Fault Tolerant Routing Service

- Diploma Thesis -

October 22, 1999

Michael Rinklef

IBM Germany, Lab Böblingen
Dept. 3291, 71032-19-165
Schönaicher Str. 220
D-71032 Böblingen

in cooperation with:
University of Stuttgart
Fakultät Informatik
Institut für Parallele und Verteilte Höchstleistungsrechner (IPVR)
Breitwiesenstr. 20-22
D-70565 Stuttgart

Thesis No.: 1773

CR-Classification: C.2.1, C.2.2, C.2.4, C.2.5, C.2.6, C.4, E.2

Abstract:

The subject of this thesis is to find a solution for the problem of possible router failures within an untypical network structure. The intranet discussed here consists of two Ethernets with a number of hosts being connected to either both or only one network. TCP/IP is the communication protocol used in this intranet and each Ethernet represents a subnet in the IP address domain. Since hosts being only connected to one network must be able to communicate with each other, a router is necessary. Each host having two running interfaces can be a potential router between both subnets, i.e. it can relay datagrams from one network to the other. In contrast to the Internet, it is not desired to select only one specific host as a router. The reason is, that the system has to cope with failures, especially with router failures. If a router fails for some reason, the Fault Tolerant Routing Service must find another host with two running interfaces which then becomes the new router.

Since TCP/IP offers several solutions for routing and even to cope with router failures, the most interesting possibilities are discussed first. After this overview of the capabilities of TCP/IP concerning routing, a strategy for realizing datagram relaying in the Fault Tolerant Routing Service is selected.

Another task is to find a host with two running interfaces. This is not trivial since a host itself doesn't reliably know whether both of its interfaces are running correctly. Two protocols are developed which both enable a host to determine its own and each other host's link state. The first solution described here is the one which was finally implemented because it causes less network overhead than the second one. It is based on each host periodically broadcasting LINK STATE messages on each network it is connected to. The content of these messages is a list of hosts the sender has a connection to on the respective other network. Thus, on receipt of a message, the receiver knows that it has a connection to the sender on the receiving network, and learns about the sender's connection on the respective other network. With one message per interface from each host the receiver can now determine if there are hosts with two running interfaces. Among all router candidates, hosts with only one connection select a router and update their routing table.

Contents

1 Introduction

Fault-tolerant systems are a prerequisite for many different application areas like database management systems, e-commerce, etc. One approach to provide fault-tolerant hardware is to replicate important components of a system to achieve redundancy. If a system consists of several CPU's which communicate using a local network, the fault-tolerance of the communication can be improved by using two independent networks with each CPU having two interfaces, each one connected another network. Having this kind of a redundant network structure is the approach of an industrial project which is the environment of this thesis.

The physical network structure given here is based on two equivalent but independent Ethernets and a number of connected hosts giving a fully symmetric network structure. The communication protocol used within this network is TCP/IP. Each Ethernet is realized by a 10/100 Mbit unmanaged hub and represents a separate subnet in the IP address domain, further on referred to as Net 0 and Net 1. Hosts connected to this network can be divided up into two classes. Class One hosts have two interfaces and are connected to both Ethernets, whereas Class Two hosts only have one interface and are connected to only one Ethernet. Having two interfaces in two different subnet domains makes a Class One hosts a multi-homed hosts with two IP addresses, one for each interface to an Ethernet. Class Two hosts have only one IP address for their single network interface. Interfaces to other networks are not taken into account because this system is viewed as an isolated network, an intranet, without any connection to other networks (the extranet). And yet even if there was a connection to another network, it is assumed and demanded, that none of the hosts would be configured as a gateway between the intranet and the extranet. From the outside, all intranet hosts must not be reachable. Hosts inside the intranet have no idea about the existence of other networks anyway, because they are per definitionem not interested at all in communicating with other hosts outside the intranet.

The initial system design provides two Class Two hosts in the network, one on each Ethernet. According to their responsibility in the system, they are called System Controller (SC), in contrast to Hardware Controller (HC), which are Class One hosts and can occur in larger numbers. The only reason for mentioning the different roles in the system is to give a naming scheme for hosts, making it easier to distinguish between hosts in the same class. The following discussion doesn't differentiate between both types of controllers (system or hardware, respectively), but only between the number of possible connections to the network. SCs have only one interface which will ever be connected to this network, so they will always be Class Two hosts. HCs initially have two interfaces and so they belong to Class One, but on failure of one interface, they become a Class Two host and vice versa. If a failed network connection is brought up again, an HC changes its class membership from Class Two back to Class One. Hosts without a running interface and thus not having any network connection are not not taken into account, because they are not a part of the system any longer.

The following figure gives an overview of the system's network structure:

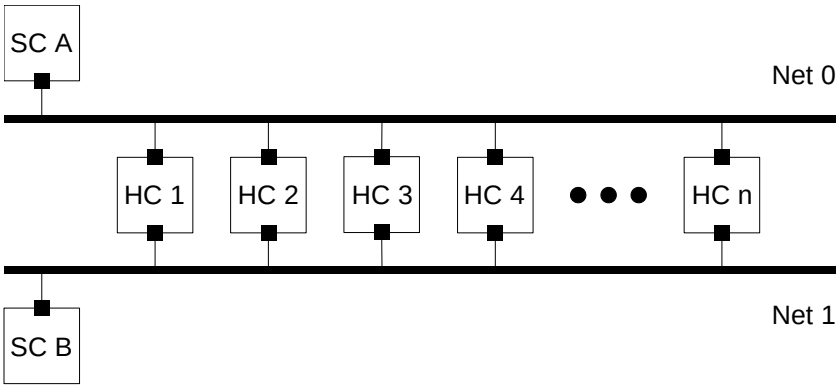

Figure 1.1: Class One hosts are connected to both Ethernets, whereas Class Two hosts only have a single Ethernet interface.

The reason for this network structure is to increase the degree of redundancy. On failure of one Ethernet, communication can still be handled using the other one. Similarly, HCs can be reached over both interfaces. Even if one of them fails, an HC is still able to communicate using its interface to the other Ethernet. At this point it should be mentioned, that communication between hosts on the network is provided by an already existing communication service, which is used to enable host applications to communicate with each other without using both of the destination host's IP addresses. A host within this system is identified by a unique host ID. Given this ID, the communication service determines both IP addresses and tries to connect to either one of them. Higher level applications using the communication service don't know which interface is used for communication, but they are aware of whether a connection could be established or not. In fact, this should be enough an application is interested in.

Thus, applications running on hosts in this system don't need to know about the redundant network structure which was designed due to the demand for high reliability and availability.

The demand for high availability is obvious. It means, that a Class One host must be able to communicate with other hosts on the network, even if one of its network connections is down (e.g. by either an interface card malfunction, or by a connection failure like a broken cable, hub port, etc.). Reliability means, that if a path between the sending and the destination host exists, communication between these hosts must be assured, no matter to which of the Ethernets each of them is connected to.

Looking at figure 1.1 again, it should be noticed immediately that there is no direct connection between Class Two hosts on different Ethernets like between SC A and SC B even in a system without failures. Of course they do have to communicate with each other, which implies that at least one of all Class One hosts has to relay datagrams between both subnets. To give a naming scheme, a host relaying datagrams within this system is called a router. This may be confusing somehow, because the term router is normally associated with a host within a TCP/IP network which is connected to two or more subnets making complex routing decisions by using routing tables and supporting protocols to communicate with other routers. This complexity is not needed here because only two subnets are involved, and no traffic from or to any other subnet outside this system will ever occur.

The subject of this diploma thesis is to provide a service which reliably assures the existence and availability of a router to enable communication between hosts of Class Two on both subnets. This service is called the Fault Tolerant Routing Service.

2 System Analysis

To understand the need for a Fault Tolerant Routing Service within this network using TCP/IP as the communication protocol, this chapter provides an analysis of the given network structure and the resulting problems implied by the failure of a Class One host's network connection.

2.1 Routing Granularity

As shown in Figure 1.1, even if all system components run properly, both SCs cannot communicate without the existence of a router which relays datagrams because there is no physical connection between them. Generally, each Class Two host on one Ethernet needs a router to be able to communicate with hosts of Class Two on the other Ethernet.

The communication path between Class Two hosts on both subnets always includes one datagram relaying host of Class One:

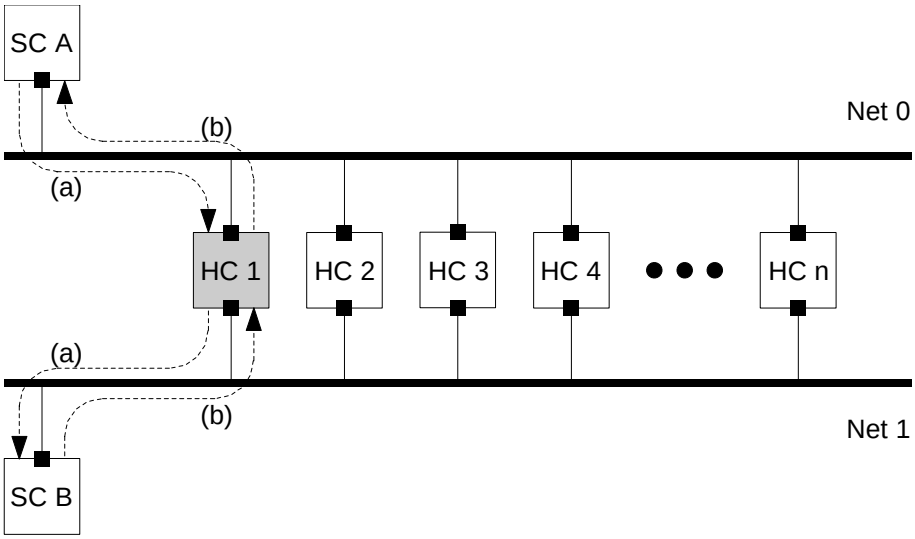

Figure 2.1: The router HC 1 relays datagrams between both subnets.

Unlike shown in this example, it doesn't necessarily have to be HC 1 or the 'leftmost' Class One host (whatever that may mean, e.g. the one with the smallest ID) being selected as the router between both subnets. In principle, each Class One host in the system is a candidate router and may be selected by the Fault Tolerant Routing Service to relay datagrams. Only if the purpose of a specific HC doesn't allow to handle more than its own network traffic, this host must be ignored by the algorithm selecting the router from the pool of candidates, although they may have both network interfaces running properly. The following discussion doesn't take such hosts into account, but restricts to hosts being designed to act as routers, if necessary.

In figure 2.1, only one Class One host is selected to be a router. This single router relays datagrams in both directions, from Net 0 to Net 1 (a) and vice versa (b), which may cause a significant additional load on this machine depending on the amount of datagrams it has to relay. The more Class Two hosts exist on both subnets, the more network traffic is to be relayed by this host. Whether or not this may cause a major impact on the router's performance depends of course on the amount of datagrams to be relayed, but also on the host's free capacity limited by the applications running on it.

A single router within this system also appears to be a single point of failure. Loosing at least one of both network connections, the router is no longer capable of relaying datagrams. Thus no more communication between both subnets is possible until another router is selected by the Fault Tolerant Routing Service.

In contrast to this single router solution, another possibility would be to have two routers within this system, one relaying datagrams from Net 0 to Net 1, and the other one relaying datagrams from Net 1 to Net 0. In TCP/IP networks this is quite possible.

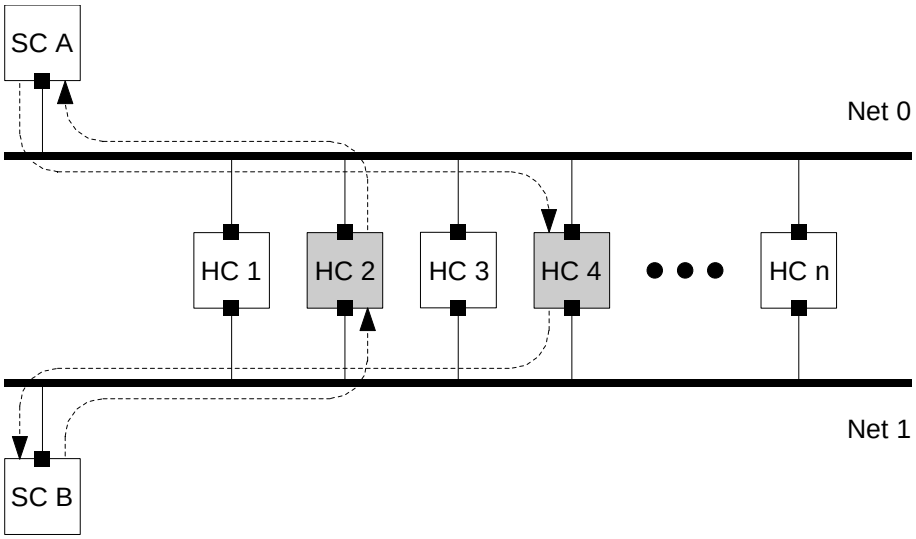

Figure 2.2: For each direction, one router is selected to relay datagrams.

Unlike the single router solution, the whole network traffic between both subnets is shared between two hosts, each one relaying datagrams in another direction. Thus, a router doesn't have to handle the whole network traffic which causes less performance impact. In addition, the system doesn't have a single point of failure with respect of the whole communication between both subnets. If one of the two routers fail, the system just lacks of the possibility to communicate into one direction until another router is established by the Fault Tolerant Routing Service. According to figure 2.2, if HC 4 fails for some reason, communication from Net 0 to Net 1 is down, whereas Class Two hosts on Net 0 can still be reached by Class Two hosts on Net 1 using HC 2 as a router.

Of course, this is just true with connectionless protocols like UDP. As soon as a connection-oriented protocol like TCP is used for the communication between Class Two hosts on both subnets, the failure of either router causes the complete failure of communication.

The idea of having more than one router in the system can be extended to a scenario where each Class Two host in the system selects its 'personal' router[1]. All datagrams to be relayed to the other subnet are sent to this Class One host. Thus, each router only has to handle additional network traffic from only one host. If such a router fails, all other Class Two hosts are still able to send because each one has its own router selected and doesn't depend on the failed one. Again, communicating with a connection-oriented communication protocol still suffers from a router failure. But in contrast to the 'one-router-for-all' solution, the 'personal-router' solution affects only the connection-oriented communication to and from a single Class Two host.

The more routers the system supports, the finer is the routing granularity used to control communication between both subnets. A system supporting only one router has the coarsest granularity in contrast to a system which supports an exclusive router for each Class Two host. In the latter case, the finest granularity is reached.

A major advantage of improving the granularity is that the whole network traffic between Class Two hosts on both subnets is shared between multiple routers, reducing the additional load on each one to a minimum. Also, not the whole system is affected on the failure of a single router, but only one Class

1 This can be assumed, because on failure of a network connection, a service engineer should be informed to repair the malfunctioning hard- or software as soon as possible. Thus the number of failed network connections should always be less than the number of remaining Class One hosts being able to relay datagrams.

Two host on one subnet being the 'client' of the failed router[2]. As a significant drawback the increase of administration overhead must be mentioned, which makes the service more complicated. In case of a single router for the whole system, only the information about which Class One host is the systemwide router must be maintained and announced to all Class Two hosts in contrast to the 'personal-router' solution. In the latter case the service must maintain one router per Class Two host. Thus, on failure of a router interface the service has to determine two new routers among all remaining Class One hosts, one for the Class Two host, the failed router was assigned to, and one for the failed router, which itself became a Class Two host and thus also needs a router to be able to communicate to other Class Two hosts on the other subnet.

Which strategy for the router granularity should be chosen by the implementation of the Fault Tolerant Routing Service depends on the desired goals to be achieved.

2.2 Router Failure

The reason why this system was designed with this degree of redundancy is obvious: hardware and software failures may occur at any time, affecting the communication between any two hosts. Each component is viewed as a source of potential failures and thus it is not possible to design the system with one static router running forever and reliably relaying datagrams from the beginning (i.e. system startup) until eternity or system shutdown (whatever may come first).

Each host can loose one or both of its network connections. However, the impact to the system is different, depending on whether the failed interface is one of a Class Two host, or one of a Class One host. A Class Two host interface failure just removes the respective host from the system. The failure of a Class One host interface can be divided up into whether the host was selected as a router or not. If it was not, the interface failure 'just' changes the host's class membership from Class One to Class Two. This affects only this host itself. However, on failure of a router interface, the communication between both subnets is disabled for a number of Class Two hosts. Depending on the routing granularity, this may impact only one Class Two host on only one subnet, or all Class Two hosts in the whole system. No matter which granularity is used, at least one Class One host must become the new router.

The following discussion restricts to a single router solution which describes the coarsest granularity, but can be generalized by the assumption, that the investigated part of the system used for explanation purposes here is just a subset of all hosts.

Figure 2.3 shows the same situation like figure 2.1 where HC 1 is selected as the systemwide router, but this time with a broken connection to Net 0 (e.g. caused by either an interface card malfunction, or by a connection failure like a broken cable). Without having HC 1 being connected to Net 0, Class Two hosts on this subnet are not capable any longer to send datagrams to HC 1 (a). Thus, Communication to both, HC 1 and all Class Two hosts on Net 1 is disabled[3]. Sent datagrams from Class Two hosts on Net 1 still reach HC 1 (b), so communication between them is still possible, but datagrams received by HC 1 which are destined to be relayed to Net 0, will never reach their destination host on Net 0. Communication from and to Class One hosts in the system is not affected at all by a router failure. This holds for both, HC 1 (c), and all other hosts in the system.

2 This is just true if communication using UDP is assumed. Using TCP, communication between the 'disabled' Class Two host and all other (Class Two) hosts on the other subnet is disabled.

3 In fact, HC 1 has already become a Class Two host because it has just one connection to a subnet left, but as it is still viewed as a router by all other hosts in the system, this is ignored for explanation purposes.

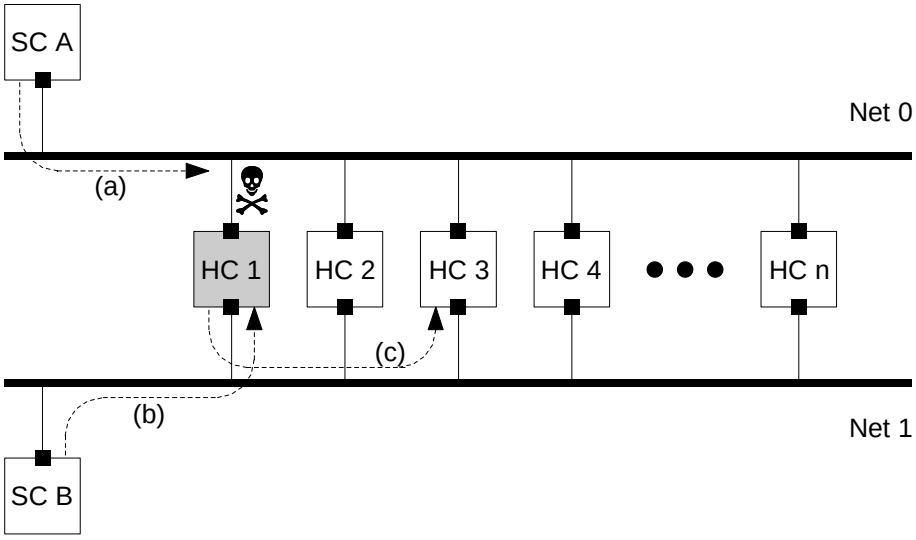

Figure 2.3: When the router HC 1 looses its connection to Net 0, it cannot be directly reached any longer by any Class Two host on Net 0 (a). But communication from and to hosts connected to Net 1 is still possible (b) (c).

It is obvious that this situation showed in figure 2.3 should be an intermediate state, because communication between Class Two hosts on both subnets must be enabled again as soon as possible. The situation now is, that HC 1 has become a Class Two host, having only one connection remaining to one of the subnets, here to Net 1. Instead of two, the system now has three Class Two hosts, SC A on Net 0 and SC B and HC 1 on Net 1, respectively. Because HC 1 was the only router within the system, but is not capable any longer to relay datagrams between both subnets, communication between Class Two hosts on both subnets (e.g. SC A on Net 0 and HC 1 on Net 1) is disabled unless another Class One host becomes a router.

Which of the remaining Class One hosts will be selected as the new router by the Fault Tolerant Routing Service is not a subject here. This will be discussed later. For now some magic is assumed, which selects a Class One host to become the the new router and informs all Class Two hosts about this change.

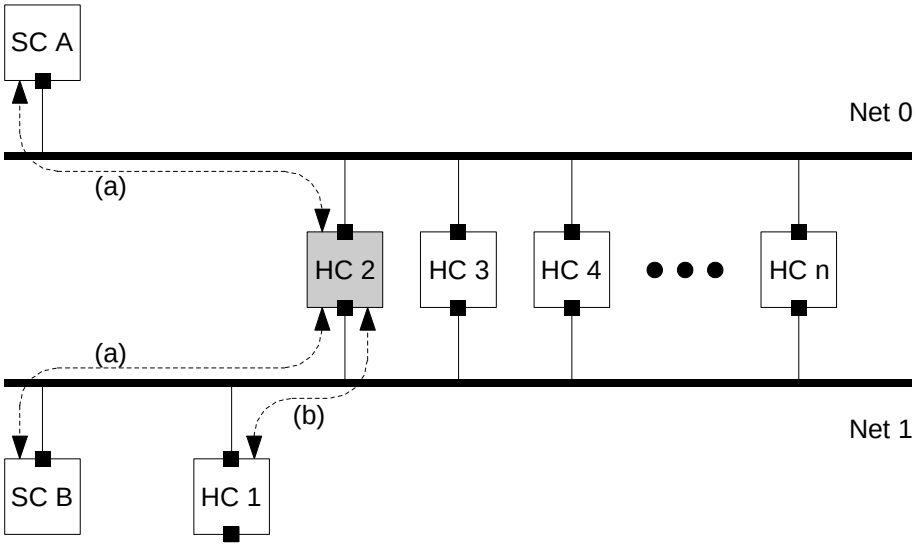

Figure 2.4: After failure of a router, another router must be selected to relay datagrams between Class Two hosts on both subnets (a). Routers do also communicate with other hosts in the system, like with HC 1 (b).

As soon as a new router exists within the system, communication between both subnets is enabled again. Class Two hosts are informed about the existence of a new router and send their datagrams destined for the other subnet not any longer to the failed router. The failed router itself is now handled as any other Class Two host in the network. Looking at figure 2.4 one can easily see that communication between both subnets is now assured by HC 2 representing the new router (a). To make clear, that being a router does not mean that the only purpose for this host remains to relay datagrams, figure 2.4 shows a communication arrow between the router itself and another host in the system, here HC 1 (b). Relaying datagrams is a task the router does in addition to its normal tasks which originally defined the purpose of this host within the system.

Like in the example above, the new router itself can fail, too. This causes the Fault Tolerant Routing Service again to find another router in the system.

Figure 2.5 shows the same situation like figure 2.4 with respect of communicating pairs of hosts. After the failure of the router's network interface 1, HC 2 can't relay datagrams any longer and thus another host must become a router. Communication between both SCs doesn't substantially change, because datagrams still have to be relayed, only the router between both hosts has changed (a). Thus, both SCs do only have to change the information about the next-hop router datagrams destined for the other subnet have to be send to. For the communication between HC 1 and HC 2 the situation is different and seems to be slightly more complicated. Before the failure of interface 1 they had a direct connection they used to exchange datagrams. Now that interface 1 of HC 2 is down, they must send datagrams to the router to communicate with each other (b). The interesting part of this situation is, that instead of communicating over HC 2's interface 1, communication now is established using its interface 0. This switch from interface 1 to interface 0 also implies the use of different IP addresses as the destination IP address for datagrams, because each interface belongs to another subnet. However, this is not really a big deal for the applications, because the above mentioned communication service covers exactly these situations and thus makes an interface failure transparent for applications.

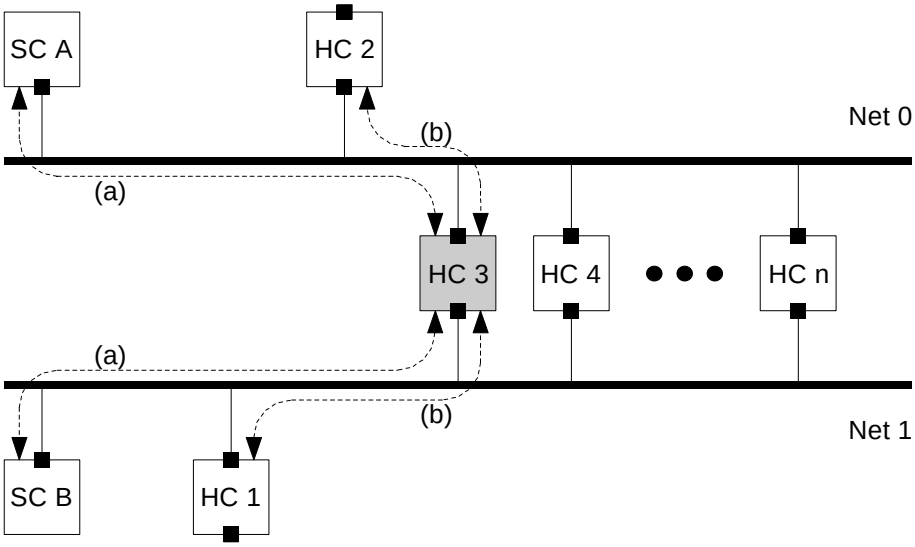

Figure 2.5: After failure of HC 1 and HC 2 as routers, HC 3 is now the new router.

Taking this into account, HC 1 now tries to contact HC 2 over its other interface using its second IP address, which, as already said, belongs to a different subnet. The path to the other subnet already includes a next-hop router, and this is why HC 1 sends datagrams destined for HC 2 automatically to the router. But this is in no way different to the situation between both SCs. Before HC 2 failed as a router, HC 1 also sent datagrams to hosts on the other subnet (e.g. to SC A) to the router, because they didn't have a direct connection, too. Each Class two host within this system has the information which Class One host relays datagrams, otherwise communication between them would be impossible. Thus, after the failure of HC 2 as a router, the same happens to HC 1 (and all other Class Two hosts in the

network) like what happened to the SCs: only the information has to be changed, which router is to be used within the network to relay datagrams from one subnet to the other.

Selecting a new router within the pool of Class One hosts and propagating this information to all Class Two hosts are the Fault Tolerant Routing Service's main tasks.

2.3 Service Tasks

The discussion above gave an overview of the system's behavior with respect of routing datagrams from one subnet to the other for both situations, a properly running system, and in case of connection failures. The redundancy within this system is designed to guarantee, that on failure of any component there is always another one to be able to take over and act as a backup. For the routing purpose, this implies, that the system must have a router as long as there is at least one Class One host in the system. If a router fails, another host being connected to both subnets must become a router.

Of course, this takeover between routers doesn't happen automatically by some magic but is the task of the Fault Tolerant Routing Service. The tasks set to this service are the following:

1. Establish a router on system startup.
2. Detect the failure of the current router.
3. If a router failure is detected, select another Class One host to become the new router.

   This must be assured as long as there is a Class One host existing in the system. Otherwise the system consists of two separated network segments.

3 Communication In TCP/IP Networks

TCP/IP is a protocol suite with a lot of possibilities to solve the problem of relaying datagrams. The purpose of the Fault Tolerant Routing Service is to find a host being able to relay datagrams, and prepare itself and all Class Two hosts for this task. The way this is done by TCP/IP affects the way of preparing the participating hosts. Thus, a very short but sufficient introduction of the TCP/IP stack is given in this chapter, followed by an analysis of the single steps taken by the datagram exchange procedure and their implications on the different layers of the protocol stack. The actions described here can be considered as a general TCP/IP procedure, but instead of a general description, the underlying system structure is taken into account wherever needed. So, the focus is still on the 'problem' of communicating from Net 0 to Net 1 and vice versa via a single intermediate host relaying IP packets.

3.1 The TCP/IP Stack

In the chapters above and in the following discussion the terms *packet* and *datagram* are used synonymously although the literature strictly distinguishes between them (see [3], [4], [15], [20] and [21]). To understand why both terms are used as synonyms, a short explanation and definition of these two terms in addition to a few other related to them is given here.

In the following figure the TCP/IP stack is shown with its particular layers compared to the ISO/OSI[4] reference model (see [23]). The ISO/OSI reference model consists of seven layers, each one representing a different abstraction layer providing specific functions to the layer above. The same applies to the different layers within the TCP/IP stack, but unlike the ISO/OSI reference model, a TCP/IP stack was designed with only four layers.

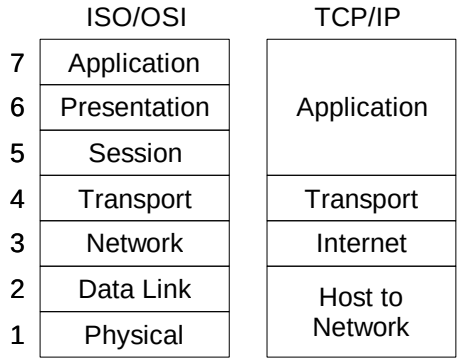

Figure 3.1: The layers of the ISO/OSI reference model compared to the TCP/IP stack.

Figure 3.1 shows the corresponding layers of both, the ISO/OSI reference model, and the TCP/IP stack. Although the following discussion restricts to the TCP/IP stack, it might be useful to see how each layer corresponds to one or more of the well-defined layers in the ISO/OSI reference model.

The topmost level in either reference model is the application layer. Unlike in the ISO/OSI reference model, an application using the TCP/IP stack accesses the transport layer directly, because there is no presentation and session layer in between. Thus, the application itself has to provide the functionality of both layers, if desired.

The transport layer provides several end-to-end protocols like the connection oriented TCP (Transmission Control Protocol) and the connectionless UDP (User Datagram Protocol). These protocols allow communication between peer entities (applications) on the source and destination hosts.

The connectionless internet layer provides access to the packet-switching network. If a transport layer protocol wants to send data to another host, it passes a *message* to the internet layer plus the informa-

4 ISO = International Standards Organization
OSI = Open Systems Interconnection

tion where this message should be sent to, namely the destination host's internet address and the destination port number. The internet layer in turn adds an Internet Protocol (IP) header to the message which includes among other things the destination IP address. Both the message and the IP header form an IP *datagram*. Now, this IP datagram has to be sent over the physical network which provides frames of a defined length, referred to as the maximum transfer unit (MTU). For standard 10 Mbit Ethernet the MTU size is 1.500 bytes. If the IP datagram fits into such a frame, it can be sent in one piece as the payload of a frame. However, if the frame is too small to carry the whole datagram, the datagram must be fragmented to fit the frame's size. Each *fragment* gets the same IP header as the original IP datagram with respect of the destination host's address and related information[5] and can be sent over the network media within a frame. The payload of a frame is called an IP *packet*. Thus, an IP packet is either the whole datagram (IP header plus message), or consists of a fragment (IP header plus message portion). On receipt of packets which belong to a fragmented datagram they have to be reassembled again. For routing purposes it is not important whether this happens on each intermediate host before a datagram is forwarded, or only on the destination host before the datagram's content, the message, is reached to the transport layer protocol. A router doesn't care about a packet's content (datagram or fragment) because it is only interested in the correct destination IP address in the packet's IP header.

Dependent on the receiving host's configuration, IP datagrams are completely reassembled before they will be forwarded or not. In either way IP packets containing either fragments or complete datagrams are routed through the internet exactly the same way using the destination host's address in the IP header of each packet. Thus, the terms IP datagram and IP packet are used synonymously, because with respect of routing it makes no difference what a frame contains traveling the network as long as a valid IP header with a valid destination IP address exists.

The term 'packet-switching' network means that IP packets are injected by a host's internet layer into the network and then are forwarded independently by the internet layer of other hosts along their path until the destination host is reached. For every single packet a new routing decision is made on each forwarding host (router) the packet passes on its way through the internet. This is desired, because one requirement is, to enable communication even in case of lost network connections in the middle of an already ongoing communication or if a new and/or faster connection is brought up. In both cases a router can choose a different network interface or a different destination host to forward the packet. Thus, as long as there is at least one path from the sender to the destination host, communication is assured. Deciding a route for each packet independently can result in an out-of-order arrival of IP packets at the destination host. On the destination host, the data portion of the IP datagram, the message, is passed to the corresponding protocol within the transport layer. If in-order delivery is desired, the transport layer protocol is asked to put the messages into the right order again. The internet layer has not even an idea of a packet order.

The host-to-network layer is not well defined within the TCP/IP stack. It is just said, that there must be some way to send IP packets over a physical media. This is normally done by sending a frame containing an IP packet over whatever network media. Strictly speaking, it isn't interesting at all which underlying physical network technology is used. The only important thing is, whether an IP packet could be delivered properly or not.

3.2 IP Packet Processing Overview

With respect of routing IP packets the most interesting layer within the TCP/IP stack is the internet layer. Both the internet layer's interaction with its adjacent layers (the host-to-network layer and the transport layer) and the processing of IP packets within the internet layer is shown in this section. The complete path of an IP packet within the TCP/IP stack is followed by the discussion below. It starts with the reception of an incoming Ethernet frame and ends with the sending of one. All single intermediate steps like processing an incoming IP packet, forwarding it and sending an outgoing IP packet to its destination is discussed in detail in the following sections.

5 Of course a fragment's IP header does have to have some information about fragmentation to enable reassembling the original IP datagram. But this is ignored as it is not a subject here.

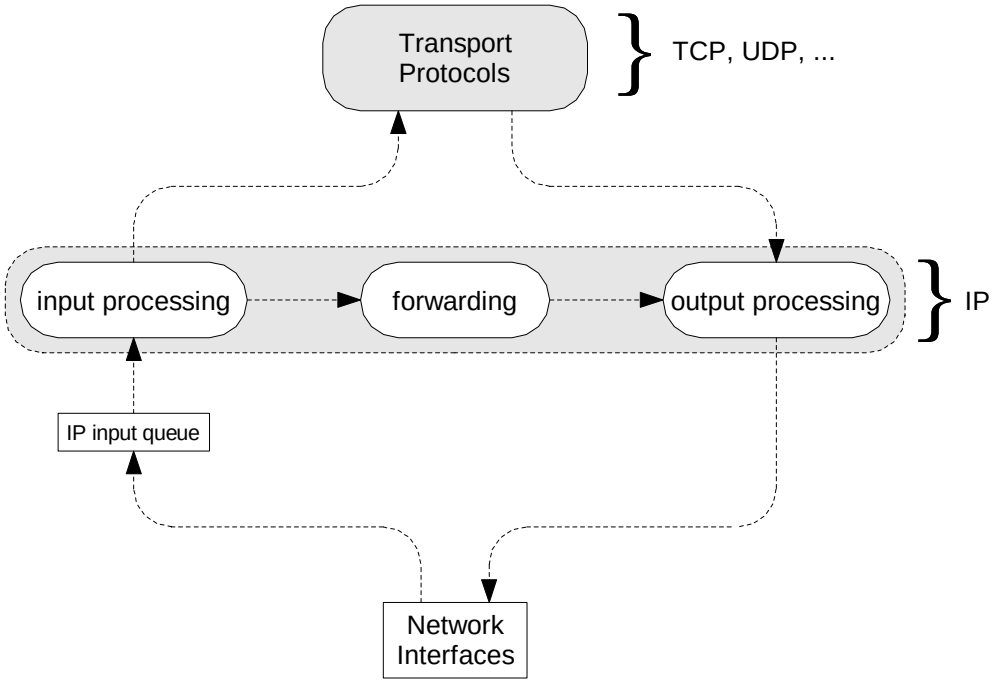

Figure 3.2: An illustration for the processing scheme of IP packets.

On arrival of an IP packet it is checked, whether it has reached its final destination. If so, the datagram has to be reassembled if necessary (the following discussion assumes that this happens transparently) and the message portion is passed to the corresponding transport layer protocol. If the receiving host is not the destination host, the packet has to be forwarded if the system is configured to act as a router. The forwarding procedure tries to find either the destination, or the next-hop router and sends the packet to the output procedure. Transport layer protocols also pass messages to the output procedure which selects a network interface to transmit the datagram. See Figure 3.2 for an illustration.

IP packets which cannot be processed are mostly silently discarded. Only in some situations it is desired to return an ICMP message to the packet's original sender to inform him about the lost packet and, what is more important, about the reason why the packet was discarded. However, these ICMP messages have no implication on routing and are thus not considered in this document.

Also, the following discussion ignores internal aspects like memory allocation and possible fragmentation of received IP packets within the memory caused by the method they are stored. By accessing an IP packet in memory it is considered to be available and complete. A complete IP packet means, the logical packet length determined from the IP header is equal or less than the physical packet length stored in memory. Implementation details like the data structures used to store IP packets or the access methods are not considered here.

3.2.1 Receiving IP Packets

Network interfaces like Ethernet cards always listen to the media traffic. Generally, there are two possibilities of processing valid frames arriving at a network interface card. The first possibility is to compare its own hardware address[6] to the destination address in the frame header. If they are equal, or the destination address is the hardware broadcast address[7], then a hardware interrupt is invoked calling the device driver which places the packet into the appropriate input queue for the protocol specified by the frame type field in the Ethernet frame header. For IP version 4 (IPv4) the value of the frame type field equals to 0x0800.

6 Each Ethernet interface card is identified by its own globally unique hardware address which is 48 bits long.
7 All 48 address bits are 1.

The second method would be to invoke a hardware interrupt on each arrival of a valid frame, no matter whether the receiving interface card is the desired destination. In this case, the interface card is called to run in *promiscuous mode*. Although this can be quite useful for network diagnostics, it is not always desired, because apart from performance implications by raising interrupts for all packets instead of only the own ones, reading other frames may also violate security constraints. For the purpose here only frames arriving at the desired destination are of any interest.

After placing the incoming IP packet into the Internet Protocol's input queue, the device driver schedules a software interrupt for the input processing routine. The input processing function called by the kernel on the occurrence of the software interrupt processes all packets in the input queue until it is empty.

The input processing routine dequeues a packet and verifies whether it is a valid IP packet[8]. If it is not, the packet is silently discarded. A valid IP packet's options are processed next. It is assumed that either no options are present, or that the options don't have any implication on the following discussion.

Now the packet's destination IP address is compared to all of the host's addresses to determine whether the packet has reached its final destination. Two system models were invented to distinguish between two strategies:

The *strong end system model* only compares the destination address with all addresses assigned to the receiving interface. If packets arrive on an interface other than the one specified by the destination address, the packet is discarded. In contrast, a *weak end system model* accepts those packets as long as there is any interface being configured with this address.

Dependent on whether the weak end or strong end system model is implemented, either all defined unicast addresses assigned to all interfaces are used for the comparison, or only the ones of the receiving interface. If no unicast address matches, the destination address is compared to all broadcast addresses of the receiving interface. Three broadcast addresses have to be considered: the network broadcast address, the subnet broadcast address (which can be the same as the network broadcast address) and the limited broadcast address.

In either way, if the final destination is reached, the data portion of the IP datagram is passed to the corresponding transport layer protocol. This can be done either by putting the data into the transport layer protocol's input queue or by calling an input processing function of the appropriate transport layer protocol. The IP input processing function can then access the next packet from the input queue.

If the final destination is not reached and the destination IP address is not a broadcast address, the IP packet is forwarded, if the system is configured to act as a router. This is usually done by setting the flag for IP Forwarding.

3.2.2 Forwarding IP Packets

Before an IP packet can be forwarded, the packet's header must be modified by decrementing the time-to-live field by 1[9]. If the value is 0 after this operation, the packet is discarded. Because of the modified value, a new checksum must also be computed and stored in the header.

Then the next hop must be located. This is done by searching the routing table for an entry which matches the packet's destination IP address. If such an entry is found, the packet can be forwarded. See Section 3.4 for a discussion of the routing table and its entries.

If no entry can be found in the routing table for the packet's destination IP address, the packet is dropped and an ICMP host unreachable error is returned. If the receiving host is the first hop router on its way through the internet, an entry is found in the routing table and the specified outgoing interface for this route is the same like the one on which the packet was received, an ICMP redirect message is

8 A valid IP packet has an IP version of 4 in its header, and the result of the header checksum computation is 0.

9 According to RFC 791 the time-to-live field is measured in seconds, but almost all TCP/IP implementations just decrement the value of this field at each hop the packet passes on its journey through the internet.

sent to the sending host if the system is configured to do so. See Section 3.5.1 for a slightly more detailed discussion on this subject.

3.2.3 Sending IP Packets

As shown in Figure 3.2, the IP output routine receives datagrams from both, the IP forwarding routine, and the transport layer protocols.

If a forwarded datagram must be sent, the IP header is already filled in almost complete as well as a route is already found, so only little additional work has to be done. However, a datagram received from neither transport layer protocol has neither an IP header nor an IP address for the next hop determined by searching the routing table. So both the IP header must be constructed and a route must be found for the destination IP address passed along with the message.

Next, the IP datagram along with the IP address of the next hop is passed to the output routine of the interface device driver. This routine fills an ethernet frame with the datagram[10] and sends it over the network media to the next hop. The next hop can either be the final destination of the IP datagram or a router. If it is the final destination, both IP addresses are the same, the one in the IP header which is always the destination host's IP address and the next hop's address indicated by the routing entry. If the next hop is a router, the IP address in the routing entry is different from the one in the datagram's IP header.

In either way, the next hop IP address must be converted into the hardware address of the next hop's receiving interface. To determine this hardware address, ARP is used which is discussed in more detail in Section 3.3. First, the ARP cache is searched for an entry matching the next hop IP address. If such an entry is found, the corresponding hardware address is copied into the frame's destination hardware address field. If the search fails, the interface driver broadcasts an ARP Request to determine the hardware address corresponding to the next hop IP address. This assignment is always unique, because an interface has only a single hardware address, but could be configured with one ore more IP addresses.

On receipt of an ARP Reply, which includes the desired next hop's hardware address, the frame can finally be sent over the network media. The hardware address is also used to update the ARP cache. Thus, subsequent datagrams which have to be sent to the same destination won't cause another ARP Request but can be sent immediately to the next hop.

3.3 ARP Packet Processing Overview

The Address Resolution Protocol (ARP) is always used by the interface device driver to determine the hardware address for a given IP address the driver can't resolve. ARP is limited to a single physical network which supports broadcast on the hardware level. A high level description of the usage of ARP is given in Section 3.2.3, but to really understand how ARP gathers the desired information, a closer look is necessary.

Like other protocols, ARP exchanges information using packets which are sent over the network. The Ethernet device driver identifies ARP packets by the *frame type* field in the Ethernet header and queues them into the ARP input queue. This procedure is similar to the one for IP packets.

10 If the interface's MTU size is smaller than the size of the datagram, the datagram is fragmented. But as already mentioned, fragmentation is ignored.

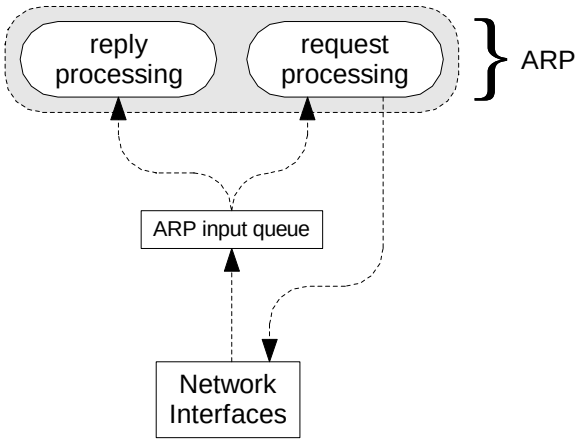

Figure 3.3: ARP packets are identified by the ethernet *frame type* field and are then put into the ARP input queue. Requests are answered by sending a reply whereas incoming replies are used to update the ARP cache.

Each incoming Ethernet frame is demultiplexed by the interface device driver according to the *frame type* field in the Ethernet header which contains 0x0806 to identify ARP packets. These packets are then processed by the ARP routine according to the value in the *operation* field of the ARP packet header. See Appendix A for the format of an ARP packet.

3.3.1 Determining a Hardware Address

Each time the Ethernet device driver gets an IP datagram from the IP layer, it tries to send it to the target host's interface identified by the IP address in the routing entry. Again it is mentioned, that this IP address is not necessarily the same like the destination IP address given in the IP header. Only if the next hop is not a router, but the final destination host, both IP addresses are equal.

To resolve the hardware address from an IP address, the device driver uses ARP. First, the ARP cache is searched for an entry matching the target IP address. If one is found, the corresponding hardware address stored in this entry is copied to the Ethernet frame header along with the sender's interface hardware address and a *frame type* of 0x0800 identifying the frame as and IP datagram. The frame can now be sent over the network.

In case the cache has no matching entry for a given IP address, the hardware address is obtained by issuing an ARP Request. After filling the protocol specific parts in the ARP packet header and setting the value for the *operation* field to 1 (ARP Request), the sender copies its own hardware and IP address into the packet's sender address fields and the unresolved IP address into the packet's target IP address field. The target Ethernet address field is left unfilled, because this is exactly what is tried to be obtained by the request. The exact format for ARP packets is shown in Appendix A.3.

An ARP Request is always broadcast over the Ethernet the sending interface is connected to, thus it is assured that each host connected to that Ethernet receives the request. Thus, the Ethernet frame's destination address is the hardware broadcast address (FF:FF:FF:FF:FF:FF), the source address is the senders hardware address, and the *frame type* is set to 0x0806, identifying the frame data as an ARP packet. The host with the unresolved IP address answers the request with an ARP Reply containing the desired hardware address of its interface.

After a little while an ARP Reply from the target host arrives at the interface the request was sent on. ARP Replies are identified by an *operation* field value of 2 and contain the target host's hardware address. After updating the ARP cache with the new information, the hardware address can finally be used to send all pending datagrams to the target host.

3.3.2 Answering ARP Requests

On receipt of an ARP Request, the first thing done by the receiving host is to use the senders IP and hardware address to update the local ARP cache. Next, the unresolved IP address is compared with the IP addresses assigned to the receiving interface. If there is a match, the host answers the request. This is usually done by changing the value of the *operation* field to 2, swapping both, the hardware and the IP addresses, and copying its own hardware address into the sender hardware address field. Then the ARP Reply is sent back directly to the originator of the ARP Request by copying the target hardware address (the former sender hardware address) into the Ethernet frame's destination address field. Also, the sender's hardware address field is filled into the Ethernet frame's source address field, and the frame type is set to 0x0806 (ARP).

3.3.3 Proxy ARP

The original intention of ARP is obvious. If a hosts wants to send datagrams to another one on the same physical network and only knows the target protocol address but not the corresponding hardware address, it just asks for the latter. Broadcasting the ARP Request assures that all connected hosts receive the request and the host being configured with the unresolved target protocol address answers.

It is not assumed that any other host than the one configured with the target IP address answers the ARP Request. But this is exactly what Proxy ARP does: incoming requests are answered by the proxy ARP agent with the own hardware address although the answering host's interface is not configured with the target IP address.

Proxy ARP can generally be used for three purposes:

- If there are hosts connected to the same physical network which don't have ARP implemented, a proxy ARP agent answers ARP Request on behalf of them. A proxy ARP agent has to maintain a table containing all IP addresses and the corresponding hardware addresses of the other hosts connected on the same physical network. Figure 3.4 illustrates this:

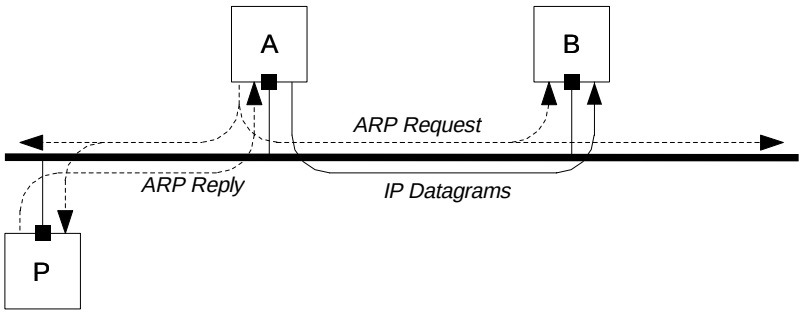

Figure 3.4: A proxy ARP agent answers requests on behalf of hosts which don't have ARP implemented.

  In the example in Figure 3.4, host A broadcasts an ARP Request over the network asking for B's hardware address. Because host B has not implemented ARP, it cannot answer. Instead the proxy ARP agent P answers the request with an ARP Reply containing B's hardware address. Now host A can address frames containing IP datagrams destined for host B with the correct hardware address.

- An ARP agent can act as a router between two physical networks by fooling the sender of an ARP Request into thinking, that the router is the target host. The router answers requests by sending its own hardware address back to the sender as a reply for the target IP address. The sender then sends its packets to the router which in turn forwards them to the real destination on one of its other interfaces. Usually the incoming and sending interface is different, otherwise it would not make a lot of sense. The proxy ARP agent must be configured as a router to be able to forward IP packets.

- The third use of proxy ARP is to hide two physical networks from each other and make them appear as it was a single physical network[11]. A proxy ARP agent answers requests issued by hosts on the one network on behalf of hosts on the other network. Arriving packets are just relayed between both networks. Thus it seems like there exists only one physical network instead of two. Like in the example above the proxy ARP agent must be configured as a router, too.

  In Figure 3.5 this is illustrated in more detail.

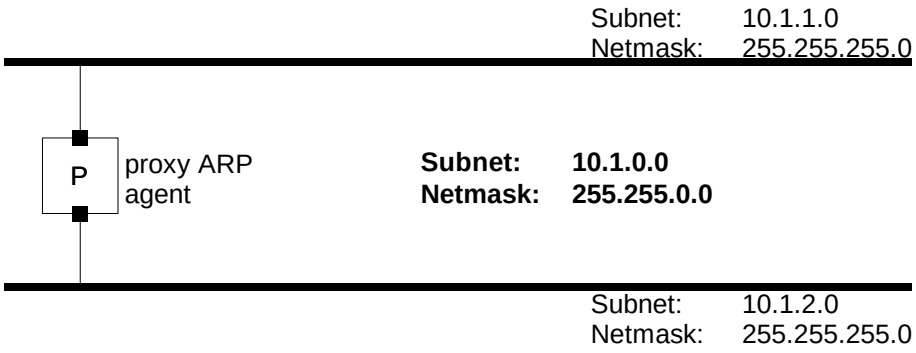

Figure 3.5: The two physical networks, each one configured with a subnet mask of 255.255.255.0, appear as a single physical network with a subnet mask of 255.255.0.0.

In the example shown in Figure 3.5 two physical Ethernets represent the subnets 10.1.1.0 and 10.1.2.0, respectively, each one with a subnet mask of 255.255.255.0. However, hosts on either network are configured with a network ID of 10.1.0.0 and a subnet mask of 255.255.0.0 which make them think they are attached to the same physical network representing the subnet 10.1.0.0.

Thus, if a host on one physical network, say the one representing subnet 10.1.1.0, wants to send datagrams to a destination host on the other one, subnet 10.1.2.0, it proceeds exactly like if the destination was directly connected to the same physical network. The destination IP address is compared to the network ID, the interface is configured with. Since the interface is configured with a subnet mask of 255.255.0.0, the resulting network ID is 10.1.0.0. After the datagram is passed to the device driver, an ARP Request is broadcast on subnet 10.1.1.0 asking for a hardware address for the destination IP address on subnet 10.1.2.0. Since Ethernet broadcasts are still limited to one physical network, the destination host is not able to answer because it doesn't receive the request. Instead, the proxy ARP agent P sends a reply with its own hardware address for the interface connected to subnet 10.1.1.0 to fool the sender which uses P's hardware address to send datagrams to hosts on the other subnet. P, of course, must be configured as a router to be able to rely those datagrams.

Looking at the routing table for the sending host, an entry like the following one must occur:

```
destination  gateway    mask           flags   interface
10.1.0.0     0.0.0.0    255.255.0.0    U       eth0
```

This technique can also be used to hide two physical networks from each other. If hosts on one physical network are configured with a network mask of 255.255.255.0 and don't have a routing table entry for the other subnet, this other subnet is hidden and unreachable. They have no idea how to send datagrams to hosts on the hidden network except to send them along the default route.

Changing the subnet mask is not the only way to make hosts on the other network reachable. Another possibility is to leave the subnet mask and add a routing table entry for the other subnet using the same interface. Instead of one entry for the (virtual) subnet 10.1.0.0 with the corresponding subnet mask of 255.255.0.0, the routing table now has two entries for both subnets 10.1.1.0 and 10.1.2.0, each one having a subnet mask of 255.255.255.0.

11 This is why Proxy ARP is also called *promiscuous ARP* or *the ARP hack*.

Routing table entries for this way of configuration can look like the following:

```
destination  gateway    mask            flags   interface
10.1.1.0     0.0.0.0    255.255.255.0   U       eth0
10.1.2.0     0.0.0.0    255.255.255.0   U       eth0
```

Looking at Figure 3.5 again, the network structure reminds of the one being investigated by this document. And indeed, for the purpose of this document, the last possible use of proxy ARP is interesting and may be used as a solution of the routing problem between Class Two hosts on different networks. But as mentioned above, the remaining interface on all HCs becoming Class Two hosts must be reconfigured with either a different network mask or with an interface change for the subnet, the broken interface was connected to.

3.3.4 Gratuitous ARP

Usually, ARP Requests are issued to resolve IP addresses for other hosts on the network. At first glance it doesn't make a lot of sense asking for its own hardware address. But on a closer look those gratuitous ARP Requests still make sense.

If a host's interface fails and was replaced by another one, it now has a new hardware address. This doesn't affect hosts which never communicated to it before, because on start of conversation, ARP Requests are issued to determine the (new) hardware address. But hosts which already have an ARP cache entry for that host will use the wrong hardware address because they have no idea about the change. Ethernet frames sent with the old hardware address never reach their destination.

To prevent those situations where invalid ARP cache entries may exist, a host sends ARP Request at bootstrap time, asking for its own hardware address. Each host on the network receiving this Request first updates its local ARP cache. If an entry for the host's IP address already exists, the hardware address is updated. Otherwise a new entry is added to the cache. Because the sender asked for its own hardware address, no reply is sent back to him until another host on the network is configured with the same IP address. If this happens, the system is not configured properly.

The other possibility of gratuitous ARP is to enable a backup host to take over transparently if the primary host fails. As soon as the backup recognizes the failure, it sends an ARP request asking for the IP address of the primary host, but giving its own hardware address. This causes an update of all ARP caches, and subsequent packets are now sent to the backup host. Other hosts on the network have no idea of the primary host's failure.

If proxy ARP is used to solve the routing problem of relaying datagrams from Net 0 to Net 1 in the underlying network structure, Gratuitous ARP can be used to cope with router failures. As soon as a new router is selected by the Fault Tolerant Routing Service, each Class Two hosts on both networks have to be informed by sending a gratuitous ARP Request.

For every Class Two host H_1 on Net 1, an ARP Request must be send on Net 0 asking for H_1's or the router's hardware address, filling the *sender IP address* field in the ARP packet with H_1's IP address and the *sender hardware address* field with the router's hardware address on Net 0. Similarly, for each host H_0 on Net 0 gratuitous ARP Request is sent on Net 1 with a combination of H_0's IP address and the router's hardware address on Net 1.

There is one big problem with this method. Because ARP Requests are usually broadcast, all hosts on the sending network media receive these requests. With 'all', both types of hosts are meant, Class Two and Class One hosts. While Class Two hosts are wanted to receive them, it wreaks havoc if Class One hosts update their ARP cache[12] on receipt of those gratuitous ARP Requests.

A closer look on what happens on a Class One host on receipt of such a gratuitous ARP Request makes it clear. For explanation purposes it is assumed that the incoming ARP Request associates the IP address of a host H on Net 1 with the hardware address of the router's interface connected to Net 0. After updating the ARP cache, the Class one host wants to communicate with host H on Net 1. Since

12 It is assumed that the ARP cache on a host is global. I.e. there exists only one cache per host for all interfaces. Supporting an exclusive ARP cache for each interface would avoid this problem.

it has an interface to Net 1, the device driver sends the packets directly to this interface. Before the packet can be sent, ARP is used to determine H's hardware address to fill the *destination address* field in the Ethernet frame. Unfortunately, ARP first examines the cache, which makes the device driver think, H's IP address is already resolved, but the hardware address is not the one of H's interface connected to Net 1, but the hardware address of the router's interface connected to Net 0. Not knowing this, the device driver uses the wrong hardware address from the cache and sends the frame over Net 1. Because no interface exists with this address on Net 1, nobody receives the frame, especially not H, which was the dedicated receiver!

To avoid this, gratuitous ARP Requests must selectively be sent only to Class Two hosts, which means a significant increase of complexity. The router (or any other authoritative instance within this system selected by the Fault Tolerant Routing Service) must send ARP Requests for each Class Two host on Net 1 to each Class Two host on Net 0, and vice versa. With n_0 Class Two hosts on Net 0 and n_1 Class Two hosts on Net 1, this leads to a total amount of $2 \cdot n_0 \cdot n_1$ gratuitous ARP requests.

The number of requests depend on the number of routers and the number of Class Two hosts, the router relays datagrams to. E.g. if the system consists of m routers and each router relays datagrams between only a subset of all Class Two hosts on Net 0 to only another subset of Class Two hosts on Net 1, the total amount of gratuitous ARP Requests reduces to $2 \cdot n_{0,m} \cdot n_{1,m}$. Where $n_{0,m}$ is the number of hosts in the subset on Net 0 and $n_{1,m}$ the number of hosts in the subset on Net 1.

3.4 The Routing Table

Every time a packet must be sent over the network, the routing table is consulted to determine the next hop. There are three entry types in a routing table and a matching entry is searched with the following priority:

1. First, an exact match for the destination IP address is searched.
2. A matching network address is searched next. This can either be a net ID or a combination of a net ID and a subnet ID.
3. The default entry in the routing table matches all destination IP addresses and is used if none of the searches for the entry types above were successful.

None of the entry types is mandatory in the routing table. It is even possible for a host to have an empty routing table with no entries at all, not even a default entry, although this would hardly make any sense for a host being connected to a network.

Routing table entries can be distinguished between two different kinds of routes, *direct routes* and *indirect routes*. A direct route specifies that the next hop to send the packet to is the destination host itself. This implies that the destination host is connected to the same physical network like the outgoing interface on the local (forwarding) host. Thus, a routing table entry specifying a direct route also specifies which local interface should be used to send packets to the destination host. Indirect routes send packets to another router within the network. This next-hop router is also connected to the same physical network like the interface specified in the routing table entry.

Because the underlying network structure can be considered as an intranet without any communication path to other hosts outside this system, the routing table contains only direct routes.

3.5 Updating Routing Tables

Routing tables are not considered to be static, i.e. their entries are set up at bootstrap time and then never changed. Quite the reverse can occur on running systems. Because the network topology could completely change, routing tables must be able to be adapted to reflect the changes and assure reliable routing even if former routes are not available any longer.

Thus, routing tables can be modified during operation. To achieve a routing table modification, several possibilities exist on different levels. TCP/IP itself supports a transparent way of routing table modification using ICMP Redirect Messages and ICMP Router Discovery Messages. Whereas ICMP

Redirect Messages are completely generated and processed by the network layer, a user process (a daemon) handles ICMP Router Discovery Messages.

Furthermore there exist some application level protocols which also perform routing table changes. The most popular ones are the Routing Information Protocol (RIP) and the Open Shortest Path First (OSPF) protocol. Both protocols have daemons running on each host exchanging messages with sufficient information to build the local routing table and to reflect changes in the network topology.

To enable user processes to access the routing table, the `ioctl()` system call is provided to be used with the request parameter set to either `SIOCADDRT` to add a routing table entry or `SIOCDELRT` to delete one. To list the routing table entries, either a routing socket must be created by a kernel process with subsequent write and read operations to determine the information, or the `sysctl()` call must be used.

3.5.1 ICMP Redirect Messages

As mentioned above, an ICMP Redirect Message is send by a first hop router if the interface used to forward the packet is the same like the one the packet was received on. This is, because the sending host could have sent the packet directly to the correct next hop router. The rules for sending such a redirect message are the following:

- A packet must be received and resent on the same interface,
- the selected route must not have been itself created by an ICMP Redirect Message and
- it must not be the default route.

This ensures that a router only propagates information it is authorized for.

On receipt of an ICMP Redirect Message, the sending host updates its routing table and adds an entry for the destination address (and only the destination address!) with the correct information about the first hop router. All following packets to that particular destination address are then sent directly to the correct first hop router.

While redirects should be limited to hosts in the internet environment, the drawbacks using host redirects in the underlying network structure are even worse than with Gratuitous ARP. Not only each Class Two host must be informed selectively about the router change. Also, for each Class Two host on the other subnet an ICMP Redirect Message would have to be generated. Thus, having n_0 hosts on Net 0 and n_1 hosts on Net 1, the total amount of necessary messages would be $2 \cdot n_0 \cdot n_1$.

Using network redirects reduces the amount of messages having to be sent over the network to n_0+n_1, because each Class Two host on either subnet would receive only one ICMP Redirect Message for the other subnet.

Broadcasting these redirect messages is not an option: Class One hosts should not change their routing tables because they can communicate directly with each destination host.

3.5.2 ICMP Router Discovery Messages

The general concept of these ICMP messages is that hosts can initialize their routing tables with a default entry by broadcasting or multicasting a request, an ICMP Router Solicitation Message, instead of using some kind of static local configuration file. All routers on the network which receive these messages answer by sending an ICMP Router Advertisement Message, containing one or more router addresses. Additionally, all routers broadcast or multicast their advertisements periodically. Thus, the hosts can update their default route if necessary. ICMP Router Discovery Messages are explained in more detail in [6] and [20].

For the purpose here, an ICMP Router Solicitation Message should only be sent by Class Two hosts which just started running, or by a Class One hosts just having lost one of its interfaces. The answer would be an ICMP Router Advertisement Message sent by the router. Although there might be still some Class One hosts in the system, advertisements could be broadcasted. This would not cause any trouble, since only the default route is affected by the receipt of an ICMP Router Advertisement Message. Having two running links implies that exactly one routing table entry exists for each interface

with its destination network being the same as the subnet, each Ethernet represents. Thus, adding a default route would not cause any trouble in finding the correct outgoing interface for each subnet.

Although the possible solution using ICMP Router Discovery Messages seem to be quite reasonable, it still suffers from major drawbacks:

- A Class One host, which not happens to be the router between both subnets, still has to detect whether both of its interfaces are running properly. And if it finally recognizes an interface failure, it must remove the routing table entry for that failed interface to begin using the default route. If there is no default route entry an ICMP Router Solicitation Message must be sent to get an answer from the router.
- Router failures have to be detected by both, the router itself, and the Class Two hosts depending on it. If there is no automatic takeover between (potential) routers, the default route would just time out without being replaced. Since no other router is defined, no more ICMP Router Advertisement Messages would be sent to add a new default route entry or update the invalid one. Similarly, if an interface goes down and the router doesn't recognize it, it continues sending advertisements although it is not able any longer to relay datagrams.
- A potential router must know from some authoritative instance that is has two running interfaces before it can start sending ICMP Router Advertisement Messages.
- If each Class One host was sending advertisements to cover router failures, the receiving Class Two hosts would update their routing tables on receipt of each advertisement. This could lead to an undesired high frequency of table updates.

3.5.3 Routing Protocols

Common Routing protocols like RIP (Routing Information Protocol) and OSPF (Open Shortest Path First) are also concerning about routing, but since they are designed for a completely different kind of network topology they don't offer a proper solution for the purpose here. A detailed description for those protocols is given in [3], [8], [20], [23] many RFCs also deal with this subject.

4 Routing Service Strategies

Based on the discussion of the TCP/IP stack and its implementation details in the previous chapter, the variety of realizing routing and dealing with router failures is quite large. Each solution comes along with several actions both partners have to perform, the Class One host which becomes the new router and Class Two hosts which start using it. And, to make the discussion a little bit more complex, the new router can be either elected between all Class One hosts themselves, or selected by one or more hosts of Class Two.

Whatever solution is chosen to be implemented, one thing always remains the same: in case of a Class One host recognizing a failure of one of its interfaces, its routing table must be changed to assure that all network traffic to the subnet, the failed interface was connected to, now passes the remaining running interface. The exact change depends on what solution is selected, but at least the entry for the failed interface must be removed. The important thing is, that at this time, the host must either already know about the new router, or must be able to find out about it, if necessary.

The following discussion distinguishes between the election and the selection approach. Both differ significantly in the ways each participating host is involved.

4.1 Router Election

Electing a router means that all Class One hosts identify themselves and/or each other as a host with two running interfaces and then start negotiating with other hosts of Class One which one of them should become the router. After all have agreed with the election, the new router has to propagate the information about its new role to all affected hosts of Class Two on both networks. Depending on the implemented routing technique, the router has to know which hosts on each network are of Class Two.

As one can see, the complexity of this solution can be quite remarkable. First, the group of router candidates must be identified. Each Class One host has to join the group since all hosts with two running interfaces are automatically router candidates. Then an election has to be performed by all group members. The winner of the election is the new router which now has to advertise its new role to all Class Two hosts.

How this advertisement is done depends on which feature of the TCP/IP stack is used. With proxy ARP, the new router has to send gratuitous ARP Requests selectively to all Class Two hosts on both networks. As already mentioned above, the number of requests is $2 \cdot n_0 \cdot n_1$, with n_0 being the number of Class Two hosts on Net 0 and n_1 being the number of Class Two hosts on Net 1. This approach requires the new router to identify each single Class Two host and its hardware address prior to sending the gratuitous ARP requests. To acquire this information the Fault Tolerant Routing Service itself has to find out this information by either accessing the internal ARP table (see. [19], [20] and [21]), or by a special protocol gathering the hardware address directly from the respective Class Two host. Whatever solution is chosen, the complexity is remarkable.

Sending ICMP Router Redirect Messages suffers from almost the same drawback. The only difference is that no hardware address has to be determined in advance. This makes it a little less complicated since a redirect message can be sent immediately to the Class Two hosts IP address. But since a redirect message is only valid for a single destination host (remember: network redirects are not supported), the amount of messages necessary to inform each Class Two host is still the same: $2 \cdot n_0 \cdot n_1$.

The situation concerning the amount of necessary messages is better with ICMP Router Discovery Messages. The new router advertises its new role by just broadcasting an ICMP Router Advertisement Message on each network after the election. All hosts update their default entry in the routing table and Class Two hosts then automatically send their messages destined for hosts on the other network to the new router. This is the most reasonable solution for the Router Election Approach. However, since ICMP Router Discovery Messages are relatively new, not each TCP/IP implementation supports them (see. [9], [10], [20] and [21]). Especially OS Open, which is the operating system of interest here doesn't support these ICMP messages and thus this solution is not applicable.

Theoretically it would be possible to support multiple routers in the election approach. However, the complexity of the Fault Tolerant Routing Service then increases significantly. Assuming the election algorithm supports n routers and m is the number of Class Two hosts within the system. The election algorithm first has to find n routers which have to divide all m Class Two hosts amongst them. Then each router has to inform all and only 'its own' Class Two hosts that it is the new router. To ensure that each Class Two host is taken into account, the whole group must know about each single Class Two host. To assure this, an additional and more complex protocol between all group members is necessary which performs the division of all m Class Two hosts into n disjunct subsets.

4.2 Router Selection

If a router is selected by Class Two hosts, the ability to relay datagrams must be inherent with each Class One host. I.e. that IP forwarding must be enabled at each host and that a Class One host usually doesn't know whether or not it is selected by a Class Two host as its router.

A Class Two host selects a router in two cases. In case of the failure of the router itself, the Class Two host must identify a new router and has to change the entry for the other network which usually is the default route. If a Class One host looses one of its interfaces, it also has to select a router to be able to continue communication with Class Two hosts on the other network. It is important, that this solution requires to enable each host identifying all router candidates. There is no need for sending any kind of 'request for router identification' since each Class One host is a potential router and can be selected by any Class Two host at any time.

The main advantage of this solution is that Class One hosts are not affected with respect of finding router candidates as long as they have two running interfaces. Thus there is no other additional overhead on Class One hosts concerning the router selection than to periodically check the state of their interfaces. The responsibility of selecting a router is completely with Class Two hosts and only in case of an interface failure a host with initially two running interfaces becomes involved in the problem of finding a router. This significantly reduces the complexity of the Fault Tolerant Routing Service. Another positive aspect is that load balancing can be implemented relatively easy. Whereas the election approach usually identifies a single router for all Class Two hosts, this selection approach enables the distribution of the network traffic between both subnets amongst more than one router. This is important since relaying datagrams may cause a significant performance impact on the router.

Another advantage of this approach is that none of the TCP/IP specific solutions must be used. The less implementation dependent details are used for a the purpose of both identifying router candidates and updating the routing table, the more likely it is that the solution will work properly. This is true since each implementation of the TCP/IP stack is different from each other (see. [19], [20] and [21]).

5 The Global Link State

The global link state reflects the state of each host's links to either network. For each host the view provides two flags, one for each network interface, signaling whether or not the host is reachable over the respective interface. In a correct view, both flags are set for a Class One host, which is connected to both networks, whereas a Class Two host has only one flag set, depending on which interface is the one running properly. Having this view is mandatory for all Class Two hosts since the router selection algorithm depends on it because it uses this view as the basis for finding router candidates.

To understand what the global link state means, an example representation is given below along with a short description. A system with five nodes is assumed, two Class Two hosts (A and B), and three Class One hosts (C, D and E). Each of the Class Two hosts is connected to a different network. Host A is connected to Net 0, and B is connected to Net 1.

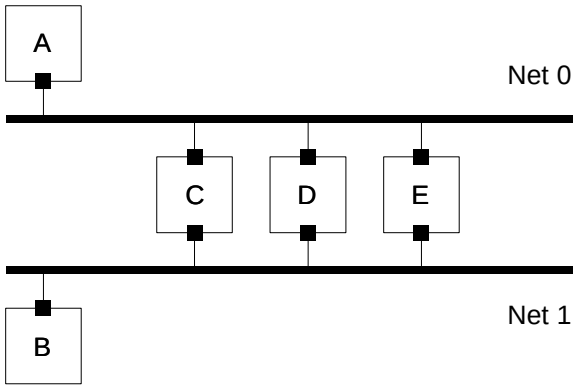

Figure 5.1: The example system used to explain the global link state.

In the bottom row of the table below each '*' (asterisk) represents a set flag indicating a running interface in the global link state.

	A	B	C	D	E
∑	* / -	- / *	* / *	* / *	* / *

The ∑-row contains a correct view of the global link state. As one can see, the flag is set for A's interface on Net 0, but not set for its Net 1 interface. Quite the reverse is true with B. The flag for its Net 1 interface is set, but not the flag for its Net 0 interface. C, D, and E have both flags set since they are hosts of Class One.

Since a protocol which provides each host with the correct view has to cope with all kinds of randomly occurring failures, the view must be rebuilt periodically, at least by each Class Two host. This is to assure that no host is selected as a router which doesn't have two running interfaces any longer or which has completely stopped operation by any reason.

The first protocol introduced here provides each host with the information about other hosts' link state by each host periodically broadcasting LINK STATE messages on each network it has a running connection to. These messages contain a list of the sender's connections to hosts on the other network than the one, the message is broadcast on. This enables Class Two hosts to learn about other hosts connections on the other network.

Then, a flooding protocol is introduced. Its characteristic is that messages which are broadcast by each host a on each connected network are forwarded by all receiving Class One hosts on the respective other network. Receiving a forwarded message makes the receiver know that the forwarder is a router candidate.

6 Connection List Protocol

A somehow different approach is made with this protocol. Each host builds its view of the global link state by processing incoming LINK STATE messages from other hosts. These messages serve two purposes: first, the receiving host knows that it has a connection to the sending host on the receiving network, and, second, the sending host provides the receiver with a list of its own connections on the respective other network. This list is contained in the message body and is especially used by Class Two hosts. Since Class Two hosts are connected to only one network, they would never learn about any connection on the other network if they were not given this list. And without any connection lists concerning the other network, a Class Two host would never be able to identify a Class One host. It would just know about other hosts it can reach over its connected network, but not whether or not the respective other one has another running interface connected to the other network. After having received a list from each host, the receiver knows about its own connections to the senders on the receiving network and learned about their respective connections to other hosts on the other network. Having this information enables it to build its view of the global link state.

6.1 Assumptions

The following assumptions are made in this chapter:

- Each host has a systemwide unique identifier, a *host ID*.
- Each message is provided with a *message sequence number* which is unique with respect to the sending host. The *message sequence number* is supposed to be an unsigned integer value which counts to infinity, i.e. no overflow must be handled.
- Since the Link State Determination Protocol introduced here uses message entities of a connectionless network communication protocol, all its restrictions also apply and must be taken into account. These are namely the possible loss of messages, the lack of an acknowledge mechanism for messages and the possibility for a disordered message delivery. I.e. a host has no idea whether sent messages are received by any other host in the right order.
- On receipt of a message:
  - The message is complete, i.e. the message is received in one piece containing all information or not at all.
  - The message is correct. No checksum mechanism is needed to verify the correctness of the message content. This functionality is provided by the underlying network communication protocol.
  - The message sender can be identified (at least by a field in the message header containing the sender's *host ID*).
- The *host ID* can be used for addressing. This is assumed to generalize the discussion of this protocol. Later, in the implementation specific part, the *host ID* must be resolved to the network protocol address. How this can be done is not a subject here, but is discussed later in Section 9.4.3.

6.2 LINK STATE Message Format

A LINK STATE message contains the following fields:

- *host ID*, identifies the message sender
- *message sequence number*, uniquely identifies a message with respect to the sending host
- *router priority*, indicates the priority with which the sending hosts wants to become a router (the higher the value, the bigger is the host's desire to become a router; zero means: only if no other host is available)
- *list length*, indicates how many connections are listed in the message body

- *connection list*; a connection to another host is represented by the *host ID* of the connected host

  Again: A connected host is a host the sender itself has received a LINK STATE message from. Thus, the sender knows about its own connection to the respective other host. E.g. if a host received three LINK STATE messages on Net 0 from A, C and D, the connection list of its own LINK STATE message being broadcast on Net 1 would contain the *host ID*s of A, C and D.

The following figure shows the format of a LINK STATE message:

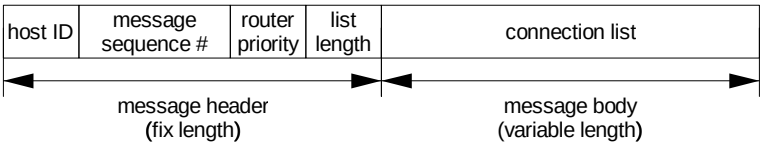

Figure 6.1: Each LINK STATE message has a fix length header and a variable length body containing a list of connections. This can also be an empty list.

6.3 The Protocol

The protocol introduced here provides each host with the necessary information to build its view of the global link state. As already mentioned above, each host periodically broadcasts LINK STATE messages on all networks it has a running connection to. The time interval between two subsequent broadcasts is called the broadcast interval and since the broadcasting of these messages is not synchronized, each host broadcasts its messages independently. Thus, messages arrive at a host's interface in an arbitrary order.

Although it is assumed that each host within the system is configured with the same value for the broadcast interval, it cannot be assured that exactly one message from each host arrives within this interval. There are several reasons causing this undesired extend of the broadcast interval on a sending host: system timers may run on different speeds; the sending of messages may be delayed due to a temporary performance leak; etc. To cope with both the randomly arriving messages and the possibility of delayed LINK STATE messages another time interval for 'checkpoints' is necessary. At checkpoint time, all messages arrived during the most recent checkpoint interval are processed. Not only the new view is built, but also new connection lists for each interface are composed for all subsequent LINK STATE messages sent by this host. Thus, the checkpoint interval must be large enough to ensure that there is enough time for messages to arrive. Otherwise a host is not able to build a complete and correct view of the global link state, not to mention the connection lists. To avoid this, at least one message from each host must be available, and thus the checkpoint interval must be larger than the broadcast interval.

Figure 6.2 below illustrates the correlation between incoming messages, the broadcast interval and the checkpoint interval.

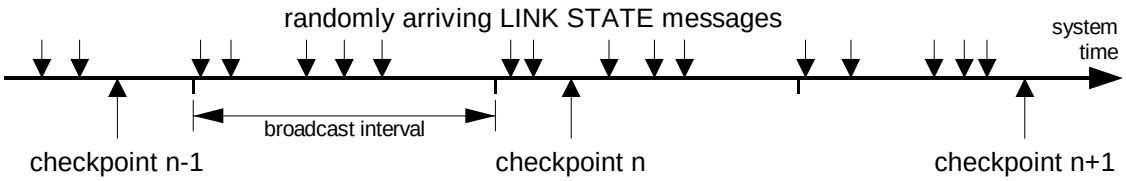

Figure 6.2: The checkpoint interval is larger than the broadcast interval to ensure that at least one message from each host can be received. The messages randomly arrive at a host's interface since broadcasting is not synchronized between them.

Since arriving LINK STATE messages are not immediately processed, they need to be temporarily stored until the next checkpoint. Then, both the new view of the global link state and the new connec-

tion lists, one for each interface, are built based on these messages. After this is done, all stored LINK STATE messages arrived during the most recent checkpoint interval are discarded.

Figure 6.3 below illustrates how the Fault Tolerant Routing Service stores randomly incoming LINK STATE messages and builds a new view of the global link state and the new connection lists at each checkpoint.

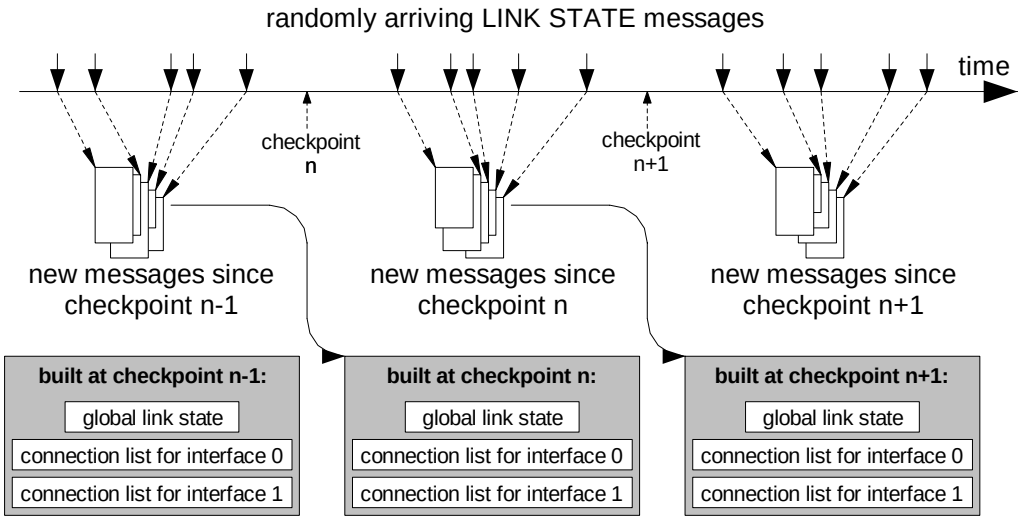

Figure 6.3: Incoming LINK STATE messages are stored until the next checkpoint. Then the host builds a new view of the global link state and connection lists, one for each interface, based on these messages. After this is done, the messages are discarded.

The following discussion is divided up into three sections. In the first section the receipt of a LINK STATE message and the necessary actions taken by the receiving host are discussed. The second section describes the subject of checkpointing including a more detailed discussion of the checkpoint interval. The third section deals with the subject of composing and broadcasting a LINK STATE message.

6.3.1 Message Receipt

Incoming LINK STATE messages are not immediately processed but are temporarily stored until the next checkpoint. The detailed actions taken by a host on receipt of a LINK STATE message are the following:

1. Discard broadcast echo.

   Broadcasting a message on a network always creates an echo on the sending interface. Now it is assured that only messages from other hosts are processed.

2. Discard old messages.

   On arrival of a message, it must be verified whether it is a new message. This is done by comparing the *message sequence number* in the message header to the one of the previously stored message received from the same host on the receiving interface. If the value is less or equal, it is either an old (delayed) message, or a duplicate. In both cases the message doesn't contain any new information and is discarded. Now it is assured that only messages containing new information are handled.

3. Save the new *message sequence number*.

   This must always be done to avoid unnecessary processing of old messages or duplicates in the previous step.

4. Store the message in an appropriate data structure for further processing at checkpoint time.

It must be assured that the receiving interface can be identified and that only the newest message from each host is contained in the data structure. If more than one message arrives from the same host between two checkpoints, only the newest message should be stored and the older message must be discarded. The newest message is the one with the largest *message sequence number* and not necessarily the one which was most recently received!

6.3.2 Checkpoints

At checkpoint time, a host processes all LINK STATE messages which arrived during the most recent checkpoint interval. If a message was processed immediately after its receipt, it could cause a major performance impact, depending on the amount of incoming messages. Instead, a host waits a specific period of time until it can be assumed to have received at least one message from each host. This period of time is called the checkpoint interval, and the actions taken after each interval is called a checkpoint.

Checkpoint Interval

A host never knows when a message arrives. This is because this protocol doesn't provide any synchronization mechanism between hosts for broadcasting their LINK STATE messages. However, since the broadcast interval is the same on each host, one message from each host is assumed to arrive within the broadcast interval plus a little latency time to cover message delays. Thus, to guarantee having at least one message from each host, which is necessary to get a complete and correct view, the checkpoint interval must always be larger than the broadcast interval.

Generally it should be avoided to get two or more messages from a single host within a checkpoint interval. Because since only the newest message is used to build the new view of the global link state, all older messages are discarded and thus the information contained within them is never used. However, this requirement can not be guaranteed. Even if the broadcast interval is the same on each host, system timer speeds can diverge between each host.

A checkpoint interval very close to the broadcast interval can cause an incomplete and thus an incorrect new view of the global link state because messages can be missing due to a delayed delivery. If these messages arrive right after the checkpoint they are not processed at checkpoint time, neither to build the new view of the global link state, nor to compose the new connection lists. Both then lack of the information contained in the delayed messages. The consequences are that other hosts don't learn about the sender's connections, and in the worst case the router selection algorithm could not be able to find a router candidate. Whether the latter happens depends on the total amount of hosts in the system: the less hosts participate in the protocol the more likely it is. But even if router candidates can be found, a different host can be selected to become the router, although the old one still works. This leads to two unnecessary changes of the routing table, because at the next checkpoint this change is very likely to be undone, because the delayed message is then processed.

Late messages are not processed but stored until the next checkpoint. Since network delays are not considered to occur each time, it is very likely for another message from the same host to arrive during the subsequent checkpoint interval. Then, the old message, which was the delayed one, is discarded and replaced by the new one. The more messages are replaced, the greater is the degree of redundancy. Having a great degree of redundancy causes unnecessary network overhead, because messages are sent over the network which are never used.

Since the consequences of an incorrect view are considered a lot more harmful than the increase of unnecessary network traffic, the checkpoint interval must be large enough to guarantee that all messages from each host can be received. For example, a checkpoint interval of 1.5 times the broadcast interval ensures that a host receives at least one message from each other host. From approximately half the number of hosts it even receives a second message which causes the receiving host to discard their previously received messages it has already stored. Thus, only 2/3 of the total amount of received messages are processed to build the new view of the global link state, and 1/3 is just discarded without ever being used.

The length of a checkpoint interval should be large enough to cover reasonable network delays and short enough to avoid too much messages being discarded.

Actions

The detailed actions taken at a checkpoint are the following:

1. Build the new view of the global link state using all messages which were received during the most recent checkpoint interval. A detailed description of this build is given in Chapter 6.4.

   Class Two hosts are required to run the router selection algorithm (see Chapter 8) based on the newly built view. If necessary, i.e. in case the algorithm selects a host as a router different from the one selected before, the routing table must be updated.

2. Build the new connection lists, one for each interface.

   This is done by adding the *host ID*s from all stored messages in the corresponding connection list.

This approach assumes that the priority to find a router is higher than the priority to send LINK STATE messages with the most actual connection list. If this is not the case or both steps are independent from each other (e.g. on multitasking systems), both steps can also be done in a different order or even at the same time.

Updating the routing table requires additional implementation specific information about hosts like their network protocol address. Since this chapter tries to keep the discussion as general as possible, the subject of how the implementation specific information can be acquired is omitted here. It is just assumed that this information is available on demand.

6.3.3 Broadcasts

Each host periodically broadcasts LINK STATE messages on each of its running interfaces. The time interval between two subsequent broadcasts is called the broadcast interval.

The connection list in a LINK STATE message which is intended to be broadcast on one network (say Net 0) contains all *host ID*s from the senders the host has received a message from the respective other network (Net 1) within its most recent checkpoint interval. Since these lists are built at each checkpoint, they are already available at broadcast time. Thus, a sending host has just to fill the field of the number of entries contained in the connection list into the message header of the new LINK STATE message and the list itself into the message body. No further computation is necessary.

6.4 Building The View Of The Global Link State

The following discussion shows how a host builds its view of the global link state at checkpoint time by processing all stored LINK STATE messages which arrived during the most recent checkpoint interval.

For explanation purposes, an example base system with four hosts is assumed. Two of them, hosts A and B, are of Class Two, which means they are connected to only one network. Host A's interface is connected to Net 0, whereas host B has its interface connected to Net 1. The other two hosts, C and D, are of Class One and thus connected to both networks, Net 0 and Net 1, respectively.

The following figure shows this example base system:

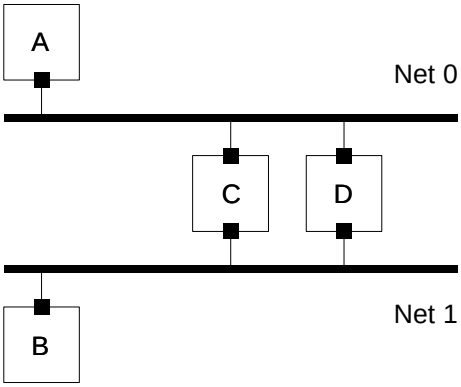

Figure 6.4: The example base system consists of two Class Two hosts (A and B), and two Class Two hosts (C and D).

Each host periodically broadcasts LINK STATE messages on each network it has an interface connected to. In general, LINK STATE messages broadcast on one network contain the sender's connections to hosts on the other network it is connected to. In this example system, a LINK STATE message broadcast by host A on Net 0 is always empty. This is because A doesn't have a connection to Net 1. B also sends empty messages on Net 1 because it is not connected to Net 0. In contrast, LINK STATE messages broadcast by hosts C and D on either network are never empty. C broadcasts on Net 0 that it has a connection to hosts B and D on Net 1. The messages broadcast on Net 1 contain the connections from C to A and D on Net 0. Similarly, D broadcasts its messages on Net 0 and Net 1 with connections to hosts B and C on Net 1, and hosts A and C on Net 0, respectively.

The following figure shows the LINK STATE messages broadcast by each host on the connected networks.

sender:	A		B		C		D	
interface:	0	1	0	1	0	1	0	1
connection list:	{ }	-	-	{ }	{ B, D }	{ A, D }	{ B, C }	{ A, C }

Figure 6.5: LINK STATE messages broadcast by each host in the example system. A and B send empty messages on Net 0 and Net 1, respectively. Both C and D send messages on each network containing the connections between each other and to A on Net 0 and to B on Net 1.

It is assumed that each host receives each LINK STATE message from each network it is connected to. Then, the messages stored by each host at checkpoint time are the following:

A		B		C		D	
0	1	0	1	0	1	0	1
C: { B, D } D: { B, C }	-	-	C: { A, D } D: { A, C }	A: { } D: { B, C }	B: { } D: { A, C }	A: { } C: { B, D }	B: { } C: { A, D }

Figure 6.6: Each host has received all messages broadcast on the connected networks. A has received two LINK STATE messages from C and D on Net 0, but none on Net 0. B has received all messages broadcast on Net 1. C and D have received all messages from both networks.

As an intermediate step, each host builds a connection table containing all connections between the hosts on each network. The two tables below shows the connection tables of A and D after having processed all stored LINK STATE messages:

A	A	B	C	D
A			K / -	K[(1)] / -
B			- / L	- / L[(2)]
C	K / -	- / L		- / L[(2)]
D	K[(1)] / -	- / L[(2)]	- / L[(2)]	

D	A	B	C	D
A			L / -	K / -
B			- / L	- / K[(3)]
C	L / -	- / L		K / K
D	K / -	- / K[(3)]	K / K	

Since the table is symmetric, either rows can be read from left to right or columns from the top to the bottom with each cell indicating the connections between the host in the leftmost column and the host in the topmost row. Connections are shown as a tuple like x / y, where x and y show the connection to another host on Net 0 and Net 1, respectively. Each of both variables x and y has one out of three possible values of either 'K', 'L' or '-'. '-' means that there is no connection between both hosts on the respective network detected. 'K' indicates that a host 'knows' about the connection to the respective other host because it has received a LINK STATE message from this host on the respective network . 'L' indicates a connection the receiving host has 'learned' from the content of an incoming LINK STATE message.

The numbers in round brackets beneath the connection tuples in the tables above LINK STATE messages and their impact to the connection tables shown in the following examples:

1. Host A has received a LINK STATE message from D on Net 0.

   By receiving a LINK STATE message, a host 'knows' about a connection to the sender. Thus, A 'knows' about its connection to D and marks this in its connection table by placing a 'K' for Net 0 in the appropriate cells.

2. In this LINK STATE message, D claims to have connections to B and C on Net 1.

   Additionally, A 'learns' about D's connections to B and C on Net 1 and marks them in its connection table by placing an 'L' for Net 1 in the appropriate cells. Since the quality of this information is different from receiving a LINK STATE message, this is also distinguished in the table above.

3. D receives an empty LINK STATE message from B on Net 1.

   D only marks its connection to B on Net 1. Since B claims not having connections to any host on Net 0, no other connection between B and other hosts is marked in D's table caused by the receipt of this message.

As one can see, each host's table contains only 'K's in its own row/column. This is easy to understand, because a host only knows about a connection to another host on a network if it has received a LINK STATE message from the other host on that network. If no message is received from a host on one network, it is very likely that there is no connection between both hosts on the respective network.

However, there may be situations when an 'L' appears in the own row of a hosts connection table instead of a 'K'. Assuming the following scenario shows this clearly: D receives a LINK STATE message from C on Net 0, containing the information that C has a connection to D on Net 1. This makes C to place a 'K' into its table for the connection to D on Net 0. However, without ever having received a message from D on Net 1, C cannot really verify the claimed connection to D on Net 1 and thus places an 'L' into its table. As soon as C receives a LINK STATE message from D on Net 1, the connection is proven to be existing and can be marked with a 'K'. This suspicious state can only occur between two Class One hosts and should always be resolved within a checkpoint interval. If this is not the case, and after the checkpoint interval a Class One host still has an 'L' instead of a 'K' in its row/column, this is a strong indication for either a lost LINK STATE message or a recent interface failure.

Since a 'K' is a stronger indication for a connection between two hosts than an 'L', an already existing 'L' for a learned connection between two hosts can be replaced with a 'K' at any time. A 'K' in turn is never replaced by an 'L', because a 'learned' connection is weaker than one a host 'knows' about. This is important for composing LINK STATE messages, because a host is only allowed to broadcast

'known' connections over the network. I.e. that only a 'K' in the host's own row/column adds the respective *host ID* to the connection list of its new LINK STATE message.

Using a single flag for both types of connections makes it easier to see differences between all hosts' connection tables. A set flag means that there is a connection between two hosts on the respective network, either a learned one, or one it knows about. Otherwise there is no connection assumed. In the tables below, a set flag is indicated by a '*' (asterisk). Also, a new row is added to each table containing the 'sum' of a host's connections, namely the one in the topmost row. All flags for a host's network connection are logically 'ored', i.e. if there is at least one flag set for a host indicating a connection to another host, it is assumed that the host's link to the suspicious network is up.

A	A	B	C	D
A			* / -	* / -
B			- / *	- / *
C	* / -	- / *		- / *
D	* / -	- / *	- / *	
∑	* / -	- / *	* / *	* / *

D	A	B	C	D
A			* / -	* / -
B			- / *	- / *
C	* / -	- / *		* / *
D	* / -	- / *	* / *	
∑	* / -	- / *	* / *	* / *

The ∑-row in each table is the respective host's view of the global link state.

An analysis of the tables shows the following:

- Class Two hosts never learn about other hosts' connections on the same network they are connected to.
  - Host A only learns about its own connection to C and D on Net 0 and their connections to each other and to B on Net 1. However, it never learns about C's and D's connections on Net 0.
  - The same happens to B on Net 1. B learns about its connection to C and D on Net 1, but never learns about C's and D's connections on Net 1.
- Class One hosts (C and D) have a complete and correct view of each hosts' connections because they get all information from both networks.
- Despite the fact that Class Two hosts suffer from a lack of information concerning connections between hosts on the same network than the respective Class Two host itself is connected to, the view of the global link state (the ∑-row) is the same for all hosts within the system!

6.5 Example Scenarios

The following sections show different network scenarios and each host's resulting view of the global link state which is built at each checkpoint. There are three most interesting cases among all others. These are namely the System Startup, the Class One Host Failure and the Class One Host Interface Failure. In the first scenario, System Startup, it is interesting to know how fast a Class Two host can identify router candidates. This is important, because communication between Class Two hosts on different networks should be established as soon as possible. The second and the third scenario mentioned above, the Class One Host Failure and the Class One Host Interface Failure, show how fast a potential router failure can be detected by Class Two hosts. This is important because in case of a router failure, a Class Two host's routing table should be updated as soon as possible to continue communication between both networks. All other scenarios are less important, but also discussed in the following sections to give a complete overview over all possible cases.

It is assumed that during a checkpoint interval each host receives at least one LINK STATE message from each other host it has a connection to. A checkpoint interval is also called an iteration. See Appendix B for a detailed description of how each host's connection table is build by processing incoming LINK STATE message broadcast on each network. The following sections only show each host's view of the global link state.

6.5.1 System Startup

In the system startup scenario the underlying system is the same as the one shown in figure 6.4. Hosts A and B are of Class Two and are connected to Net 0 and Net 1, respectively. Hosts C and D are of Class One and are connected to both networks.

In the first iteration no host has any idea about the existence of other hosts in the system. Thus, their LINK STATE messages are all empty. Each host's view of the global link state after the first checkpoint interval looks like shown below:

	A	B	C	D
A's view:	* / -	- / -	* / -	* / -
B's view:	- / -	- / *	- / *	- / *
C's view:	* / -	- / *	* / *	* / *
D's view:	* / -	- / *	* / *	* / *

While hosts C and D have the correct view, hosts A and B still don't know about C's and D's connections to the respective other networks. But since A and B are hosts of Class Two, they need this information for their router selection algorithm to find router candidates. Although hosts C and D are of Class One and don't need to find a router, they already have this information.

In the second iteration (after the first checkpoint), the connection table built by receiving only empty LINK STATE messages in the first iteration is used to build new LINK STATE messages. But this time, these messages are not empty but contain *host ID*s from hosts the sender has a 'K' in its own row/column for the connection between them. Each host's view after the second iteration is shown below:

	A	B	C	D
A's view:	* / -	- / *	* / *	* / *
B's view:	* / -	- / *	* / *	* / *
C's view:	* / -	- / *	* / *	* / *
D's view:	* / -	- / *	* / *	* / *

Now each host has the same view of the global link state. Especially A and B can now find all router candidates (C and D).

6.5.2 Class One Host Startup

The new host E is of Class One and thus connected to both networks. This is shown in the figure below:

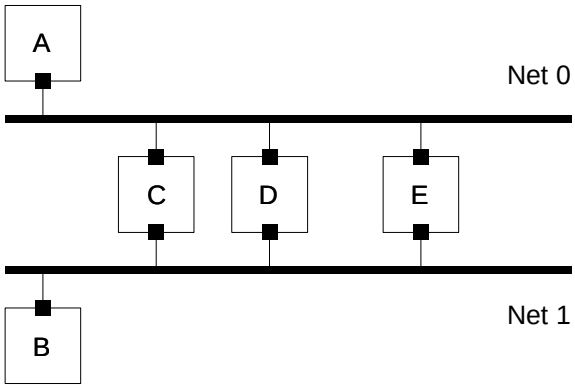

Figure 6.7: The new Class One host E is connected to both networks, Net 0 and Net 1, in the same example base system described above.

At E's boot time, all other hosts are in a stable state. I.e. that their connection tables didn't change and thus their LINK STATE messages are the same like after the second iteration in the scenario above. Only E's empty messages are added in the first iteration.

	A	B	C	D	E
A's view:	* / -	- / *	* / *	* / *	* / -
B's view:	* / -	- / *	* / *	* / *	- / *
C's view:	* / -	- / *	* / *	* / *	* / *
D's view:	* / -	- / *	* / *	* / *	* / *
E's view:	* / -	- / *	* / *	* / *	* / *

After the first iteration, A and B don't know yet that E is of Class One and thus a router candidate. Only C, D and E itself have the correct view of the global link state after this first iteration.

	A	B	C	D	E
A's view:	* / -	- / *	* / *	* / *	* / *
B's view:	* / -	- / *	* / *	* / *	* / *
C's view:	* / -	- / *	* / *	* / *	* / *
D's view:	* / -	- / *	* / *	* / *	* / *
E's view:	* / -	- / *	* / *	* / *	* / *

Now, after the second iteration, A and B also have the correct view of the global link state. Thus, E can be identified as a router candidate.

6.5.3 Class Two Host Startup

The new Class Two host F is connected to Net 0. The figure below shows the new example system.

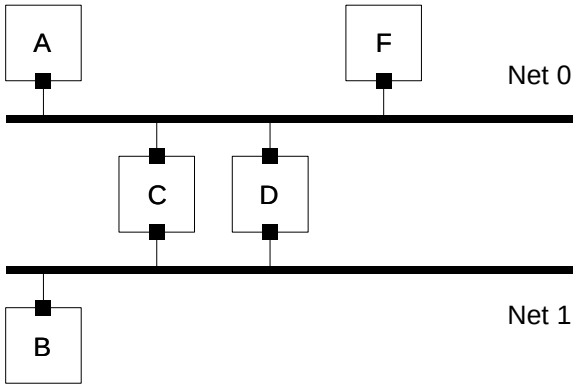

Figure 6.8: The new Class Two host F is connected to Net 0 in the same example base system described above.

At F's boot time, all other hosts are assumed to be in a stable state.

	A	B	C	D	F
A's view:	* / -	- / *	* / *	* / *	* / -
B's view:	* / -	- / *	* / *	* / *	- / -
C's view:	* / -	- / *	* / *	* / *	* / -
D's view:	* / -	- / *	* / *	* / *	* / -
F's view:	* / -	- / *	* / *	* / *	* / -

After this first iteration F has the correct view of the global link state and thus it is able to identify all router candidates. This is important since F is a host of Class Two and needs a router to be able to communicate with Class Two hosts on the other network.

Since F is only connected to Net 0, and thus broadcasts only empty LINK STATE messages, only the hosts which are also connected to this network have the correct view after the first iteration (here: A, C and D). Class Two hosts which are connected to the other network (here: B), have not yet learned about F's existence. This is done in the second iteration:

	A	B	C	D	F
A's view:	* / -	- / *	* / *	* / *	* / -
B's view:	* / -	- / *	* / *	* / *	* / -
C's view:	* / -	- / *	* / *	* / *	* / -
D's view:	* / -	- / *	* / *	* / *	* / -
F's view:	* / -	- / *	* / *	* / *	* / -

Now, B also knows about F. But since F is a host of Class Two, it will never be identified as a router candidate anyway.

6.5.4 Class One Host Interface Startup

In this scenario, the example system in figure 6.8 is assumed to be in a stable state before F's second interface to Net 1 starts working for some reason. F now becomes a host of Class One with two running interfaces, one to each network.

	A	B	C	D	F
A's view:	* / -	- / *	* / *	* / *	* / -
B's view:	* / -	- / *	* / *	* / *	* / *
C's view:	* / -	- / *	* / *	* / *	* / *
D's view:	* / -	- / *	* / *	* / *	* / *
F's view:	* / -	- / *	* / *	* / *	* / *

Only B, which is a host of Class Two connected to Net 1, and C and D, which are Class One hosts and thus also connected to Net 1, immediately update their view. Only A, which is a Class Two host on Net 0, still doesn't know that F is now a potential router.

	A	B	C	D	F
A's view:	* / -	- / *	* / *	* / *	* / *
B's view:	* / -	- / *	* / *	* / *	* / *
C's view:	* / -	- / *	* / *	* / *	* / *
D's view:	* / -	- / *	* / *	* / *	* / *
F's view:	* / -	- / *	* / *	* / *	* / *

After the second iteration, A has updated its view, too. Now F can be identified as a router candidate by the router selection algorithm running on A.

6.5.5 Class One Host Failure

The complete failure of a Class One host is shown in this section. In the example system shown in figure 6.4, which is assumed to be in a stable state, D fails for some reason. This is one of the two most interesting failure cases, since the failed Class One host can be selected as a router. And if a router fails, it should be detected as soon as possible by all Class Two hosts in the system.

After the first iteration the views of the global link state for each remaining host is like the following:

	A	B	C	D
A's view:	* / -	- / *	* / *	- / *
B's view:	* / -	- / *	* / *	* / -
C's view:	* / -	- / *	* / *	- / -

After only one iteration, the router selection algorithms running on each Class Two host don't identify D as a router candidate any longer. Although the complete failure is not detected, at least one of D's links is considered to be down. If D was a selected as a router before by one of the Class Two hosts, its failure is detected after only one iteration and another one can be selected. This is a good characteristic of this protocol.

C seems to have detected D's failure immediately, but this is just because it is the only Class One host an receives just empty LINK STATE messages from A and B. If there was at least one more host of Class One, C would receive messages from these hosts which still contain connections to D. This would lead to C thinking that D still had running interfaces.

	A	B	C	D
A's view:	* / -	- / *	* / *	- / -
B's view:	* / -	- / *	* / *	- / -
C's view:	* / -	- / *	* / *	- / -

After the second iteration, D is completely eliminated from all others' view.

6.5.6 Class Two Host Failure

The impact of the failure of a Class Two host to other hosts' view is shown here. Host A fails in the example system shown in figure 6.4. It is assumed that the system was in a stable state before A's crash.

	A	B	C	D
B's view:	* / -	- / *	* / *	* / *
C's view:	* / -	- / *	* / *	* / *
D's view:	* / -	- / *	* / *	* / *

None of the remaining hosts has detected A's failure in the first iteration. Since A was a host of Class Two, its failure is not considered to be harmful with respect of the purposes of the Fault Tolerant Routing Service.

	A	B	C	D
B's view:	- / -	- / *	* / *	* / *
C's view:	- / -	- / *	* / *	* / *
D's view:	- / -	- / *	* / *	* / *

Finally, after the second iteration, all remaining hosts have detected the failure of the Class Two host A.

6.5.7 Class One Host Interface Failure

This is the second of the two most interesting failure cases. Since the Class One host which interface fails could be selected as a router, its interface failure must be detected by all Class Two hosts to avoid a broken connection between both networks. It is assumed, that D's interface to Net 1 fails, while the system was in a stable state.

	A	B	C	D
A's view:	* / -	- / *	* / *	* / *
B's view:	* / -	- / *	* / *	* / -
C's view:	* / -	- / *	* / *	* / *
D's view:	* / -	- / *	* / *	* / *

After the first iteration, B, a host of Class Two on Net 1, is the only host in the system which has detected the D's interface failure. All other hosts, especially all Class Two hosts on Net 0, still think D has two running interfaces. This is not as good as in the case of a complete failure of a Class One host. But at least all Class Two hosts on the network, the failed interface is connected to, detect the failure and don't identify it as a router candidate any longer. However, Class Two hosts on Net 0, where D has still a running interface connected to, need a second iteration.

	A	B	C	D
A's view:	* / -	- / *	* / *	* / -
B's view:	* / -	- / *	* / *	* / -
C's view:	* / -	- / *	* / *	* / -
D's view:	* / -	- / *	* / *	* / -

Now, after the second iteration after D's interface failure, all hosts in the system have the correct view of D's link state. Especially all Class Two hosts on Net 0 are able to change their routing table in case D was selected by them as the router between both networks.

6.5.8 Summarizing Remarks

The really interesting thing is to know, when each host reflects a change of the global link state in its view. The following table summarizes all startup scenarios. Each row gives the number of necessary iterations for a Class One or a Class Two host, respectively.

	System Startup	Class One Host Startup	Class Two host Startup	Class One Host Interface Startup
Class One Host	1	1	1	1
Class Two Host	2	2	1 / 2 (1)	1 / 2 (2)

Ad (1): Class Two hosts connected to the same network like the new Class Two host are immediately aware of it. However, Class Two hosts on the other network need one more iteration until the information of the existence of the new Class Two host has reached them.

Ad (2): This is the same like with (1). Class Two hosts on the same network recognize the new interface immediately, whereas the information needs one iteration to be propagated to the other network's Class Two hosts.

All failure scenarios are summarized by the following table:

	Class One Host Failure	Class Two Host Failure	Class One Host Interface Failure
Class One Host	1	2	2
Class Two Host	1	1 / 2 (3)	1 / 2 (4)

Ad (3): Although not shown in the example scenario above, Class Two hosts connected to the same network like the failed Class Two host immediately recognize the failure. This is easy to understand, because the failed host doesn't send any messages any longer, and since it didn't have a connection on the other network, nobody claims to have a connection to the failed host in its LINK STATE message. However, Class Two hosts on the other network don't learn about the failure until the second iteration.

Ad (4): This is almost the same situation. Class Two hosts connected to the same network like the failed interface immediately recognize the failure. Class Two hosts on the other network don't learn about the failure until the second iteration.

The most important scenarios are the Class One host failure and the failure of a Class One host's interface. Since a Class One host could be selected as a router by a host of Class Two, its failure should be detected as soon as possible. And indeed, the only scenario when a Class Two host needs two iterations to recognize a failure is, when an interface on the other networks goes down. In all other cases a Class Two host immediately knows about the failure.

6.6 Optimization Possibilities

As the discussion above shows, this protocol still suffers from several shortcomings. The first one is that a new Class Two host needs a complete iteration to detect all router candidates. Depending on the value of the checkpoint interval, this could be too long since there might be applications which have to send urgent messages to Class Two hosts on the other network. But without having a router, these urgent messages have to be held until at the first checkpoint. To cope with the need for a fast router candidate detection, HELLO messages could be broadcast on each network as the first task of the Fault Tolerant Routing Service. These messages are answered by each receiving host sending LINK STATE messages directly back to the requestor. Thus, the new host can immediately build its view of the global link state and can make an entry into its routing table. Additionally, all other hosts receiving a HELLO message know about the new host and their view is then updated one iteration earlier.

Another optimization deals with the amount of unused LINK STATE messages received during a checkpoint interval. An unused message is a message, which is replaced by a newer message from the same host received during a checkpoint interval. This is very likely to happen, since the broadcast interval is shorter than the checkpoint interval. But if messages are just discarded and the information contained in them is not used, their broadcasting is an undesired overhead. A possibility to cope with this situation is to encourage each host keeping track of how many hosts are connected to it. As soon as all expected messages have arrived, the host performs a checkpoint. This optimization can significantly reduce the amount of unused messages and additionally enables a host to build its view of the global link state earlier.

Both ideas are discussed in more detail in the following sections.

6.6.1 HELLO Messages

Broadcasting a HELLO message is the first action performed by a host at the beginning of its first checkpoint interval. HELLO messages contain no other information than the sender's *host ID* and serves no other purpose than to request for a LINK STATE message from all recipients. A HELLO message is broadcast on each connected network at startup time of the Fault Tolerant Routing Service. All receiving hosts answer the request by sending back a reply directly to the sender. This reply is the most recent LINK STATE message composed by the host at its most recent checkpoint. No new LINK STATE message is composed to answer the request to reduce the performance overhead on the receiving machine.

The following figure illustrates this:

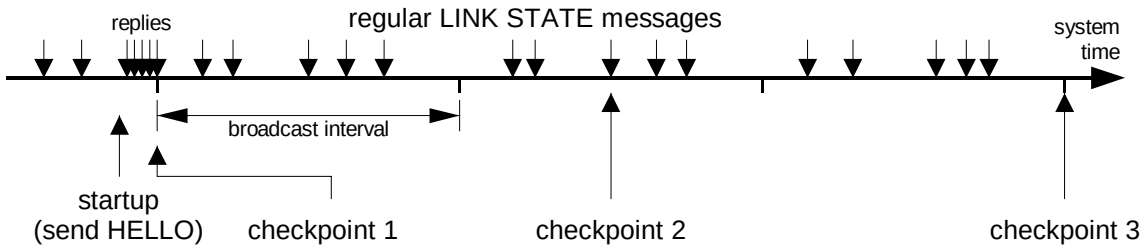

Figure 6.9: After a host has sent its HELLO message on startup time, all hosts answer with sending their actual LINK STATE message in addition to their regularly sent messages. As soon as a host has received all replies, it starts normal operation.

The reason why replies are not broadcast is, because it is very unlikely that the content has changed since the last broadcast. Sending replies directly to the requestor reduces the amount of received messages and thus causes less performance overhead on hosts other than the new one. Also, answering a HELLO message should not have any other impact to the normal processing. A broadcast reply could affect both the sender and all receivers. The sender is affected since its broadcast interval becomes significantly shorter. The receivers are affected because they suddenly receive an additional LINK

STATE message from all other hosts within a checkpoint interval. Thus it is very likely that at least two messages are received within a checkpoint interval. But since only the latest message is used, all others are discarded. To reduce the amount of unused messages, the broadcast interval could be started over as soon as a reply is sent. With all hosts doing this on receipt of a HELLO message, all subsequent broadcasts are suddenly synchronized. With each host broadcasting its LINK STATE messages at almost the same time, the probability of collisions on the Ethernet increases significantly.

The new host can receives all replies and builds its view of the global link state. In contrast to the normal way of processing LINK STATE messages, in the first checkpoint interval the new host's view is updated by each incoming message. As soon as a Class Two host has found a router candidate, the routing table entry for its default route can be added. The selected router is not necessarily the one which would have been selected at a normal checkpoint. But since the purpose of this optimization is to enable communication between both networks very fast, it is accepted that during the first checkpoint interval an inappropriate host may be selected. In this case, the first checkpoint will change the router if there are candidates detected having a higher priority than the currently selected one.

A host cannot wait for replies forever. There are two scenarios in which this is important. On system startup, there is always one host which may be the first one sending its HELLO message. If other hosts need a longer boot time, they may not receive this message and no answer is sent. Thus, after a relatively short timeout period the host should stop waiting for answers. It assumes to be the only host in the system and continues 'normal' operation. In this case this is just sending empty LINK STATE messages. As soon as the other hosts come up, it receives their HELLO messages and sends replies back to them.

Receiving a HELLO message is equal to a host's first LINK STATE message also sent at service startup time. This first message is always empty, because a new host has no idea yet about its connections to other hosts. Thus, other hosts take no advantage of HELLO messages.

6.6.2 Neighborhood Auditing

Each host makes an assumption about how many hosts are expected to send LINK STATE messages on each network. As soon as all expected messages have arrived, a checkpoint is performed and the checkpoint interval is started over again. This reduces the amount of unused LINK STATE messages since the checkpoint interval is reduced to the minimal time needed to receive all messages. Only if a message is lost or a host didn't send a message due to a failure, a host waits for the full checkpoint interval until a checkpoint is performed.

Since a host has no idea about its neighborhood at startup time, the first checkpoint interval is not shortened to be sure having received each host's LINK STATE message. Storing each sender's *host ID* enables a host to wait for their messages. If all hosts have sent their messages, the checkpoint is performed regardless whether the checkpoint interval is over. Usually this reduces the checkpoint interval to almost the same length than the broadcast interval. But if a message is delayed, the checkpoint interval is extended until the last message has arrived. In the worst case, a message doesn't arrive at all, and the host waits the full checkpoint interval before a checkpoint is performed. However, this should not happen too often.

The following figure illustrates how the length of a checkpoint interval changes dependent on the receipt of the last expected LINK STATE message:

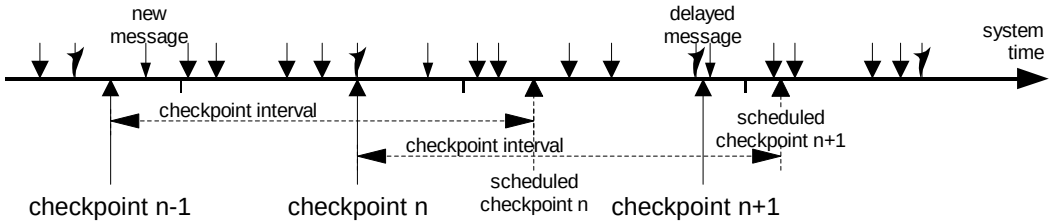

Figure 6.10: If the last expected LINK STATE message has arrived, a checkpoint is performed although the scheduled checkpoint would have been later.

Figure 6.10 also shows the case of a new host being added to the system. On receipt of a new host's LINK STATE message (or HELLO message) its *host ID* is just added to the list of IDs the receiving host waits for in the next iteration. The checkpoint interval is not affected by this, in contrast of a delayed message or even a failed link, which extends the checkpoint interval. Also, if a message from a certain host was not received during a checkpoint interval, its *host ID* is not taken into account during the next interval.

The advantages of the neighborhood auditing approach are the following:

- Class Two hosts get their results sooner, which makes them more actual. However, if a router looses one or both of its interfaces, and thus doesn't send any LINK STATE messages any longer, all other hosts still have to wait the full checkpoint interval until they recognize the router failure.
- The amount of unused messages is decreased. Again, an unused message is undesired since the information contained in them is never taken into account for the router selection but caused both additional network traffic and an overhead by processing the arriving message. Normally, since the checkpoint interval is longer than the broadcast interval, it is very likely to receive more than one message from a number of hosts. This probability is significantly decreased since the checkpoint interval is shorter than normal, and thus subsequent messages from a single host are very likely to arrive in subsequent checkpoint intervals, too.

7 Flooding Protocol

Here, flooding is used to determine each host's link state. This flooding protocol works on a simple principle: each host frequently broadcasts a (flooding) message on its local interfaces. All hosts connected to the same physical network which receive this message forward it on all other of their interfaces (each of them usually connected to a different physical network). Looking at the underlying network structure, a message sent by a host on one physical network (say Net 0) is relayed by all Class One hosts to the other physical network (Net 1).

In what way does a flooding protocol like this helps the hosts to determine the global link state? It is easy to understand that the bare fact of a host H_1 receiving a flooding message from a host H_2 implies that H_1 and H_2 have a running connection between themselves. If the message was forwarded by a Class One host H_3 and the information about the path between H_1 and H_2 over H_3 can be determined (assuming the information about the intermediate host H_3 is available), then H_3 can be considered as a candidate router by the receiving host H_1.

This is exactly what is done here. Initially, H_2 has broadcast a flooding message on a network, say Net 0. Each host on this network receives this message and if it has two running links, the message is forwarded after the forwarding hosts have filled the field for the forwarder's *host ID* in the message. This is what H_3 has done on receipt of the flooding message from H_2 before it has broadcast the message on Net 1. This fields serves other hosts as an indication that this message has already been forwarded and thus should not be forwarded again. On arrival of a forwarded message at H_1, the receiver knows that the forwarder is a router candidate, and that the message creator has at least one interface connected to the other network.

This flooding protocol differs from the Connection List Protocol in the way the information is gathered, but the rest is quite identical. There are also broadcast and checkpoint intervals. The discussion of the flooding protocol is similar to the discussion of the Connection List Protocol above.

7.1 Assumptions

The same assumptions are made for this Flooding Protocol like for the Link State Protocol (see Section 6.1).

7.2 Flooding Message Format

A flooding message contains only three protocol specific fields:

- The *host ID* of the message creator.
- The *host ID* of the forwarding host. If the message is not forwarded, the value of this field is zero.
- A *message sequence number* to uniquely identify the message sent by the message creator.

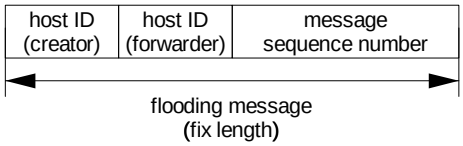

Figure 7.1: A flooding message has a fix length and consists of only three fields.

7.3 The Protocol

A host periodically broadcasts messages on each interface (e.g. whenever a predefined period of time is over). Each message is unique which is assured by assigning a different *message sequence number* to each message.

7.3.1 Message Receipt

1. Discard broadcast echo.

   Broadcasting a message always creates an echo on the sending interface. Now it is assured that only messages sent from other hosts are processed.

2. Discard old messages.

   This is done by comparing the *message sequence number* in the received message with the one being stored for the sending host. If the incoming message has a number less than the stored one, the messages has already been processed by the receiving host and is discarded. The test for equality of both numbers must not be made since there is a multitude of forwarded messages with all of them having the same *message sequence number* but different *host ID*s of the forwarding hosts. Now it is assured that only new and forwarded messages are processed.

3. Check if it is a forwarded message.

   This is done by checking the field for the *host ID* of the forwarder. If it is not zero, then the message is a forwarded one and the forwarding host can be marked as a candidate router. This is done by setting both flags of the forwarding host in the receiver's new view of the global link state. Additionally the flag for the message creator's interface on the other network can be set since it has broadcast the message on that network. Then discard the message.

4. Check if the message has not yet been forwarded.

   If the field for the forwarding *host ID* is zero, the message was received directly from the broadcasting host. The receiving host then knows that the sender's interface is up and sets the corresponding flag in its new view of the global link state.

5. Forward message.

   Before the message is broadcast on the other network, the receiving host has to fill in the field for the *host ID* with its own one to mark this message as a forwarded one.

7.3.2 Checkpoints

At checkpoint time no additional computation is necessary to get the new view of the global link state. Since the flags are set immediately whenever a message arrives, and since during one checkpoint interval each host has broadcast a flooding message which was forwarded by all other hosts, the view is completely build after only one iteration and can be searched by Class Two hosts without any further processing.

After a router is found, all flags are deleted. This ensures that a router candidate doesn't remains one forever even if one of its interfaces fails.

7.3.3 Broadcasts

After each broadcast interval the host just sends its flooding messages on each of its networks. The field for the forwarding host is always zero. This is obvious since the sender is the message creator and the message is not yet forwarded.

7.4 Protocol Analysis

For each message broadcast on one of the subnets, n_1 messages are generated on the other subnet, where n_1 is the number of forwarding Class One hosts. With n_2^0 and n_2^1 being the number of Class Two hosts on network 0 and 1, respectively, the total amount of messages generated, if each host sends a message on each interface, computes to:

$$n_2^0 \cdot n_1 + n_2^1 \cdot n_1 + 2 \cdot n_1 \cdot (n_1 - 1) = n_1 \cdot [n_2^0 + n_2^1 + 2 \cdot (n_1 - 1)] =$$
$$n_1 \cdot (n_2^0 + n_2 1 + 2n_1 - 2) = n_1 \cdot (n_2^0 + n_2^1 + n_1 + n_1 - 2) =$$
$$n_1 \cdot (N + n_1 - 2)$$

It can be assumed that the number of Class Two hosts is very small compared to the number of Class One hosts. Thus, the system can be considered to contain almost only Class One hosts. Because the difference between n_1 an N is very small, n_1 can be assumed to be equal to N. By substituting n_1 with N in the above formula, a new approximation for the maximum number of messages is given:

$$N \cdot (N + N - 2) = N^2 + N^2 - 2 \cdot N = 2 \cdot N^2 - 2 \cdot N$$

$$\Rightarrow O(N^2)$$

This large amount of messages is the main reason why this protocol is not implemented. Although there are several advantages like having less overhead at checkpoint time, the network traffic grows quadratic with the number of participating host. This is significantly more than with the Link State Protocol, where the network traffic is of O(N), where N is the total number of participating hosts. This is easy to understand since each host sends at most two messages which are never forwarded by any other host. With N hosts each sending at most 2 messages, the total amount of LINK STATE messages is never greater than 2·N. Even if the Flooding Protocol's broadcast intervals is twice as long as the one used in the Link State Protocol, the network traffic still remains significantly more, but the time necessary to detect a failure is exactly the same.

8 Router Selection

The last thing remaining to do at checkpoint time is to finally select a router. After a host has built its view of the global link state, it knows all Class One hosts which are all router candidates. However, since relaying datagrams from one network to the other can cause a significant overhead on this machine, depending on the amount of traffic between both networks, not all hosts may be wanted to do this job. This is because of time critical applications running on them or because of limited resources like free CPU time. Before selecting such a host, all other hosts should be selected first.

Different policies in selecting a router may be implemented. Of course, a host's priority must be taken into account, but additional factors could be considered by the router selection algorithm. In the sections below, different possibilities are discussed to fulfill either one or both needs: the guarantee that hosts with the highest priority are taken with the highest probability, and the wish for a well-balanced distribution of traffic between all or some router candidates.

The following discussion refers to a system with 10 hosts. In the table below the global link state is shown as well as each host's *host ID* and priority.

	A	B	C	D	E	F	G	H	I	J
∑	* / *	* / *	* / *	* / *	* / *	* / *	* / *	* / *	* / *	* / *
ID	1	2	3	4	5	6	7	8	9	10
priority	10,000	0	90,000	90,000	1	10,000	50,000	50,000	50,000	50,000

Only Class One hosts are considered here since Class Two hosts can never be selected as a router. Each host's priority is given here as an unsigned integer value. '0' means, that the host is not designed to become a router, but can be selected if there is no other host available. In fact, this is the only case in which a host with zero priority is allowed to be selected.

8.1 Single Router System

As mentioned in Chapter 2, there are several possibilities for the routing granularity. If only one host is desired, all hosts should select the same router which has to cope with all the traffic between both subnets. The easiest way to implement this is to compare each router candidate's priority and identify the subset with the highest priority. If the subset contains only one host, this one becomes the router. If a system consists of multiple hosts with all having the same priority, another property must be used to enable each host to identify the same router. The easiest way to do this is using the *host ID*. Since this is a unique identifier, no two hosts have the same combination of routing priority and *host ID*. Amongst all hosts with the same routing priority, always the one with the highest ID is selected to become the router. It is very likely then, that each host selects the same router, at least for most of the time.

In the example system with the 10 hosts shown in the table above, host D is selected as the router. Amongst all routers with the highest priority (90,000) D's *host ID* is greater than C's. If one of the other Class One hosts (which all have a greater *host ID*) was changing their priority to 90,000, too, this one would automatically be selected as the new router. Similarly, if D decreases its priority to a value less than 90,000, C automatically becomes the new router.

8.2 Multiple Router System

For a solution with multiple routers, each hosts selects exactly one from a subset of all router candidates. Which router candidates are included in this subset depends on the strategy:

- All hosts are taken into account. This means, that the subset contains each router candidate with a priority greater than 0. If there is none, all candidates with a priority of 0 are contained in the subset.

- All hosts with the same highest priority are taken into account.
- Either all hosts with the same priority are taken into account, or at least n hosts with the greatest priorities.

8.2.1 Selection Algorithm

To select a router from a subset of router candidates with a priority >0, the sum of all priorities from all hosts are considered to be 100%. A random number between 0 and 1 then shows which host is actually selected as the router. The following figure illustrates this.

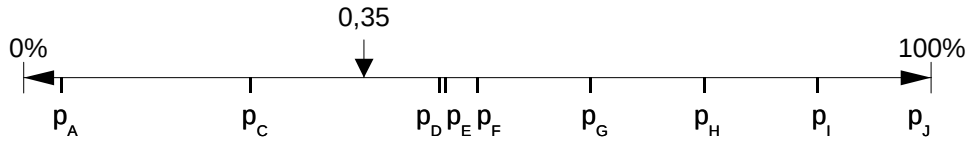

Figure 8.1: Each hosts priority p_x is appended to the one of its predecessor. All priorities together are considered to be 100%. Then, a random variable is assigned a number between 0 and 1 and is used to select the router.

Which hosts should participate in the router selection is the subject of the following sections.

8.2.2 Selection Among All Candidates

If the subset consists of all router candidates, it is possible, but also unlikely, that even hosts with a significant smaller priority can be selected as a router. I. e., even E has a chance to become a router, although its priority of 1 is very small compared to the priorities of others.

8.2.3 Selection In The Same Priority Class

Only hosts in the same priority class are taken into account. If there is only one host remaining with the highest priority, this host remains the only router in the system. Only if this last one fails, the next priority class is searched for a router. Assuming C has failed. Then, D is the only host remaining in its priority class and is selected by all Class Two hosts as the router. If D fails, too, each host selects one from the next priority class containing hosts G, H, I and J.

8.2.4 Priority Class, Extended To A Minimal Number Of Hosts

This is almost the same algorithm like above. However, there is a minimal number of hosts desired to be contained in the subset of router candidates. Assuming the desired number of routers is 5, then not only hosts C and D are contained in the subset, but also hosts J, I and H. G is not contained because its host ID is less than the ones of the other hosts in its priority class. The probability for the hosts C and D being selected as the router is still higher than for the others in the next smaller priority class. However, if the difference between their priority values is too small and the number of hosts with higher priorities is significant smaller than the number of hosts with lower priorities, the probability of the host with the highest priority being selected decreases. To avoid this, one could think of multiplying a host's priority with a value of 2 before adding it to the total value. Either all priorities could be multiplied, or only those of the highest priority/priorities.

9 Implementation Specifics And Restrictions

Now, as it comes to the implementation level, the development environment along with their specific characteristics and restrictions is introduced here. But since not all restrictions applied for the purpose of implementing the Fault Tolerant Routing Service application, only those are discussed in more detail which had a major impact. Other less restrictive characteristics detected during implementation are just mentioned to complete the list.

9.1 Development Environment

The host's hardware platform was a PowerPC 604 based system with two 10 Mbit Ethernet interfaces, further on referred to as a *box*. A box didn't have anything but a BIOS which enabled booting an image via one of its Ethernet interfaces using the standard BOOTP protocol. The image contained both, the operating system and all applications running on top of it. After the image was loaded using TFTP, the BIOS gave control to the operating system which in turn started all applications, each one in a separate thread. The operating system running on each box was the IBM proprietary realtime operating system OS Open and one of the applications was the Fault Tolerant Routing Service.

The image, and especially the Fault Tolerant Routing Service application was built on a seperate machine. After having compiled the application files, the object files are linked together with all OS Open libraries to a single object file. Then, another tool built the image using this object file as its input. This machine was running AIX, an IBM Unix derivate, and an IBM C/C++ Compiler.

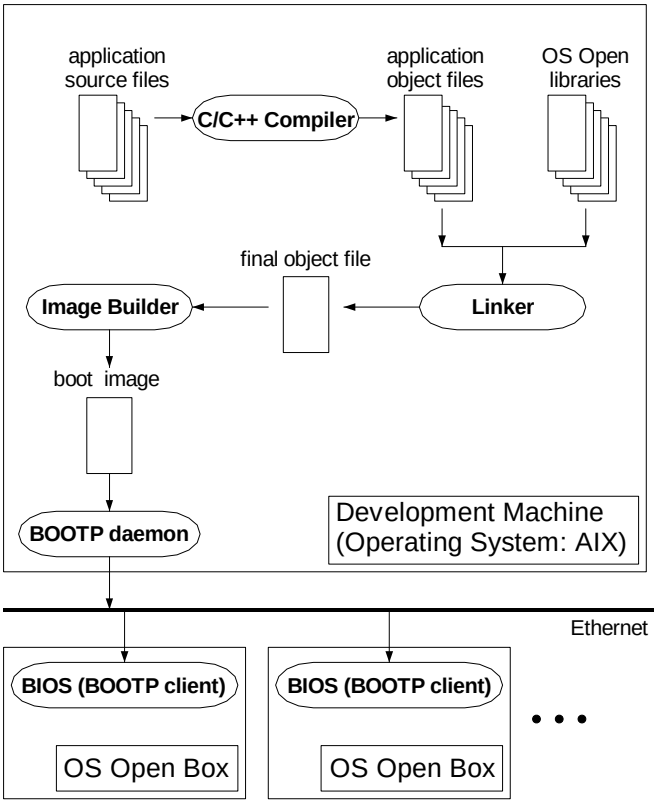

Figure 9.1: The development machine is used to create the boot image which is loaded by each OS Open box using one of their Ethernet interfaces.

9.2 OS Open

Although OS Open is hardly a widely spread operating system, a detailed description is not really necessary since it is a realtime operating system based on 4.3BSD. Thus, almost all functionality is available as one might be used by working with other Unix Systems. See [9] and [10] for a more detailed description.

9.3 C/C++

The programming language used for the implementation of the Fault Tolerant Routing Service was C++. Along with this language some restrictions apply which have to be handled.

9.3.1 Message Sequence Number Overflow

The message sequence number is used to uniquely identify messages broadcast by a host. It is incremented each time a message is sent. Having this number enables receiving hosts to decide whether the message is a new one, or whether the same message has already been received. However, since the implementation of the messages sequence number reduces the value to 2^{32}, the range is limited. The more frequent a message is broadcast, the sooner an overflow will occur.

Depending on the broadcast interval an overflow is more or less likely to appear. The table below shows this:

broadcast interval	overflow after approx.
1 millisecond	49 days
10 milliseconds	1 year and 4 months
100 milliseconds	13 years and 7 months
1 second	136 years
10 seconds	1362 years

Avoiding an overflow by setting the broadcast interval to an appropriate value is not a good idea since nobody knows how long the system will run without a shutdown. Also it may turn out that broadcast intervals have to be changed during the system's runtime. And since a failure of a host may happen at any time, other hosts must recognize another host's re-start of the message sequence number, anyway. This is almost the same as an overflow.

The implementation of the Fault Tolerant Routing Service deals with the problem of a failed host by deleting the entry of a host in its table containing the most recent message sequence number if it has not received a message within the last n checkpoint intervals. This is quite suitable since a failed host must be repaired, which is usually not be done within a broadcast interval. To cope with normal overflows, the receiving host compares the new message sequence number with the one it has stored for the sending host. If the stored one is 'very close' at the end of the possible range and the new one is 'very close' to zero, an overflow is assumed and the message is considered to be new. The term 'very close' means that there is no need for the number switch from the maximum value to zero, but might switch from max(2^{32})-n to m. I.e. that some messages might be missing, which can always happen since the protocol used is a connectionless one, namely UDP.

9.3.2 Memory Allocation Failures

Since the amount of incoming messages and their respective length is never known in advance, the amount of memory necessary to store them can be more than available. A failure in allocating memory, either by `malloc()` or `new`, causes the Fault Tolerant Routing Service to immediately stop operation.

Another possibility would be to cope with this failure by not taking a message into account. This behavior absolutely makes sense in real systems, it may even be a necessity. But for the purpose here it

is enough to show the principle of the protocol, and not to cover all possible kinds of failures which are not related to the protocol itself.

9.4 TCP/IP

Although there are a lot of standards dealing with TCP/IP, it is still a protocol suite with a wide variety of different implementations. This makes it difficult to talk about 'the' TCP/IP protocol in general. However a few assumptions concerning the configuration of each host can be made which hold for all kinds of implementations, especially for the one coming along with OS Open.

9.4.1 Assumptions

- Each interface is configured with only one IP address. Or at least, each interface is configured only with IP addresses of one subnet. This is not really a restriction, because a physical interface configured with IP addresses of more than one subnet can be seen as two logical interfaces, each of them belonging to one subnet only.

  This is assumed to ease the processing of messages because the logical network structure matches the physical network structure. Having an interface belonging to multiple subnets, a host needs to send one message per subnet on the same interface to enable other hosts on each subnet to determine the sender's link state.
- On receipt of a message:
  - The sender's IP address can be determined. This is assured by using the `recvfrom()` function call which delivers the sender's IP address on receipt of a UDP message.
  - There is no depiction of a host's IP address to its *host ID* (and vice versa).

9.4.2 Maximum Length of Connection List

Generally the protocol is not restricted to a certain number of participating hosts. However, using pure UDP as the underlying communication protocol restricts the length of the connection list in a LINK STATE message and thus the maximum number of hosts. Since the length of a UDP message is limited by the maximum length of an IP datagram, which is 2^{16} = 65,535 bytes, the remaining space for the connection list is 65,497 bytes. Assuming a length of the host ID of 16 bit, the maximum number of participating hosts is 32,748. A detailed calculation is shown below:

# bytes	description
65,535	max. length of an IP datagram
20	length of the IP header length
8	UDP header length
65,507	⇒ max. length of UDP datagram
10	LINK STATE message header length
65,497	⇒ max. length of connection list
2	length of *host ID*
32,748	⇒ max. # of entries in connection list

But: many TCP/IP implementations restrict the length of a UDP datagram to less than 65,535 bytes! See [3], [4], [15], [19], [20] and [21] for a more detailed description.

9.4.3 Protocol Address Determination

In the discussions above, the hosts have been always addressed using their *host ID*s. This was all right as long as the Fault Tolerant Routing Service was discussed in a general way. But now, as it comes to

the implementation specific level, addressing of messages must be provided by using the network protocol addresses, namely their IP addresses. Although LINK STATE messages are always broadcast, and the Fault Tolerant Routing Service is not designed to exchange other messages, IP addresses are still needed to be able to update the routing table. Thus, as soon as a host wants to add or update an entry in its routing table, the router's IP address must be known.

In general, there are two ways to get the necessary information: to receive a UDP message, the function call `recvfrom()` or `recvmsg()` should be used. These calls not only receive a UDP message, but also return, among other things, the IP address of the message sender. Another possibility would be to implement some kind of a higher level address resolution protocol, which maps host IDs to the corresponding IP addresses on each network. This is very similar to the already known ARP for resolving IP addresses to their corresponding hardware addresses, implemented by almost every TCP/IP implementation.

The Fault Tolerant Routing Service maintains a table, which maps a host's ID to its corresponding IP address on each network. Of course, there must be either two tables, one for each network, or an entry for a host must support two IP addresses.

`recvfrom()` Function Call

Since `recvfrom()` is mostly implemented by calling `recvmsg()`, and `recvmsg()` doesn't return any other additional data, which is of any interest for the purpose of getting the sender's IP address, only the `recvfrom()` call is mentioned here (see [19]).

The implementation of the Fault Tolerant Routing Service uses the `recvfrom()` function call to wait for and receive LINK STATE messages from a network. If a messages arrives, the call returns and delivers the message along with the sender's IP address. Having both, the receiving host can read the sender's *host ID* from the message header and add or update the entry for the sender in its table. If this happens each time a message arrives, the table contains always the most actual IP addresses, even if a host's interface should be reconfigured while its system is running.

As soon as the router selection algorithm has selected a host to become a router, the Fault Tolerant Routing Service application then searches its table to find the corresponding IP address. For a Class Two host, which are the only hosts needing to run the router selection algorithm, this address must always be found in the table. It is easy to understand why this must be the case. Assuming that the Class Two host A selects the Class One host D as a router between both networks means, that A has identified D as a host with two running links. Since A is only connected to one network (Net 0), it never gets information about other hosts' connections on the same network, but it only receives messages with information about connections on the other network (Net 1). Class Two hosts on the same network don't have any connection, and thus their messages are empty. However, Class One hosts do have connections on Net 1, and A learns about them and fills its connection table. This is how A learned about D's connection on Net 1, but the connection on Net 0 must be detected by receiving a message from D itself. Since this message is unlikely to be empty, but contains at least one connection, this one message received from D is sufficient to enable A to identify D as a Class One host.

Ignoring the fact, that Class One hosts have no reason to find a router candidate, they do not necessarily have to have an entry. A Class One host may have learned about D's connection by received messages from other Class One hosts, without ever having received a message from D itself. This is unlikely, but if the network consists of several hubs, and the connection between two hubs is broken on each network, D may be unreachable for A, but other Class One hosts may still be able to communicate with D. Again, a Class One host normally never runs the router selection algorithm. But even if it was, it would be very unlikely that it has never reached a message from the selected router.

Higher Level Address Resolution

The other approach for mapping *host ID*s to IP addresses is similar to the ARP implementation of a TCP/IP stack. As soon as a Class One host wants to update its routing table, it broadcasts an 'Fault Tolerant Routing Service ARP Request' on the network. This request gives the *host ID* and asks for the IP address. Each host on the network receives this request, and the one with the questioned *host ID* answers this request by sending a reply with the desired IP address. This protocol can be implemented almost exactly like the TCP/IP ARP implementation, resolving IP addresses to hardware addresses.

10 Fault Tolerant Routing Service Application

This chapter gives a description of the Fault Tolerant Routing Service application.

10.1 Application Structure

The Fault Tolerant Routing Service is implemented as an application on top of OS Open as the operating system running on the hardware controllers. It consists of four threads. The broadcast thread periodically broadcasts LINK STATE messages on each network the host has a running interface to. The connection lists of the messages are created by the checkpoint thread at each checkpoint. This thread periodically processes all messages received by both of the receive threads and builds both, the new connection lists for the broadcast thread and the new view of the global link state. If necessary, the routing table is updated by the checkpoint thread, too. The receive threads both wait for messages, one on Net 0, and the other on Net 1. Each incoming message is stored in memory, if it is considered to be valid message. Also, additional information about the sender is updated, like the *message sequence number*, the router priority and, last but not least, the IP address.

The following figure shows the application structure and the data items accessed by each thread.

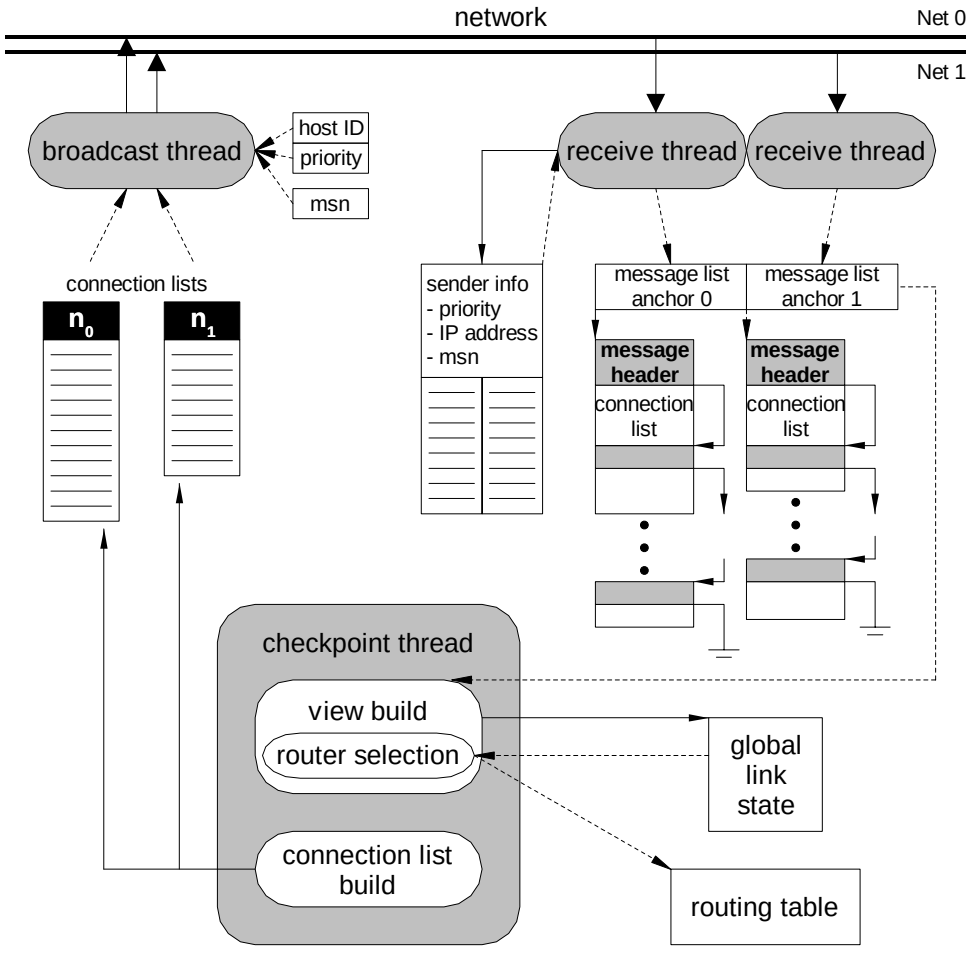

Figure 10.1: The Fault Tolerant Routing Service consists of three main threads. The broadcast thread just frequently broadcasts LINK STATE messages on each network. The receive thread collects all incoming messages which are processed by the checkpoint thread. At checkpoint time this task builds a new view of the global link state and new connections lists for the broadcast thread. On Class Two hosts, the checkpoint thread runs the router selection algorithm to find router candidates and updates the routing table, if necessary.

The concurrent access of the same data structures is serialized using mutex semaphores which are created at application startup. In the sections below the startup and each thread is discussed in more detail and their respective actions are explained.

10.2 Application Startup

At application startup time, the Fault Tolerant Routing Service initializes all variables, creates all necessary objects and mutex semaphores, and examines the system capabilities. The most interesting part is the examination of available interfaces. Each interface is represented by a separate object. Whether or not this interface exists and/or is active can be determined using `ioctl()` function calls (see [9], [10] and [19]).

Then, on hosts with two interfaces, both receive threads are created. If a host has only one interface, one receive thread is enough, since no additional resources like CPU time and memory are unnecessarily used then. Creating threads is done using the `pthread_create()` function call (see [9] and [10]). Since a separate object is created each time a LINK STATE message is received, a receiving thread must be created with a reasonable stacksize to avoid memory allocation failures. After their creation they immediately start working.

The broadcast thread is created next. No special preparation is necessary since the application initialization has already allocated all necessary message objects to be sent. The same objects are accessed by the checkpoint thread to update the connection list.

After all the above mentioned threads have been started, the main thread calls the checkpoint thread function and never returns. This is because the Fault Tolerant Routing Service is considered to be a service application running all the time until system shutdown or a major host failure.

10.3 Receive Thread

Each receive thread is responsible for one of both interfaces. Since the tasks are identical for each interface, both receive threads are treated equal.

The actions taken by each receiving thread is given in some kind of pseudo code:

```
loop forever
{
   if interface is configured and up
   {
      wait for message;                    // blocking, i.e. only if a message
                                           // arrives, the thread continues
      if message sender is not me and the message sequence number is valid
      {
         acquire mutex semaphore;
         update message sequence number for sending host in table;
         set router priority for sending host in table;
         set IP address for sending host in table;
         store message in memory;
         release mutex semaphore;
      }  // if
   }
   else
   {
      wait for an appropriate period of time and try again;
   }  // if
}  // loop
```

Acquiring and releasing the mutex semaphore is necessary for synchronization with the checkpoint thread which also accesses the stored messages at each checkpoint.

10.4 Broadcast Thread

There is not much to do for the broadcast thread but to periodically broadcast its LINK STATE message on each running interface.

The pseudo code describing the actions taken by the broadcast thread:

```
loop forever
{
   loop for both interfaces
   {
      if interface is configured and up
      {
         acquire mutex semaphore;
         broadcast already prepared message;
         release mutex semaphore;
      }  // if
   }  // loop

   sleep until broadcast interval is over;

}  // loop
```

Again, the mutex semaphore, one for each message, is needed to synchronize with the checkpoint thread which prepares both messages at each checkpoint.

10.5 Checkpoint Thread

Most of the work is done here.

The pseudo code:

```
loop forever
{
   wait checkpoint interval;

   acquire both mutex semaphores for stored messages;
   remove all messages from both lists;
   release both mutex semaphores;

   if this is a host of Class Two
   {
      build view of global link state;
      select a router from all router candidates;
      update routing table;
   }  // if

   acquire mutex semaphores for broadcasting messages;
   create new connection lists;
   release mutex semaphores for broadcasting messages;

}  // loop
```

11 Summary

The purpose of this thesis was to deal with the problem of assuring communication between hosts in a redundant network structure in case of interface failures. Since TCP/IP was chosen as the communication protocol within the system, a solution was desired which takes as much advantage of already existing mechanisms provided by this protocol suite. This lead to an detailed investigation of the TCP/IP stack's functionality and all possible methods not only to make a host relaying datagrams, but also to enable other hosts using the router. After several possibilities have been identified, they were checked if a suitable solution could be implemented. Most of the possible solutions would have led to an election mechanism among all router candidates. But since this would have caused a lot more complexity to cope with than a selection algorithm, the choice was not to use one of the special features of TCP/IP. An ordinary routing table update after the router was selected by a Class Two host, this is how the Fault Tolerant Routing Service was finally implemented.

To enable a host selecting a router needs him to be able to identify router candidates. A router candidate is a host which has two running interfaces. This must be detected by a Class Two host. For this purpose, two protocols have been developed, a Link State Protocol and a Flooding Protocol. Both protocols require that each host periodically sends a message on each network. In the flooding protocol, a message is forwarded by each host which causes a quadratic message overhead proportional to the amount of forwarding hosts. This is avoided by the link state protocol. A LINK STATE message contains a host's connections on the respective other network and receiving Class Two hosts then know about 'the other side' of the network. However, a host needs one more iteration to recognize all important failures, but the message overhead is linear with the number or hosts. This is why the link state protocol is chosen to be implemented.

The Fault Tolerant Routing Service was implemented in C++ using OS Open as the operating system. This was determined by the project's hardware and software environment.

Appendix A: Frame, Packet and Message Formats

A.1 Ethernet Frame

Every single data being sent over an Ethernet is encapsulated in an Ethernet frame which has always the same format, no matter what upper layer protocol is used. Figure A.1 shows the format of an Ethernet frame.

Figure A.1 An Ethernet frame includes both the destination and the source address, a type field and the data followed by a checksum.

The first two fields contain the target's *destination* and the sender's *source address*. An Ethernet address is 6 bytes long and is worldwide unique. I.e., no two Ethernet interface cards can ever have the same hardware address.

The 2-byte *type* field identifies the frame's payload, i.e. the type of the packet in the data portion. According to the value the packets are sent to the appropriate input queue. Amongst other values, the following two are of interest in this context:

- 0x0800 Internet Protocol Version 4 (IPv4)
- 0x0806 Address Resolution Protocol (ARP)

After the type field, the variable length *data* portion begins with a minimum length of 46 bytes and a maximum length of 1.500 bytes. If a packet is smaller than the required 46 bytes, additional padding bytes must be appended. If a packet is larger, it must be fragmented to fit into the frame.

The 4-byte Ethernet checksum (*CRC*) at the end of each frame is usually generated and verified by the hardware and thus not considered harmful.

A.2 IP Datagram

IP datagrams are encapsulated in an Ethernet frame with the frame type field set to 0x0800. This ensures that the Ethernet device driver passes the frame's data portion to the IP input queue.

Each datagram has an IP header with various fields preceding the datagram's data portion. If any options are set, they are included between the header section and the data portion. Figure A.2 shows the format of an IP datagram.

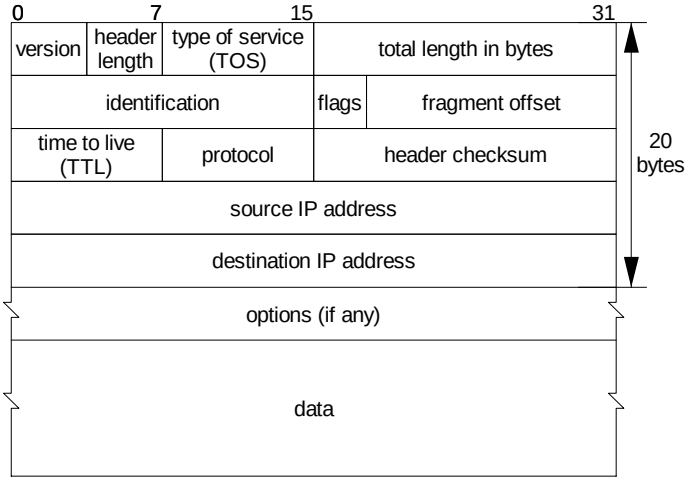

Figure A.2: An IP datagram consists of an IP header, some possible options and the data.

Each IP packet has the 4-bit *version* field at the beginning of the IP header set to the value of 4, which describes IPv4. Future implementations will have other values to distinguish between different IP versions such as IPv6.

The IP header has a minimum length of 20 bytes but can exceed to up to 60 bytes, if any options are appended. The limit of only 40 bytes for all options per packet results from the length of the *header length* field. The 4-bit value counts the number of 4-byte words used for both, the header and the options.

In the *type of service* (*TOS*) field the sender can specify certain qualities which are desired for the delivery of IP datagrams.

Since the *total length* is a 16-bit field, the maximum datagram size (including the header and all options) is limited to 65.535 bytes. With an Ethernet MTU size of 1.500 bytes, IP datagrams do have to be fragmented anyway, if they become larger.

Each packet sent by a host is uniquely identified by the *identification* field. This is useful in case of fragmentation which is controlled by the 3-bit *flags* field and the 12-bit *fragmentation offset* field.

An upper limit for intermediate routers can be set by the *time to live* (*TTL*) field. Each time a router forwards an IP packet, this value is decreased by 1.

IP uses the *protocol* field to demultiplex all incoming packets stored in the data section to the right transport layer protocol queue. The values are 6 for TCP, 17 for UDP and 1 for ICMP .

In the *header checksum* field the checksum over the IP header and all options is stored. Before the computation, the value of this field is set to 0.

The *source address* field and the *destination address* field contain the sending host's and the final destination host's IP address, respectively.

The length of the *option* section can be determined by subtracting the minimum IP header length in 4-byte words (5) from the value of the header length field (5 - 15).

After the options the *data* section begins.

A.3 ARP Packet

An ARP Packet has a fix length of 28 bytes. Figure A.3 shows the single fields and their lengths in bytes.

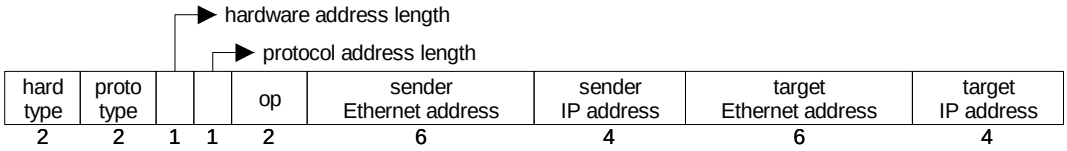

Figure A.3: The format of an ARP packet is always the same, no matter whether it contains a Request or a Reply.

In the *hard type* field the type of hardware is specified. For Ethernet the value is 1. Similarly, the protocol type is specified in the *prot type* field, which is set to 0x0800 for IP.

The next two fields specify the *hardware address length* and the *protocol address length*. Ethernet hardware addresses have a length of 6 bytes, and the length of IP addresses is 4 bytes.

In the *op* field the operation is specified. Possible values are 1 for an ARP Request or 2 indicating an ARP Reply. There are two other possible values for an RARP Request (3) and RARP Reply (4).

The *sender Ethernet address* and the *sender IP address* specify both the sender's hardware and protocol address, respectively. These are always used on the receiving host to update the local ARP cache.

The value of the *target Ethernet address* is not set in ARP Requests, because this is exactly the desired information the sender tries to obtain by giving the *target IP address*.

A.4 ICMP Redirect Message

Figure A.4 shows the format of an ICMP Redirect Message:

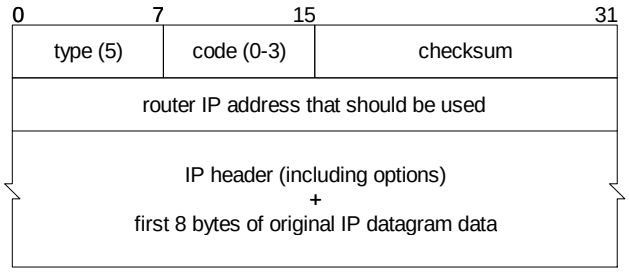

Figure A.4: ICMP Redirect Messages have a 4 byte header followed by the new router's IP address and the beginning of the IP datagram which was sent to the wrong router.

The overall length of an ICMP Redirect Message can be calculated by adding 8 (the 4-byte header plus the length of the IP address field), 20 (the length of an IP header), 0-40 (the length of the options portion) and 8 (the first 8 bytes of the original IP datagram). Thus, an ICMP Redirect Message has a minimum length of 36 bytes and a maximum length of 76 bytes.

The *type* field has a length of 1 byte and its value is always 5, indicating an ICMP Redirect Message.

In the *code* field, also 1 byte, the value for the redirect message type is given. There are four different values:

- 0: redirect for network
- 1: redirect for host
- 2: redirect for type-of-service and network
- 3: redirect for type-of-service and host

The *checksum* (2 bytes) is generated over the whole ICMP message. The value is considered to be 0 at calculation time.

The new router's IP address is given in the next 4 bytes followed by the beginning of the IP datagram sent to the wrong router (the one which generated the redirect message).

A.5 ICMP Router Solicitation Message

Figure A.5 shows the format of an ICMP Solicitation Message:

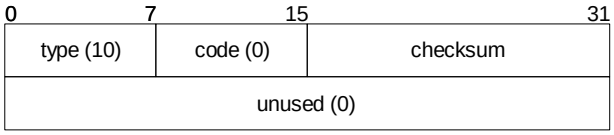

Figure A.5: ICMP Router Solicitations Messages always have a fix length of 8 bytes with the last 4 bytes being unused (sent as 0).

The *type* field has a length of 1 byte and its value is always 10, indicating an ICMP Router Solicitation Message.

The *code* field value is always 0.

The *checksum* (2 bytes) is generated over the whole ICMP message. The value is considered to be 0 at calculation time.

The next 4 bytes are unused and must be 0.

A.6 ICMP Router Advertisement Message

Figure A.6 shows the format of an ICMP Router Advertisement Message:

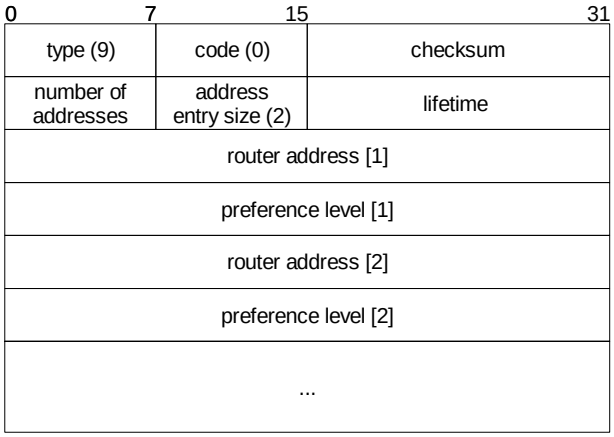

Figure A.6: ICMP Router Advertisement Messages always have a header of a fix length of 8 bytes followed by a list of router addresses with their corresponding preference level.

The *type* field has a length of 1 byte and its value is always 9, indicating an ICMP Router Advertisement Message.

The *code* field value is always 0.

The *checksum* (2 bytes) is generated over the whole ICMP message. The value is considered to be 0 at calculation time.

Number of addresses contains the number of router IP addresses being advertised by a router.

The *address entry size* is the number of 32-bit words for each router address. The value is always 2, because each 32-bit router IP address is followed by a 32-bit preference level.

Lifetime contains the number of seconds that the advertised addresses can be considered valid.

Appendix B: Connection Tables

B.1 General Remarks

Since the new view is build by LINK STATE messages composed by the content of other hosts' old connection tables, incoming messages can be processed by a host on random order. But for the ease of understanding, the impact to the connection tables of each hosts in the system by an incoming message sent by a host on a network is shown below in detail for each host and always in the same order.

B.2 System Startup

At system startup time, no host within the system has an idea about the existence of other hosts. Each host just sends an empty LINK STATE message and waits for incoming messages from other hosts on all networks it has a running connection to. The following discussion shows the development of each host's connection table and the according view of the global link state after having processed all incoming LINK STATE messages received in one checkpoint interval.

The underlying example system is the one described in Section 6.4. It consists of four hosts, two of Class One (C and D) and two of Class Two (A and B).

First Iteration

- A sends on Net 0: no connections to other hosts on Net 1

  ⇒ all on Net 0 know about their connection to A

A	A	B	C	D
A				
B				
C				
D				

B	A	B	C	D
A				
B				
C				
D				

C	A	B	C	D
A			**K** / -	
B				
C	**K** / -			
D				

D	A	B	C	D
A				**K** / -
B				
C				
D	**K** / -			

- A sends nothing on Net 1 since it has no interface which is connected to this network.
- B sends nothing on Net 0 since it has no interface which is connected to this network.
- B sends on Net 1: no connections to other hosts on Net 0

  ⇒ all on Net 1 know about their connection to B

A	A	B	C	D
A				
B				
C				
D				

B	A	B	C	D
A				
B				
C				
D				

C	A	B	C	D
A			K / -	
B			- / **K**	
C	K / -	- / **K**		
D				

D	A	B	C	D
A				K / -
B				- / **K**
C				
D	K / -	- / **K**		

- C sends on Net 0: no connections to other hosts on Net 1

  ⇒ all on Net 0 know about their connection to C

A	A	B	C	D
A			**K** / -	
B				
C	**K** / -			
D				

B	A	B	C	D
A				
B				
C				
D				

C	A	B	C	D
A			K / -	
B			- / K	
C	K / -	- / K		
D				

D	A	B	C	D
A				K / -
B				- / K
C				**K** / -
D	K / -	- / K	**K** / -	

- C sends on Net 1: no connections to other hosts on Net 0

  ⇒ all on Net 1 know about their connection to C

A	A	B	C	D
A			K / -	
B				
C	K / -			
D				

B	A	B	C	D
A				
B			- / **K**	
C		- / **K**		
D				

C	A	B	C	D
A			K / -	
B			- / K	
C	K / -	- / K		
D				

D	A	B	C	D
A				K / -
B				- / K
C				K / **K**
D	K / -	- / K	K / **K**	

- D sends on Net 0: no connections to other hosts on Net 1

  ⇒ all on Net 0 know about their connection to D

A	A	B	C	D
A			K / -	**K** / -
B				
C	K / -			
D	**K** / -			

B	A	B	C	D
A				
B			- / K	
C		- / K		
D				

C	A	B	C	D
A			K / -	
B			- / K	
C	K / -	- / K		**K** / -
D			**K** / -	

D	A	B	C	D
A				K / -
B				- / K
C				K / K
D	K / -	- / K	K / K	

- D sends on Net 1: no connections to other hosts on Net 0

  ⇒ all on Net 1 know about their connection to D

A	A	B	C	D
A			K / -	K / -
B				
C	K / -			
D	K / -			
∑	* / -		* / -	* / -

B	A	B	C	D
A				
B			- / K	- / **K**
C		- / K		
D		- / **K**		
∑		- / *	- / *	- / *

C	A	B	C	D
A			K / -	
B			- / K	
C	K / -	- / K		K / **K**
D			K / **K**	
∑	* / -	- / *	* / *	* / *

D	A	B	C	D
A				K / -
B				- / K
C				K / K
D	K / -	- / K	K / K	
∑	* / -	- / *	* / *	* / *

These last four tables are each hosts' new connection tables with their resulting views of the global link state in the bottom line of each table. The tables are used in the next (second) iteration to build LINK STATE messages. As one can easily see, each host only knows about its own connections to other hosts because of received (empty) LINK STATE messages. Thus, no two hosts have the same connection table. But since hosts C and D are both of Class One and receive exactly the same messages from all other hosts but themselves, their view of the global link state is the same.

At this time it is not possible for Class Two hosts (here: A and B) to expect a valid result from the router selection algorithm. Because only empty LINK STATE messages are received, no other information about the network topology is given than the connections between the message sender and the receiver. Thus, during this first iteration each host acquires information about its own connections to other hosts only, but it learns nothing about other hosts' connections.

Second Iteration

After each host has build its connection table in the first iteration, the quality of the LINK STATE messages sent on the network changes to a more detailed description of the sender's connections to other hosts. A receiving host learns about those connections and gets a more sophisticated view of the global link state.

- A sends on Net 0: no connections to other hosts on Net 1

  ⇒ all on Net 0 know about their connection to A

A	A	B	C	D
A				
B				
C				
D				

B	A	B	C	D
A				
B				
C				
D				

C	A	B	C	D
A			**K** / -	
B				
C	**K** / -			
D				

D	A	B	C	D
A				**K** / -
B				
C				
D	**K** / -			

- B sends on Net 1: no connections to other hosts on Net 0

  ⇒ all on Net 1 know about their connection to B

A	A	B	C	D
A				
B				
C				
D				

B	A	B	C	D
A				
B				
C				
D				

C	A	B	C	D
A			K / -	
B			- / **K**	
C	K / -	- / **K**		
D				

D	A	B	C	D
A				K / -
B				- / **K**
C				
D	K / -	- / **K**		

- C sends on Net 0: connections to hosts B and D on Net 1

  ⇒ all on Net 0 know about their connection to C
  ⇒ all on Net 0 learn about C's connections to B and D on Net 1

A	A	B	C	D
A			**K** / -	
B			- / **L**	
C	**K** / -	- / **L**		- / **L**
D			- / **L**	

B	A	B	C	D
A				
B				
C				
D				

C	A	B	C	D
A			K / -	
B			- / K	
C	K / -	- / K		
D				

D	A	B	C	D
A				K / -
B			- / **L**	- / K
C		- / **L**		**K** / **L**
D	K / -	- / K	**K** / **L**	

  Now, D's connection table shows the suspicous state mentioned above. It knows, that it has a connection to C on Net 0, because it has received a LINK STATE message from C on Net 0. Thus, it can mark this connection with a 'K'. In this message, host C claims to have a connection to D on Net 1, but since D didn't receive a message on that interface (yet), it cannot verify this information, and thus marks this connection with an 'L'.

- C sends on Net 1: connections to hosts A and D on Net 0

  ⇒ all on Net 1 know about their connection to C
  ⇒ all on Net 1 learn about C's connections to A and D on Net 0

A	A	B	C	D
A			K / -	
B			- / L	
C	K / -	- / L		- / L
D			- / L	

B	A	B	C	D
A			**L** / -	
B			- / **K**	
C	**L** / -	- / **K**		**L** / -
D			**L** / -	

C	A	B	C	D
A			K / -	
B			- / K	
C	K / -	- / K		
D				

D	A	B	C	D
A			**L** / -	K / -
B			- / L	- / K
C	**L** / -	- / L		K / **K**
D	K / -	- / K	K / **K**	

  By the receipt of a message from C on Net 1, D can finally verify this connection and replaces the former 'L' with a 'K'.

- D sends on Net 0: connections to hosts B and C on Net 1

  ⇒ all on Net 0 know about their connection to D
  ⇒ all on Net 0 learn about D's connections to B and C on Net 1

A	A	B	C	D
A			K / -	**K** / -
B			- / L	- / **L**
C	K / -	- / L		- / L
D	**K** / -	- / **L**	- / L	

B	A	B	C	D
A			L / -	
B			- / K	
C	L / -	- / K		L / -
D			L / -	

C	A	B	C	D
A			K / -	
B			- / K	- / **L**
C	K / -	- / K		**K** / **L**
D		- / **L**	**K** / **L**	

D	A	B	C	D
A			L / -	K / -
B			- / L	- / K
C	L / -	- / L		K / K
D	K / -	- / K	K / K	

  Again, the suspicous state appears, this time in C's connection table. D sent a message on Net 0 and claims to have a connection to C on Net 1.

- D sends on Net 1: connections to hosts A and C on Net 0

  ⇒ all on Net 1 know about their connection to D
  ⇒ all on Net 1 learn about D's connections to A and C on Net 0

A	A	B	C	D
A			K / -	K / -
B			- / L	- / L
C	K / -	- / L		- / L
D	K / -	- / L	- / L	

B	A	B	C	D
A			L / -	**L** / -
B			- / K	- / **K**
C	L / -	- / K		L / -
D	**L** / -	- / **K**	L / -	

C	A	B	C	D
A			K / -	**L** / -
B			- / K	- / L
C	K / -	- / K		K / **K**
D	**L** / -	- / L	K / **K**	

D	A	B	C	D
A			L / -	K / -
B			- / L	- / K
C	L / -	- / L		K / K
D	K / -	- / K	K / K	

  Like with D before, C changes the quality of its connection to D on Net 1 from 'L' to 'K' by the receipt of a LINK STATE message from D on Net 1.

B.3 Class One Host Startup

Now, another Class One host E is added to the system. The following discussion shows how E's connection table is build and how other hosts begin to know about E in the first iteration and learn about its connections in the second. At E's boot time, the other hosts' connection tables are like the ones shown below. Only E has an empty table.

A	A	B	C	D	E
A			K / -	K / -	
B			- / L	- / L	
C	K / -	- / L		- / L	
D	K / -	- / L	- / L		
E					

B	A	B	C	D	E
A			L / -	L / -	
B			- / K	- / K	
C	L / -	- / K		L / -	
D	L / -	- / K	L / -		
E					

C	A	B	C	D	E
A			K / -	L / -	
B			- / K	- / L	
C	K / -	- / K		K / K	
D	L / -	- / L	K / K		
E					

D	A	B	C	D	E
A			L / -	K / -	
B			- / L	- / K	
C	L / -	- / L		K / K	
D	K / -	- / K	K / K		
E					

E	A	B	C	D	E
A					
B					
C					
D					
E					

First Iteration

As long as no LINK STATE message from E is received by hosts A, B, C and D, the content of their tables don't change compared to the System Startup scenario. Only E's table is filled with new information.

- A sends on Net 0: no connections to other hosts on Net 1

E	A	B	C	D	E
A					**K** / -
B					
C					
D					
E	**K** / -				

- B sends on Net 1: no connections to other hosts on Net 0

E	A	B	C	D	E
A					K / -
B					- / **K**
C					
D					
E	K / -	- / **K**			

- C sends on Net 0: connections to hosts B and D on Net 1

E	A	B	C	D	E
A					K / -
B			- / **L**		- / K
C		- / **L**		- / **L**	**K** / -
D			- / **L**		
E	K / -	- / K	**K** / -		

- C sends on Net 1: connections to hosts A and D on Net 0

E	A	B	C	D	E
A			**L** / -		K / -
B			- / L		- / K
C	**L** / -	- / L		L / L	K / **K**
D			L / L		
E	K / -	- / K	K / **K**		

- D sends on Net 0: connections to hosts B and C on Net 1

E	A	B	C	D	E
A			L / -		K / -
B			- / L	- / **L**	- / K
C	L / -	- / L		L / L	K / K
D		- / **L**	L / L		**K** / -
E	K / -	- / K	K / K	**K** / -	

- D sends on Net 1: connections to hosts A and C on Net 0

E	A	B	C	D	E
A			L / -	**L** / -	K / -
B			- / L	- / L	- / K
C	L / -	- / L		L / L	K / K
D	**L** / -	- / L	L / L		K / **K**
E	K / -	- / K	K / K	K / **K**	
Σ	* / -	- / *	* / *	* / *	* / *

Now, E's table is complete and won't change during this first iteration (all hosts in the system have sent their messages but E). The last row shows E's view of the global link state. The tables of A, B, C and D look like shown at the beginning of this chapter.

- E sends on both, Net 0 and Net 1: no connections to other hosts on either network, Net 1 and Net 0, respectively

  ⇒ all hosts on both networks know about their connection to E

A	A	B	C	D	E
A			K / -	K / -	**K** / -
B			- / L	- / L	
C	K / -	- / L		- / L	
D	K / -	- / L	- / L		
E	**K** / -				
∑	* / -	- / *	* / *	* / *	* / -

B	A	B	C	D	E
A			L / -	L / -	
B			- / K	- / L	- / **K**
C	L / -	- / K		L / -	
D	L / -	- / L	L / -		
E		- / **K**			
∑	* / -	- / *	* / *	* / *	- / *

C	A	B	C	D	E
A			K / -	L / -	
B			- / K	- / L	
C	K / -	- / K		K / L	**K** / **K**
D	L / -	- / L	K / L		
E			**K** / **K**		
∑	* / -	- / *	* / *	* / *	* / *

D	A	B	C	D	E
A			L / -	K / -	
B			- / L	- / K	
C	L / -	- / L		K / K	
D	K / -	- / K	K / K		**K** / **K**
E				**K** / **K**	
∑	* / -	- / *	* / *	* / *	* / *

Hosts A, B, C and D now know about E's existence, but have no idea about its connections to other hosts. However, this information is needed by Class Two hosts (A and B) to be able to identify E as a Class One host, and thus as a router candidate.

Second Iteration

E's table won't change, because the content of incoming LINK STATE messages doesn't change with respect of the sending hosts A, B, C and D. Only their respective connections to E is a new content, but this is a redundant information with respect to E, because by receiving a message, E automatically knows about these connections. Thus, only the impact of E's LINK STATE messages on the tables of hosts A, B C and D is shown below.

Below are shown each hosts' connection table after all hosts but E have sent their LINK STATE message with their new connection(s) to E.

A	A	B	C	D	E
A			K / -	K / -	
B			- / L	- / L	
C	K / -	- / L		- / L	- / L
D	K / -	- / L	- / L		- / L
E			- / L	- / L	

B	A	B	C	D	E
A			L / -	L / -	
B			- / K	- / L	
C	L / -	- / K		L / -	L / -
D	L / -	- / L	L / -		L / -
E			L / -	L / -	

C	A	B	C	D	E
A			K / -	L / -	
B			- / K	- / L	
C	K / -	- / K		K / L	
D	L / -	- / L	K / L		L / L
E				L / L	

D	A	B	C	D	E
A			L / -	K / -	
B			- / L	- / K	
C	L / -	- / L		K / K	L / L
D	K / -	- / K	K / K		
E			L / L		

- E sends on Net 0: connections to hosts B, C and D on Net 1

  ⇒ all hosts on Net 0 know about their connection to E
  ⇒ all hosts on Net 0 learn about E's connections to B, C and D on Net 1

A	A	B	C	D	E
A			K / -	K / -	**K** / -
B			- / L	- / L	- / **L**
C	K / -	- / L		- / L	- / L
D	K / -	- / L	- / L		- / L
E	**K** / -	- / **L**	- / L	- / L	

B	A	B	C	D	E
A			L / -	L / -	
B			- / K	- / L	
C	L / -	- / K		L / -	L / -
D	L / -	- / L	L / -		L / -
E			L / -	L / -	

C	A	B	C	D	E
A			K / -	L / -	
B			- / K	- / L	- / **L**
C	K / -	- / K		K / L	**K** / **L**
D	L / -	- / L	K / L		L / L
E		- / **L**	**K** / **L**	L / L	

D	A	B	C	D	E
A			L / -	K / -	
B			- / L	- / K	- / **L**
C	L / -	- / L		K / K	L / L
D	K / -	- / K	K / K		**K** / **L**
E		- / **L**	L / L	**K** / **L**	

- E sends on Net 1: connections to hosts A, C and D on Net 0

  ⇒ all hosts on Net 1 know about their connection to E
  ⇒ all hosts on Net 1 learn about E's connections to A, C and D on Net 0

A	A	B	C	D	E
A			K / -	K / -	K / -
B			- / L	- / L	- / L
C	K / -	- / L		- / L	- / L
D	K / -	- / L	- / L		- / L
E	K / -	- / L	- / L	- / L	
∑	* / -	- / *	* / *	* / *	* / *

B	A	B	C	D	E
A			L / -	L / -	**L** / -
B			- / K	- / K	- / **K**
C	L / -	- / K		L / -	L / -
D	L / -	- / K	L / -		L / -
E	**L** / -	- / **K**	L / -	L / -	
∑	* / -	- / *	* / *	* / *	* / *

C	A	B	C	D	E
A			K / -	L / -	**L** / -
B			- / K	- / L	- / L
C	K / -	- / K		K / K	K / **K**
D	L / -	- / L	K / K		L / L
E	**L** / -	- / L	K / **K**	L / L	
∑	* / -	- / *	* / *	* / *	* / *

D	A	B	C	D	E
A			L / -	K / -	**L** / -
B			- / L	- / K	- / L
C	L / -	- / L		K / K	L / L
D	K / -	- / K	K / K		K / **K**
E	**L** / -	- / L	L / L	K / **K**	
∑	* / -	- / *	* / *	* / *	* / *

Now, after E has broadcast its connections to other hosts, each hosts' connection table is filled completely and all hosts within the system have the same view of the global link state.

B.4 Class Two Host Startup

Now, the new host F is assumed to be of Class Two and has its interface connected to Net 0. At F's boot time, the other hosts' connection tables are like the ones shown below. Only F has an empty table.

A	A	B	C	D	F
A			K / -	K / -	
B			- / L	- / L	
C	K / -	- / L		- / L	
D	K / -	- / L	- / L		
F					

B	A	B	C	D	F
A			L / -	L / -	
B			- / K	- / D	
C	L / -	- / K		L / -	
D	L / -	- / D	L / -		
F					

C	A	B	C	D	F
A			K / -	L / -	
B			- / K	- / L	
C	K / -	- / K		K / D	
D	L / -	- / L	K / D		
F					

D	A	B	C	D	F
A			L / -	K / -	
B			- / L	- / K	
C	L / -	- / L		K / K	
D	K / -	- / K	K / K		
F					

F	A	B	C	D	F
A					
B					
C					
D					
F					

First Iteration

As long as no LINK STATE message from F is received by hosts A, B, C and D, the content of their tables don't change compared to the System Startup scenario. Only F's table is filled with new information.

- A sends on Net 0: no connections to other hosts on Net 1

F	A	B	C	D	F
A					**K** / -
B					
C					
D					
F	**K** / -				

- B sends on Net 1: no connections to other hosts on Net 0

  ⇒ since F is not connected to Net 1, this message is never received by F

- C sends on Net 0: connections to hosts B and D on Net 1

F	A	B	C	D	F
A					K / -
B			- / **L**		
C		- / **L**		- / **L**	**K** / -
D			- / **L**		
F	K / -		**K** / -		

- C sends on Net 1: connections to hosts A and D on Net 0

  ⇒ since F is not connected to Net 1, this message is never received by F

- D sends on Net 0: connections to hosts B and C on Net 1

F	A	B	C	D	F
A					K / -
B			- / L	- / **L**	
C		- / L		- / L	K / -
D		- / **L**	- / L		**K** / -
F	K / -		K / -	**K** / -	

- D sends on Net 1: connections to hosts A and C on Net 0

  ⇒ since F is not connected to Net 1, this message is never received by F

Now, F's table won't change any more in this first iteration because it has received all messages broadcast by other hosts on Net 0.

F's view of the global link state is the following:

F	A	B	C	D	F
∑	* / -	- / *	* / *	* / *	* / -

- F sends on Net 0: no connections to other hosts on Net 1

  ⇒ all hosts on Net 0 know about their connection to F

A	A	B	C	D	F
A			K / -	K / -	**K** / -
B			- / L	- / L	
C	K / -	- / L		- / L	
D	K / -	- / L	- / L		
F	**K** / -				
∑	* / -	- / *	* / *	* / *	* / -

B	A	B	C	D	F
A			L / -	L / -	
B			- / K	- / D	
C	L / -	- / K		L / -	
D	L / -	- / D	L / -		
F					
∑	* / -	- / *	* / *	* / *	- / -

C	A	B	C	D	F
A			K / -	L / -	
B			- / K	- / L	
C	K / -	- / K		K / D	**K** / -
D	L / -	- / L	K / D		
F			**K** / -		
∑	* / -	- / *	* / *	* / *	* / -

D	A	B	C	D	F
A			L / -	K / -	
B			- / L	- / K	
C	L / -	- / L		K / K	
D	K / -	- / K	K / K		**K** / -
F				**K** / -	
∑	* / -	- / *	* / *	* / *	* / -

Second Iteration

F's table doesn't change, because no other information will be received like the ones before.

- A sends on Net 0: no connections to other hosts on Net 1

  ⇒ since this is nothing new, no new content is added to anyone's table compared to the first iteration

- B sends on Net 1: no connections to other hosts on Net 0

  ⇒ since this is nothing new, no new content is added to anyone's table compared to the first iteration

- C sends on Net 0: connections to hosts B and D on Net 1

  ⇒ since this is nothing new, no new content is added to anyone's table compared to the first iteration

- C sends on Net 1: connections to hosts A, D and F on Net 0

  ⇒ all hosts on Net 1 learn about C's connection to A, D and F on Net 0

A	A	B	C	D	F
A			K / -		
B			/ L		
C	K / -	- / L		- / L	
D			- / L		
F					

B	A	B	C	D	F
A			**L** / -		
B			- / **K**		
C	**L** / -	- / **K**		**L** / -	**L** / -
D			**L** / -		
F			**L** / -		

C	A	B	C	D	F
A			K / -		
B			- / K		
C	K / -	- / K			
D					
F					

D	A	B	C	D	F
A			**L** / -	K / -	
B			- / L	- / K	
C	**L** / -	- / L		K / **K**	**L** / -
D	K / -	- / K	K / **K**		
F			**L** / -		

- D sends on Net 0: connections to hosts B and C on Net 1

  ⇒ since this is nothing new, no new content is added to anyone's table compared to the first iteration

- D sends on Net 1: connections to hosts A, C and F on Net 0

⇒ all hosts on Net 1 learn about D's connection to F on Net 0

A	A	B	C	D	F
A			K / -	K / -	
B			/ L	- / L	
C	K / -	- / L		- / L	
D	K / -	- / L	- / L		
F					

B	A	B	C	D	F
A			L / -	**L** / -	
B			- / K	- / **K**	
C	L / -	- / K		L / -	L / -
D	**L** / -	- / **K**	L / -		**L** / -
F			L / -	**L** / -	

C	A	B	C	D	F
A			K / -	**L** / -	
B			- / K	- / L	
C	K / -	- / K		K / **K**	
D	**L** / -	- / L	K / **K**		**L** / -
F				**L** / -	

D	A	B	C	D	F
A			L / -	K / -	
B			- / L	- / K	
C	L / -	- / L		K / K	L / -
D	K / -	- / K	K / K		
F			L / -		

- F sends on Net 0: no connections to other hosts on Net 1

A	A	B	C	D	F
A			K / -	K / -	**K** / -
B			/ L	- / L	
C	K / -	- / L		- / L	
D	K / -	- / L	- / L		
F	**K** / -				
Σ	* / -	- / *	* / *	* / *	* / -

B	A	B	C	D	F
A			L / -	L / -	
B			- / K	- / K	
C	L / -	- / K		L / -	L / -
D	L / -	- / K	L / -		L / -
F			L / -	L / -	
Σ	* / -	- / *	* / *	* / *	* / -

C	A	B	C	D	F
A			K / -	L / -	
B			- / K	- / L	
C	K / -	- / K		K / K	**K** / -
D	L / -	- / L	K / K		L / -
F			**K** / -	L / -	
Σ	* / -	- / *	* / *	* / *	* / -

D	A	B	C	D	F
A			L / -	K / -	
B			- / L	- / K	
C	L / -	- / L		K / K	L / -
D	K / -	- / K	K / K		**K** / -
F			L / -	**K** / -	
Σ	* / -	- / *	* / *	* / *	* / -

The only change in the tables during this second iteration is that all hosts connected only to Net 1 (here: B) learn about other hosts' connection to F on Net 0. Hosts connected only to Net 0 (here: A) just know about F's existence, but never learn about other hosts' connections to F. Class One hosts benefit from both, they know about F by the receipt of its LINK STATE messages, and learn about other hosts' connection to it.

B.5 Class One Host Interface Startup

Host F is assumed to have had a failed interface to Net 1 which suddenly comes up again. Thus F changes from a Class Two host to a host of Class One. Before F broadcasts its LINK STATE messages on Net 1, too, each hosts' table look like the following.

A	A	B	C	D	F
A			K / -	K / -	K / -
B			- / L	- / L	
C	K / -	- / L		- / L	
D	K / -	- / L	- / L		
F	K / -				

B	A	B	C	D	F
A			L / -	L / -	
B			- / K	- / D	
C	L / -	- / K		L / -	L / -
D	L / -	- / D	L / -		L / -
F			L / -	L / -	

C	A	B	C	D	F
A			K / -	L / -	
B			- / K	- / L	
C	K / -	- / K		K / D	K / -
D	L / -	- / L	K / D		L / -
F			K / -	L / -	

D	A	B	C	D	F
A			L / -	K / -	
B			- / L	- / K	
C	L / -	- / L		K / K	L / -
D	K / -	- / K	K / K		K / -
F			L / -	K / -	

F	A	B	C	D	F
A					K / -
B			- / L	- / L	
C		- / L		- / L	K / -
D		- / L	- / L		K / -
F	K / -		K / -	K / -	

First Iteration

LINK STATE messages received by F on Net 0 don't change its table compared to the Class Two Host Startup scenario, because the content in the first iteration doesn't change. No host on Net 1 knows yet about F's new connection this network. Instead, messages received by F on Net 1 do change its table, because it never has received any messages from there before.

- A sends on Net 0: no connections to other hosts on Net 1

F	A	B	C	D	F
A					**K** / -
B					
C					
D					
F	**K** / -				

- B sends on Net 1: no connections to other hosts on Net 0

F	A	B	C	D	F
A					K / -
B					- / **K**
C					
D					
F	K / -	- / **K**			

- C sends on Net 0: connections to hosts B and D on Net 1

F	A	B	C	D	F
A					K / -
B			- / **L**		- / K
C		- / **L**		- / **L**	**K** / -
D			- / **L**		
F	K / -	- / K	**K** / -		

- C sends on Net 1: connections to hosts A, D and F on Net 0

F	A	B	C	D	F
A			**L** / -		K / -
B			- / L		- / K
C	**L** / -	- / L		**L** / L	K / **K**
D			**L** / L		
F	K / -	- / K	K / **K**		

- D sends on Net 0: connections to hosts B and C on Net 1

F	A	B	C	D	F
A			L / -		K / -
B			- / L	- / **L**	- / K
C	L / -	- / L		L / L	K / K
D		- / **L**	L / L		**K** / -
F	K / -	- / K	K / K	**K** / -	

- D sends on Net 1: connections to hosts A, C and F on Net 0

F	A	B	C	D	F
A			L / -	**L** / -	K / -
B			- / L	- / L	- / K
C	L / -	- / L		L / L	K / K
D	**L** / -	- / L	L / L		K / **K**
F	K / -	- / K	K / K	K / **K**	
∑	* / -	- / *	* / *	* / *	* / *

- F sends on Net 0: no connections on Net 1

  ⇒ since this is an empty LINK STATE message, no changes appear in the tables of hosts A, C and D

- F sends on Net 1: connections to hosts A, C and D on Net 0

A	A	B	C	D	F
A			K / -	K / -	K / -
B			- / L	- / L	
C	K / -	- / L		- / L	
D	K / -	- / L	- / L		
F	K / -				
∑	* / -	- / *	* / *	* / *	* / -

B	A	B	C	D	F
A			L / -	L / -	**L** / -
B			- / K	- / L	- / **K**
C	L / -	- / K		L / -	L / -
D	L / -	- / L	L / -		L / -
F	**L** / -	- / **K**	L / -	L / -	
∑	* / -	- / *	* / *	* / *	* / *

C	A	B	C	D	F
A			K / -	L / -	**L** / -
B			- / K	- / L	
C	K / -	- / K		K / K	K / **K**
D	L / -	- / L	K / K		L / -
F	**L** / -		K / **K**	L / -	
∑	* / -	- / *	* / *	* / *	* / *

D	A	B	C	D	F
A			L / -	K / -	**L** / -
B			- / L	- / K	
C	L / -	- / L		K / K	L / -
D	K / -	- / K	K / K		K / **K**
F	**L** / -		L / -	K / **K**	
∑	* / -	- / *	* / *	* / *	* / *

This is some kind of an asymmetric case. Because no host claims to have a connection to F on Net 1, no Class Two host on Net 0 (here: A) has an idea about F's connections on Net 1. However, Class Two hosts on Net 1 not only learn about other hosts connections to F on Net 0, but also receive a LINK STATE message from F which enables them to identify F as a router candidate. Thus, after only one iteration, F's change from a host of Class Two to one of Class One is detected by all Class Two hosts on the newly connected network (here: B on Net 1), and F can be selected by them as a router candidate.

Second Iteration

- A sends on Net 0: no connections to hosts on Net 1

  ⇒ since A tells nothing new, no new content is added to anyone's table

- B sends on Net 1: no connections to hosts on Net 0

  ⇒ since B tells nothing new, no new content is added to anyone's table

- C sends on Net 0: connections to hosts B, D and F on Net 1

A	A	B	C	D	F
A			K / -		
B			- / L		
C	K / -	- / L		- / L	- / L
D			- / L		
F			- / L		

B	A	B	C	D	F
A					
B					
C					
D					
F					

C	A	B	C	D	F
A			K / -		
B			- / K		
C	K / -	- / K			
D					
F					

D	A	B	C	D	F
A				K / -	
B			- / L	- / K	
C		- / L		K / L	- / L
D	K / -	- / K	K / L		
F			- / L		

F	A	B	C	D	F
A					K / -
B			- / L		- / K
C		- / L		- / L	K / L
D			- / L		
F	K / -	- / K	K / L		

- C sends on Net 1: connections to hosts A, D and F on Net 0

A	A	B	C	D	F
A			K / -		
B			- / L		
C	K / -	- / L		- / L	- / L
D			- / L		
F			- / L		

B	A	B	C	D	F
A			**L** / -		
B			- / **K**		
C	**L** / -	- / **K**		**L** / -	**L** / -
D			**L** / -		
F			**L** / -		

C	A	B	C	D	F
A			K / -		
B			- / K		
C	K / -	- / K			
D					
F					

D	A	B	C	D	F
A			**L** / -	K / -	
B			- / L	- / K	
C	**L** / -	- / L		K / **K**	**L** / L
D	K / -	- / K	K / **K**		
F			**L** / L		

F	A	B	C	D	F
A			**L** / -		K / -
B			- / L		- / K
C	**L** / -	- / L		**L** / L	K / **K**
D			**L** / L		
F	K / -	- / K	K / **K**		

- D sends on Net 0: connections to hosts B, C and F on Net 1

A	A	B	C	D	F
A			K / -	**K** / -	
B			- / L	- / **L**	
C	K / -	- / L		- / L	- / L
D	**K** / -	- / **L**	- / L		- / **L**
F			- / L	- / **L**	

B	A	B	C	D	F
A			L / -		
B			- / K		
C	L / -	- / K		L / -	L / -
D			L / -		
F			L / -		

C	A	B	C	D	F
A			K / -		
B			- / K	- / **L**	
C	K / -	- / K		**K** / **L**	
D		- / **L**	**K** / **L**		- / **L**
F				- / **L**	

D	A	B	C	D	F
A			L / -	K / -	
B			- / L	- / K	
C	L / -	- / L		K / K	L / L
D	K / -	- / K	K / K		
F			L / L		

F	A	B	C	D	F
A			L / -		K / -
B			- / L	- / **L**	- / K
C	L / -	- / L		L / L	K / K
D		- / **L**	L / L		**K** / **L**
F	K / -	- / K	K / K	**K** / **L**	

- D sends on Net 1: connections to hosts A, C and F on Net 0

A	A	B	C	D	F
A			K / -	K / -	
B			- / L	- / L	
C	K / -	- / L		- / L	- / L
D	K / -	- / L	- / L		- / L
F			- / L	- / L	

B	A	B	C	D	F
A			L / -	**L** / -	
B			- / K	- / **K**	
C	L / -	- / K		L / -	L / -
D	**L** / -	- / **K**	L / -		**L** / -
F			L / -	**L** / -	

C	**A**	**B**	**C**	**D**	**F**
A			K / -	**L** / -	
B			- / K	- / L	
C	K / -	- / K		K / **K**	
D	**L** / -	- / L	K / **K**		**L** / L
F				**L** / L	

D	**A**	**B**	**C**	**D**	**F**
A			L / -	K / -	
B			- / L	- / K	
C	L / -	- / L		K / K	L / L
D	K / -	- / K	K / K		
F			L / L		

F	A	B	C	D	F
A			L / -	**L** / -	K / -
B			- / L	- / L	- / K
C	L / -	- / L		L / L	K / K
D	**L** / -	- / L	L / L		K / **K**
F	K / -	- / K	K / K	K / **K**	
∑	* / -	- / *	* / *	* / *	* / *

F's table is now completed and not considered any more in the following discussion of F's LINK STATE messages.

- F sends on Net 0: connections to hosts B, C and D on Net 1

A	A	B	C	D	F
A			K / -	K / -	**K** / -
B			- / L	- / L	- / **L**
C	K / -	- / L		- / L	- / L
D	K / -	- / L	- / L		- / L
F	**K** / -	- / **L**	- / L	- / L	

B	A	B	C	D	F
A			L / -	L / -	
B			- / K	- / K	
C	L / -	- / K		L / -	L / -
D	L / -	- / K	L / -		L / -
F			L / -	L / -	

C	A	B	C	D	F
A			K / -	L / -	
B			- / K	- / L	- / **L**
C	K / -	- / K		K / K	**K** / **L**
D	L / -	- / L	K / K		L / L
F		- / **L**	**K** / **L**	L / L	

D	A	B	C	D	F
A			L / -	K / -	
B			- / L	- / K	- / **L**
C	L / -	- / L		K / K	L / L
D	K / -	- / K	K / K		**K** / **L**
F		- / **L**	L / L	**K** / **L**	

- F sends on Net 1: connections to hosts A, C and D on Net 0

A	A	B	C	D	F
A			K / -	K / -	K / -
B			- / L	- / L	- / L
C	K / -	- / L		- / L	- / L
D	K / -	- / L	- / L		- / L
F	K / -	- / L	- / L	- / L	
∑	* / -	- / *	* / *	* / *	* / *

B	A	B	C	D	F
A			L / -	L / -	**L** / -
B			- / K	- / K	- / **K**
C	L / -	- / K		L / -	L / -
D	L / -	- / K	L / -		L / -
F	**L** / -	- / **K**	L / -	L / -	
∑	* / -	- / *	* / *	* / *	* / *

C	A	B	C	D	F
A			K / -	L / -	**L** / -
B			- / K	- / L	- / L
C	K / -	- / K		K / K	K / **K**
D	L / -	- / L	K / K		L / L
F	**L** / -	- / L	K / **K**	L / L	
∑	* / -	- / *	* / *	* / *	* / *

D	A	B	C	D	F
A			L / -	K / -	**L** / -
B			- / L	- / K	- / L
C	L / -	- / L		K / K	L / L
D	K / -	- / K	K / K		K / **K**
F	**L** / -	- / L	L / L	K / **K**	
∑	* / -	- / *	* / *	* / *	* / *

After the second iteration, Class Two hosts on the 'old' network (here: A on Net 0) can identify F as a router candidate, too. They don't only receive F's LINK STATE message, but also learn about F's connections to other hosts on the other network by the content of both F's and of other hosts' LINK STATE messages.

B.6 Class One Host Failure

Host D fails, i.e. both interfaces go down at the same time and no more LINK STATE messages are sent by D any longer. Before D's failure the tables look like the following.

A	A	B	C	D
A			K / -	K / -
B			- / L	- / L
C	K / -	- / L		- / L
D	K / -	- / L	- / L	

B	A	B	C	D
A			L / -	L / -
B			- / K	- / K
C	L / -	- / K		L / -
D	L / -	- / K	L / -	

C	A	B	C	D
A			K / -	L / -
B			- / K	- / L
C	K / -	- / K		K / K
D	L / -	- / L	K / K	

D	A	B	C	D
A			L / -	K / -
B			- / L	- / K
C	L / -	- / L		K / K
D	K / -	- / K	K / K	

First Iteration

- A sends on Net 0: no connections to other hosts on Net 1
- A sends nothing on Net 1
- B sends nothing on Net 0
- B sends on Net 1: no connections to other hosts on Net 0
- C sends on Net 0: connections to hosts B and D on Net 1
- C sends on Net 1: connections to hosts A and D on Net 0

Since D has failed, no more LINK STATE messages are broadcast.

A	A	B	C	D
A			K / -	
B			- / L	
C	K / -	- / L		- / L
D			- / L	
Σ	* / -	- / *	* / *	- / *

B	A	B	C	D
A			L / -	
B			- / K	
C	L / -	- / K		L / -
D			L / -	
Σ	* / -	- / *	* / *	* / -

C	A	B	C	D
A			K / -	
B			- / K	
C	K / -	- / K		
D				
Σ	* / -	- / *	* / *	- / -

The tables of hosts A and B still contain a connection between host D and C, because C has broadcast this connections based on its old connection table which was true at that time. But neither host A nor host B will identify D as a router candidate any longer because only one connection from/to C and D, respectively, is indicated in each one's table. Thus, after only one iteration, the failure of a router candidate is detected and another router is selected by the router selection algorithm.

Second Iteration

- A sends on Net 0: no connections to other hosts on Net 1
- A sends nothing on Net 1
- B sends nothing on Net 0
- B sends on Net 1: no connections to other hosts on Net 0
- C sends on Net 0: connection to host B Net 1

A	A	B	C	D
A			**K** / -	
B			- / **L**	
C	**K** / -	- / **L**		
D				

B	A	B	C	D
A				
B				
C				
D				

C	A	B	C	D
A			K / -	
B			- / K	
C	K / -	- / K		
D				

- C sends on Net 1: connection to host A Net 0

A	A	B	C	D
A			K / -	
B			- / L	
C	K / -	- / L		
D				
∑	* / -	- / *	* / *	- / -

B	A	B	C	D
A			**L** / -	
B			- / **K**	
C	**L** / -	- / **K**		
D				
∑	* / -	- / *	* / *	- / -

C	A	B	C	D
A			K / -	
B			- / K	
C	K / -	- / K		
D				
∑	* / -	- / *	* / *	- / -

After the second iteration no more hints for the existence of D are given in anyone's connection table.

B.7 Class Two Host Failure

Host A fails, i.e. its interface to Net 0 goes down.

A	A	B	C	D
A			K / -	K / -
B			- / L	- / L
C	K / -	- / L		- / L
D	K / -	- / L	- / L	

B	A	B	C	D
A			L / -	L / -
B			- / K	- / K
C	L / -	- / K		L / -
D	L / -	- / K	L / -	

C	A	B	C	D
A			K / -	L / -
B			- / K	- / L
C	K / -	- / K		K / K
D	L / -	- / L	K / K	

D	A	B	C	D
A			L / -	K / -
B			- / L	- / K
C	L / -	- / L		K / K
D	K / -	- / K	K / K	

First Iteration

- A sends nothing on both networks
- B sends nothing on Net 0
- B sends on Net 1: no connections to other hosts on Net 0
- C sends on Net 0: connections to hosts B and D on Net 1
- C sends on Net 1: connections to hosts A and C on Net 0
- D sends on Net 0: connections to hosts B and D on Net 1
- D sends on Net 1: connections to hosts A and C on Net 0

A	A	B	C	D
A				
B				
C				
D				

B	A	B	C	D
A			L / -	L / -
B			- / K	- / K
C	L / -	- / K		L / -
D	L / -	- / K	L / -	
∑	* / -	- / *	* / *	* / *

C	A	B	C	D
A				L / -
B			- / K	- / L
C		- / K		K / K
D	L / -	- / L	K / K	
∑	* / -	- / *	* / *	* / *

D	A	B	C	D
A			L / -	
B			- / L	- / K
C	L / -	- / L		K / K
D		- / K	K / K	
∑	* / -	- / *	* / *	* / *

B's table looks the same like before, but C and D don't have a 'K' any longer in the field for the connection to A on Net 0. Thus, subsequent LINK STATE messages won't contain this connection any longer.

Second Iteration

- A sends nothing on both networks
- B sends nothing on Net 0
- B sends on Net 1: no connections to other hosts on Net 0
- C sends on Net 0: connections to hosts B and D on Net 1
- C sends on Net 1: connection to host C on Net 0
- D sends on Net 0: connections to hosts B and C on Net 1
- D sends on Net 1: connection to host C on Net 0

A	A	B	C	D
A				
B				
C				
D				

B	A	B	C	D
A				
B			- / K	- / K
C		- / K		L / -
D		- / K	L / -	
∑	- / -	- / *	* / *	* / *

C	A	B	C	D
A				
B			- / K	- / L
C		- / K		K / K
D		- / L	K / K	
∑	- / -	- / *	* / *	* / *

D	A	B	C	D
A				
B			- / L	- / K
C		- / L		K / K
D		- / K	K / K	
∑	- / -	- / *	* / *	* / *

All connections between A and other hosts are eliminated in each table during this second iteration.

B.8 Class One Host Interface Failure

Host D's interface to Net 1 fails, i.e. it becomes a Class Two host connected only to Net 0. Below are the connection tables of each host before the interface failure.

A	A	B	C	D
A			K / -	K / -
B			- / L	- / L
C	K / -	- / L		- / L
D	K / -	- / L	- / L	

B	A	B	C	D
A			L / -	L / -
B			- / K	- / K
C	L / -	- / K		L / -
D	L / -	- / K	L / -	

C	A	B	C	D
A			K / -	L / -
B			- / K	- / L
C	K / -	- / K		K / K
D	L / -	- / L	K / K	

D	A	B	C	D
A			L / -	K / -
B			- / L	- / K
C	L / -	- / L		K / K
D	K / -	- / K	K / K	

First Iteration

- A sends on Net 0: no connections to other hosts on Net 1

D	A	B	C	D
A				**K** / -
B				
C				
D	**K** / -			

- A sends nothing on Net 1
- B sends nothing on Net 0
- B sends on Net 1: no connections to other hosts on Net 0
- C sends on Net 0: connections to hosts B and D on Net 1

D	A	B	C	D
A				K / -
B			- / **L**	
C		- / **L**		**K** / **L**
D	K / -		**K** / **L**	

- C sends on Net 1: connections to hosts A and D on Net 0

- D sends on Net 0: connections to hosts B and C on Net 1

A	A	B	C	D
A			K / -	**K** / -
B			- / L	- / **L**
C	K / -	- / L		- / L
D	**K** / -	- / **L**	- / L	
∑	* / -	- / *	* / *	* / *

B	A	B	C	D
A			L / -	
B			- / K	
C	L / -	- / K		L / -
D			L / -	
∑	* / -	- / *	* / *	* / -

C	A	B	C	D
A			K / -	
B			- / K	- / **L**
C	K / -	- / K		**K** / **L**
D		- / **L**	**K** / **L**	
∑	* / -	- / *	* / *	* / *

D	A	B	C	D
A				K / -
B			- / L	
C		- / L		K / L
D	K / -		K / L	
∑	* / -	- / *	* / *	* / *

- D sends nothing on Net 1

D's interface failure is not detected by all Class One hosts and all Class Two hosts on Net 0 (here: C, D, and A) during this first iteration. C and D are hosts of Class One, and thus they don't run the router selection algorithm. However, running the router selection algorithm on A may result in erroneously selecting D as a router. Class Two hosts on the other subnet (here: B on Net 1) immediately detect D's failure because they don't receive a LINK STATE message from D any more.

Second Iteration

- A sends on Net 0: no connections to other hosts on Net 1
- A sends nothing on Net 1
- B sends nothing on Net 0
- B sends on Net 1: no connections to other hosts on Net 0
- C sends on Net 0: connection to host B on Net 1

A	A	B	C	D
A			**K** / -	
B			- / **L**	
C	**K** / -	- / **L**		
D				

B	A	B	C	D
A				
B				
C				
D				

C	A	B	C	D
A			K / -	
B			- / **K**	
C	K / -	- / **K**		
D				

D	A	B	C	D
A				K / -
B			- / **L**	
C		- / **L**		**K** / -
D	K / -		**K** / -	

- C sends on Net 1: connections to hosts A and D on Net 0

A	A	B	C	D
A			K / -	
B			- / L	
C	K / -	- / L		
D				

B	A	B	C	D
A			**L** / -	
B			- / **K**	
C	**L** / -	- / **K**		**L** / -
D			**L** / -	

C	A	B	C	D
A			K / -	
B			- / K	
C	K / -	- / K		
D				

D	A	B	C	D
A				K / -
B			- / L	
C		- / L		K / -
D	K / -		K / -	

- D sends on Net 0: no connections to other hosts on Net 1

A	A	B	C	D
A			K / -	**K** / -
B			- / L	
C	K / -	- / L		
D	**K** / -			
∑	* / -	- / *	* / *	* / -

B	A	B	C	D
A			L / -	
B			- / K	
C	L / -	- / K		L / -
D			L / -	
∑	* / -	- / *	* / *	* / -

C	A	B	C	D
A			K / -	
B			- / K	
C	K / -	- / K		**K** / -
D			**K** / -	
∑	* / -	- / *	* / *	* / -

D	A	B	C	D
A				K / -
B			- / L	
C		- / L		K / -
D	K / -		K / -	
∑	* / -	- / *	* / *	* / -

- D sends nothing on Net 1

Now, also A has the right view of D's connections and the router selection algorithm won't result in identifying D as a router candidate any longer.

References

[1] Jean Bacon. 1997. Concurrent Systems. Operating Systems, Database and Distributed Systems: An Integrated Approach, 2nd Edition, Addison-Wesley, ISBN 0-201-17767-6

[2] Ye-In Chang, Ming-Hon Hwang. 1997. An ordered and reliable broadcast protocol for distributed systems. Computer Communications 20, pp. 487-499

[3] Douglas E. Comer. 1995. Internetworking with TCP/IP. Volume 1: Principles, Protocols and Architecture. Prentice-Hall. ISBN 0-13-216987-8

[4] Douglas E. Comer, David L. Stevens. 1999. Internetworking with TCP/IP. Volume 2: Design, Implementation and Internals. Prentice-Hall. ISBN 0-13-973843-6

[5] Douglas E. Comer, David L. Stevens. 1997. Internetworking with TCP/IP. Volume 3: Client-Server Programming and Applications. Prentice-Hall. ISBN 0-13-020272-X

[6] S. Deering. 1991. ICMP Router Discovery Messages. RFC 1256

[7] Ross Finlayson et al. 1984. A Reverse Address Resolution Protocol. RFC 903

[8] C. Huitema, 1995. Routing in the Internet. Prentice-Hall. ISBN 0-13-132192-7

[9] IBM. 1995. OS Open User's Guide. Seventh Edition

[10] IBM. 1995. OS Open Reference Guide

[11] IETF. R. Braden. 1989. Requirements for Internet Hosts - Communication Layers. RFC 1122

[12] M. Frans Kaashoek, Andrew S. Tanenbaum, et al. 1989. An Efficient Reliable Broadcast Protocol.

[13] Vern Paxson. 1997. End-to-End Routing Behavior in the Internet, IEEE/ACM Transactions On Networking, Vol. 5, No. 5

[14] David C. Plummer. 1982. An Ethernet Address Resolution Protocol. RFC 826

[15] J. Postel. 1981. Internet Protocol. RFC 791

[16] J. Postel. 1981. Internet Control Message Protocol. RFC 792

[17] J. Reynolds, J. Postel. 1994. Assigned Numbers. RFC 1700

[18] R. Sedgewick. 1999. Algorithms in C++. 3rd Edition. Addison-Wesley. ISBN 0-201-35088-2

[19] W. Richard Stevens. 1998. UNIX Network Programming, Volume 1: Network APIs: Sockets and XTI. 2nd Edition, Prentice-Hall, ISBN 0-13-490012-X

[20] W. Richard Stevens. 1997. TCP/IP Illustrated, Volume 1: The Protocols. Addison-Wesley. ISBN 0-201-63346-9

[21] W. Richard Stevens, Gary R. Wright. 1995. TCP/IP Illustrated, Volume 2: The Implementation. Addison-Wesley. ISBN 0-201-63354-X

[22] W. Richard Stevens. 1995. Advanced Programming in the UNIX Environment. Addison-Wesley. ISBN 0-201-56317-7

[23] Andrew S. Tanenbaum. 1996. Computer Networks. 3rd Edition. Prentice-Hall. ISBN 0-13-349945-6