarantees correct convergence, even with a history including Byzantine faults. Self-stabilizing algorithms were introduced in [Dij]. Self-stabilizing algorithms guarantee correctness once any malfunctioning nodes are disconnected from the network, but do not make any guarantees about behavior while a malfunctioning node is participating in the network. The original ARPANET algorithm was described and proven to be self-stabilizing, by this definition, in [Taj]. The "new" ARPANET algorithm, described in [MRR], does not have this property, as documented in [Ros]. Improvements to the ARPANET algorithm that would ensure the self-stabilizing property were suggested in [Pe].

Byzantine Detection Another form of robustness is an algorithm that may not operate correctly in the face of Byzantine failure, but in which the identity of the failed node can be easily discovered. Such an algorithm, especially if it also has the "self-stabilizing" property, is for practical reasons almost as good as an algorithm that has full Byzantine Robustness, since once the identity of the malfunctioning equipment is known, it can be disconnected from the network[1], and the network will continue normal operation.

[1] Disconnection of a malfunctioning node might require robust network management, for instance an out of band method of node elimination

With this combined form of robustness (self-stabilization plus Byzantine detection) even a malicious malfunctioning node can do little damage, since it must remain active to cause disruption, and its identity can be easily determined.

This form of robustness has not previously been attempted.

Byzantine Robustness An algorithm with Byzantine Robustness is defined to be one that exhibits correct behavior while arbitrarily malfunctioning nodes (nodes with Byzantine failures) participate in the network. In [Dol], an attempt to provide communication in an environment containing Byzantine failures was made. The observation was made in that paper that flooding would theoretically provide a foundation on which communication could be provided, but the unrealistic assumptions were made that:

1. Links have infinite capacity.
2. Nodes have infinite processing power.

Byzantine robustness in a realistic network model has not been previously attempted.

Note that Byzantine Robustness does not imply Byzantine detection. In other words, it is possible for a network with Byzantine robustness not to have the property of Byzantine detection. In practice, although it is very desirable for a network to continue operation in the presence of faults, it is also desirable for the network to detect faults and identify faulty components while the number of such faults is still small enough that the network can continue operation. The alternative is that the network would continue operation, oblivious to the existence of faults, until so many components were faulty that the network was nonfunctional, at which point the existence of faults would be obvious. Thus the ideal network would have both Byzantine Robustness and Byzantine Detection.

This thesis presents a basic design of a Network Layer with Byzantine Robustness, with some measure of Byzantine Detection as well. Variants of the design achieve slightly different versions of "robust". The exact form of robustness of each variant will be given with the definition of the scheme. Most forms achieve some form of robustness approximating the following definition.

If a path of currently nonfaulty processors and links exists between currently nonfaulty processors A and B, and no more than t faults (Byzantine or simple) simultaneously exist in the network, A and B will be able to communicate.

Note that we use the term "currently nonfaulty", to indicate that a faulty node should be able to participate correctly in a network after it has been repaired. In other words, no form of failure should prevent a node from participating correctly in the future, after it has been restored to correct operation.

1.7 Our Model of Network Layer

A Network Layer is a distributed algorithm implemented cooperatively at the *nodes* composing a network. Each node has a finite amount of processing power, a finite (and small) amount of non-volatile storage, and a finite amount of volatile storage. A node may fail and restart, in which case it loses any information in non-volatile storage. A failure of this type is considered to be common, and is not considered a Byzantine failure.

In addition to nodes, the network contains some number of *links*. Each link, L, has a pair of nodes, A and B, as its endpoints. The link enables bidirectional communication between A and B. Each link has finite capacity, and may lose, misorder, duplicate, or corrupt messages. There is an amount of time, γ, such that with high probability a message is delivered within γ, or not delivered at all. Links may also fail and restart, and such failures are not considered to be "Byzantine".

Since simple failures of links and nodes are considered part of "normal operation", the network must efficiently deal with such failures. Byzantine failures, in which nodes or links do not follow the protocol, are also allowed, but are considered more rare, and the network performance is allowed to degrade somewhat in the presence of Byzantine failures.

We choose this model because it is realistic. Network nodes are computers, with finite amounts of memory and processing capacity. Links in networks provide a fixed amount of bandwidth. Links in networks often have a Data Link Layer protocol implemented which attempts to make communication between the neighbor nodes reliable (no duplicates, no lost messages, no out of order messages). But we do not need to assume a reliable link, and it is safer not to rely on it since sometimes the underlying technology behind what appears to the network to be a "link" is an entire multi-hop network. Some Data Link Layer protocols, in particular those on LANs, provide only best effort service, and even those links with a "reliable" Data Link Layer protocol may lose, duplicate, or misorder messages when the link fails and restarts.

We make no provision in any of our designs for allocation of processing resources, since it is

possible to engineer a node so that its processing capacity is not a bottleneck. The processing capacity need only be large enough to keep up with the combined speed of the links. We assume therefore that each node has sufficient processing capacity. It is essential that a node have sufficient processing capacity, since if it is forced to discard packets before it has read them, there is no way to ensure fairness. Packets for a particular conversation might always be dropped.

The Network Layer accepts a packet from a higher layer "client", with a specified destination address. The Network Layer must, with high probability, deliver the packet to the specified destination. We define a Network Layer with "Byzantine robustness" to be one that, with high probability, delivers packets between a pair of nonfaulty processors if they are connected via a nonfaulty path (a path consisting of nonfaulty processors and nonfaulty links). We strive for this form of robustness in our design.

We explicitly retain the character of a "datagram" Network Layer, which is not required to be error-free. In particular:

- No guarantee is made that all messages sent by A for B will arrive at B; just that each packet independently has a high probability of reaching its destination. Node A can compensate for lost packets by using acknowledgments and retransmissions at a higher layer.

- No attempt by the Network Layer is made to keep conversations private. If privacy is necessary, encryption must be done at a higher layer.

- The Network Layer does not certify packets that it delivers. Although the robust form of Network Layer guarantees that each packet launched by a nonfaulty processor has a high probability of arriving undamaged, it does not guarantee that only those packets get delivered. For instance, some faulty processor C might send data packets with source address A, and those packets might get delivered in addition to the packets being delivered from genuine source A. Also, some faulty processor could corrupt the data in the packets from A, and the damaged copies might get delivered to B, again in addition to the undamaged packets.

  Again, it is up to A and B to have a higher level mechanism to extract correct data from the data that gets delivered. This is a solvable problem, using cryptography at the higher layer, given that the network does deliver the correct data.

- No guarantee is made about delivery of data generated by a faulty source.

1.8 Motivation

In a practical sense designing networks with Byzantine robustness is important, and is growing more so for the following reasons.

network size Networks are becoming very large. Hardware faults can cause arbitrary behavior, as evidenced by [Ros]. As the number of nodes grows, the probability increases that the network contains a malfunctioning node with just the right form of fault to disrupt the network.

Also, networks unfortunately do not come equipped with a "reset" button. The difficulty of restoring a disrupted network to proper operation grows with the size of the network. Restoring the ARPANET after the disruption documented in [Ros] involved reloading every node in the net with patched software, one at a time, and then after all nodes were successfully loaded with patched software designed to quiet the disturbance, each node needed to be reloaded again with the original program.

The ARPANET maintainers were very lucky in this incident. The ARPANET was reasonably small (hundreds of IMPs), all IMPs were identical (so that a single patched version of software was required), and the people maintaining the net were the same ones that designed the algorithm and wrote the code. Typical field service personnel are not the algorithm designers and network implementors.

standards As standards emerge, nodes will be implemented using different hardware by different organizations. Each organization has some probability of interpreting an ambiguous specification differently from other organizations, misinterpreting an unambiguous specification, or simply delivering an implementation with some incorrect code. In addition to increasing the probability of failure, having a network implemented with different software in different nodes makes the approach in [Ros] of quelling a disturbance much less viable. In order to reload every node with patched code in such a network, program sources for every implementation would need to be acquired, along with individuals familiar enough with each implementation to devise the correct patch. During this time, the network would remain nonfunctional.

distributed network management As different portions of the net are managed by different individuals, the probability increases that parameter settings or manual databases will be inconsistent across different portions of the net.

sabotage Additionally, sabotage is a real threat, especially as networks become more prevalent for financial or military institutions, or we become dependent on them for basic societal needs, such as transportation and communication. A network that may remain broken after a history of Byzantine failure (a network that lacks self-stabilization) is particularly vulnerable to sabotage, since an intruder can inject a few bad packets into the network and leave the scene before the problem is diagnosed. A network that is guaranteed to converge despite history, provided that no Byzantine nodes currently exist, is much less vulnerable to sabotage, because the intruder (or intruder's equipment) must remain active in order to keep the network from returning to proper operation. And once the equipment is found, simple disconnection of it will return the network to correct operation.

1.9 Current Network Robustness Designs

In this section, we will discuss current approaches towards building robustness into networks, together with ideas for enhancing the approaches for increased robustness.

1.9.1 Prevention of Byzantine Faults

The ARPANET requires the routers to checksum the Network Layer code periodically, to prevent a hardware or software error from modifying the code.

This sort of check can safeguard against many nonmalicious faults that typically occur in networks. However, note that only the code itself (not the databases which are dynamically changing) is tested with this form of checksum.

In Digital's DNA architecture, several robustness enhancing features were included in the design:

- All routing control information is generated by the source with a (non-cryptographic) checksum that is not modified by succeeding nodes. Thus it is very unlikely for a node to inadvertently modify another node's routing information.

- The distance vector routing scheme, upon which early phases of DNA were built, was guaranteed correct and self stabilizing (in the absence of active Byzantine faults) provided

that each node started with a correct idea of its own ID. The ID was thus stored in multiple places and the node was required to check that all locations agreed on the ID before generating a routing message.

- In later phases of DNA, when routing was built upon a Link State Routing scheme, and Link State Packets contained a (non-cryptographic) checksum, the nodes were required to prestore the computed value of the checksum following the initial portion of the Link State Packet (the portion that contained the source ID, and other static information) and use the prestored value when computing a checksum for the generated Link State Packet. In this way, if the node's image of its own ID became corrupted, it would probably no longer generate Link State Packets with correct checksum.

- DNA requires all routing data to be regenerated and rebroadcast periodically.

Note that all the above checks rely on the faulty node to be honest in its self-diagnosis. A node with a true Byzantine fault must be assumed capable of bypassing any sorts of self-checks. So while in practice these sorts of checks are sometimes useful, they give no guarantees for correct behavior for the sorts of faults we consider in this thesis.

1.9.2 Non-Automatic Networks

One approach to making more robust networks is to rely more heavily on manual maintenance. For instance, if routes are precalculated and manually maintained at each source node (as done in SNA), then the network cannot, by definition, miscalculate routes.

On the other hand, manual databases introduce another form of Byzantine failure, namely incorrect databases. Manual databases often are entered incorrectly at a node, entered incompatibly in a set of nodes, or become incorrect due to topological changes occurring without manual table updates. The ARPA EGP protocol [EGP] requires manually configured databases, and the ARPA internet has experienced many network-wide disruptions as the result of a single node's incorrect database.

The SNA architecture is not vulnerable to this form of network-wide disruption in the case of incorrect databases. If a source's routes are correct, they will work, regardless of the state of the databases at other nodes.

1.9.3 Data Corruption Prevention

A Data Link Layer checksum can be thought of as a safeguard against Byzantine failure of a single link, in that it protects against the link corrupting data. In practice, this sort of failure is so common that virtually all Data Link Layer protocols include a checksum.

However, data corruption does not occur only while a packet is traversing a link. Data can be corrupted at a relay between the time the relay verifies its correctness after being received (and strips off the checksum which had been placed by the Data Link Layer protocol on the receiving link), and the time a new Data Link Layer checksum is computed for transmission across the next link. Data Link Layer checksums cannot guard against this sort of data corruption.

Another mechanism which can introduce data corruption by the Network layer protocol is fragmentation and reassembly. In typical Network Layer fragmentation and reassembly, the source Network Layer writes a "unique packet identifier" (UPI) into each packet. This UPI enables the destination Network Layer to recognize fragments of the same datagram. If a UPI with the same source/destination pair is used within a packet lifetime, a fragment from an old packet might mistakenly be pieced in with data from a newer packet. UPI reuse can occur because the source has crashed and lost state, or because of wrap-around. Large packet lifetimes exist due to queuing delays and Data Link retransmissions of damaged packets. In the ARPA IP [IP] protocol and the ISO Connectionless Network Layer protocol, the UPI is only 16 bits long. At 56 KB transmission speed, typical for when ARPA IP was designed, and 256 byte packets, wraparound would occur in roughly 1/2 hour, making UPI wraparound without source failure a very low probability event. However, as technology pushes transmission speeds beyond 1 Mbit, UPI wraparound becomes more likely (unless the protocol evolves with the technology, and increases the length of the UPI field).

Thus data corruption, not detectable by Data Link Layer checksum, can and does occur in networks. The solution is an end-to-end checksum, computed at the source, and not modified until the packet reaches the destination. TCP [TCP] and the ISO Class 4 Transport [TP4] protocols utilize a Transport layer checksum to reduce vulnerability from Network layer packet corruption. In practice, Network layer packet corruption is a frequent event.

Suggested Modification – End to End Network Layer Checksum

The Transport Layer checksum does identify packets damaged due to Network Layer data corruption. However, it does not allow the faulty intermediate node to be isolated. A better approach is to have an end-to-end Network Layer checksum. The advantage of a Network Layer checksum is that intermediate nodes can check (but not modify) the checksum at each hop, and the node one hop closer to the destination than the faulty node will detect immediately that the packet has been damaged, and the culprit is identified.

It may be deemed too computationally expensive for intermediate nodes to check the checksum on every packet they forward. Thus if a Network Layer checksum is employed, it might be desirable to have a flag in the packet indicating whether intermediate nodes should check the checksum. In ordinary operation, the flag would be off, for efficiency. When some destination D starts detecting corrupted packets from some source S, D can notify S so that S can flag future packets to D for fault isolation.

However, if the efficiency option is chosen, the Network Layer might not isolate the faulty node, since the corrupting node might corrupt the flag, so that subsequent nodes would not check the checksum.

In practice, the Network Layer check would be useful, since a node that is corrupting packets is likely to sometimes leave the flag on (if the node is corrupting packets due to hardware problems and not malice). But there are options that will guarantee a Network Layer check's effectiveness:

- The efficiency option could be eliminated, forcing all nodes to compute the checksum on every packet. If the checksum is implemented in hardware, and engineered to keep up with the speed of the link, then it is completely practical to compute the checksum on every packet.

- The efficiency option could be selected on a per node basis instead of on a per packet basis. In other words, an intermediate node could be told to check all packets, or all packets destined to a particular destination, or all packets with a particular source/destination pair. In this way a source could use "binary search" on a route to find the culprit node. When source S is informed by destination D that the S-D packets are arriving corrupted, S can inform the midpoint of the route, node M, to start computing checksums on packets. If M reports no checksum problems, but packets still arrive damaged, then some node on

the path starting at M must be the culprit.

- Packets could contain two checksums, one for the header portion (including the "check data checksum" flag), and one for the data. Since the header is reasonably short, it would not be so much of a computational burden to require every node to check the header checksum on every packet.

The third scheme (two checksums) has the disadvantage that a node might be "selectively faulty", in that it might only corrupt data in packets in which the "check data checksum" flag was off. Thus a faulty node could force the network to use the more expensive form of packet forwarding at all times.

A Network Layer checksum need not include all of the Network Layer header. For simplicity, it might be desirable to exclude from the checksum any fields in the Network Layer header that must be modified by intermediate nodes, such as a hop count field. If protection of those fields is deemed vital, a checksum that allows incremental updating may be employed.

A simple checksum is sufficient to guard against non-malicious data corruption, for instance, due to hardware error. Cryptographic public key signatures can be utilized to guard against malicious threats.

None of the popular Network Layer protocols employ end to end Network Layer checksums. Thus all the popular protocol suites allow corrupted data to be discarded (by the Transport Layer), but they do not allow easy detection of the relay which is corrupting the data. The simple change of moving the end-to-end checksum down one layer introduces no extra overhead in terms of processing, memory, or header size. Utilizing the extra capability it presents (the ability to isolate the faulty node on a data corrupting path) does add a computational burden on forwarding nodes, but as pointed out above, this capability can be turned off when not needed, and does not increase computational burden if done in hardware.

1.9.4 Firewalls

Hierarchical Routing consists of partitioning a network into subnetworks. Routing within one subnetwork is done completely independently from routing within another subnetwork.

A separate layer of routing concerns itself with routing between subnetworks. This layer of routing is also usually independent of the routing within the subnetworks.

Hierarchical routing allows important savings in database overhead. It also protects the network by preventing malfunctions from spreading:

- If routing within one subnetwork becomes nonfunctional, it will not affect routing within a different subnetwork.

- If hierarchical routing is carefully designed, a malfunctioning subnetwork will not affect routing between subnets either (except that traffic to and from the nonfunctional subnetwork would not work).

- If the inter-subnetwork routing algorithm becomes nonfunctional, then inter-subnet routing would no longer work, of course, but again if the hierarchical routing scheme is designed carefully, failure of the inter-subnet routing should not affect intra-subnet routing.

1.9.5 Fault Isolation

Network designs often build mechanisms into the architecture for determining what is wrong when routes don't work. Various popular approaches are:

Network Management Network Management is a distributed service that collects data from all the network layers within a node, and communicates with other Network Management modules in order to exchange information. Using Network Management, it is possible to ascertain what routes have been computed, and the direction in which each node would forward a packet to a particular destination.

However, it is often the case that a route will not work, even though a correct route has been computed by the network. In cases such as that, further tools must be available to discover where and why packets are being lost.

Route Recording Route Recording is usually designed as an option to be selected on a per packet basis, in which nodes that forward the packet add their IDs to a route which is being collected in the packet's header.

This mechanism is not very effective at discovering why a route does not work, since data is only collected when the recorded route arrives at the destination.

Trace Packets This mechanism is also usually designed as an option on a per packet basis. When this option is selected, each node that forwards the packet additionally sends a packet back to the source, informing the source that the packet had proceeded up to that point.

1.9.6 Overcoming Failure

Some networks provide "source routing" as a backup mechanism to force route delivery when the routes computed by the Network Layer do not work. Source Routing allows a route to be placed in the packet header, and the network will route the packet according to the route in the header, instead of using a dynamically computed route.

Such a mechanism can often succeed in communicating across a malfunctioning network. The ARPA and ISO (connectionless) network layer protocols allow source routing for this purpose, but no guidance is given as to how the route would be obtained by the source. The source node must generate the source route, and often the source node is an "endnode" (ARPA terminology is "host", ISO terminology is "end system") that does not receive dynamic routing information. Therefore, the burden is presumably placed on manually maintained databases, or on a human to know the network topology and guess where the faults may lie.

In Chapter 3, we will recommend use of a form of source routing, though in our scheme a router (ARPA terminology is "IMP", ISO terminology is "Intermediate System"), which is a node participating in the routing algorithm (as opposed to a human, or an "endnode") computes the route to be placed in the header.

1.9.7 Legal Topologies

Some architectures place certain requirements on topologies. If these topological principles are violated, then the Network Layer will not work.

Examples of restrictions are:

1. The simplest form of "bridge", a node that performs forwarding at the Data Link Layer, is one that simply forwards every packet it sees on one link to all its other links. This form of routing will work provided that the topology is a tree. If the topology is not a tree, consequences can be disastrous since packets will not only loop forever, but proliferate at each hop.

2. Routers have finite capacity. If a network is larger than a router is configured to handle, then information must be selectively discarded, or the router must shut down entirely.

3. Hierarchical networks frequently require certain topological restrictions, such as that subnetworks be physically intact, that the net consisting of inter-subnet routers be physically intact, or that no links exist between intra-subnet routers in different subnetworks.

4. Networks which rely a great deal on manual databases require the physical topology and manual databases to agree. For instance, the ARPA EGP protocol [EGP] has been plagued by problems when manual databases are incorrect.

If a topology is legal, it is important that all subsets of that topology also be legal, if at all possible, since a network manager can configure a legal network, which can become an illegal topology because of simple node or link failure.

Approaches to dealing with topological restrictions (in order from weakest to strongest) are:

1. Do nothing – rely on network managers to carefully read and follow the documentation.

2. Have the machines recognize and report a misconfiguration, without attempting to control behavior while the net is misconfigured.

3. Have the machines prevent misconfiguration, for instance by having machines shut down links or shut down completely if misconfiguration is detected.

4. Design the architecture so that all topologies are legal.

The last approach has been accomplished in some instances. In the IEEE 802.1 bridge architecture, a spanning tree algorithm was adopted so that physical topology could be an arbitrary mesh, from which a spanning tree subset was automatically and continuously calculated by the bridges. [Pe2]

Various designs have been proposed to allow the requirement that subnets be physically intact to be eliminated. For instance, in [Pe4], a design is proposed in which level 2 routers detect a partitioned subnetwork, and send a copy of a packet destined for a partitioned subnetwork into each partition. In a contribution to ANSI [ANSI], the partitioned subnetworks are automatically "repaired" by incorporation of inter-subnetwork paths as intra-subnet links.

1.10 Overview of Approach

In Chapter 2, we present a design for robust packet delivery based on flooding. The key to accomplishing this is to structure databases in such a way that every source is guaranteed resources, and utilize authentication so that only the source can use its reserved resources. This is accomplished as follows.

1. Distribute public keys for every node in the network, through a "trusted node service", consisting of some number of nodes that broadcast a list of node/key pairs throughout the network.

2. Consider each node/key pair to be a separate entity, entitled to its own resources. In this way, if a Byzantine "trusted node" advertises a fictional node, or an incorrect public key for a network node, it will not interfere with resources reserved for the real network nodes.

3. Reserve one buffer for each source/key pair at each node.

4. Use a public key cryptographic signature, so that it is guaranteed that only a node with knowledge of the associated private key generated a particular packet.

5. Use a nonwrapping sequence number on each packet, so that earlier packets from a particular source will not compete for resources with its later data packets.

6. Scan the database of packets to be transmitted round-robin, so that the latest packet from every source will be output on the link.

In Chapter 3, we present a design for robust packet delivery in which routes are calculated, and packets are directed along a particular path (instead of flooded throughout the network). This is accomplished as follows.

1. As before, employ a "trusted node service" to broadcast public keys, so that every node in the net knows the identities and public keys for all other nodes in the network.

2. Have nodes ascertain the identities of their neighbors.

3. Have each node generate a "Link State Packet", containing a list of the node's neighbors.

4. Use the robust flooding of Chapter 2 for dissemination of Link State Packets.

5. Use the database of Link State Packets to compute routes in the network.

6. Have the source compute a route and place the route in the packet header.

7. Use a route calculation algorithm capable of calculating, when available, node disjoint paths in the case in which a calculated route does not work.

8. Also use fault diagnosis to isolate malfunctioning nodes and links.

Chapter 2

Robust Flooding

2.1 Overview

As stated in the introduction, we will begin by designing robust routing based on flooding. We are doing this for two reasons.

1. Robust flooding will work as a data packet delivery system, although it is expensive in terms of bandwidth.

2. Flooding can be used as the delivery mechanism for control messages in a more intelligent routing protocol. In Chapter 3, we will use the flooding mechanism designed in this chapter.

Flooding is the simplest form of routing. In basic flooding, each data packet is forwarded to each neighbor except the one from which the packet was received. Some route history is recorded in the data packet, so that packets can eventually be discarded. Some typical strategies are:

- The route history consists of a count, indicating the number of nodes that the packet has visited. A packet is discarded after it has traversed some bounded number of nodes. [Git]

- The route history consists of a list of nodes that the packet has visited. A packet is discarded if it revisits a node (i.e., a node B discards a packet if B is included in the list of nodes in the packet. If B is not included in the list of nodes, B adds itself to the list and forwards a copy of the packet to each neighbor). [DP]

With flooding, no routing control messages are required and no routing computation is performed.

Note the communications overhead involved in basic flooding is exponential, since there are an exponential number of routes in a network with moderate connectivity. For instance, if every node had k neighbors, and the hop count strategy of packet discard were used, with a limit of h hops, then every data packet would spawn on the order of k^h copies.

Since flooding does not involve route computation, it is vacuously true that a faulty node cannot interfere with route computation. Also, since in flooding, packets traverse every possible path between source and destination, it might be thought that flooding ensures packet delivery provided a path of nonfaulty processors exists between source S and destination D. However, because networks have finite resources, this is not true. Links have finite capacity, so packets must be queued at the nodes waiting to forward packets onto the links. The switching nodes can be engineered so that their processing capacity can keep up with the maximum speed of all attached links; however, they will have finite buffering capacity. Thus some packets will need to be dropped when buffering capacity is exceeded. If no intelligent strategy is designed for dropping packets, the packets for the conversation from S to D might always get dropped, preventing S from successfully getting a packet to D. To ensure successful communication, even across a path of nonfaulty processors, a method of resource allocation that ensures fairness is required.

This chapter presents a Network Layer protocol design based on flooding. The top level description of the algorithm is that each node stores the most recently generated packet from each source node, and ensures fairness in link utilization.

To implement this, the following aspects of the algorithm are required.

- At each node, one fixed length buffer is reserved for each source.
- To assure a packet generated by the proper source node occupies the buffer reserved for that source, a digital signature scheme based on public key cryptography is used.
- To limit the amount of manual configuration necessary to manage the network, the list of nodes, together with public keys, is broadcast by a "trusted node service".
- To prevent a "trusted node" with a Byzantine failure from disrupting the network by broadcasting ficticious nodes, or incorrect keys for network nodes, resources are reserved for (node, key) pairs. In this way, if *any* trusted node is nonfaulty, and therefore broadcasts the correct pair (A,key) for node A, then A will be assured resources in the network.

- To enable nodes to compare generation times of packets from the same source, a source-specific monotonically increasing sequence number is included with each packet.

- To assure orderly use of link bandwidth, waiting packets are served in round robin order according to source node. Associated with each packet stored in memory are flags indicating onto which links the packet needs to be transmitted. The database is scanned round robin, so that every source is guaranteed access to each link.

- Reliability is assured by explicit hop by hop acknowledgments and retransmissions, and state is reacquired after recovery from simple failures.

- We do not require that a node keep its own sequence number in non-volatile storage. Instead, we provide a mechanism for a source to reacquire its own sequence number following its own simple failure.

Note that many networks are configured such that there are "routers" ("IMP"s, in ARPA terminology, "DCE"s in ISO terminology) and "endnodes" ("hosts" in ARPA terminology, "DTE"s in ISO terminology). Each router serves as the point of attachment into the network for tens of endnodes. Thus there is typically an order of magnitude more endnodes than routers. Furthermore, within an endnode, there may be many different processes that are holding conversations across the network. In this thesis, we assume that an endnode internally enforces fairness among its resident processes, and that a router locally enforces fairness among its attached endnodes. Thus endnodes (and processes within endnodes) need not be visible to the rest of the network. The entities with which the distributed Network Layer is concerned are the set of *routers*. Thus "source" refers to the first router that receives the packet from one of its attached endnodes, and reserved network resources are proportional to the number of *routers* in the network.

This chapter starts with a detailed description of the basic design, and then argues its correctness. At the end of the chapter, variants of the design are presented.

2.2 A Robust Flooding Design

In this section we will present the design of a robust flooding-based routing strategy. To meet our robustness goals, we require some number of distinguished "trusted" nodes, responsible for distribution of public keys for each node in the network to each node in the network.

The flooding design we present in this chapter assures that each packet generated by node A destined for node B will have a high probability of reaching B provided that:

- Nodes A and B are nonfaulty.
- At least one path between A and B exists such that for each link L and node C along the path:
  1. L is nonfaulty[1]
  2. C is nonfaulty
  3. C is connected, via a path of nonfaulty processors and links, to at least one nonfaulty "trusted" node.
- Node A waits a sufficient amount of time before generating a new packet such that the packet is not "overtaken" by the source's next packet.

Note that we do not place a limit on the number of faults in a network. This scheme is robust in the above sense, regardless of how many faults occur in the network.

2.2.1 Overview

Routing based on flooding is conceptually simple because no computation of routes is performed. If a packet is received by nonfaulty node A, it will be forwarded to nonfaulty neighbor, B, of node A, provided that A is not forced to drop the packet due to lack of resources.

There are three types of resources that are required for a packet to make progress from node to node:

bandwidth The transmitting node must guarantee, in reasonable time, to output the packet onto the link.

memory The receiving node must have room in memory to store the received packet.

processing capacity The receiving node must have enough processing capacity to read the packet and store it in the appropriate location in memory. We assume the node is engineered properly so that there is sufficient processing power.

[1] Although links usually do not involve sophisticated intelligence, they can exhibit Byzantine behavior such as corrupting data, or discriminatorily passing through packets based on characteristics such as packet length or specific bit patterns. It is also possible for a link to include an intelligent node, such as a Data Link Layer relay, so that more sophisticated Byzantine behavior might be possible.

The subtleties in our design are the result of:

1. ensuring fair usage of finite memory and link bandwidth,

2. minimizing the amount of manual configuration necessary to maintain the network, and

3. minimizing the use of non-volatile storage.

The database at each node B is structured so that B reserves a fixed number (e.g., one) of buffers for packets generated by each possible source. B does not know, a priori, the set of possible sources. Instead, we employ a "trusted node service" that broadcasts the identities of the sources. The set of trusted nodes must be known a priori.

In order to prevent a node A from generating packets with source address S, and thereby using the buffer reserved for S at some node B, (causing packets generated by S to be dropped by B because of lack of buffers) we utilize a signature scheme based on public key cryptography. The "trusted node service" broadcasts the public key of each node, in addition to the identity of the node.

Thus each node B is dynamically informed of the complete set of network nodes, together with a public key for each node. This allows B to reserve resources for each node, and allows B to verify that a packet was generated by node S before devoting resources reserved for S to that packet. However, B only reserves a fixed number, x, of buffers for S. If S generates more than x packets, and B receives more than x packets generated by S, then B will need to drop some of S's packets. It is necessary to give B enough information so that it can make an intelligent choice as to which of S's packets to retain. The packet which B should retain is the packet most recently generated by S. In order for B to recognize the packet most recently generated by S, we include in the packet a "sequence number", which is a monotonically increasing counter maintained by S.

We do not require S to keep its own sequence number in non-volatile storage. We instead provide for S to reacquire the highest sequence number it had used prior to its own simple failure, by having the packet with highest sequence number rebroadcast automatically to any portion of the net which has lost knowledge of the packet with highest sequence number generated by S. This is accomplished through two mechanisms:

1. The network persistently attempts to ensure that the packet with highest sequence number from each (source, key) pair reaches all portions of the network. This is accomplished

by having a node B transmit a received packet to each neighbor C, until C acknowledges the packet. If C restarts, it informs neighbor B, and B then retransmits all packets in its database to C.

2. If a node B receives a packet from neighbor C with source S, and sequence number smaller than that stored in B's memory, B transmits the packet in memory (also with source S, but higher sequence number than the one received from C) to C.

In order to defend against faults in the "trusted node service", we allow multiple "trusted nodes", and reserve resources for each (ID, key) pair broadcast by any "trusted node". In this way, if a "trusted node" is faulty, and broadcasts nonexistent nodes, or incorrect public keys for existing nodes, resources will still be reserved for the set of true (node, key) pairs, provided that at least one "trusted node" is nonfaulty.

The types of packets required in our scheme are:

Data These packets are generated by a higher layer, and are the packets which the Network Layer is to deliver. The Network Layer does not interpret the contents of a data packet, though it adds control information (a "header") for its own purposes. This control information is deleted by the destination Network Layer so the original packet is delivered unmodified to the destination.

Public Key List (PKL) These packets are generated by "trusted nodes" and broadcast throughout the network. They contain the list of (ID, key) pairs for all nodes in the network.

Restart Notification This packet is generated by node C to inform C's neighbor B that C has restarted.

Data ACK The Data ACK packet serves as a neighbor to neighbor acknowledgment of a Data packet.

PKL ACK The PKL ACK packet serves as a neighbor to neighbor acknowledgment of a Public Key List packet.

Restart Notification Ack This packet serves as a neighbor to neighbor acknowledgment of a Restart Notification packet.

Because we do not require a node to keep its own sequence number in stable storage, it is possible for a node, after its own simple failure, to issue a Data or PKL packet with a sequence number it had used prior to its failure. We wish one packet to successfully flood. In this way, either the source's most recently generated packet will successfully flood throughout the network (which is the desired result), or the old packet will successfully flood, reaching the source. If this occurs, and the source is nonfaulty, the source will receive the older packet, recognize it is not the packet the source just generated, and the source will reissue the newer packet with a higher sequence number.

In order to assure one of the packets will flood, in the case of packets with equal sequence numbers, we use the packet signature as a "tie breaker". For Public Key List Packets, to defend against the very unlikely case in which two distinct Public Key List Packets have the same sequence number and the same valid signature, we additionally use the data inside the PKL as a tie breaker.

2.2.2 Databases

A Priori Information

Each node must have stable storage for the following information which is manually entered and maintained:

its own identity
its own keys (public and private)
N, an upper bound on the total number of network nodes
the identities of, and public keys for, each of the t "trusted" nodes
the size of the maximum sized data packet

All other state information can be kept in volatile storage. When a node is initialized, for the first time or after a crash, it knows only the information from stable storage.

Dynamic Database

Each node keeps the following database in volatile storage:

OWN-SEQ-NUM	The next sequence number to be assigned when this node generates its next data packet
OWN-SEQ-NUM-PKL	The next sequence number to be assigned when this node generates a Public Key List packet (this is only kept by "trusted nodes", since other nodes do not generate Public Key List Packets). Note that there is no correlation between "OWN-SEQ-NUM" and "OWN-SEQ-NUM-PKL". If a node is operating both as a "trusted node" and an ordinary data node (generating data packets), it keeps both counters, and the values of the counters are not correlated with each other.
Public Key Lists	For each trusted node: • The Public Key List Packet with highest sequence number received from that trusted node, in a buffer of sufficient size to hold a Public Key List Packet listing N nodes • For each neighbor of this node, two flags: 1. Send-flag – indicating whether this Public Key List Packet needs to be transmitted to this neighbor 2. Ack-flag – indicating whether an ACK for this Public Key List Packet needs to be transmitted to this neighbor
Data Packets	For each node/public key pair reported by any trusted node: • The data packet with highest sequence number from that node/public key • For each neighbor of this node, two flags: 1. Send-flag – indicating whether this packet needs to be transmitted to this neighbor 2. Ack-flag – indicating whether an ACK for this packet needs to be transmitted to this neighbor
Restart Flags	for each neighbor, "Send-Restart" and "Send-Restart-ACK" flags

2.2.3 Public Key Distribution

Our design requires each node to have knowledge of the identity of, and public key for, every other node in the network. This is accomplished with a "trusted node service", which generates and broadcasts Public Key List packets.

Some number, say t, of nodes are designated to cooperate in providing this service.

- Each "trusted node" contains a manually maintained database consisting of the identities

of, and public keys for, each other node in the network.

- Every node in the network contains a manually maintained database consisting of the identities of, and public keys for, each "trusted node".

- Each "trusted node" periodically floods (using the robust flooding described in this chapter) a packet containing a list of all nodes in the net together with public keys for each node. Since we are assuming that keys for the "trusted nodes" are manually maintained at each network node, there is no recursive problem here – packets from "trusted nodes" that are validly signed will automatically be accepted and forwarded by nonfaulty network nodes.

- Installation of a new "trusted node" requires manual modification of all nodes in the network. Installation of a new network node requires manual modification of all the "trusted nodes". Changing of a node's public key requires modification of all the "trusted nodes".

Note that information does not "expire" in this scheme. If a trusted node T ever issues a report of source/key pairs, that information remains with the network, and is reacquired by any node A that has lost state unless one of the following occur:

1. T issues a new report (with higher sequence number).

2. T is manually "deinstalled" by having its (manually maintained) public key modified or deleted at node A.

3. The manually maintained parameter N, which is an upper bound on the number of nodes in the network, and is manually maintained independently at each node in the network, (see Section 2.2.2), is modified at node A to be smaller than the number of source/key pairs listed in T's last report.

4. All nodes in the net lose state at about the same time so that all memory of T's last report is lost.

2.2.4 Packet Types

Data Packets

The purpose of a data packet is to allow delivery of data from a higher layer process at a "source" node, to a higher layer process at a "destination" node. The "user data" is furnished by the upper layer process and is not interpreted by the Network Layer. In addition to furnishing the data to be delivered, the upper layer process informs the Network Layer process of the identity of the destination node. The Network Layer at the source node then furnishes the "header" of the packet, consisting of all fields defined below except "user data". A data packet contains the following.

source node	The identity of the source node
destination node	The identity of the destination node
sequence number	Assigned by the source node, in a monotonically increasing way, except after a loss of state by the source node.
public key	The public key under which this packet has been signed.
user data	Information provided by a layer higher than the Network Layer at the source node, that is not meant to be interpreted in any way by the Network Layer, but is to be delivered without modification to the peer layer at the destination node.
packet signature	A digital signature, verifiable based on the public key listed in the packet, covering the entire contents of the packet.

Public Key List Packets

A "Public Key List" packet is generated by each "trusted node" and it contains the identities of, and public keys for, every node in the network (other than "trusted nodes", whose keys must be manually maintained at every node in the network). It contains:

source	The trusted node that generated this packet
sequence number	Assigned as in a data packet by "source"
key list	Identity/public key for each node in the network
packet signature	A digital signature, verifiable based on the (manually maintained) public key for the trusted node, covering the entire contents of the packet.

A "Public Key List" packet will be quite large, but the maximum size can be derived from N, an upper bound on the number of network nodes, a manually configured parameter (see Section 2.2.2). Given that its size is known, a buffer of appropriate size can be allocated. The only reason a "Public Key List" packet might need to be fragmented is because the link technology connecting two neighbors might limit the size of a packet. If this is the case, then some single hop fragmentation and reassembly protocol can be invoked, (which is just a straightforward encoding problem), so that for the purpose of this thesis we can assume a

"Public Key List" packet can be transmitted and stored intact.

Data ACK

The next form of packet is a Data ACK packet. Its purpose is to acknowledge receipt of a particular Data Packet from a neighbor. It is transmitted only between neighbors, and never forwarded.

Note that the node generating the ACK does not sign the ACK, or even include its ID. All fields in the ACK are copied from the packet being ACK'ed. In this chapter, in which we are building a Network Layer based on flooding, the identity of the neighboring node is irrelevant. The "neighbor" is really the link itself, and a (nonfaulty) node is assumed capable of distinguishing which link a packet was received from.

Thus the information in the ACK is just enough so that the neighbor can match the acknowledgment with the packet to be acknowledged. In the case of a Data ACK, it is necessary to include the packet signature of the packet being ACK'ed in addition to the sequence number, since it is possible there are two distinct packets in the network from the same source, with the same sequence number.

Note that there is no way to verify the packet signature, based solely on receipt of the ACK. Nothing prevents a faulty neighbor from claiming to have received a packet it in fact never did receive. This does not concern us, because even if we made it impossible for a faulty node to ACK a packet unless it had indeed successfully received the packet, there would be no way to prevent that same fa ..y node from discarding the packet after it acknowledged it.

source node	The source of the packet being ACK'ed
sequence number	The ACK'ed packet's sequence number
public key	The public key under which the ACK'ed packet was signed
packet signature	The signature copied from the ACK'ed packet

PKL ACK

The next form of packet is a PKL ACK packet. Its purpose is to acknowledge receipt of a particular PKL Packet from a neighbor. It is transmitted only between neighbors, and never forwarded.

Like a Data ACK, there needs to be enough information in the PKL ACK to unambiguously determine which packet is being ACK'ed. Because we use the data inside the PKL packet as a tie breaker in the case of distinct packets with identical sequence numbers and signatures, we

include the PKL's data in the ACK. Thus a PKL ACK contains the entire PKL which is being ACK'ed.

source	Source of PKL
sequence number	Sequence number from PKL
key list	Data from PKL
packet signature	Signature from PKL

Restart-Notification

The next form of packet is a Restart-Notification. Its purpose is to inform a node M's neighbor, that M has lost state.

No information is necessary within a Restart-Notification except enough to identify the packet as a Restart-Notification. As with the ACK packet, no verification of the source of the Restart-Notification is necessary, because the node can tell which link the packet was received from.

Restart-Notification-ACK

The next form of packet is a Restart-Notification-ACK. Its purpose is to inform a node M's neighbor B, that B has received M's Restart-Notification message.

No information is necessary within a Restart-Notification-ACK except enough to identify the packet as a Restart-Notification-ACK, because of the assumption that the node can tell which link the packet was received from.

2.2.5 Packet Reception Rules

Receipt of Data Packet

The following is executed when node V receives the following Data Packet from neighbor W, (with contents of unspecified fields irrelevant).

source address	S
destination address	Dest-rcv
public key	p
sequence number	sn-rcv
packet signature	PSig-rcv

First, verify that a buffer is reserved for source/key pair S/p (as a result of having received S/p in a valid Public Key List Packet). If not, drop the packet.

Next, verify that PSig-rcv is valid, based on public key p. If not, drop the packet.

Next check if S=V (and the public key matches this node's). If so, check if the sequence number on the received packet is greater than this node's OWN-SEQ-NUM (or equal to, but PSig-rcv does not match the signature of the most recently generated packet by this node). If so, set OWN-SEQ-NUM to sn-rcv + 1, and regenerate the most recently generated data packet.

Now assume that the packet stored in memory in the buffer reserved for source/key pair S/p is:

source address	S
destination address	Dest-mem
public key	p
sequence number	sn-mem
packet signature	PSig-mem

- If sn-mem < sn-rcv, then overwrite the packet in memory and send copies of the received packet to all neighbors except W. This is accomplished by setting the flags for this packet as follows:

  1. Set "Send-Flag", and clear "ACK-Flag" for all neighbors except W.
  2. Clear "Send-Flag" and set "ACK-Flag" for W.

  Also, if Dest-rcv = V, then deliver the packet to the higher layer process that is the client.

- If sn-mem = sn-rcv:

  1. If PSig-mem = PSig-rcv, assume the packet received is a duplicate to the one in memory. Drop the received packet and set "ACK-flag" for W.
  2. Else, (PSig-mem $\neq$ PSig-rcv), use the packet signature as a tie breaker, so that the packet with the numerically higher signature is flooded.
     - If PSig-mem < PSig-rcv, overwrite the packet in memory and send copies of the received packet to all neighbors except W, by setting "Send-Flg" and clearing "Ack-Fig" for all neighbors except W, and clearing "Send-Flg" and setting "Ack-Flg" for W.
     - If PSig-mem > PSig-rcv, drop the received packet and send the one in memory to W, by setting "Send-Flg" and clearing "Ack-Flg" for W.

- If sn-mem > sn-rcv, then transmit the one from memory (the one with the larger sequence number) to W, the single neighbor from which the one with the smaller sequence number

was received. This is accomplished by clearing "Ack-Flag" and setting "Send-Flag" for W.

Receipt of PKL Packet

Receipt of a PKL Packet from neighbor W is handled almost identically with receipt of a Data Packet from neighbor W. As with a Data Packet, the received packet is first checked for validity, and then compared to the stored packet to determine whether it is older, a duplicate, or newer.

The only modification necessary when dealing with a PKL is that an additional check is made, in the case where the sequence numbers and signatures match. If the sequence numbers and signatures match, the "key list" field is compared. If the "key list" matches in the two packets they are truly duplicates. Otherwise, a lexicographic comparison of the "key list" fields serves as a tie breaker to determine which packet is considered newer.

Receipt of a Data ACK Packet

When receiving a Data ACK Packet from neighbor W, find the corresponding packet in memory. If no such packet, drop the ACK packet with no further processing. If the ACK does match a packet in memory, clear "Send-Flag" for W for that packet.

Receipt of a PKL ACK Packet

As with a Data ACK Packet, find the corresponding packet in memory. If no such packet, drop the ACK packet with no further processing. If the ACK does matchh a PKL packet in memory, clear "Send-Flag" for the neighbor from whom the ACK was received.

Receipt of a Restart Notification

If a Restart Notification is received from neighbor W, set "Send-Restart-ACK" for W, and, for all packets (Data and PKL), clear "ACK-Flag" and set "Send-Flag" for neighbor W.

Receipt of a Restart Notification ACK

If a Restart Notification ACK is received from neighbor W, clear "Send-Restart" flag for W.

2.2.6 Transmission Rules

Reliability of the broadcasts is done by acknowledgments and retransmissions. The rules above set flags, indicating the need to accomplish certain actions in the future. The actual transmission of packets is done as follows:

When the link to a neighbor is ready to transmit a packet, scan the database in a round robin order, starting at the flag that caused the last packet transmission. Any round robin ordering is legal, as long as it guarantees to scan every flag on every packet. Find the first place in which a flag is set. The flag will be associated with some neighbor, W. The flag is one of the following:

1. If the flag is "Send-Restart", queue a Restart Notification packet to W.
2. If the flag is "Send-Restart-ACK", queue a Restart Notification ACK to W and clear "Send-Restart-ACK" for W.
3. If the flag is "Send-Flag" for a data packet or a PKL, queue that packet to W.
4. If the flag is "ACK-Flag" for a data packet or a PKL, queue an ACK for that packet to W and clear "ACK-Flag" for that packet for W.

2.2.7 Restarting

The last event which causes modification to the database is restarting. When restarting, initialize the database and set "Send-Restart-Notification" flag for each neighbor.

2.2.8 Additional Check on "Public Key List" Packet

Two distinct packets with the same sequence number can be generated by source S if S is faulty, or if S uses a sequence number following its own simple failure, which it had used prior to its own simple failure. In this case, we'd like one packet or the other to successfully flood through the network. If the most recently generated packet is successfully flooded, then everything works properly. If the older packet floods instead of the more recently generated packet, then the older packet will flood back to the source, which will recognize that the packet is not its own most recently generated packet, and the source will retransmit the latest packet, with higher sequence number.

With data packets, when sequence numbers are equal, the check for whether the received packet is a true duplicate of the one in memory is made by comparing the packet signatures.

If two distinct packets mapped to the same packet signature, this scheme would not detect a conflict, in a very low probability occurrence. Since the Network Layer is a datagram service, loss of a data packet under this circumstance would not be an issue.

However, it is more critical that "Public Key List" packets propagate throughout the network. Thus we require that the data itself be compared against the data in the packet in memory, in order to determine if the packet is a duplicate.

If the received packet is a duplicate of the one in memory, it is dropped. If it is not a duplicate, and the packet signatures are distinct, then the packet signature will act as a "tie breaker" resulting in the successful flooding of one of the packets. If the packet signatures are equal (perhaps because a malicious "trusted node", through use of its own private key, was capable of constructing two distinct packets with the same packet signature), then the mechanism does not allow one packet to successfully flood. Instead, one portion of the network may retain one packet and a different portion of the network may retain a distinct packet.

Thus in the case of Public Key List packets, we will not assume packets are duplicates if the signatures match, but instead require a comparison of the data. If the signatures in two Public Key List packets match, but the data does not, then the data itself acts as a "tie-breaker" in this case.

Thus for data packets, the signature acts as a low order portion of the sequence number field (and we ignore the potential of a data packet being lost due to appearing to be a duplicate). For Public Key List packets, the data field is an even lower order field, to be considered only in the case of ties with both the sequence number and signatures.

This portion of the design may be overly defensive. It is extremely unlikely for two distinct PKL packets to have the same valid signature, and a comparison of the data may be too computationally expensive, especially as it must be done every time a duplicate PKL is received. It may be preferable to treat PKL packets the same way as data packets (assume packets with equal sequence numbers and equal signatures are duplicates, with no further processing).

2.3 Costs of This Design

The costs of the above design over basic flooding are:

memory In each node, $O(t * N)$ buffers must be reserved, one for each possible source/public key pair. (In contrast to basic flooding in which O(1) buffers are reserved.)

As mentioned before, the "N" in the $O(N)$ refers to the number of *routers*, not the number of endnodes. Provided that each router ensures fairness among the endnodes that it serves, signatures and buffer reservations are on behalf of the router.

A separation of nodes into "routers" and "endnodes" is a simple form of introducing hierarchy into the network, which is further discussed at the end of the chapter.

manual maintenance A database must be maintained consisting of the identities of, and public keys for, the set of "trusted nodes".

extra nodes Some nodes must act as the "trusted node" service. There must be enough trusted nodes so that at least one will not exhibit a *Byzantine* fault. A simple fault by a "trusted node" is not a problem, assuming the trusted node issued a Public Key list packet before it crashed, and the network retains its Public Key list packet (which it will, as long as some nonfaulty nodes remain up at all times).

communications bandwidth Extra overhead in data packets is required for sequence numbers and signatures. Also, the public key list must be periodically broadcast by each "trusted node".

processing power Each node forwarding a data packet must cryptographically check the signature (to check that it was the source node that generated the packet and that no part of the packet has been corrupted).

The gains of the above design over basic flooding are:

robustness This design achieves robustness in the face of Byzantine failures.

communications bandwidth In basic flooding, each packet spawns an exponential number of copies. With this design, since nodes keep state about the latest packet, a node normally floods a particular packet once. Thus each packet traverses each (one-way) link once. Since there are at most N^2 links, this form of flooding is dramatically more efficient (in terms of communications bandwidth usage) than basic flooding. In fact, networks are usually designed in such a way that nodes have a fixed maximum number of neighbors. In this case, there are actually $O(N)$ links in the network.

2.4 Motivations Behind the Above Design

2.4.1 Review of the Design

We wish to assure that if a nonfaulty path exists between a pair of nonfaulty nodes, a packet will successfully travel between the pair of nodes. The critical resources that must exist in order for the packet to successfully reach the destination along the nonfaulty path are memory at each node, and bandwidth at each link.

We reserve a buffer for each source at each network node, and scan the database in an orderly fashion to guarantee every stored packet access to the finite link bandwidth. To assure that only the source could have generated the packet that is to occupy the reserved buffer, we include a signature in the packet.

To verify a signature, every node must have a "public key" for every other node in the network. To assure this, with a minimal amount of necessary manually configured information, we use "trusted nodes" which act as a public key distribution service. Keys for all nodes are manually configured at the trusted nodes, and the trusted nodes flood a list of (node, key) pairs throughout the network. To enable the successful flooding of the Public Key List Packets, public keys for the trusted nodes must be manually entered at all the network nodes.

Flooding is accomplished by having each node recognize whether a received packet is older, a duplicate, or newer than the packet from the same source that is stored in the database. This is accomplished by use of a "sequence number" which is a monotonically increasing counter kept by the source. If the received packet is deemed older, the packet from memory is transmitted to the neighbor which sent the older looking packet. If the received packet is a duplicate, it is ignored. If the received packet is deemed newer, the packet in memory is overwritten and the received packet is transmitted to all neighbors except the one from which it was received.

We allow a source to lose state regarding its own sequence number after it experiences a simple failure. Most likely, a source will reacquire its own sequence number because the network persistently attempts, through neighbor to neighbor acknowledgments, to keep databases at all nodes up to date. If the source issues a packet with a lower sequence number than it had used prior to its failure, the source will eventually (if the net retains memory of the pre-crash packet) receive the pre-crash packet, and the source will reissue its post-crash packet with higher sequence number.

If the source issues a packet with a sequence number matching one it used prior to its own

failure, the packet signature (and in the case of a PKL, additionally the data inside the PKL) acts as a low order field to the sequence number, so that either the post-crash packet will flood successfully through the net (despite the sequence number reuse), or the source will receive the pre-crash packet, and then reissue its post-crash packet with higher sequence number.

2.4.2 Buffer Pool

With flooding, since every packet theoretically traverses every possible path, communication between two nonfaulty nodes should be possible if at least one nonfaulty path connects them. However, in practice, since networks have finite bandwidth and finite buffering capabilities, packets must be dropped. If packets are dropped indiscriminately, there is no way to guarantee fairness. A particular traffic stream (source/destination pair) might have all of its packets dropped.

We wish to design a network in which a traffic stream has a high probability of successful delivery of a single packet at a time. In other words, if a source does not generate a new packet until the old packet has had time to be delivered, then each packet should arrive safely at the destination, provided a path of nonfaulty processors existed during the packet delivery time, and provided that a reliable Data Link layer recovered from occasional corruption caused by the data link.

It is sufficient for the network to assure fairness on a per source basis (instead of on a per source/destination pair basis, or worse yet, on a per source process/ destination process pair basis), since the source can be held responsible for assuring fairness between traffic streams it originates.

To assure fairness on a per source basis, with N being the number of sources, a node needs to keep $O(N)$ state. Otherwise it may continually drop the same source's packets.

Since faulty "trusted nodes" can report nonexistent nodes, or false public keys for existent nodes, we actually require $O(t*N)$ buffers, where t is the number of trusted nodes and N is the maximum number of nodes any single "trusted node" is assumed capable of reporting.

If an otherwise nonfaulty node A has too low a value for N, i.e. a nonfaulty "trusted node" does indeed report more than N nodes, then A becomes essentially a faulty node, and its behavior under the circumstances becomes irrelevant (though if it reports the problem to Network Management, the problem is easily diagnosed and repaired – either the "trusted node" reporting more than N nodes is faulty, or A's value for N is incorrect and should be modified).

Since buffers are reserved on a per source basis, a source S must authenticate itself in order to allow its packet to occupy the buffer. Otherwise, a faulty node could claim to be S and have its own packets occupy buffers reserved for S.

It is necessary to include all the contents of the packet in the signature, so that no node other than the source can modify any portion of a packet.

Delivery of a corrupted packet to the destination is not a problem. We have assumed that a higher layer protects the destination from mistaking faulty data as valid. Our design does not protect against delivery of faulty data, since a faulty trusted node could construct a private key/public key pair for some node S, broadcast the fraudulent public key for S, and issue packets, masquerading as S. We have specifically required that higher layers recover from such problems.

We only require that nonfaulty data get delivered. Thus it is essential that a corrupted packet for S not compete for network resources with the nonfaulty copy.

An alternative scheme, attractive for performance reasons, is to require the signature just to cover the header of the packet. It takes less computation to verify a signature if the object being signed is smaller. Thus it might be attractive to attempt to design a scheme whereby only the header of a packet is signed. Then faulty nodes might corrupt the data, but a higher layer protocol will protect the destination.

The problem with this approach is that distant nodes will not be able to detect (without looking at all the bytes of the packet) that a corrupted packet and the original packet are not simply duplicates. If they are assumed to be duplicates, then the corrupted packet might be delivered *instead of* the correct copy, and the scheme does not meet the robustness criteria we have required.

Thus it is necessary for forwarding nodes to examine the complete contents of each data packet. If duplicate detection were done by comparing the data, with a signature scheme that only covered the header of the packet, then a distant node would detect that at least one of the copies was a corrupted packet, but it would not be able to determine which one was the valid packet. This would prevent the Network Layer from meeting the correctness condition that the correct copy be guaranteed resources. If a node cannot differentiate the correct copy from corrupted copies, then no fixed number of buffers will suffice to ensure a correct copy gets delivered, since there is no (practical) limit to the number of variations of the data that a faulty node can generate.

Thus the signature is required to distinguish the correct packet. Once the signature covers the entire packet, it is not necessary to further do a byte by byte comparison of the data, since different data will yield distinct signatures[2] (with overwhelming probability).

2.4.3 Trusted Node Service

Every node must be capable of verifying each other node's signature, without being able to forge signatures. This can be done with a public key scheme, provided that every node knows every other node's public key.

Manual databases at every node could be used for distribution of public keys. However, maintenance of such databases would be extremely tedious. Adding a new node to the network would require modification of all the existing nodes.

With a trusted node service, each node need only know, a priori, the number of trusted nodes, together with public keys for each of the trusted nodes. When a new node is to be added to the network, or a node's public key is to be changed, manual modification only of the trusted nodes is required. However, if a new trusted node is to be added to the network, or if the public key of a trusted node is to be modified, manual modification of all the nodes is required.

Usually, all nonfaulty trusted nodes will have the same public key for each node. However, when their databases are in the process of changing (being manually modified), due to a node's public key changing or a new node being added, nonfaulty trusted nodes will have non-identical versions of the network node identities and keys.

Faulty trusted nodes can advertise faulty public keys for other nodes, and even collaborate with other faulty trusted nodes so that a majority of trusted node reports might contain the same faulty public key for some node or group of nodes.

Routing will continue to work between nonfaulty nodes A and B despite faulty trusted nodes, provided that at least one reachable trusted node did not exhibit a *Byzantine* failure. That is because we have required each nonfaulty node to reserve a buffer for each possible source/public key pair. Since there are a fixed number, t, of trusted nodes, each of which is limited to reporting identities of and keys for N nodes, there are at most $N*t$ possible source/public key pairs that any network node could know about at any time. We call this the "source/key pair" scheme.

[2]Of course, byte by byte examination of data is required in order to verify the packet signature, so the performance benefit derived from not needing to compare the data on two packets is not dramatic.

An alternative approach is to require receipt of the same public key for some node S from a **majority** of trusted nodes, before the public key for S is used. We call this the "majority" scheme. The majority scheme has the advantages of:

smaller memory requirements With the majority scheme, only one buffer is required per source, since the trusted nodes in effect "vote" on the public key to be used for that buffer.

more efficient communications bandwidth usage With the majority scheme, each source S can utilize $1/N$ of the communications bandwidth of each link (where N is the total number of nodes in the network).

With the source/key pair scheme, if all but one of the t trusted nodes were faulty, and the faulty nodes each broadcast a distinct (and incorrect) set of N source/key pairs, then the true sources' access to communications bandwidth is decreased by a factor of t.

less manual information With the majority scheme, a faulty trusted node cannot report nonexistent nodes in an attempt to force network nodes to allocate extra memory and communications bandwidth, since no new node or node/key pair will be believed by network nodes unless a majority of the trusted nodes report it.

With the source/key pair scheme, an extra manually maintained parameter, N, an upper bound on the size of the network, must be maintained so that there is a limit to the amount of resources that a faulty trusted node can impose on the network by reporting and simulating nonexistent network nodes.

However, we decided the enhanced robustness offered by our scheme outweighed the performance implications. With our scheme, only a single trusted node need be reachable and nonfaulty (in the Byzantine sense) in order for communication to be possible. This has the following fortunate consequences.

1. Fewer trusted nodes are needed. With the majority scheme, at least $2*f+1$ trusted nodes are required, where f is the number of faulty trusted nodes the network should tolerate. With the source/key pair scheme, only $f+1$ trusted nodes would be required.

   Note that f is actually the number of *Byzantine* faults the network can handle. Simple faults by trusted nodes are not a problem, as long as the trusted node issued a node list report before its simple failure, and at least one network node was up at any time (with

enough overlap so that network state can be passed to the node that will be up), so that memory of its report persists in the network.

2. With the majority scheme, network partitions could cause all routing in the network to cease, even in the absence of any Byzantine faults. If the network partitioned such that no partition contained a majority of trusted nodes, then routing throughout the network would cease (unless a majority of nonfaulty trusted nodes had issued node list reports before the partition occurred, and knowledge of the reports persists in the partition.)

   With the source/key pair scheme network partitions are, of course, not a problem. Routing will operate correctly in any partition containing at least one nonfaulty trusted node.

3. With the majority scheme, if a node changes its own public key, there will be a potentially long period during which it will not be able to send packets, since it cannot use its new key until the new key has been installed on the majority of trusted nodes. With the source/key pair scheme, a node can use its new key as soon as it has been installed on a single nonfaulty trusted node.

The performance penalty associated with rejection of the majority scheme might at first seem severe. However, in practice the source/key pair scheme will perform as well as the majority scheme because:

- In practice, Byzantine failure of a trusted node can be assumed very unlikely. Thus, one or perhaps two trusted nodes will be sufficient.

  Note that either scheme would work with only a single trusted node, provided that node never experienced any Byzantine failures, since the network will not even detect a simple failure of the trusted node.

  Once a trusted node issues a node report, there is no reason for it to ever issue a new one unless the information has changed (new node, node deletion, or key change.)

- Byzantine failure of a trusted node is very easy to detect.

  Any node in the network can be alerted, and report a potential problem, if it receives conflicting reports of any node's public key, or receives nonidentical lists of nodes from different trusted nodes.

If the faulty information is an incorrect public key for a node A, then A knows which trusted node is faulty. If the faulty information consists of leaving a node A out of the list of nodes, again A knows which trusted node is faulty. If a trusted node adds new, nonexistent nodes to its node list, there is no obvious single node that can know for certain whether the trusted node with the expanded list is faulty, or a different node is faulty because of omitting nodes. However, the faulty node cannot report more than N nodes, so if N is not significantly larger than the true network size, a Byzantine trusted node will not be able to add many new nonexistent nodes without either leaving out a true node (in which case that node will know the trusted node is faulty), or reporting more than N nodes (in which case its reports will not be believed by any node whose network bound is set to N.)

Since Byzantine failure of trusted nodes is quickly diagnosable and correctable, in practical networks the number of trusted nodes can be very small.

In practical networks, we can provide 2 buffers per source plus a few (say x) extras. These $2 * N + x$ buffers would allow routing to continue for normal operational scenarios (key changes for x nodes simultaneously), and allow a single one of the trusted nodes to be maliciously Byzantine (which would be quickly detected by all the other nodes if the faulty trusted node were really reporting N false node/key pairs, since each nonfaulty node would either be left off the list, or be reported with a false public key).

2.4.4 Sequence Number

Public key cryptography can prevent a faulty node from generating a packet that looks like it was generated by a different source node S, usurping resources reserved for S. However, without additional design, a faulty node could replay old packets from S, and have the old packets usurp the resources, locking out S's current packets from the resources necessary for successful delivery.

For this reason we require a sequence number, which is a counter maintained by each source node, incremented for each packet generated by that source node. Sequence numbers from different sources have no relationship.

Sequence numbers traditionally introduce problems [Pe]. The basic problems are:

- Sequence Number reaching maximum value – due to the field being of fixed length

- Incorrect Ordering – due to the finite length field being allowed to "wrap-around"
- Loss of State by Source – so that after a crash its new packets are incorrectly ordered with respect to pre-crash packets.

There are various strategies for dealing with a fixed length sequence number field:

1. **Wrapping** – With this strategy, when the field reaches its highest numerical value it is allowed to "overflow" back to 0. Arithmetic comparisons can be made in a circular space. If the size of the sequence number space is n (with n odd), then an ordering $a\ LT\ b$ is defined as:

   - $|a - b| < n/2$ and $a < b$, or
   - $|a - b| > n/2$ and $a > b$

   With this strategy, the size of the sequence number space is chosen to be sufficiently large so that a source (under normal circumstances) would not use as much as half the sequence number space within a packet lifetime.

   However, this non-well ordered space can cause severe problems. It was just this ordering that caused the ARPANET disaster documented in [Ros][3].

2. **Resets** – Another strategy is to allow a sequence number to be "reset" after it reaches its maximum value, by some process involving flooding of a reset packet throughout the net.

   This strategy is very risky as well, for two reasons:

   (a) If not all memory of the previous use of high numbers for the sequence number is purged, the nodes that remember the high value might reintroduce memory of the high values into the network. A single node with memory of a high value can reflood the net with the high value, undoing the reset operation.

[3]The sequence number was being used to order Link State Packets in a Link State form of Network Layer. The basic algorithm was that a node would accept a Link State Packet and flood it if it was newer than the one stored in memory from that source. Then one day a malfunctioning source generated packets with three sequence numbers a, b, c such that $a\ LT\ b\ LT\ c\ LT\ a$. These packets proliferated around the network like a virus, since each node, when accepting one of the LSPs, would make several copies for each of its neighbors. Furthermore, the order in which LSPs would be flooded by a node were a, b, c, a, b, etc. – just the order so that the neighbor would regard each packet received as newer and flood it (making even more copies).

(b) If one of the packets that execute the reset continues to reside inside the network, it can cause a reset at a later time. This creates a new flood of reset packets, creating the possibility that in the future the reset can again mistakenly be restarted, with no guarantee that the process will ever halt.

3. **Large Enough Field** If the sequence number field is sufficiently large, it should never need to wrap around. For instance, if the field is 64 bits long, and a source generates packets every microsecond, it would take over 400,000 years for the field to overflow.

   However various types of errors in a traditional system could cause the field to prematurely reach its maximum value:

   (a) A faulty node other than the source could masquerade as the source, generating packets with the source's ID, and with a high value for the sequence number.

   (b) The source itself could fail and issue packets with high sequence numbers on behalf of itself.

   (c) A node forwarding the source's packet could corrupt the sequence number.

Another problem with sequence numbers is loss of state following simple failures. This is particularly critical when a node loses state about its own sequence number.

If a source S were to restart with the lowest sequence number, its new packets (the ones generated after the crash) would be ignored by the rest of the network until S's post crash sequence number increments past the pre-crash value.

Some obvious approaches for a node to acquire its own pre-crash sequence number do not work:

1. The source, upon recovery, can query a neighbor as to the pre-crash value of its sequence number. This does not work because the neighbor could also have lost state. Also, if naively done, the neighbor could give the source a faulty answer, such as an overly high sequence number which might prematurely cause the field to overflow.

2. The source might broadcast a message requesting anyone with knowledge of its pre-crash value to respond. Again, this does not work because the network might be partitioned at the time of the source's recovery, and knowledge of the pre-crash value might exist only in some partition that is temporarily unreachable at the time of the request.

These problems which are normally associated with sequence numbers are avoided with our design.

1. We allocate a "large enough" field, precluding resets and wraparound.

2. Nodes are persistent about keeping state synchronized with their neighbors. When a node informs its neighbor that it is restarting, the neighbor marks all packets in the database as needing to be retransmitted to that neighbor. Thus, theoretically, the packet with latest sequence number from each source should automatically be reflooded into any portion of the network that has lost state.

3. Additional mechanism for recovery of state following a crash is accomplished by having a node send a stored packet with a higher sequence number in response to receipt of a packet with a lower sequence number. In this way, if a source issues a packet with sequence number k, and any reachable node R has memory of a packet from that source with higher sequence number h, then when packet k reaches R, it will cause a reflooding of the packet h back into the region that does not have memory of that packet, including the source, which will know at this point that it must increase its sequence number to $h + 1$, and reissue the last packet.

   Theoretically, this mechanism should be redundant – without it, the source should eventually receive the old packet with higher sequence number anyway. However, there are reasons for including this mechanism:

   - It is basically free – it adds no extra memory or communications bandwidth, nor does it significantly complicate the algorithm.

     The entire mechanism consists of the rule, "If neighbor B transmits a packet from source S, with sequence number smaller than the one in memory for source S, set "Send-Flag" on the packet it memory for neighbor B." Theoretically, "Send-Flag" should already be set for B, if this node has a packet with sequence number larger than B has yet seen, or if B has lost state. But there is no significant burden placed on the protocol to make sure "Send-Flag" for B is in the correct state (set) when this case occurs.

   - In practice, with nodes restarting, lost messages, delayed messages, and out of order messages, (which can occur even on a point to point link with some technologies),

it is possible for neighbors to get out of synchronization. If this were to occur, this mechanism causes resynchronization automatically (again, at no additional cost).

- It may be desirable to modify the robust flooding, especially for data packets, to be less "persistent", since the Network Layer is assumed to be a datagram service. See Section 2.6.2 for more discussion of this issue.

  In the modified scheme (where "Send-Flag" for W for all data packets is not set as a result of receipt of a "Restart-Notification" from W), the mechanism ensures that data packets that really do need to be reflooded (those informing the source of its own sequence number) will. Without this mechanism, in the modified scheme, the source might not reacquire its old sequence number, and its newly issued packets might not successfully flood in the network until their the source issued enough packets so that the sequence number became larger than the previously used sequence number.

4. Premature overflow of the "large enough" field due to faulty nodes other than the source is avoided through the use of cryptography. No node other than the source can (with non-negligible probability) generate a packet with the source's address, since it cannot generate a signature. Likewise, no node other than the source can modify a packet, since that will make the signature invalid.

5. Thus the only node which can cause the field to prematurely overflow is the source. Through Byzantine fault, the source can cause its own sequence number to overflow. If this occurs, then the faulty source will no longer be able to generate packets, which is not a problem (we make no guarantees that faulty nodes be able to communicate). However, if the source is repaired, the algorithm must then enable the source to communicate, since at that point it will be a nonfaulty node. This is accomplished by changing the source's public key, which will not compete for buffers with packets issued and signed with the source's previous public key.

An alternative method of marking packets so that the times of their generations can be ordered is to use a timestamp, globally synchronized throughout the network. A timestamp of global significance has nice properties.

- The space is totally ordered (it is "large enough"), avoiding wraparound and reset issues.

- Recovery of pre-crash values is not an issue, since a node is not considered "recovered" until it is synchronized with the other nodes' clocks.

- Faulty timestamps (e.g., those for the future) can be recognized, and packets with faulty timestamps can be discarded.

However, global clock synchronization is not an easy problem, and there are currently no practical implementations of networks with globally synchronized clocks, especially globally synchronized clocks robust against Byzantine failure. Thus we prefer to build our design upon sequence numbers, since it can be implemented without special hardware, and without solving the global clock synchronization problem.

It is possible for a source to reuse a sequence number if

- the source is faulty (Byzantine fault), or

- the source experienced a simple failure, and lost state regarding its own sequence number.

In this case we use the signature (and possible the data itself in the case of PKL packets) as a low order field appended to the sequence number, so that in the case of sequence number reuse, one packet is unambiguously considered "newer" than the other. If the correct packet (i.e., the post-crash packet) is deemed newer, then everything works properly. If the pre-crash packet is deemed newer, then it will flood instead of the post-crash packet, reach the source, and the source (if nonfaulty) will reissue the post-crash packet with higher sequence number.

2.4.5 Public Key in Packet Formats

We have included the field "public key" in Data Packets, and Data Packet ACKs, but not in PKL Packets or PKL ACKs.

It is not necessary to include "public key" in the Data Packet format. Instead, we could require nodes to infer the public key being used, by trying all public keys known for that source node, until one yields a valid signature. At most t public keys could be known for a particular source node, since at worst each trusted node could report a distinct key for that source. The decision about whether to include "public key" is a tradeoff between communications bandwidth (the necessity of increasing the header length to include the extra field) and processing overhead (the necessity to try multiple public keys).

The field "source node" in Data Packets is also not necessary. The identity of a node could be its public key. Of course if the field "source node" is excluded, then the field "public key" would

be essential, since otherwise the only method of identifying the source node would be trying all known public keys, of which there can be up to $t * N$. We have included the field "source node" for conceptual clarity, and since the field is likely to be very small (certainly small compared to a public key, or a signature), a network protocol designer wishing to minimize header length would be better off excluding the "public key" field rather than the "source node" field.

However, even if the field "public key" is excluded from the Data Packet, it remains essential in the Data Packet ACK Packet. Suppose the source node S has public key p_1. Suppose a malicious trusted node, T, broadcasts the pair (S,p_2). Then T is capable of constructing validly signed packets from (S,p_2). It is possible that knowledge of the private key associated with public key p_2 enables T to construct a Data Packet for which a particular signature would be valid. In that case, T could, on receipt of packet with sequence number x and signature s from (S,p_1), construct a packet with sequence number x and signature s from (S,p_2). In this case, without the field "public key" in the Data Packet ACK, the ACK packet would be ambiguous as to which packet was being acknowledged.

In PKL packets, the field "public key" is not necessary because each node holds only one key for each trusted node (the key which is manually maintained).

2.5 Fault Detection

Certain failures are easy to detect with the flooding scheme described in this chapter. Others are not. This section will discuss which sorts of errors can be automatically diagnosed and reported, so that a network manager can then investigate the problem.

2.5.1 Faulty Trusted Nodes

Since the task of a "trusted node" is to generate a Public Key List Packet with the latest information, failure of a "trusted node" can only consist of one of the following:

1. Failing to send *any* Public Key List packet.

2. Having its Public Key List packet with highest sequence number contain incorrect information:

   (a) It can contain too many nodes (more than N).

   (b) It can leave out nodes.

(c) It can contain incorrect nodes.

(d) It can contain correct nodes with incorrect public keys.

Symptoms of such failures do not necessarily prove failure of the "trusted node", since the same symptoms can also result from other causes. However, as long as these symptoms can be detected by at least one nonfaulty node, they can be reported and investigated.

Failure to generate Public Key List Packets

If a trusted node T fails to send *any* Public Key List packet, then every nonfaulty node will detect that, due to having a manual key for trusted node T, and never receiving a Public Key List packet with source T. However, this symptom (no knowledge in a nonfaulty node of any Public Key List packet generated by T) could also occur with nonfaulty T, for instance if all of T's neighbors are faulty and fail to forward T's Public Key List packet to the rest of the network.

Including Too Many Nodes in PKL

If T's latest (one with highest sequence number) Public Key List Packet contains M nodes, then any node V with parameter N set such that $N < M$ will detect and report the problem. "The problem", in this case can be either that T is faulty, or that V's parameter N is set incorrectly.

Omitting nodes or Reporting Incorrect Keys

If T's latest Public Key List Packet leaves out node V, or reports V with an incorrect public key, then if V receives T's Public Key List Packet, V will detect and report the problem. T's latest Public Key List Packet will reach V if a path of nonfaulty nodes connects T with V, and there is no other validly signed Public Key List Packet from V with the same sequence number.

Inclusion of Nonexistent Nodes

If T's PKL contains incorrect nodes, but its report also contains all the nonfaulty nodes, together with correct keys for them, this problem will not be automatically detected by the network, unless a comparison is made between T's PKL and the PKL issued by a different "trusted node". However, as long as T's PKL is correct regarding the nonfaulty network nodes, the inclusion of incorrect nodes will not interfere with the correct functioning of the Network Layer.

2.5.2 Faulty Forwarding Nodes

Detection of faulty forwarding nodes is not easy with our scheme. The very robustness of our scheme is somewhat of a disadvantage because a problem will not be detected unless *no* nonfaulty path exists between source and destination. For instance, if several paths exist between S and D, but failures have occurred along some of those paths, it would be desirable to detect and report the problem so that those paths can be repaired, even though they are not needed at the time. It is undesirable to detect failures only after *all* paths have failed, and the network is no longer operating.

Let us assume the following conditions in the network:

1. At least one nonfaulty path exists between all pairs of nonfaulty nodes in our network.

2. Some number of faulty forwarding nodes exist, that fail to forward packets, but do acknowledge them correctly when received from a neighbor.

This situation cannot be detected with our scheme. Every nonfaulty node will successfully receive the packet with largest sequence number from every other node, and no hint of trouble will be evidenced by the state of the "Send-Flag" and "Ack-Flag" on each packet.

It is only when all nonfaulty paths fail that any evidence exists that there is a problem, evidenced by some nonfaulty nodes not receiving the latest packet from some sources. This situation cannot be detected automatically by the Network Layer with our scheme, since nodes do not know whether later packets exist, unless they receive them. However, the upper layer protocol can inform the Network Layer when it fails to get acknowledgments to its packets launched to a destination node.

In this case (when no nonfaulty route exists), node by node query can determine which nodes have received the latest packet from a particular source/key pair, which will allow a network manager to have a good idea of which nodes have failed. This manual query assumes that the network manager knows the topology of the network. With our scheme, the Network Layer is not aware of the identity of neighbors, so a node by node query done without knowledge of the topology will just yield a list of nodes reporting they have seen the latest sequence number, and list of nodes reporting they have not. Without knowledge of the topology, this information does not narrow down the candidate failing nodes at all – any node on either list can be either faulty or nonfaulty.

2.6 Variants

2.6.1 Multiple Outstanding Packets

The above design assumes S will not generate a packet with sequence number $k+1$ until its packet with sequence number k has been delivered. If S fails to follow this rule, and instead issues two packets in rapid succession, its second packet may overtake its first packet, and the older packet will be dropped before delivery.

Note however, that there might be slow paths in the network, in addition to fast paths. If the source issues packet $k+1$ after packet k has been successfully delivered over the fastest path in the network, but while k is still in transit over slower paths, $k+1$ may overtake k over some of the slower paths, causing no ill effects (in fact, causing the positive effect of limiting some redundant traffic).

Unfortunately, the source has no way of knowing when at least one copy of its packet has been successfully delivered to the destination. It can learn of that fact in approximately twice the time, since the destination can send an acknowledgment. If the source wishes to maximize its throughput, therefore, it must estimate the time of delivery. If the source issues packets too quickly, some of its packets may get lost due to being overtaken by packets with higher sequence numbers. But since the Network Layer is assumed to be a datagram service this is in fact legal.

A modification to our algorithm allows safer "pipelining" of packets. ("Pipelining" is allowing multiple packets from the same source/key pair to be in transit simultaneously.) The modification is to require nodes to keep, instead of a single buffer per source (and per key), m buffers per source/key pair, where m is the number of packets desired in the pipeline. If the largest sequence number seen so far by node B from source/key pair S/p is k, then node B keeps any packets with sequence numbers between $k-m+1$ and k with source/key pair S/p.

2.6.2 Less Persistent Data Packet Flooding

It is not essential that a node B reacquire the database of data packets received prior to its own simple failure. Most likely, these packets have already reached their destinations, or (if the only path to the destination was through B, which was down at the time), the packets are so old that upper layers no longer need them. In fact, it is usually preferable, from the point of view of the higher layer protocols, that very old data packets do not get delivered. Public Key List

packets are different, since it is essential that every node store the most recently issued PKL from each trusted node. The "persistent" design is necessary for PKLs.

In the design above, we used the same mechanism for distributing data packets and PKL packets. Since the design is overly reliable for data packets, it could be modified to be less "persistent" in the case of data packets.

The modification consists of not modifying "Send-Flag" for neighbor W on all data packets, when receiving a Restart Notification message from W. As a result, recovering node W does not receive old data packets that it had acknowledged prior to its own failure. The only case in which an old data packet should be reflooded is the case in which a source is issuing packets when memory of packets from that source with higher sequence numbers remains in the network. The mechanism of having receipt of an older looking packet trigger reflooding of the stored packet ensures that the reflooding will occur in this case.

The modification results in some bandwidth efficiency gain, since the entire database of old data packets need not be transmitted on each link to a recovering node. If the node were nonfunctional for long enough, it would never have acknowledged most of the data packets stored by its neighbors, so upon recovery the majority of the data packet database will be transmitted anyway.

Temporarily wasting bandwidth on links to a recovering node is not a very important problem, since the links were of no benefit to the network while the node was nonfunctional, and wasting the bandwidth after the node recovers is equivalent to the node having been down slightly longer. However, a more radical modification prevents wasting the bandwidth. The more radical modification consists of *clearing* "Send-Flag" for W on all data packets, upon receipt of a "Restart-Notification" from W.

The theory behind this proposal is that the majority of packets in the database either have an alternative path to the destination, or were generated sufficiently long ago that they will be of no use if delivered now. Only packets generated prior to W's recovery will fail to be flooded through W. A source cannot expect packets to reach a destination through a node which is nonfunctional. Packets generated after W's recovery will be transmitted to W.

2.6.3 Elimination of Acknowledgments

It is interesting to note that acknowledgments are needed only as an optimization, for efficient use of bandwidth. They are not needed for correctness.

The design would be correct if we eliminated all of the following:

- Packets
    1. Data Ack
    2. PKL Ack
    3. Restart Notification
    4. Restart ACK
- Flags
    1. Per Packet, Per Neighbor "Send-Flag" and "Ack-Flag", for both Data Packets and PKL Packets
    2. Per Neighbor, Restart Flags

The result is a design with only two types of packets:

1. Data Packets
2. Public Key List Packets

The volatile database consists only of the node's own sequence numbers, and the latest Data and PKL packet from each source/key pair.

The database is scanned in order. Every packet in the database is transmitted in order. If no new packet is received from a particular source/key pair, its old packet will be retransmitted every time the database is rescanned.

Since packets are transmitted without maintaining state regarding acknowledgments, there is no need for a node to be informed when its neighbor restarts. Thus the need for Restart Notification packets is also eliminated.

This modification yields a simpler, equally robust design, but it is far less efficient in bandwidth usage.

2.6.4 Hierarchical Networks

An $O(N)$ database can be impractically large in very large networks. The same trick of adding hierarchy to make a routing algorithm tractable can be used to make robust flooding practical. In this section we present a design for accomplishing flooding in a hierarchical network. We

present a two level hierarchy, though the scheme can be easily extended to arbitrary numbers of levels, for even larger networks.

Topology and Addressing

The network will be partitioned into subnetworks, such that each subnetwork is of manageable size.

Addressing will be hierarchical, consisting of two parts,

SUBNET This portion specifies which subnetwork the node belongs to.

NODE This portion specifies the individual node within the subnetwork SUBNET.

Within a subnetwork, all nodes will have the same value for the SUBNET portion of their address, and all will have distinct NODE values.

There will be two types of routing:

1. Level 1 routing – this type of routing concerns itself with all the individual nodes and links within a subnetwork.

2. Level 2 routing – this type of routing concerns itself with paths to subnetworks, and the subnetwork consisting of level 2 routers, but does not concern itself with the details inside of subnetworks[4].

Nodes that participate only in level 1 routing are known as "level 1 routers". Nodes that participate in level 2 routing are known as "level 2 routers". Level 2 routers reside in a subnet, and additionally participate in level 1 routing within the single subnetwork in which they reside.

The subnetwork consisting of level 2 nodes is known as the "level 2 subnetwork".

General Routing Pattern

When source and destination nodes of a packet are in the same subnetwork, the packet is flooded only within that subnetwork. When source and destination are in different subnetworks,

1. First the packet is flooded throughout the source subnetwork.

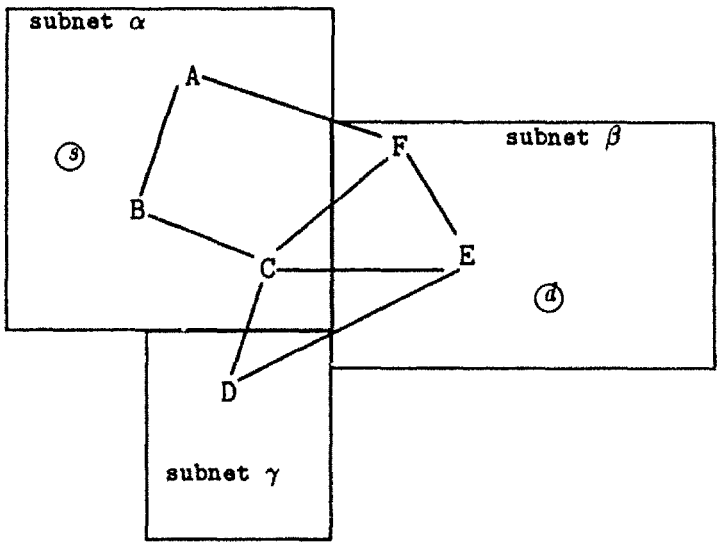
[4]Note that we assume the subnetwork of level 2 routers is connected directly (though not fully connected). In some models of hierarchical networks, the level 2 routers are connected to each other via paths through subnetworks.

2. Next the packet is picked up by each of the Level 2 routers in the source subnetwork, and signed.

3. Each of those Level 2 routers flood their copy of the packet throughout the Level 2 subnetwork.

4. Each of the Level 2 routers in the destination subnetwork pick up and sign each copy of the packet, and flood them within the destination subnetwork.

Thus the destination will receive multiple copies of the packet, equal to the product of the number of level 2 routers in the source subnetwork and the number of level 2 routers in the destination subnetwork.



For example, in the picture above, source s generates a packet with destination d. The SUBNET portion of s's address indicates that s resides in subnet α. The SUBNET portion of d's address indicates that d resides in subnet β.

The packet is first flooded throughout subnet α, using the resources and signed by the public key of s. Since the destination address indicates the destination is in a different subnet, all level 2 routers residing in subnet α, i.e. A, B, and C, pick up the packet for flooding throughout the level 2 subnetwork.

Each of A, B, and C independently supply their signature and sequence number to the packet. Since the nodes in the level 2 subnet, and the nodes in subnet β have no knowledge of the ultimate source s, s's identity, sequence number, and signature are at this point irrelevant

pieces of information in the packet, from the point of view of the Network Layer[5].

Thus each of A, B, and C independently overwrite the header fields:

- source node
- sequence number
- public key
- packet signature

with their own ID, sequence number, public key, and signature.

At this point, the packet becomes three independent packets as far as the level 2 subnet can detect.

When one of the level 2 routers in subnet β (E or F) receives a flooded packet destined for subnet β (as indicated by the SUBNET portion of the "destination node" field in the packet header), it picks up the packet for flooding throughout destination subnet β. As before, since nodes A, B, and C are not known within subnet β, the fields overwritten by A, B, and C (source node, sequence number, etc.) are now irrelevant inside of subnet β. Thus each of E and F independently overwrite the same header fields:

- source node
- sequence number
- public key
- packet signature

with their own ID, sequence number, public key, and signature.

Since E and F cannot correlate the packets they receive from A, B, and C as all having originated from the same ultimate source, (because they cannot remember more than a constant number of packets from each level 2 router, and the packets from A, B, and C resulting from s's original packet may arrive at different times), each of E and F will originate three separate packets into subnet β, one for each packet of A, B, and C.

[5]Of course the higher layer process at d which is the ultimate destination of the packet will want to know the identity of the ultimate source, but that can and should be handled at a higher layer.

Databases

As in the non-hierarchical flooding scheme, nodes within a subnetwork keep state about all the other nodes in their own subnetwork. In other words, public keys are kept for, and buffers are reserved for, each other node in the subnetwork. Level 2 routers participate within a single subnetwork, but in addition keep state about all the other level 2 routers. A level 2 "trusted node service" broadcasts public keys for all level 2 routers, within the level 2 subnetwork. Each level 2 router keeps public keys and buffers for each other level 2 router.

Why This Works

- Flooding within a subnetwork (Source ID and Destination ID have identical "SUBNET" fields) works identically with flooding in a nonhierarchical network.

- When Source and Destination are in different subnets, each level 2 router acting on behalf of the source subnetwork must guarantee fairness for all sources within that subnetwork. Then each source in the source subnetwork is guaranteed some resources within the level 2 subnetwork ($1/N$ of the resources guaranteed to the level 2 router, where N is the number of level 2 routers).

  Each level 2 router R which introduces level 2 traffic into the destination subnetwork must guarantee fairness to each level 2 router (each "source" in the level 2 subnetwork), for the introduction of traffic into the destination subnetwork. Then each level 2 router will be guaranteed $1/M$ of the resources guaranteed to R within the destination subnetwork, where M is the total number of level 2 router ID/key pairs within the level 2 subnetwork.

  This assures that source subnetworks are guaranteed access into the destination subnetwork. Thus the fraction of bandwidth guaranteed to source node S within a foreign destination subnetwork is the fraction of bandwidth guaranteed per node in the source subnetwork, times the fraction of bandwidth guaranteed to each level 2 router in the level 2 subnetwork, times the fraction of bandwidth guaranteed per node in the destination subnetwork.

2.6.5 Flooding Without Network Layer Cryptography

It is interesting that a flooding scheme can meet the Byzantine robustness criteria without Network Layer cryptography and without $O(N)$ buffers[6]. We present here a scheme that theoretically accomplishes the robustness goal without using Network Layer cryptography, but is totally impractical due to the negligible performance it achieves.

This scheme requires use of reasonably accurate elapsed time timers (as opposed to globally synchronized clocks). Some threshhold, say 10%, is set, such that if a node's elapsed time clock is not within that percentage of true elapsed time, the node would be considered faulty.

The robustness achieved by this scheme is "A packet from nonfaulty source A to nonfaulty destination D will have a high probability of reaching D provided that at least one path of nonfaulty processors and links connects A and D, regardless of the number of other faulty components in the network".

A Priori Knowledge

The manually configured information required at each node consists of:

- H – the maximum path length in the network
- PktSize – the maximum sized data packet to be processed
- Nbrs – the maximum number of neighbors of any node in the network
- ClkTol – the maximum allowable ratio of measured time between two nonfaulty processors. To ensure this, we require the ratio of time measured by any nonfaulty processor to true elapsed time to be between $1/\sqrt{\text{ClkTol}}$ and $\sqrt{\text{ClkTol}}$.

A data packet contains the following:

source node	The identity of the source node
destination node	The identity of the destination node
remaining hops	The number of hops further this packet is allowed to traverse
user data	to be delivered, but not interpreted by the Network Layer

Each node keeps H buffers, each of size PktSize. Each buffer is reserved for packets with a specific hop count. A packet occupies the buffer corresponding to the hop count specified in the packet's "remaining hops" field.

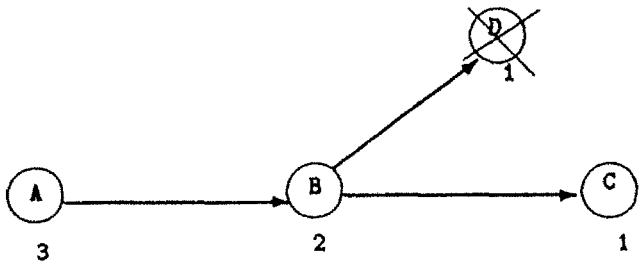
[6]Cryptography at a higher layer will still be needed to recognize falsely injected packets and corrupted packets, but the point in this section is that the Network Layer, and in particular intermediate nodes which are forwarding packets, do not need to use cryptography.

Since routing is via flooding, each packet must be transmitted to all neighbors except the one from which it was received. Thus associated with each buffer is a flag for each neighbor, indicating whether the packet occupying the buffer still needs to be transmitted to that neighbor. When all flags are clear, the buffer is free for acceptance of another packet with the specified "remaining hops".

When a node S initiates a packet, it initializes "remaining hops" to H[7]. When a node receives a packet, it decrements "remaining hops". If "remaining hops" becomes 0, or is not in the proper range (0 through H), the packet is discarded.

The hop count strategy of organizing buffer pool is taken from the Merlin-Schweitzer deadlock avoidance scheme [MS]. The M/S scheme does prevent deadlock (in the absence of Byzantine faults in the network), but it does not assure fairness. To ensure fairness, we employ a round-robin use for each buffer by each neighbor. Each neighbor in turn is given the opportunity to transmit a packet with hop count $i+1$ when buffer $\#i$ becomes vacant. Thus additional state is kept with each buffer, indicating the last neighbor from whom a packet was received for that buffer.

The M/S scheme, with per neighbor fairness added, assures fairness and packet progress in the absence of Byzantine failures. However, a single Byzantine failure in the network can cause zero throughput on a nonfaulty path in the network.



In the picture above, the path A-B-C is nonfaulty, and A has a packet with "remaining hops" of 3, waiting to be transmitted to B. Nonfaulty B has a packet with "remaining hops" of 2, waiting to be transmitted to D, who is faulty and will never accept a packet from B. Thus A

[7]Actually, S could initialize "remaining hops" to any number H or smaller. Since timer values, as explained later, increase exponentially with the value of "remaining hops", S could first attempt to reach the destination with smaller values for "remaining hops", and increase the initial value of the field until it successfully reaches H. However, given that the scheme obviously has no practical utility, optimizations to it seem unwarranted.

will never be able to transmit to B even though B is nonfaulty, with this scheme. To prevent Byzantine failures from blocking the progress of a packet through a nonfaulty path, we must also add timers.

We make the rule that a nonfaulty processor **must** process a packet within some time limit, say 1 time unit. Thus, if a processor P holds a packet with "remaining hops" of 1, it will expect that the processor Q to which the packet is queued will be ready to receive the packet within 1 time unit * the number of Q's neighbors. The multiplicative factor of the number of neighbors is required due to the round robin scheduling of the buffer.

Since P does not know how many neighbors Q has, P multiplies by "Nbrs", the manually configured parameter that gives an upper bound on the number of neighbors of any node in the network. Also, since we only require elapsed time estimates to be within ClkTol, an additional factor of ClkTol is multiplied to the waiting time.

Thus, if processor P holds a packet with "remaining hops" of 1, waiting to be forwarded to processor Q, P waits ClkTol*Nbrs for Q to receive the packet. If Q fails to request a packet from P with hop count 1 within that time, P drops the packet.

Unfortunately, each extra hop requires an additional factor of Nbrs. Thus if processor P holds a packet with "remaining hops" of k, queued for neighbor Q, P must hold the packet for $(\text{ClkTol}*\text{Nbrs})^k$ time units before discarding it, in case Q has not yet requested the packet.

This scheme can be shown to meet the correctness conditions by induction:

> Suppose a nonfaulty path $A_1, A_2, \ldots A_k$ exists between nonfaulty processors S and D. Assume that the path is exactly H long, and that S initialized "remaining hops" to H. Since D, assuming it is nonfaulty, will process each packet within 1 time unit, and since D has at most Nbrs neighbors, D will accept the packet within Nbrs time units (times the fudge factor of ClkTol).
>
> Assuming A_k is nonfaulty, it will forward each packet occupying the buffer for "remaining hops" of 1 each time the neighbor to which it is queued requests it. Since a neighbor is required to request a packet with "remaining hops" of 1 within Nbrs time units, from each neighbor, A_{k-1} can expect that A_k will be ready for receiving the packet within an extra factor of Nbrs.
>
> If the path is actually shorter than H, the proof still holds. The only consequence of setting "remaining hops" higher than necessary (though a value greater than

H is illegal and would be discarded), is that the timers are longer than would be necessary for the S to D path.

This scheme has the obviously undesirable property that the throughput on a nonfaulty path of m hops is only guaranteed to be one packet for every $(\mathrm{ClkTol}*\mathrm{Nbrs})^m$ time units, making this scheme clearly without practical utility.

An additional point of note is that the scheme does not necessarily yield a storage advantage over the crytographic, per-neighbor scheme discussed in the main portion of this chapter, since H is likely to be $O(N)$. In order to meet the robustness requirement that nonfaulty node pair A and B should be able to communicate provided that *any* nonfaulty path between them exists, then H must be at least as large as the *longest* possible path between any pair of nodes in the network. Thus H is quite likely to be very close to $N - 1$, the longest possible path in any network of N nodes.

Chapter 3

Robust Link State Routing

3.1 Overview

A Link State Algorithm consists typically of the following steps:

1. Each node probes its neighbors, in order to discover the identity and state of the link to each of its neighbors.
2. Each node constructs a "Link State Packet", containing its own identity and a list of its neighbors.
3. These Link State Packets (LSPs) get flooded throughout the network.
4. Each node stores a copy of the latest LSP from each other node in the network
5. The LSP database gives complete topological information about the network, and each node uses its database to calculate paths in the network.

In Chapter 2 we presented a design of a Network Layer based on flooding in which communication between nonfaulty processors A and B was guaranteed provided that any nonfaulty path existed between A and B. Unfortunately, that scheme required that every packet traverse every link.

In this chapter we present the design of a Network Layer in which packets traverse a specified path. In this way, if a source S has packets for destinations D1 and D2, where the path to D1 does not intersect the path to D2, then the bandwidth used by the S-D1 conversation does not diminish the bandwidth available for the S-D2 conversation. This increases the total bandwidth available (over that provided by robust flooding) to S by a factor that (depending on topology) can be as great as the total number of links in the network.

The components of our Link State based Network Layer are:

- We utilize a "trusted node service" exactly as in Chapter 2, so that we may assume each node has a complete list of (ID, key) pairs, including (but unfortunately not necessarily restricted to) all the real nodes in the network.
- We require some subtlety in the protocol for automatic discovery of the identity of neighbors, to defend against Byzantine nodes creating fictitious links between nonfaulty nodes.
- We use the flooding design in Chapter 2 for propagation of Link State Packets.
- We include a route in the header of a packet, generated at the source Network Layer. This allows a packet to be routed successfully even if Link State databases at nodes are not synchronized, and allows alternate paths to be explored easily if the primary route chosen does not work.
- We use computation of node disjoint paths and fault isolation of a faulty path, to facilitate selection by the source of a nonfaulty route.

Our design assures that communication between nonfaulty nodes A and B will succeed provided that the number of node disjoint paths connecting A and B is greater than the number of Byzantine failures in the network. In practice, however, the network will approach the robustness exhibited in Chapter 2 (if a path of nonfaulty processors connects nonfaulty nodes A and B, they will be able to communicate, regardless of the total number of Byzantine faults in the network) when fault isolation is used to intelligently eliminate suspicious nodes from the database.

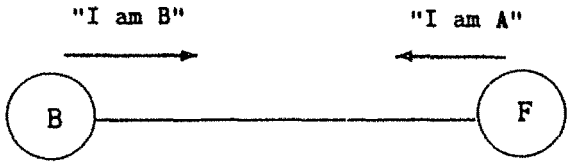
3.1.1 Manifestations of Byzantine faults

Byzantine behavior can create problems for many different aspects of a Link State Algorithm. These problems are briefly introduced here, but discussed at length after the description of the algorithm.

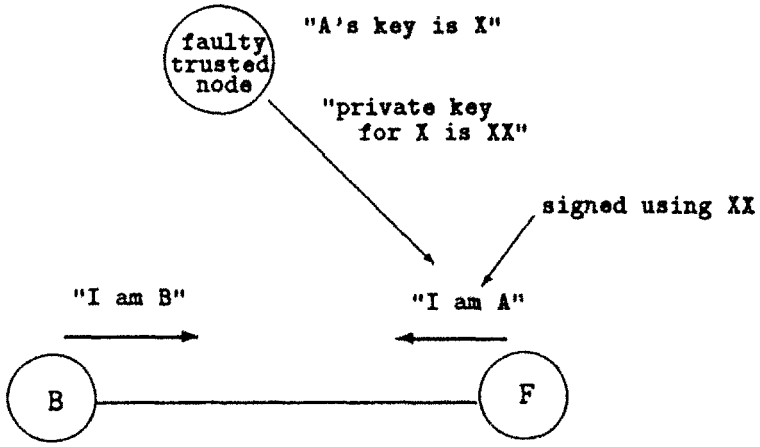
Discovering Neighbors

In a traditional Link State algorithm, discovering the identity of neighbors is simple – a handshake is executed, in which each endpoint of a link transmits a packet with its own identity across the link.

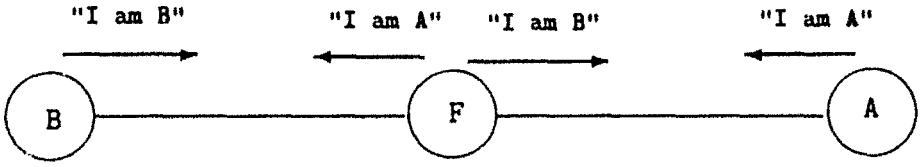
If the neighbor F is faulty, however, F can incorrectly identify itself.



If identification messages are signed, F may be prevented from claiming to be a nonfaulty node A. However, if F conspires with a faulty "trusted node" that constructs and broadcasts an incorrect public key for A, F can use A's incorrect public key and sign an identification message claiming to be A.



Even without collusion with a faulty trusted node, a faulty node F can confuse its nonfaulty neighbor B into thinking B has a link to nonfaulty node A. This can be accomplished, assuming both A and B are neighbors of F, by having F act as a "dumb relay", forwarding messages between A and B.



Our safeguards against such behavior are:

- Identification Messages (which identify a node to its neighbor) explicitly contain the public key being used by the neighbor. Thus a neighbor is an (ID,key) pair, not simply an ID.

  If A's true public key is key_1, and a faulty trusted node broadcasts key_2 for A (and also shares the associated private key for key_2 with a faulty network node F), then F can masquerade as the pair (A,key_2), but F cannot simulate the true node A, because that is pair (A,key_1).

- All messages except identification messages between neighbors are encrypted, using the receiver's public key (so that a faulty neighbor, acting as a "dumb relay", cannot discriminatorily forward packets).

  Currently, private key schemes are more efficient than public key schemes. A private key can be used for encryption between neighbors. Once a pair of neighbors discover each other's identity through use of public key signatures, they can use public key encryption to exchange information allowing them to share a private key, and future encryption on the link can be accomplished with private key technology. [GMT]

- Data messages are acknowledged hop by hop, and statistics are kept by the transmitting node to determine the reliability of the link. If the link does not satisfactorily forward data packets, perhaps because the link includes a faulty neighbor acting as a "dumb relay" for unencrypted identification messages, but not data messages, then the link is no longer considered operational.

Propagating LSPs and Public Keys

Propagation of LSPs and Public Key List Packets is done through flooding. As seen in Chapter 2, there are many attacks a faulty node can execute that will interfere with traditional flooding algorithms. However, as described in Chapter 2, there are also satisfactory solutions to all the threats. Thus the design in Chapter 2 can be used to ensure Byzantine behavior will not interfere either with propagation of LSPs or Public Keys.

In the ARPANET, propagation of LSPs is accomplished with a scheme based on flooding [MRR]. An improved design was described in [Pe]. Surprisingly, the flooding design in Chapter 2 is simpler than the ones recommended in [MRR] and [Pe], which is ironic since the ones in [MRR] and [Pe] are not robust against active malfunctioning nodes.

The flooding design in Chapter 2 is simpler because:

- Sequence Number wraparound need not be considered. Given a large enough field, sequence number wraparound will only occur because of Byzantine failure by the source itself. After such an event, it is reasonable to require manual intervention, in the form of reregistering the source with a new public key, with the "trusted service". Thus the manipulation of sequence numbers becomes much simpler.

  Manual intervention of the same form is not possible in a truly distributed scheme such as in [Pe], since there is no central repository of stable storage (no analog to the "trusted node service").

- There is no need for an age field in Link State Packets. There is never any reason to purge LSPs, or consider them "too old". If the information in them is out of date then the source will issue a new LSP, unless the source is not reachable. If the source is not reachable, then the contents of its last LSP are irrelevant, because there is no path to it.

- If a node really has permanently left the network, its LSP need not clutter all the other nodes' databases, since de-registering the departed node with the "trusted service" will signal other nodes that they no longer should keep that LSP. (They only keep LSPs for nodes registered with the trusted service, and for which they have an up-to-date public key).

However, this simplicity is bought at a price. The additional cost of the Byzantine Update Process is:

- Nodes must have more stable storage. The only a priori knowledge required in the scheme in [Pe] is a node's own ID. In the Byzantine Update Process additional stable storage is required for the node's own private key, and the public keys of the "trusted nodes".

- Several nodes must be reserved as "trusted nodes"

- New nodes must register public keys with the trusted nodes before they can operate in the network

- Other nodes must verify signatures on all LSPs.

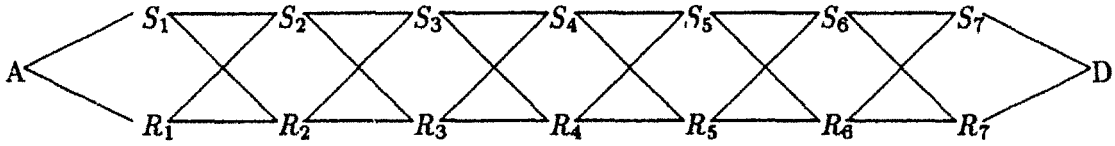
Calculating Routes

Since route calculation is not done in a distributed fashion, a faulty node cannot interfere with the route calculation done by a nonfaulty node as long as the database from which the nonfaulty node calculates routes is reasonably correct.

However, with Byzantine faults, there is no way to know what subset of the reported topology is truly operational. A node can operate correctly in all aspects of the protocol except forwarding of data packets. Thus we provide the ability for the source to calculate multiple routes, and specify a route to the destination in each packet.

Unfortunately, the number of routes that might need to be tried before a working route is found can be too large for practical purposes.

For instance, consider the following topology:



In the above diagram, there are 2^7 paths between A and B, since at hop i, either R_i or S_i can be chosen. (Actually there are more paths than that if paths longer than 8 hops are considered.)

A simple failure of a node or link will be detected by the routing algorithm, and the network will automatically calculate routes that do not include the nonfunctional component. However, a Byzantine failure is harder to detect, since a node may participate correctly in all aspects of the Network Layer protocol, and then fail to forward packets. Thus the Network Layer cannot automatically calculate *functional* routes.

With flooding, any number of faults is tolerated, provided that at each hop, at least one of R_i or S_i is nonfaulty. With explicit route calculation, if there were 7 failures, such that exactly one of R_i or S_i were faulty for each i, then 2^7 paths must be tried to discover the one nonfaulty path.

It is possible to calculate a set of independent paths, using the Max Flow problem [Orl], [GT]. In the above topology, given any path, there is only one other independent path between A and B. If there were a single Byzantine failure in the above topology, switching to the second independent path will guarantee a functional route, with no need to identify the malfunctioning

component. However, if there were more than one Byzantine failure, there might still be many functional routes, but the Max Flow solution will probably not find a working one, since it will only calculate 2 routes.

There are various approaches that can be taken to deal with this problem.

- Require stronger conditions to be met by the network. For instance, limit the number of faults to be tolerated to be less than some constant c, and require the network to have more than c independent paths between each pair of nodes.

  With this approach, the Network Layer only guarantees delivery of packets between nonfaulty sources A and B when the number of Byzantine failures in the network is less than the number of node disjoint paths connecting A and B.

- Use flooding as a backup – if the route calculated by the Network Layer fails to work, then select flooding as an option for delivery on a per packet basis.

  This approach yields a Network Layer as robust as one built entirely on flooding. In other words, it guarantees delivery of packets between nonfaulty sources A and B provided that any nonfaulty path connects them. Unfortunately, its performance in the presence of many Byzantine faults is no greater than the simpler design, of building the Network Layer entirely on flooding (as in Chapter 2).

- Use fault diagnosis as a backup – if a route fails, then the source must investigate to discover and report the problem node or link.

  This form of Network Layer has the advantage in practice that Byzantine failures will be discovered and reported before so many occur that the network no longer functions. However, since each Byzantine failure also casts suspicion on a potentially nonfaulty node (if the 4th hop on a packet's route claims to have received the packet but the 5th hop claims not to have received it, then either 4 is faulty for not forwarding the packet or 5 is faulty for not receiving it), it take impractically many attempts to find a functional route, in the presence of many Byzantine faults. Thus the theoretically attainable robustness of this approach is the same as the disjoint paths approach, but in practice this approach will be even more robust than flooding, since Byzantine faults will be detected.

3.2 A Robust Link State Design

3.2.1 Overview

We use the same mechanism as in Chapter 2 to broadcast a list of (node, key) pairs. Thus we can assume that every node knows the set of network nodes, together with public keys for each node. Also, as in Chapter 2, because of the possibility of a Byzantine "trusted node", some of the (node, key) pairs may be incorrect, though we assume that the list of true (node, key) pairs is contained in the received list.

We provide a mechanism for each node B to discover the identity of its neighbors. This information is included in a "Link State Packet", which is generated by B and flooded through the network, again using the mechanism described in Chapter 2.

The Link State Packet database contains complete topological information about the network, and each node uses its database to compute routes. In traditional packet switching networks, each node makes an independent decision as to the direction in which a packet should be forwarded. We instead require that the entire route be computed by the source node. Then the source includes the route in the packet header, and the packet is routed according to the source's computed path.

Thus we have added a new type of packet, a "Link State Packet", and we have modified the Data Packet to include a route. Routing of Data Packets is very similar to routing in Chapter 2, in that sequence numbers are still used, and only the data packet with highest sequence number is stored from each (source, key) pair. The only modification is that, when a new Data Packet is received, only one "Ack-Flag" will be set (the one for the neighbor from whom the packet was received, which must also be the neighbor preceding this node in the route or the packet will be deemed invalid and dropped), and one "Send-Flag" will be set (the one for the next node specified in the route).

Other modifications include:

- We require stable storage for Data packet sequence numbers. The reason for this is that we do not wish Data packets to be flooded in this scheme (otherwise, the design in Chapter 2 would suffice), and so we did not have a satisfactory means for a source to reacquire its own sequence number after a failure.

- We use hop by hop encryption, so that in the case of a faulty node acting as a relay, creating a link between two nonfaulty nodes, the relay node will not be able to discriminatorily

corrupt or drop packets.

- We add an additional packet type "Destination Data Packet Ack" so that the source Network Layer can discover, without hints from the client layer, whether the computed route successfully delivers packets. All previous Acknowledgment packets have been neighbor to neighbor only.

3.2.2 Packet Types

To prevent two nonfaulty nodes, A and B, from mistakenly assuming they have a functional link between them, when instead the link is being simulated by a faulty node acting as a relay, B encrypts all packets it is forwarding to A using A's public key. Upon receipt, A decrypts all packets it receives. In this way, if a faulty node is acting as a relay, it cannot distinguish packets, and as such cannot discriminatorily fail to forward packets based on such characteristics as type of packet, source node, or destination node. To prevent discrimination based on packet length, all packets should be padded to maximum length.

All packet types below are encrypted and decrypted at each hop, except for the "Identification" packet, which is sometimes not encrypted.

Identification

The purpose of an "Identification" packet is to inform the node at the other end of a link of this node's identity. The public key is also included, because a node is really defined as an ID/key pair, to differentiate the real node from an imaginary node with the same ID and different key, that can be created by a faulty trusted node.

This is the only form of packet that is sometimes not encrypted as it is transmitted over a link. An "Identification" packet is transmitted only between neighbors, and never forwarded (except perhaps by faulty nodes).

identity	The identity of the node
public key	The node's public key

Public Key List Packets

A "Public Key List" packet is generated by each "trusted node" and it contains the identities of, and public keys for, every node in the network (other than "trusted nodes", whose keys must be manually maintained at every node in the network). It is flooded, using the robust

flooding defined in Chapter 2. It contains:

source	The trusted node that generated this packet
sequence number	Assigned as in a data packet by "source"
key list	Identity/public key for each node in the network
signature	A digital signature, verifiable based on the public key for the trusted node, manually maintained at each node, covering the entire contents of the packet.

LSP

The purpose of a Link State Packet (LSP) is to inform other nodes about the links in the network. Each node generates a Link State Packet identifying its own neighbors. This packet is flooded, using the robust flooding in Chapter 2. Each node keeps the most recently generated Link State Packet from each other node in the network.

A Link State Packet must be smaller than some network-wide defined maximum length. It contains the following fields:

source node	The identity of the source node
destination node	The identity of the destination node
sequence number	Assigned by the source node in a monotonically increasing way for each packet generated by that source, (regardless of destination address), except after a loss of state by the source node.
public key	The public key under which this LSP has been signed.
neighbor list	A list consisting of the identities of, and costs to, each neighbor of this node.
signature	A digital signature, verifiable based on the public key listed in the packet, covering the entire contents of the packet.

Data Packet

The next form of packet is a data packet. It is generated by a layer higher than the Network Layer, and passed to the Network Layer for delivery to a specified destination node.

There is a network-wide maximum length for a data packet. A data packet contains the following information:

source node	The identity of the source node
destination node	The identity of the destination node
sequence number	Assigned by the source node, in a monotonically increasing way, except after a loss of state by the source node.
public key	The public key under which this packet has been signed.
route	A route, written by the source node, that the packet is to follow. The route consists of a sequence of node ID/public key pairs, or else is omitted, indicating the source wants this data packet to be flooded.
user data	Not interpreted by Network Layer.
signature	A digital signature covering the entire contents of the packet.

Note that the sequence number node A writes into the LSPs it generates is totally independent of the sequence number node A writes into the data packets that it generates.

Destination Data Packet Ack

The purpose of a Destination Data Packet Ack is to inform the source Network Layer that the packet was received by the destination Network Layer. Since the Network Layer is a datagram service, without an explicit acknowledgment from the destination, or feedback from a higher layer protocol, there is no way for the Network Layer to know whether a route works.

It is quite likely for a network to require feedback from the higher layer to the Network Layer, so that the responsibility would be placed on the higher layer to discover when a route did not work. In this case, the Destination Data Packet ACK would not be necessary.

The Destination Data Packet Ack contains the following:

source node	Source of data packet being Ack'd
destination node	The destination of data packet being Ack'd
sequence number	Sequence number from data packet
source key	The public key of the data packet's source.
destination key	The public key of the data packet's destination.
route	copied from data packet
signature	Copied from data packet
Ack signature	A digital signature covering the entire contents of the ACK packet, generated by the Destination (which is generating the ACK packet).

The Destination Data Packet Ack travels along "route", but in the reverse direction. If "route" indicates the data packet was to be flooded, then the Destination Data Packet Ack is also flooded.

The Ack packet is stored with the source's data packet, rather than stored with resources associated with the destination node.

LSP-ACK

The next form of packet is an LSP-ACK packet. Its purpose is to acknowledge receipt of a particular LSP from a neighbor. It is transmitted only between neighbors, and never forwarded.

Note that if public key hop by hop encryption is done, the node generating the LSP-ACK needs to sign the packet, (see "ACK-signature" field, below), to protect against a faulty node, F, acting as a relay on the link between two nonfaulty nodes, A and B, generating LSP-ACKs on behalf of B and thereby lowering the reliability of the LSP flooding, by making A assume the LSPs it had forwarded successfully reached B.

Although an LSP-ACK transmitted by B and sent to A is encrypted, so that F cannot distinguish an LSP-ACK from any other encrypted packet, encryption alone does not prevent F from forging an LSP-ACK using A's public key. Thus it is necessary for B to sign the LSP-ACK, even though B also encrypts it.

However, if private key hop by hop encryption is done, for efficiency, then the ACK-signature is not needed.

source node	The identity of the LSP's source
sequence number	The LSP's sequence number
public key	The public key under which the LSP was signed
data signature	Copied from LSP
header signature	Copied from LSP
ACK-signature	Signature proving neighbor generated the ACK

Restart-Notification

The next form of packet is a Restart-Notification. Its purpose is to inform a node M's neighbor, that M has lost state.

The only information in a Restart-Notification is the packet type ("Restart-Notification").

Restart-Notification-ACK

The next form of packet is a Restart-Notification-ACK. Its purpose is to inform a node M's neighbor B, that B has received M's Restart-Notification message.

No information is necessary within a Restart-Notification-ACK except enough to identify the packet as a Restart-Notification-ACK.

Note that even though hop by hop encryption is assumed (to defend against a faulty node F acting as a relay between nonfaulty nodes M and B), F can capture a previous Restart-Notification transmitted by M, and cause extra traffic on the M-B link because B will assume the entire packet database needs to be retransmitted to M (since M restarted). Also, F can capture a previous Restart-Notification-Ack transmitted by M, fail to forward B's Restart-Notification (if it can guess which packet is a Restart-Notification), and transmit M's Restart-Notification-Ack to B, causing M not to retransmit the packet database to B.

However, neither of these threats cause great harm to the network, so we do not add extra mechanism into our main scheme for this purpose. This is discussed further in Section 3.6.3.

3.2.3 Stable Storage for Sequence Numbers

In Chapter 2, we did not require stable storage for a node's own sequence numbers, since the mechanism whereby a node reacquired its own sequence number after a crash was simple and low cost. With flooding, the way a node reacquires its sequence number is:

- The packet with largest sequence number is flooded everywhere, and reflooded when portions of the network lose state, so a recovering source would automatically receive its own packet with highest sequence number (assuming network nodes don't lose synchronization with each other).

- In case nodes lose synchronization, and fail to reflood the largest sequence number after a portion of the net loses state, the mechanism of reflooding upon receipt of a packet with lower sequence number causes the source's largest sequence number to be reflooded back to the source as soon as the source issues its first post-crash packet.

- The use of the signature field (plus data field, in the case of a Public Key List packet) as a tie breaker handles the case where the post-crash and pre-crash sequence numbers are the same.

As a result of the above, in our robust flooding design, stable storage was required only for information that remained reasonably static, and only changed with manual modification. In fact, the storage required in Chapter 2 could have been read-only technology. In contrast, a node's own sequence number changes dynamically, and must be automatically updated in the stable storage. Thus this requirement does indeed have implications beyond the additional space required.

In this chapter we do require a node to keep an "estimate" of its own sequence numbers on stable storage. The reason for this will be discussed in detail later on in the chapter. Briefly, we require this because an extension of the reaquisition mechanism for sequence numbers on flooded packets into a scheme for directed data packets would allow a Byzantine node to easily increase the traffic in the network, to the point where this scheme would approach the data traffic burden of a scheme in which data packets are flooded.

Given that this chapter requires stable storage for a sequence number for data packets, there is no reason for avoiding requiring a node to keep similar state regarding its sequence numbers for PKLs and LSPs. Keeping state regarding PKL and LSP sequence numbers offers a modest simplification to flooding.

Depending on the technology, it might be inconvenient to update the stable copy of the sequence number after every packet. Indeed there is no reason to keep the stable copy consistent with the sequence numbers actually in use. The only constraint is that the copy on stable storage be larger than the one in use, so that if the node restarted and used the number on stable storage as a starting point, it would be guaranteed to be greater than any sequence number the node had generated in the past.

Thus the sequence number on stable storage can be set to, say, 100,000 more than the current value, and when the sequence number in use gets close to the one on stable storage, the one on stable storage can be increased by another 100,000.

3.2.4 Propagation of Public Keys

Our design requires knowledge of public keys for every node in the net. This is accomplished as described in Chapter 2.

Since we have required keeping a value for a nodes' own sequence number for data packets on stable storage, we might as well require the node to additionally keep a similar value for use on its own Public Key List packets, in the case of a restart.

No design simplifications result. To check for a Byzantine "trusted node", we still require nodes to compare signatures and data in Public Key List packets to determine whether two packets with the same sequence number are indeed duplicates. If indeed reuse of the same sequence number was detected, (given that we are requiring nodes to keep their own sequence numbers on stable storage) we might record such packets so that the "trusted node" could be investigated for malfunction.

Given that in the robust flooding design, the mechanism for reacquiring a sequence number was so simple and efficient, we did not think the increase in use of stable storage was warranted in Chapter 2. We have only required the additional stable storage here because it was already required for data packet sequence number.

3.2.5 Information in Non-Volatile Storage

Each node must have stable storage for the following information which is manually entered and maintained:

its own identity
its own keys (public and private)
N, an upper bound on the total number of network nodes
the identities of, and public keys for, each of the "trusted nodes"
the size of the maximum sized LSP (or alternatively, the maximum number of neighbors any node can have, from which the maximum sized LSP can be derived)
the size of the maximum sized data packet

In addition, the following information must be kept in stable storage, but this information is dynamically and automatically maintained.

a safely large data packet sequence number to use on restart
a safely large LSP sequence number to use on restart
a safely large PKL sequence number to use on restart

3.2.6 Dynamic Database

Each node keeps the following database in volatile storage:

OWN-LSP-SEQNUM	The next sequence number to be assigned when this node generates its own next LSP
OWN-DATA-SEQNUM	The next sequence number to be assigned when this node generates a data packet
OWN-PKL-SEQNUM	The next sequence number to be assigned when this node generates a PKL packet. This is kept only by nodes acting as "trusted nodes"
LSP database	For each node/public key pair reported by any trusted node: • The LSP with highest sequence number from that node/public key • For each neighbor of this node, two flags: 1. Send-Flag – indicating whether this LSP needs to be transmitted to this neighbor 2. Ack-Flag – indicating whether an ACK for this LSP needs to be transmitted to this neighbor
data packets	For each node/public key pair reported by any trusted node: • The data packet with highest sequence number from that node/public key • For each neighbor of this node, two flags: 1. Send-Flag – indicating whether this data packet needs to be transmitted to this neighbor 2. Ack-Flag – indicating whether an ACK for this data packet needs to be transmitted to this neighbor Note that for data packets (other that ones for which the "flooding" option was selected), Send-Flag will be on for at most one neighbor, for any particular data packet, and likewise Ack-Flag will be set for an most one neighbor, for any particular data packet. • The Destination Data Packet Ack associated with this data packet • For each neighbor of this node, two flags: 1. Send-Flag – indicating whether this end to end Ack packet needs to be transmitted to this neighbor 2. Ack-Flag – indicating whether a (next hop) ACK for this end to end Ack packet needs to be transmitted to this neighbor
neighbors	identity of and public key for each neighbor
Restart Flags	for each neighbor, "Send-Restart" and "Send-Restart-ACK" flags

3.2.7 Propagation of LSPs

Propagation of LSPs is done via the robust flooding scheme described in Chapter 2. Although the scheme in Chapter 2 is quite robust, it does not ensure absolute reliability. The justification in Chapter 2 for the adequacy of the scheme (in the absence of total reliability) was that the occasional lost packets, or forged packets, would be recovered from by a higher layer process. However, with LSP propagation, there is no higher layer process, since LSPs are destined to the Network Layer itself.

In particular, the following can occur:

- A faulty trusted node can broadcast an incorrect public key for nonfaulty node A, and can then forge packets on behalf of A, which would get delivered *in addition to* packets from genuine source A.

  We prevent this situation from being a problem by identifying nodes by ID/public key pair. In other words, if there are two public keys broadcast for node A, then the Network Layer treats (A/key1) as one node, and (A/key2) as a different node.

- A faulty trusted node can broadcast up to N incorrect node/key pairs, and then generate LSPs on behalf of all those node/key pairs. In this case, enough of the LSP database is incorrect that it might be thought difficult for any node to compute a correct route.

  This is not a problem because we partition the LSP database and compute paths based on the assumption of correctness of a particular "trusted node". As a consequence, if one or more faulty trusted nodes broadcast imaginary nodes, or imaginary ID/key pairs, the extraneous information will be filtered with an efficiency degradation factor equal at most to the total number of trusted nodes.

  In other words, if only one of the t trusted nodes is nonfaulty, and all the others broadcast imaginary ID/key pairs and simulate the imaginary nodes by generating LSPs on behalf of them, then only $1/t$ of the LSPs in the database are legitimate. If a node were to naively try to calculate routes from an unstructured version of the database, it would be very unlikely to find a correct route.

  If instead it did t different calculations, each assuming a different one of the trusted nodes was nonfaulty, one of the calculations would be operating on legitimate data.

  When using the view of trusted node T:

1. Ignore LSPs generated by node/key pairs not listed by T.

2. Ignore links to node/key pairs not listed by T, even if reported in the LSP of a node/key pair listed by T.

- LSPs take time to propagate, and during the propagation time of an LSP, databases at nodes will be different. Non-identical LSP databases will not be a problem with our scheme. We use source routing rather than hop by hop incremental decisions on a packet's route. Therefore, if the database at the single source node is reasonably correct, and the source is able to calculate a working route, the state of the databases at other nodes is of no relevance.

- A nonfaulty source F may issue two packets in quick succession, in which case the one with the higher sequence number may "overtake" the one with the lower sequence number, resulting in the earlier packet not being delivered.

  In the case of LSPs, this is exactly the service we want, so it is not a problem. We only want the latest LSP to be delivered.

- A faulty node may include incorrect information in its own LSP.

  This is not a problem because we recognize that a path may not work due to Byzantine fault of some component along the computed path, and enable the source to compute alternate and independent paths, in the event that a path does not work.

3.2.8 Route Calculation

The LSP database gives each node a complete view of the network, so that any efficient graph algorithm for computing paths can be utilized, such as the SPF algorithm used in the ARPANET [MRR], based on Dijkstra's algorithm [Dk]. However, there are certain problems:

Byzantine Trusted Nodes

Not all the information in the LSP database is necessarily correct. In particular, a faulty trusted node could disseminate incorrect public keys for network nodes, and then generate LSPs on behalf of those nodes. The LSP dissemination scheme guarantees each node will have the LSP with largest sequence number from each node/key pair reported by any trusted node.

Thus each faulty trusted node can introduce up to N nonexistent node/public key pairs. If the set of node/key pairs were treated equally, then it would take an impractical number of route calculation attempts to discover a set of node/key pairs that really existed. However, if the node/key pairs are structured according to the assumption of a particular trusted node being nonfaulty, then at most t independent route calculations need be performed, one for each "view" of the network (consisting of a list of node/key pairs) reported by a trusted node. If all trusted nodes agree on the list of nodes and public keys, then only one view is necessary. If one trusted node's list differs from all the others, but the others all agree, then two views are required.

If more than one view exists, then routes to each destination can be computed for each view. Packets can be duplicated, with a copy for each route computed based on different views. Or routes can be tried in turn, with a route based on a different view attempted only when a previous route is discovered not to work. (Based on lack of receipt of a Destination Data Packet ACK.)

Byzantine Nodes

The information in the LSP database is not guaranteed to be correct, even if the trusted node's node/key list being used is correct, since a Byzantine node can generate incorrect information in its own LSP, or even generate a correct LSP but fail to forward packets correctly.

One easy defense against false information in LSPs is to require links to be two-way in order to be used for route calculation. In other words, if A's LSP reports a link to B, the link is only used in calculating routes if B's LSP also reports a link to A. In this way a faulty node F's report of a link to nonfaulty node A will be ignored by nonfaulty network nodes unless A also reports the link. Likewise, if faulty node F failed to include its link to nonfaulty node A, the network would also ignore the A-F link.

However, even if information in all LSPs was magically guaranteed to be correct (all reported links actually existed), there is still no guarantee that a computed route would work. A node along the path could still fail to forward data packets, forward data packets in the wrong direction, or corrupt data packets it forwards.

If it were known that at most k nodes could exhibit Byzantine behavior, then to be guaranteed of finding a functioning route (assuming at least one existed between source and destination), it would be necessary to try all possible ways of excluding k or fewer nodes from the

route computation. In other words, a guess would be made as to which nodes were nonfaulty, and routes computed according to the LSPs of those nodes alone. This obviously presents too many routes to attempt and still be practical.

A better method, which yields slightly less robust behavior, is to compute node disjoint paths, meaning paths between source and destination that do not share intermediate nodes.

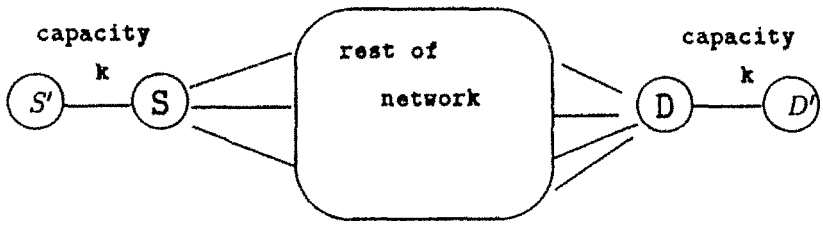
This can be accomplished with any solution to the Max Flow problem. First transform the graph into a new directed graph, as follows.

1. Represent each node M in the original graph by two nodes, M_{in}, and M_{out}.
2. For each node M in the original graph, add a directed link of capacity one in the new graph from M_{in} to M_{out}.
3. For each link in the original graph, say between nodes A and B, add a directed link of capacity infinity from A_{out} to B_{in} and a directed link of capacity infinity from B_{out} to A_{in}.

Then, use the Max Flow problem on the transformed graph. This yields the maximum possible number of link disjoint paths in the transformed graph, which maps to the maximum possible number of node disjoint paths between a pair of nodes in the original graph.

If it is desired to compute the set of independent paths with minimal cost (and still the maximum possible number of paths in the set), the Min Cost Max Flow problem, again with capacity 1 for each link, will yield the desired set. [Orl]

If some bound, k is assumed on the total number of Byzantine failures in the network, it might be desired to calculate at most $k-1$ independent paths. Again, the solutions to the Max Flow and min cost Max Flow problems can be used to calculate a limited number of paths.



The method of applying the Max Flow problem to compute at least k independent paths between source S and destination D is to do the following:

1. Mark each network link as having capacity "1".

2. Add two dummy nodes, S' and D'.

3. Add a link of capacity k between S' and S, and a link of capacity k between D and D'.

4. Solve the Max Flow (or Min Cost Max Flow) problem between S' and D'.

3.3 Neighbor Discovery

As discussed briefly in Section 3.1.1, neighbor discovery in a Byzantine environment can be difficult, since a faulty node can act as a transparent forwarder.

In some sense, this is not a problem. If a link between A and B appears to work, and only exists due to the cooperation of faulty nodes, it still does exist: A and B still communicate. However, there are two possible problems caused by such links:

1. The link might work in a discriminatory fashion, such as forwarding routing control information but not data packets.

2. The node acting as a relay could forge neighbor to neighbor control packets.

The result of these attacks is that a link now has a capability of becoming an intelligent Byzantine component in the network. This was not an issue in the flooding scheme because knowledge of the identity of a neighbor was not relevant. If a link managed to exhibit intelligent Byzantine behavior, it would be handled by the network like (and indeed could not be distinguished from) two ordinary links on either side of an intelligently Byzantine node.

The defenses against an intelligently Byzantine link in Link State Routing is:

encryption All messages transmitted over the link must be encrypted and padded to maximum length so that an observer of the messages across the link could not distinguish messages based on type, length, source address, or other characteristics, and in particular has no choice other than to forward all packets faithfully, drop an acceptably small number of packets at random, or drop a sufficiently large number of packets (again at random) to convince the nonfaulty nodes at opposite ends of the link that the link is too unreliable to report to the network.

Sometimes a link might be sensitive to specific bit patterns. If probabilistic encryption is used, then when a packet is retransmitted (due to a bit pattern in the encrypted original

that could not pass over the data link), the retransmitted packet will most likely succeed. If deterministic encryption is used, a packet retransmitted by the source will be likely to succeed since it will have a different sequence number, and thus it will encrypt to a different value.

The only packet that will not be encrypted is an "Identification" message, when transmitted while the endpoint of the link is unknown. (If the identity of the neighbor is known, or assumed known, the "Identification" message is encrypted. An unencrypted "Identification" message is transmitted when a node restarts, or when a message is received that is not properly signed, given the assumption of the neighbor's identity, or when a message is received that cannot be decrypted (indicating loss of synchronization between neighbors regarding identities).

When node B is not aware of the identity of its neighbor across a particular link, B sends an unencrypted "Identification" message across the link, though B also includes its own signature. When node A receives an unencrypted "Identification" message from neighbor B, A responds with an encrypted "Identification" message, signed using A's private key, and encrypted using B's public key.

signatures Messages generated by a neighbor must be signed by the neighbor. Otherwise, a faulty node acting as a relay could forge neighbor to neighbor control packets.

performance monitoring Every packet transmitted over a link is acknowledged with our scheme. Nonfaulty nodes can therefore keep statistics on the reliability of the link. As stated above, if the link only exists due to a faulty node acting as a relay, since encryption is used, the relay can only drop packets at random.

If less than an "acceptable" percentage of packets get acknowledged, the link is no longer reported to the network, or is reported to the network with a flag warning that it is of dubious reliability.

manually configured information The above three defenses when used simultaneously prevent a faulty node acting as a relay from unduly disturbing the operation of the network. However, an alternative strategy is to manually configure, at each node, the (ID,public key) pair corresponding to the other end point of each of its links.

Since that introduces a lot of extra manual burden, and since this problem has a solution

that does not involve additional manual configuration, we advocate the use of encryption, signatures, and performance monitors (all simultaneously), rather than manually configured information (which by itself would solve the problem, assuming the manual information was correct at virtually all the nodes).

Note that a link can have only a single neighbor. A faulty node cannot cause its nonfaulty neighbor's LSP to grow overly large by causing the link to appear to contain many neighbors, since nodes know that a point to point link can have only one other endpoint. If a node were to receive "Identification" messages from more than one node, it would only believe the latest received, and never assume multiple neighbors simultaneously.

Also note that we have not discussed multiaccess links in this thesis – all links are assumed to be point to point.

3.4 Packet Forwarding

3.4.1 Source Routing

In current packet switching networks, (ARPANET, ISO connectionless Network Layer, DNA), routes will work only if consistent routes are calculated by all the nodes in the network. In these networks, each forwarding node makes an independent decision as to the direction in which a packet should be forwarded. If Link State databases in two nodes along a packet's path differ by knowledge of even a single link, the packet may loop, since each of the nodes might view the other as closer to the destination.

With Byzantine behavior, our scheme cannot guarantee that LSP databases at nodes will remain consistent for sufficiently long periods of time so that a reasonable number of data packets might successfully reach their destinations. Although a node cannot generate LSPs on behalf of a nonfaulty node, it can generate enough LSPs on behalf of itself to keep the network in an unstable state for the majority of the time.

A possible defense is to set a minimum time between generation of LSPs by any source, and ignore a source that has issued too many LSPs (assume it is faulty).

However, a timer scheme such as this would be extremely difficult to design for many reasons:

- A nonfaulty node that generates LSPs just within threshold in one portion of the network may appear as if it is generating LSPs just beyond threshold in another portion of the net, as propagation skew becomes an effect.

If the result is that some nodes assume the source faulty, and ignore its LSP, while others do not, databases at network nodes will not be identical.

- The value of the timer would need to be extremely large, since if any single node is allowed to issue an LSP every unit of time, then the average number of LSPs in the network can be N times as great, since every node is allowed to issue a new LSP within the unit time.

Thus it seemed difficult to devise a scheme in which the databases at different nodes in the network would be consistent for a sufficiently large fraction of the time to make routing acceptably reliable.

Another factor that caused us to reject the hop by hop routing decision was the need to compute alternate routes when a route was failing. If an alternate route strategy were to be performed in a distributed manner, each network node would have to be able to do the identical computation as the source node, and have knowledge about which of the routes the packet should follow. For instance, if the source computed k different routes to a particular destination, the packet could be marked by route number, such that a forwarding node would know that this packet should traverse the ith route to be computed. A strategy such as that would be computationally very complex. Having the source compute the route and place it in the header is far simpler.

Including the route in the header has the following nice properties.

- Databases at different nodes do not need to agree.
- Different nodes do not need to use the same strategy for route computation.
- Arbitrarily complex strategies for finding a working route can be used by any node, independently of the strategies at other nodes. For instance, a node can avoid links that have changed state "recently", making the assumption that links that have been up for a very long time are more likely to be reliable.

The costs of preselecting the entire route are:

- Extra room in the header to list the route.
- No ability to reroute a packet after it has been launched.

3.4.2 Initial Packet Checks

When a node A receives a data packet from neighbor B, from source/key pair S/p, it makes the following checks, dropping the packet if all the following conditions are not met:

- The pair S/p must have been reported by at least one trusted node.
- The signature in the packet must be valid for S/p.
- If a route is specified in the header (as opposed to specifying that the packet is to be flooded), then the route must contain (somewhere within) the adjacent pair B, then A.

3.4.3 Data Packet Forwarding Rules

In traditional datagram Network Layers, data packets are not acknowledged hop by hop at the Network Layer, although on links with non-negligible error rates, a reliable Data Link Layer protocol is usually employed, which does hop by hop acknowledgment and retransmission.

In our scheme, in order for the Network Layer to monitor the reliability of a link, neighbors are required to send Network Layer acknowledgments for data packets. Neighbor to neighbor Data Link Layer acknowledgments will not yield the proper information. The purpose of our Network Layer neighbor to neighbor acknowledgment is to force a Byzantine node acting as a relay to act as a *reliable* relay. A Data Link Layer acknowledgment would just verify that the packet successfully traversed the link from the nonfaulty node to the relay node. It cannot verify that the relay node will forward the packet to the node which is assumed to be the Network Layer neighbor.

Additionally, our scheme requires a node to retain indefinitely the data packet with highest sequence number received from each source/key pair. Thus, since acknowledgments are required, and storage of the packet is required, it is no extra burden on the Network Layer to enhance reliability of data packet delivery by having a forwarding node A persistently transmit the latest data packet from source S until:

1. A's neighbor (the next hop specified in the route in the packet header) acknowledges receipt of S's packet,
2. or A receives a packet with source S with later sequence number.

This is accomplished through manipulation of the "Ack-Flag"s and "Send-Flag"s.

Suppose the packet in memory at node A from source S has sequence number sn-mem. Suppose A receives, from neighbor B, a valid (i.e. signature is correct, packet is well-formed) data packet with source S and sequence number sn-rcv.

- If sn-mem $<$ sn-rcv, then overwrite the packet in memory with the received packet. If the received packet specifies (in the route field) forwarding to node C, then clear Send-Flag for all neighbors except C and set Send-Flag for neighbor C. (If C is not a neighbor, then no Send-Flag will be set for this packet). If the received packet specifies (in the route field) that the packet is to be flooded, clear Send-Flag for B, and set Send-Flag for all other neighbors.

  Also, clear ACK-Flag for all neighbors, and set ACK-Flag for B.

- If sn-mem $\geq$ sn-rcv, then clear Send-Flag for B and set Ack-Flag for B.

  Note that if sn-mem $>$ sn-rcv, the packet with smaller sequence number will not succeed in reaching the destination, since it has been overtaken by a packet with a higher sequence number traveling on a different route. This is unfortunate, but the source is not supposed to pipeline packets too quickly for this very reason. If it mis-estimates the time of arrival, and loses packets for this reason, a higher layer protocol at the source will recover (since the Network Layer has advertised it is a datagram service).

  Note also that we assume the packet with smaller sequence number is really older than the one with greater sequence number. This will indeed be the case, since a nonfaulty source is required to keep its own sequence number on non-volatile storage, and thus be assured, upon restart, of not reusing old sequence numbers.

3.5 Finding a Route

Our scheme requires the source to choose a route to the destination. As stated before, the source does not have enough information to be assured of computing a *functional* route. The Link State Database contains a list of all nonfaulty nodes and links, but it also will contain some number of faulty nodes and links. If there are many Byzantine nodes, it is unlikely that a route calculated from the database will not include at least one Byzantine node. There are exponentially many routes in the network, and the source cannot try all of them. Therefore, in order for our scheme to be practical, there must be some backup strategy the source can use to find a functional route.

Various strategies:

- node disjoint paths – As indicated earlier, the source can compute node disjoint paths. Thus if the first route calculated does not work, due to inclusion of a Byzantine node, then a node disjoint path is guaranteed not to include the same Byzantine node. However, this strategy (of using node disjoint paths) only works if there are fewer Byzantine failures than node disjoint paths.

  If the source computes node disjoint paths, and gives up if none of them work, then the form of robustness the scheme achieves is:

  > If at least $f+1$ node disjoint paths connect nonfaulty nodes A and B, then A and B will be able to communicate provided that no more than f Byzantine faults exist in the network.

  Note that the "disjoint paths" refer to the existing topology, not to the original topology. The disjoint paths must consist of nonfaulty components.

- Flood when route fails – Another strategy is for the source to flood a packet when all node disjoint paths fail. This allows increased robustness when the number of Byzantine faults exceeds the number of node disjoint paths. Backup flooding might be particularly useful for Network Management messages in a malfunctioning network.

  Note that resources should first be allotted equally to each source, and then, given the source's share of resources, allocated amongst routes of that source. With this strategy, a source that relies heavily on flooding will not be able to degrade resources for the other sources.

- Intermediate Acks – We have provided a mechanism for the destination Network Layer to explicitly acknowledge a data packet to the source. The same mechanism can be utilized for having any intermediate node along the route acknowledge a packet.

  Since an Ack from a node closer to the source (in the route) is redundant, if an Ack from a node further from the source is received, it is only necessary, in terms of database structure, for intermediate nodes to hold a single Ack associated with a data packet. If an Ack is received from a node further from the source than the Ack being held, then the Ack being held is overwritten.

The strategy of requiring intermediate Acks can be implemented in several ways:

1. We could require every forwarding node to transmit an Ack, on every packet it forwards. This causes the burden on the first link of the route to be increased by a factor equal to the number of hops in the route. Since we would like the network to work efficiently in the hopefully usual case (when there are few or no Byzantine failures), this is not a good strategy.
2. We could have the source request Acks from all intermediate nodes, on selected data packets, by including one additional bit in the packet header, indicating intermediate nodes should generate Acks for this data packet.
3. We could have the source request Acks from selected intermediate nodes, by including an extra bit "request Ack" in each hop specified in the "route" field in the data packet header.
4. We could explicitly tell an intermediate node (perhaps by flooding a Network Management packet to it), that it should Ack packets from a particular source.

All strategies except the first place no extra burdens on network resources when a route is working correctly. Of strategies 2 through 4, 2 is the simplest to implement. Strategy 3 uses the least network resources, and is not significantly more complex than strategy 2.

The disadvantage of both 2 and 3 is that it marks packets as special. A Byzantine node could forward packets with any of the bits set, and fail to forward other packets, and then the source is left without any information about which nodes along a path may be faulty. Thus to guard against that form of Byzantine failure, strategy 4 is necessary.

Note however, that if a Byzantine node forwards only packets for which the source has requested Acks, then the source can communicate with the destination through the route. It will unfortunately be somewhat more expensive than communicating over a truly non-faulty path, since all data packets along the path will require Acks, but data packets will succeed in reaching the destination.

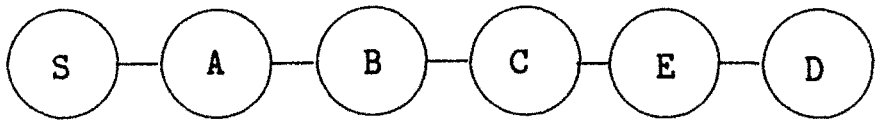
Strategy 4 is sufficiently more complex than 2 and 3 that the increased robustness it might yield does not warrant its use.

A variation of strategy 3 could be used in which the "request ACK" bits are hidden to other nodes by the use of encryption. For instance, each "request ACK" bit, concatenated

with a random number, can be encrypted according to that node's public key. Previous and next addresses for each node in the route can be encrypted in the same operation, hiding global route information from forwarding nodes. This strategy is also sufficiently more complex (and requires more header space) that it is not recommended. More space efficient mechanisms may be possible, but they will also be computationally expensive, and a simple strategy of finding a node disjoint path will probably work sufficiently well in practice.

Note that Acks cannot be forged. A node cannot generate an Ack unless it indeed has seen the data packet it is Ack'ing (since the Ack contains the signature from the data packet), and no node can forge an Ack on behalf of another node (since the node generating the Ack signs it).

Now suppose, using the information from Acks, that source S discovers that packets along route:



are Ack'ed by nodes up to and including C, but not by E. Then S can conclude that one of C or E must be faulty, and S can compute a new route that does not include nodes C or E. This gives S more freedom in route selection than requiring S to choose a route that is node disjoint from the original route. There will likely be more routes to select from if only a subset of nodes from the original route are avoided.

Note that this method of fault diagnosis gives "advice" to the source as to which nodes might be useful to exclude when computing a route. It has disadvantages.

- For each faulty node noted as suspicious, a possibly nonfaulty node is also noted as suspicious. If too many nonfaulty nodes are excluded from the database, a nonfaulty path may exist, and may not be found.
- If there are two Byzantine failures along a path, say nodes A and E in the example above, then S can be fooled into eliminating nonfaulty nodes B and C from the

database, and not eliminating either A or E. This can occur if E's fault consists of failing to forward S's data packets to D, and A's fault consists of discriminatorily failing to forward ACKs from C, so that S receives ACKs from B, but not from any nodes further along the path.

In practical networks, this form of fault diagnosis would be useful, but it does not enhance the theoretical robustness of the design. We still require the number of Byzantine faults to be smaller than the number of node disjoint paths between a pair of nonfaulty nodes in order for communication between the node pair to be guaranteed.

- Explicit query – Another possible method of route debugging is explicit query of nodes along the path through Network Management.

  Because nodes keep a record of the latest packet seen so far from a given source, it is possible for Network Management, at any time, to query intermediate nodes and discover which nodes have seen the latest packet from a given source.

  This form of fault diagnosis is most robust against Byzantine behavior, especially if nodes are queried locally, since the network is not required to route Ack packets properly. However, it is far more cumbersome to implement than Ack packets, since it involves Network Management, and is not confined to the Network Layer.

This concludes the description of our Link State Routing scheme. Following will be explanations for our choice of design.

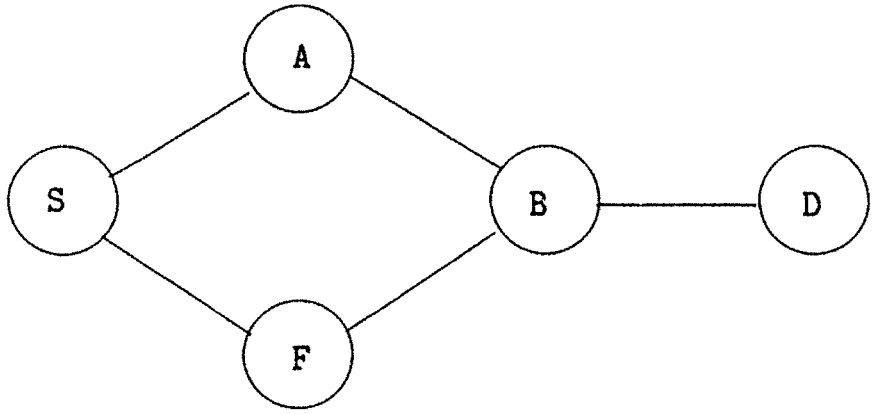
3.6 Design Choices

3.6.1 Data Packet Sequence Numbers

We wish to guarantee a certain minimum amount of resources so that a source's latest data packet will have a high probability of successfully reaching the destination, provided that the calculated path consists of all nonfaulty components. We allocate a fixed number (for simplicity, one) of buffers per source at each node.

It might be assumed that a nonfaulty path will not reorder packets from S (that contain the same route), and will not retransmit an old packet from S. Thus it might be assumed that we could avoid the use of sequence numbers on data packets. However, a faulty component can

introduce old packets from S into a node along a nonfaulty path.



For instance, if F is faulty, and at some point in the past S issued a packet with route (S,F,B,D), then F can disrupt packets along path (S,A,B,D) by reintroducing the old packet to node B, and without sequence numbers, B cannot intelligently decide which of the packets should reside in the single buffer reserved for data packets with source S, and which should be discarded.

A mechanism such as that described in Section 2.6.5 can insure progress of a packet along a nonfaulty path, but as in Section 2.6.5, the scheme would be totally impractical due to the exponentially small bandwidth it would guarantee.

Sequence numbers on data packets assure that:

- No nonfaulty node will accept the same packet twice (so that a packet cannot loop, or be reintroduced into the network at a later time, consuming resources).
- The latest packet issued by a source S is guaranteed resources, and will not compete at any node for resources with a packet issued earlier by S (since the earlier packet will have a smaller sequence number).

3.6.2 Stable Storage for Sequence Numbers

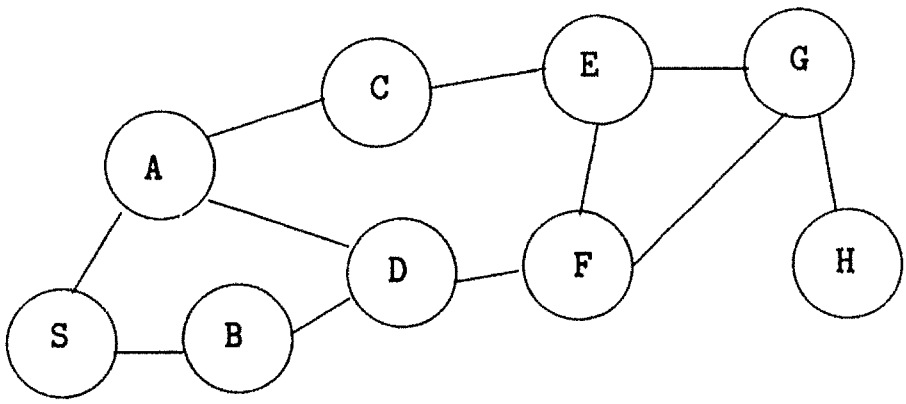
In Chapter 2, a node, A, kept its sequence numbers in volatile storage, and reacquired them dynamically from information stored in the network (if some component in the network did retain memory of A's sequence numbers). The reacquisition mechanism consisted of node B

transmitting A's stored packet to neighbor C, in response to C transmitting a packet to B with source A, and sequence number smaller than the one in B's database.

This mechanism caused a flooding of the packet with higher sequence number throughout the portion of the network without memory of A's higher sequence number packet. A Byzantine node B cannot create an extra burden on the network with this mechanism because:

- B cannot construct a packet from A with higher sequence number than the rest of the network has seen, since B cannot generate a correct signature. Thus B can only retain packets that were legitimately constructed by A.
- Once B floods a packet with sequence number k, B can create no additional network traffic with that packet, since a nonfaulty neighbor of B will simply acknowledge the packet to B, and not forward it further. Thus B can congest its own links to its neighbors, but it cannot create additional packets on any other links in the network.

In this chapter, we do not want data packets to be flooded, since we want to increase the network capacity by utlizing parallelism in the network. With data packets traveling over directed paths, network nodes will not all see the latest sequence number from a particular source.



For instance, if S's packet with highest sequence number contained route (S,A,C,E,G,H), then nodes B, D, and F would not know of S's highest sequence number.

Suppose we wanted to allow S to reacquire its own sequence number. The idea would be, as in Chapter 2, that S would always restart using sequence number 0. If its data packet with sequence number k encountered a node that had knowledge of a packet from S with higher sequence number, that node would send information back to S, informing S of the higher sequence number.

The mechanism in Chapter 2 of reaching the source was flooding. If we allowed a node to flood a later packet in response to a packet with earlier sequence number, then a Byzantine node along the path chosen by S could cause each of S's new packets to be flooded. Again, using our example network, although packets along path (S,A,C,E,G,H) are supposed to be contained within that path (and not to use resources anywhere else in the network), any of the nodes along the path could flood the packet into the rest of the network, ostensibly to inform S of the highest sequence number.

A slight improvement to having a node flood the higher sequence numbered packet from S, in response to receiving a packet with smaller sequence number, is to have a node send the higher sequence numbered packet backwards along the route from which it received a packet with smaller sequence number. This limits the ability of a Byzantine node to flood all packets, since it can only send a packet back along a directed path. The node can be further restricted by making the reverse path packet self proving. The reverse path packet must contain enough information to prove that the node initiating it has seen two packets generated by source S, and the route by which the reverse path packet is traveling was constructed by S.

Again, looking at our example topology:

- Assume S issued a packet with sequence number k and route (S,B,D,A), at some point in the past.

- Now assume source S issues a packet with sequence number j and route (S,A,C), where $j > k$.

- Node A can construct a "Reverse Path Packet" with enough of the header from the packet $\#j$ to prove S did generate a packet with sequence number j, and enough of the header (including the route) from the packet with sequence number k, to prove S generated it.

- The "Reverse Path Packet" is sent backwards along the route (S,B,D,A), consuming resources at nodes B and D and the links along the path. Each node along the path of the "Reverse Path Packet" updates its memory of the largest sequence number from source S to be j.

The problem is we can no longer enforce that packets consume resources only along the path they are directed onto, once we allow nodes along the path to "divert" the packets onto

alternate paths. This cancels the potential bandwidth advantages we gained by attempting the more complex Link State Routing instead of the Flooding Routing.

Thus, in the absence of a satisfactory scheme for reacquisition of sequence numbers, we require each node to keep an estimate of its own sequence number on stable storage.

3.6.3 Faulty Neighbor Restart Problem

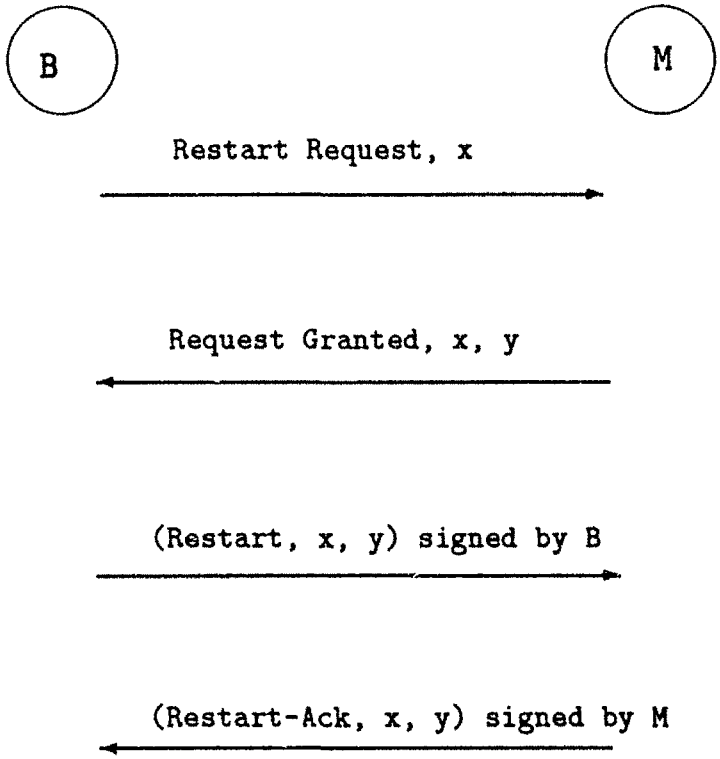
Recognizing the possibility that a faulty node F could act as a relay between nonfaulty nodes M and B, we provided that M and B encrypt all traffic across the presumed link between M and B. Thus in general F cannot differentiate packets. However, if we assume F captures a "Restart-Notification" packet transmitted by M, then F can cause B to erroneously believe its neighbor M has restarted, which will cause B to retransmit all its stored packets to B.

This is not a very serious problem, since the extra traffic will be localized to the M-B link (B, receiving a packet with the same sequence number as it has stored, will acknowledge the packet, but will not transmit it further). With M and B monitoring the quality of the link, they can declare the link down if frequent restarts become a performance problem for the link.

A second threat is that F could capture a Restart-Notification-Ack generated by M. In this case, when B lost state and issued a Restart-Notification to M, F could fail to forward B's Restart-Notification (if it could guess which packet was a Restart-Notification, which might be easy if, for instance, a Restart-Notification is usually the first packet transmitted after a long period of link idleness), and retransmit M's old Restart-Notification-Ack to B.

In this case, B will not receive an updated database from M. So, for instance, B will not receive (through its link to M) the latest Public Key List Packets. This is also not a very serious problem because if B has a nonfaulty path to the resource from which it needs packets, it will receive those packets along that path. If B does not have a nonfaulty path, then we do not guarantee B will be able to operate, and its behavior under the circumstances is irrelevant.

It is, however, possible to design a scheme to defend against the Restart-Notification and Restart-Notification-Ack replay threats. Instead of two messages (Restart-Notification and Restart-Notification-Ack), the restarting protocol would use 4 messages.



In the first message, B transmits its intention to restart, with a random number, x. M responds with its own random number y. With high probability, x and y will be different from random numbers chosen on previous restart handshakes between B and M. The third message is a Restart-Notification, including both random numbers x and y, and signed by B. The fourth message is a Restart-Notification-Ack, including x and y, signed by M. Since F cannot generate a Restart-Notification with B's signature for numbers x and y, F cannot cause an unnecessary restart. Since F cannot generate a Restart-Notification-Ack with M's signature with numbers x and y, F cannot prevent (without B's knowledge) B's restart from reaching M.

3.7 Hierarchical Networks

Just as in Chapter 2, the design in this chapter can be adapted for hierarchical networks. See Section 2.6.4 for an overview of hierarchical addressing and routing.

To support hierarchical routing, one additional field, "ultimate destination" is required in a data packet. Initially, the source places the destination address into the "ultimate destination"

field.

When source and destination nodes of a packet are in the same subnetwork (as evidenced by the "SUBNET" portion of the destination address), the source writes the destination into the "destination" field in the packet header, calculates a route to the destination, and places that route in the packet header.

When source and destination are in different subnetworks,

1. The source calculates a path to a level 2 router, L_1, writes L_1 into the "destination" field in the packet header, and places the path to L_1 in the "route" field of the packet header.

2. Next, L_1 calculates a path to a Level 2 router, L_2, in the destination subnetwork (as specified by the "SUBNET" field in "ultimate destination"). L_1 removes all of the header except "ultimate destination", and overwrites the header with its own ID, key, sequence number, L_2 as "destination", route to L_2 as "route", and generates its own signature.

3. When L_2 receives the packet, it calculates a path to the destination specified by "ultimate destination", and overwrites the header with its own ID, key, sequence number, "ultimate destination", route to "ultimate destination", and generates its own signature.

This scheme works for the same reason that hierarchical routing worked in Chapter 2. Basically, each level 2 router guarantees fair access to the level 2 net, to each source within its subnet, and each level 2 router also guarantees fair access into its subnet for each level 2 router.

3.8 Route Setup Variant of Source Routing

In this section we show an alternate form of routing which also exhibits Byzantine robustness. It is interesting because it allows data packets to be forwarded without the need for cryptographic checks by the nodes which are forwarding. The basic modification is that with this scheme, data packets travel along a route which needs to be "set up" before a conversation takes place, instead of using " packet specified source routing", where the route is placed in the packet header. In both methods the source chooses the route. Cryptography is still needed to authenticate the route setup procedure, but cryptography is not needed for forwarding data packets. Rather, a set up route is assumed by the intermediate nodes to be nonfaulty, and a data packet is assumed to be legitimate based on its being received from the expected direction (the previous hop of the route set up by the source).

This variant has advantages and disadvantages with respect to the packet specified scheme described in Section 3.2.

The advantages of this scheme are:

1. We eliminate the requirement that the packet contain the route, which saves bandwidth.

2. We eliminate the requirement that forwarding nodes verify a signature when forwarding data packets, saving processing overhead.

The disadvantages of this scheme are:

1. Memory requirements in this scheme are $O(N^2)$ per node, as contrasted with O(N) per node with the packet specified source routing scheme.

2. Fault isolation of Byzantine forwarding nodes is impossible in this scheme, because (by its very nature of avoiding cryptographic checks on data packets), data packets become impossible to trace.

   However, additional mechanism can be employed to aid fault isolation. For instance, we can continue to use sequence numbers and signatures in Data packets, eliminating the disadvantage of reduced fault isolation, but it also eliminates the advantage of relieving forwarding nodes from computing the signature.

Thus if we use cryptography in the Route Setup scheme on packet forwarding, and include the fields "signature" and "sequence number" in data packets, the tradeoff between the Route Setup method of Source Routing and the Packet Specified Route Setup scheme becomes one of memory versus communications bandwidth (smaller headers in Route Setup scheme because the route is not included in each data packet).

The portions of the Link State Routing Scheme that remain without modification are:

- Public Key List Packets are generated and propagated as before.

- Link State Packets are generated and propagated as before.

- Routes are computed as before.

The main differences between this scheme and the packet specified source routing scheme are:

- Memory is reserved to hold a route and the latest packet for each source/destination pair.

- An additional packet type is required, a "Route Setup Packet", which contains a route, and is flooded by the source.

- Data packets no longer carry the route. Additionally, if fault isolation is not considered important (for instance, if reliance on node disjoint paths is considered sufficiently robust), the sequence number and signature can also be eliminated from data packets.

3.8.1 Dynamic Database

The following information is kept by each node in volatile storage, as before:

OWN-LSP-SEQNUM	The next sequence number to be assigned when this node generates its own next LSP
OWN-PKL-SEQNUM	The next sequence number to be assigned when this node generates a PKL packet. This is kept only by nodes acting as "trusted nodes".
LSP database	For each node/public key pair reported by any trusted node: • The LSP with highest sequence number from that node/public key • For each neighbor of this node, two flags: 1. Send-Flag – indicating whether this LSP needs to be transmitted to this neighbor 2. Ack-Flag – indicating whether an ACK for this LSP needs to be transmitted to this neighbor
neighbor	identity of and public key for each neighbor
Restart Flags	"Send-Restart" and "Send-Restart-Ack" flags for each neighbor

The following information is also kept by each node in volatile storage (but is listed separately because it is different than before:

OWN-RT-SETUP-SN	The next sequence number to be assigned when this node generates a Route Setup Packet. Note that, as with LSP and PKL sequence numbers, stable storage contains a sequence number that can be used on restart, which is guaranteed larger than any previously used by this node.
Routes	For each source/destination pair: • The "Route Setup" packet with largest sequence number, for this source/destination pair. • For each neighbor: 1. Send-Flag – indicating whether this "Route Setup" packet needs to be transmitted to this neighbor. 2. Ack-Flag – indicating whether this "Route Setup" packet needs to be acknowledged to this neighbor. • Data Packet Buffer – A buffer for the most recently received data packet on this route.

3.8.2 Route Setup

A "Route Setup" packet contains the following:

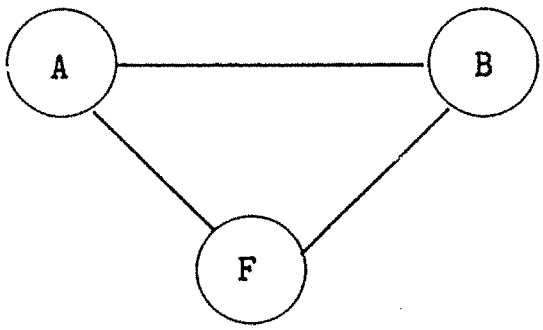
source	The node that generated this packet
destination	The destination of the route, or if 0, indicates "all destinations" (used for clearing out all of the source's routes from the network, and a packet with "0" for destination would also have "0" for route).
public key	Public key for "source"
sequence number	Assigned by "source"
route	A sequence of ID/key/link triples
signature	A digital signature, verifiable based on the public key for this node, manually maintained at each node, covering the entire contents of the packet.

Note that the same pair of nodes might have multiple links connecting them. A route must then specify a particular link. The method of specifying a particular link is just an encoding problem. For instance, we could use the ordering of the links within the Link State Packet of the previous node along the path, to specify a particular link. For example, if one hop in the path is between nodes G and H, then the route setup would specify G/key/2/H, and the link to be used when forwarding from G to H would be the 2th listed link in G's LSP that has H as the neighbor.

The encoding scheme should minimize the problems caused when node G issues a new LSP – it would be desirable if routes going through G not be disrupted because the ordering of information in G's LSP has changed. If this cannot be accomplished, the correctness of the

scheme is not affected – if a route changes and still is correct, then there is no problem. If a route changes to a nonfunctional route, then the protocol would handle it the same way as any route failure.

We could require the network not to allow multiple links between the same pair of nodes, but we cannot avoid them, due to the possibility of a common faulty neighbor F, between two nonfaulty nodes A and B (which are neighbors of each other) creating an additional link between A and B by acting as a "dumb relay".



However, the simple mechanism of explicitly stating the link in the Route Setup packet avoids any problems caused by multiple links between neighbors.

- When a source S reinitializes after having lost state, it sends a Route Setup packet with null destination and route. This packet will be flooded, and will purge from the nonfaulty nodes in the network any knowledge of previous routes involving S.

- When a source S wishes to initiate a conversation with destination D, S computes a path to D and includes that path in a Route Setup Packet.

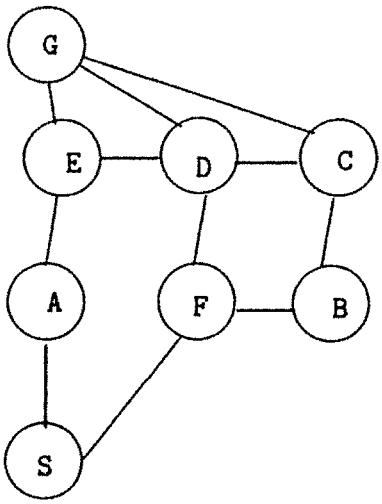
- The Route Setup Packet is flooded by the network, using the same mechanism as flooding of LSPs, except that a Route Setup Packet with specified destination D will be accepted and flooded by node B provided that B has no knowledge of any other Route Setup Packet with higher sequence number from source S that involves destination D.

  If the Route Setup Packet specifies destination D, and B has seen another Route Setup Packet with source S, different destination, and higher sequence number, the S/D Route Setup packet will still validly overwrite the S/D route stored at B. A Route Setup Packet with source S, sequence number k, and null destination received by B will overwrite every route stored at B with source S and sequence number less than k.

Thus a node can remember at most one route for each source/destination pair. The requirement to remember a route, possibly for each source/destination pair, makes the database $O(N^2)$.

Note that since a Route Setup packet is flooded, only one S/D path can exist in the network of nonfaulty nodes. If S sets up a new path to D, then nodes along the old path will overwrite their route with the new route, on which they are not included, so that they will not accept any data packets from with source S and destination D.

We could have provided that Route Setup Packets traverse the path specified in the packet, and not be flooded. In that case, knowledge of old routes would remain in the network. It is not necessary for correctness that routes ever be purged. The latest Route Setup packet generated by a source will be guaranteed to successfully set up a route (if the route selected is nonfaulty), and once set up, a nonfaulty route is guaranteed resources. We provide a mechanism for purging faulty routes, because faulty routes can utilize network bandwidth.



For instance, suppose node F is faulty, and all other nodes are nonfaulty. Assume at some point in the past, S, in attempting to find a good route to G, set up routes (S,F,D,G) and (S,F,B,C,G) before finally setting up the good route (S,A,E,G). Especially since there is no cryptographic authentication of data packets, F can generate an unlimited number of packets along both paths (F,D,G) and (F,B,C,G). Since nodes D, B, and C have no knowledge of the (S,A,E,G) route, they will assume they are part of a correct route, and will forward any packets with source/destination pair S/G that they receive from the appropriate direction according to their stored route for S/G.

In some sense the other two routes are irrelevant, since they do not interfere with S's resources along path S-A-E-G. In fact, packets fallaciously launched by F along those paths do not actually reach G, since G knows the only link from which it should be receiving packets from S is its link to E. However, network bandwidth is consumed unnecessarily. The entire reason for choosing to continue our design beyond the robust flooding provided by Chapter 2 was to enable the resources consumed by a traffic stream to be "localized", and not be seen by every node. Unless old routes are purged from the network by the source, there is no limit to the number of nodes (other than N, the total number of nodes) that can be allocating resources for a particular source/destination pair.

3.8.3 Data Packets

The only relevant information in a data packet is the source and destination (each of which is an ID/key pair). There is no need for any signature, or sequence number.

When a data packet with source/destination pair is received by node A, from neighbor B:

1. If B is not the previous node listed in the Route database, A drops the packet.

2. Else, A stores the packet for forwarding towards the next node specified in A's Route database.

Data packets are not acknowledged hop by hop, and have no sequence numbers. Once a forwarding node transmits a data packet, it deletes it from the database.

3.8.4 Assuring Fairness

Each forwarding node guarantees resources for one Route Setup packet for each source/destination pair. Additionally, each forwarding node guarantees resources for one data packet for each source/destination pair for which a route is set up.

At any time in the database, there will be some number of stored Route Setup Packets, with Send-Flag and/or Ack-Flag set. Also, there will be some number of stored data packets.

The forwarding node must cycle through the database, sending an ACK for every ACK-Flag set, forwarding each Route Setup for which Send-Flag is set, and forwarding each stored data packet.

3.8.5 Why This Works

Assuming node S calculates a nonfaulty path to destination D, S's Route Setup will succeed, because it will be the Route Setup Packet with highest sequence number. Also, once the route is successfully set up, no other node can disrupt the route, since the only situation in which the nodes along the path will discard or overwrite the information about the route is receipt of a validly signed Route Setup packet with higher sequence number, with source/destination pair S/D.

Assuming the set up route is nonfaulty, each node along the path is responsible for making sure packets traversing the route never arrive from the wrong direction. Thus, assuming the route is nonfaulty, a node outside the route cannot inject packets into the flow along the route.

3.8.6 Performance

The chief advantage of the Route Setup method over the Packet Specified Source Routing method is bandwidth efficiency – packets no longer need to carry a route.

An additional benefit is that nodes which are forwarding data packets do not need to do cryptography as part of the forwarding process, saving computation for what would presumable be the vast majority of network packets (data packets). (However, this advantage creates the disadvantage of decreased ability to diagnose why a route has failed.) End to end cryptography by the source and destination of each data packet, and signature verification by each forwarding node along the path of a Route Setup packet would still be performed, however.

With the Source Routing method of packet delivery, a source can utilize $1/N$ of the bandwidth of any link for a conversation to a particular destination (it will get more than that usually since usually not all sources will be directing packets along that link at any time).

With the Route Setup method, each source is guaranteed only $1/N^2$ of the bandwidth of a link for a conversation to a particular destination, if resources are allocated in a simple round-robin fashion for stored routes. However, allocation of the link can be done per source, and then the source's share can be suballocated for each of the (up to $N-1$) routes with source S, in which case with the Route Setup method the source would still be able to utilize $1/N$ of the bandwidth of any link.

The advantage of the Packet Specified Source Routing method is that memory per node is $O(N)$, whereas in the Route Setup method memory is $O(N^2)$ per node.

The Route Setup method would probably prove impractical unless Data Packets continued

to include sequence numbers and signatures, since forwarding nodes with Byzantine faults nodes would be difficult to isolate. Adding sequence numbers to data packets (so that forwarding nodes could be queried as to receipt of specific packets) without also adding signatures would not be useful in the presence of true Byzantine failures, since nodes along the path could corrupt sequence numbers, or generate faulty data. Without cryptographic checks there is no way to verify that the sequence number was generated by the source. Thus, if the path from source S to destination D consists of nodes (S,A,B,C,D), and node B were faulty, then B could fail to forward S's true data packets, and instead inject bogus packets with the same sequence numbers, and since (by the very nature of the scheme) nodes C and D do not verify the packets in any way, all nodes along the path would claim to have seen the packets with those sequence numbers, but node D would not receive them (cryptographic verification still needs to be done at the destination).

Likewise, with an unverified sequence number scheme, any node could generate data packets with source address S, and arbitrarily high sequence numbers, making any scheme in which packets were rejected based on comparison of sequence numbers non-robust.

Thus the Route Setup variant is probably only a viable alternative if cryptography is still performed by intermediate nodes. In this case, the real tradeoff is memory (Route Setup is $O(N^2)$ vs $O(N)$ for packet specified source routing) vs bandwidth (Packet Specified Source Routing requires longer headers, to contain the route, though this is partially offset by the bandwidth consumed by Route Setup packets in the Route Setup method).

Chapter 4

Conclusions

4.1 Results

In Chapter 2, we presented a form of routing based on flooding in which a pair of nonfaulty nodes are guaranteed to be able to communicate provided that a nonfaulty path connects them. Although based upon flooding, a simple form of routing in which communications bandwidth consumed per packet can be exponential in the size of the network, the design in Chapter 2 actually gives reasonable performance because each packet will traverse each link only once (except for link retransmissions due to transmission errors on the link, or restart of the adjacent node, or Byzantine behavior by one of the endpoints of the link). When Byzantine behavior is likely to be common, a network based upon this design should be practical and robust.

The key to the design was structuring databases and providing authentication in such a way that fairness was enforced.

In Chapter 3, we presented a form of routing based on Link State Routing, in which a pair of nonfaulty nodes are guaranteed to be able to communicate provided that at least $f+1$ node disjoint paths connect them, and at most f failures exist simultaneously in the network. This form of guarantee makes a network with the design in Chapter 3 considerably less robust. Design modifications yield higher robustness at greater cost.

1. Flooding (as done in Chapter 2) could be selected as a backup option when a functional route cannot be easily found. In this case, the robustness is equivalent to that provided in Chapter 2, but the performance is no longer guaranteed superior to that provided by the simpler scheme in Chapter 2.

2. Fault isolation by the source can be executed to attempt to discover which components are malfunctioning. This enables the source to find a functional route with higher probability.

Since in non-military settings, Byzantine behavior would probably be fairly rare, a design based on that presented in Chapter 3 would probably be practical and sufficiently robust. If many Byzantine nodes are expected in a network, the design presented in Chapter 2 would probably be preferable, since it is simpler and more robust.

We concluded in Chapter 3 that hop by hop routing decisions were not viable, and instead the entire route should be computed by the source. Two variants of specifying the route were given, one in which the route was specified in the packet header, and one in which the route was first set up by the source and stored in the intermediate nodes. The packet specified option was more memory efficient and the route setup option was more bandwidth efficient, but both are practical.

4.2 Basic Tools

To accomplish Byzantine robustness, we made use of certain "tricks". First we describe the observations and mechanisms that made robust flooding possible.

4.2.1 Flooding Mechanisms

Bootstrapping Service We require certain information to exist throughout the network; in particular, the complete list of nodes and public keys. Rather than requiring manual maintenance of such information at all nodes in the network, we instead employ a central repository of the information, where the information is maintained. This centralized service then broadcasts the information throughout the network.

This approach minimizes the amount of manually maintained state that must exist at each network node (the information necessary to enable the bootstrapping service to do a network-wide broadcast). It also minimizes the locations at which the large database must be maintained (it must be maintained at the central repository only).

Signatures We use public key cryptography, so that any node can verify a source's signature, but only the source can generate a correct signature for itself. This technique, of course, is well known.

(Node,key) pairs By considering each distinct (node,key) pair as a different node, we ensure no disruption of service to a node that is in the process of changing its own key, or to a

node whose key is being mis-reported by one or more "trusted nodes" (as long as at least one "trusted node" reports the node's key correctly).

Non-wrapping, Finite length Sequence Numbers We require a method of packet comparison so that distant nodes can intelligently decide which packet generated by a particular source should occupy resources reserved for that source.

Our sequence number mechanism is extremely simple because we do not require sequence number wrap-around, or resets. This is accomplished because:

1. The sequence number field is chosen to be large enough so that exhaustion of the field is a rare event.
2. A mechanism is provided for a source, following a simple failure, to reacquire the value of its own sequence number in use prior to the simple failure. The primary mechanism for this purpose is persistence by the network (hop by hop acknowledgments and retransmissions, and retransmissions following node restart). An additional mechanism, particularly crucial when a source reuses the *same* sequence number following a simple failure, is for smaller (or conflicting) sequence numbers to cause a reflooding of the packet with larger sequence number (or a notification that a sequence number has been reused).
3. A source whose sequence number does reach the limit can become a "new node", and restart with the lowest sequence number, by changing its public key. This is accomplished by choosing a new public key and manually installing it at the "trusted nodes".

Structured Database A node can receive an unlimited number of packets. We structure the database so that the latest packet from each source is guaranteed storage.

Guaranteed Fairness We provide rules for scanning the database in an orderly manner, so that every packet that needs to be transmitted is guaranteed fair access to link bandwidth.

4.2.2 Link State Mechanisms

Additional mechanisms used to provide directed routing were:

Source Routing The source chooses the packet's complete path. This enables routing to work even if routing databases are not synchronized and even if nodes choose different

strategies for route computation.

We use two variants of source routing. In the first part of Chapter 3, we use the form of source routing in which the route is placed in the header of each data packet. In Section 3.8 we use the form of source routing in which the route is carried in a special "route setup" packet which must be flooded prior to a conversation.

Disjoint Path Computation We use disjoint paths when primary paths do not function. The computation of disjoint paths is a trivial extension of the min cost max flow problem, which has been extensively studied in the literature.

Link Encryption Because a Byzantine node can simulate a link between two nonfaulty nodes, we use Network Layer hop by hop encryption to prevent a Byzantine component, along what is assumed to be a link by the Network Layer, from discriminatorily forwarding packets.

Fault Isolation Because nodes keep the data packet with highest sequence number from each source, querying of the network nodes can reconstruct a packet's progress and find two nodes along the path, of which at least one is faulty.

Packet Verification Based on Direction of Receipt In the Route Setup option, we concluded that correctness could be ensured without cryptographic checks when forwarding Data packets, by having a forwarding node reject packets unless they arrive from the expected direction. In this case, packets will travel correctly on nonfaulty paths, and no guarantees are made about packets on faulty paths.

Parallel Networks A single faulty "trusted node" can simulate many faulty network nodes. Directed routing will not efficiently find a functional path if too much of the Link State database contains faulty nodes. Thus we model the network as t parallel networks, one for each of the "trusted nodes". If all "trusted nodes" agree on the list of (node, key) pairs, then all parallel networks are equivalent. However, if they differ, and at least one "trusted node" is nonfaulty, then one of the "networks" will be (mostly) correct. This model allows efficient computation of a functional route no matter what information is given by faulty "trusted nodes".

4.3 Further Application of These Ideas

Some of the mechanisms designed for Byzantine robustness can be applied to simplifying or increasing the robustness of other protocols.

For example, a noncryptographic adaptation of the flooding scheme in Chapter 2 can be used as the algorithm for propagation of Link State Packets in a traditional Link State Routing scheme, yielding a more efficient and robust Link State Packet distribution scheme than current designs.

The most complicated aspects of a traditional LSP propagation scheme are the timers and the wrapping sequence numbers. Instead, we can use the approach in Chapter 2 and avoid both problems. We will give an overview of a noncryptographic Link State Propagation scheme that does not provide Byzantine robustness, but achieves self-stabilizing robustness (like the design in [Pe] – the design in [MRR] is not even self-stabilizing), is simpler than the designs in [Pe] and [MRR], and is comparable in implementation costs to the designs in [Pe] and [MRR].

Associated with each node in the network is a "key", analogous to the "public key" used in this thesis. However, in this design, the "key" is not used for computation – instead it acts as a descriptor to the ID of a node, to identify what is roughly an "incarnation" of a node.

We use a "trusted node service", like the "trusted node service" in the thesis, which broadcasts a list of (ID,key) pairs. To cover simple failures of the trusted node, there could be several nodes in the service, and the accepted list of (ID,key) pairs would be the union of all the lists received from all the "trusted nodes".

As in the thesis, each node in the network would need to be manually configured with the ID and key of each of the "trusted nodes". Likewise, the "trusted nodes" would need to be manually configured with the complete set of (ID,key) pairs in the network.

Link State Packets would contain a sequence number, but no age field (an age field is used in the designs in [Pe] and [MRR]). The sequence number would be monotonically increasing, and chosen to be of sufficient length so that reaching the maximum value would be a rare event. If a node B (due to its own Byzantine fault, or due to actually reaching the maximum value of a sequence number with moderate range) did reach the sequence number limit, the node would change its "key" and the "trusted nodes" would need to be manually modified to associate the new "key" with B.

Nodes would keep the LSP with highest sequence number from each (ID,key) pair listed by

any "trusted node". If a node is to permanently leave the network, it can be deinstalled, and all resources associated with it released, by manually removing its entry at the "trusted nodes".

If a "trusted node" is to be added or deleted from the network, manual modification of all the network nodes is required. This inconvenience can be avoided by making the entire "trusted node service" appear to the network like a single node. This can be accomplished by the techniques employed in distributed databases. The group of trusted nodes can elect a "leader" which will broadcast (ID,key) pairs. In the event that the leader crashes, the group of trusted nodes can elect a new "leader", which can take over transparently. Updates to the database can be manually done at each site, or a protocol to share updates can be carried out among the "trusted node" servers.

4.4 Future Research

This is a first attempt at construction of an implementable Network Layer robust in the face of active Byzantine failures. It is possible that more efficient, or more strongly robust designs are possible. The impractical scheme presented in Chapter 2, for flooding without any use of cryptography, gives an existence proof that radically different designs may achieve the same functionality.

Also, this thesis concentrated narrowly on Network Layer protocols. It is possible that some of the techniques used can be extended to the design of other types of protocols.

References

[ANSI] "Information processing systems - Data communications - Intermediate System to Intermediate System Intra-Domain Routing Exchange Protocol", Contribution ISO/IEC JTC 1/SC 6 N4945 to ANSI X3S3.3 Committee, October 1987.

[BG] D. Bertsekas and R. Gallager, "Data Networks", Prentiss-Hall, 1987, pp 403-405.

[Dij] E.W. Dijkstra, "Self-Stabilization in Spite of Distributed Control", Comm. ACM, Nov. 1974.

[Dk] E.W. Dijkstra, "A Note on Two Problems in Connection with Graphs," Numer. Math. Vol. 1, pp. 269-271, 1959.

[DH] W. Diffie and M. Hellman, "New Directions in Cryptography", IEEE Trans. Inf. Theory, vol. IT-22, pp. 644-654, Nov. 1976a.

[Dol] D. Dolev, "Unanimity in an Unknown and Unreliable Environment", Proc 22nd IEEE Symposium on the Foundations of Computer Science, 1981, pp 159-168.

[DP] R. Dixon and D. Pitt, "Addressing, Bridging and Source Routing", IEEE Network, January 1988.

[EGP] "Exterior Gateway Protocol Formal Specification", RFC 904, 1984.

[Fis] M. Fischer, "The Consensus Problem in Unreliable Distributed Systems (A Brief Survey)", Yale University Technical Report YALEU/DCS/RR-273, 1983.

[FL] M. Fischer and N. Lynch, "A Lower Bound for the Time to Assure Interactive Consistency", Information Processing Letters 14, 4, 183-186, 1982.

[FLP] M. Fischer, N. Lynch, and M. Paterson, "Impossibility of Distributed Consensus with One Faulty Process", JACM, 32, 2, 374-382, 1985.

[Git] I. Gitman, R.M. Van Slyke, and H. Frank, "Routing in Packet-Switching Broadcast Radio Networks,", IEEE Transacations on Communications, vol COM-24, August, 1976.

[GMT] S. Goldwasser, S. Micali, and P. Tong. "Why and How to Establish a Private Code on a Public Network", Proc. 23rd IEEE Symposium on Foundations of Computer Science, 1982, pp 134-144.

[GT] Goldberg, A.V., and Tarjan, R.E. "Solving Minimum Cost Flow Problem by Successive Approximation", Proc. 19th ACM Symposium on the Theory of COmputation, 1987.

[IP] Department of Defense, "Military Standard Internet Protocol", MIL-STD-1777, August 1983.

[LSP] L.Lamport, R. Shostak, and M. Pease, "The Byzantine Generals Problem", ACM Transactions on Programming Languages and Systems, Vol. 4, No. 3, July 1982, pp 382-401.

[Mcq74] J.M. McQuillan, "Adaptive Routing Algorithms for Distributed Computer Networks", Bolt Beranek and Newman, Inc., BBN Rep. 2831, May 1984.

[MRR] J.M. McQuillan, I.Richer, and E.C. Rosen, "The New Routing Algorithm for the ARPANET", IEEE Transactions on Communications, Vol. COM-28, No. 5, May 1980.

[MS] P.M. Merlin and P.J. Schweitzer, "Deadlock Avoidance – Store and Forward Deadlock", IEEE Transactions on Communications, March 1980.

[Orl] Orlin, J.B., "Genuinely Polynomial Simplex and Non-Simplex Algorithms for the Minimum Cost Flow Problem", Technical Report No. 1615-84, Sload School of Management, MIT, Cambridge, 1984.

[Pe] R. Perlman, "Fault-Tolerant Broadcast of Routing Information", Computer Networks, December 1983.

[Pe2] R. Perlman, "A Protocol for Distributed Computation of a Spanning Tree in an Extended LAN", Ninth Data Communications Symposium, Vancouver, 1985.

[Pe3] Perlman, Radia, "Incorporation of Multiaccess Links Into a Routing Protocol", Eighth Data Communications Symposium, Massachusetts, 1983.

[Pe4] Perlman, Radia, subnet partition problem paper [PR] reference on ARPA Packet Radio Network design, I believe November 1978 Transactions on Communications

[PSL] M. Pease, R. Shostak, L. Lamport, "Reaching Agreement in the Presence of Faults", JACM 27, 2, 228-234, 1980.

[Ros] E.C. Rosen, "Vulnerabilities of Network Control Protocols: An Example", Computer Communications Review, July 1981.

[RSA] R. L. Rivest, A. Shamir, and L. Adleman, "On Digital Signatures and Public Key Cryptosystems, "Communications ACM, vol. 21, pp. 120-126, Feb. 1978.

[RYB] A. Rybczynski, "X.25 Interface and End-to-End Virtual Circuit Service Characteristics", IEEE Transactions on Communications, April 1980.

[SNA] R. J. Cypser, "Communications Architecture for Distributed Systems", Reading, Mass, Addison-Wesley, 1978.

[Taj] W.D. Tajibnapis, "A Correctness Proof of a Topology Information Maintenance Protocol for Distributed Computer Networks", Communications of the ACM, Vol 20, No. 7, July 1977, pp. 477-485.

[Tan] Andrew Tanenbaum, Computer Networks, Prentiss Hall, 1981.

[TCP] Department of Defense "Military Standard Transmission Control Protocol", MIL-STD-1778, August 1983.

[TP4] Internation Organization for Standardization, "Connection Oriented Transport Protocol", DP 8073, 1983.

[Zim] H. Zimmerman, "OSI Reference Model – The ISO Model of Architecture for Open Systems Interconnection", IEEE Transactions on Communications, Vol. COM-28, No. 4, April 1980, pp. 425-432.