NASA-CR-194,923

NASA Contractor Report-194923

NASA-CR-194923
19940032989

Advanced Information Processing System: Authentication Protocols for Network Communication

Richard E. Harper, Stuart J. Adams, Carol A. Babikyan, Bryan P. Butler,
Anne L. Clark, Jaynarayan H. Lala
THE CHARLES STARK DRAPER LABORATORY, INC., CAMBRIDGE, MA 02139

Contract NAS1-18565
June 1994

National Aeronautics and
Space Administration
Langley Research Center
Hampton, Virginia 23681-0001

NASA Contractor Report-194923

Advanced Information Processing System: Authentication Protocols for Network Communication

Richard E. Harper, Stuart J. Adams, Carol A. Babikyan, Bryan P. Butler, Anne L. Clark, Jaynarayan H. Lala

THE CHARLES STARK DRAPER LABORATORY, INC., CAMBRIDGE, MA 02139

Contract NAS1-18565

June 1994

National Aeronautics and
Space Administration
Langley Research Center
Hampton, Virginia 23681-0001

This page intentionally left blank.

Executive Summary

Real-time distributed systems for mission- and life-critical applications typically employ processing sites of various levels of redundancy which communicate with one another as well as with I/O devices by means of fault tolerant intercomputer and I/O networks. The Charles Stark Draper Laboratory has developed and implemented a number of reconfigurable circuit-switched networks for input/output and intercomputer communication. Such networks currently are considered fundamental building blocks for Draper designed distributed fault tolerant systems such as the Advanced Information Processing System (AIPS). They appear as various I/O networks and as the Intercomputer (IC) network for the AIPS system.

The design of a network for this type of mixed redundancy distributed system poses numerous challenges. The following are the principal attributes desired in such an inter-computer network:

Possess Theoretically Sound Basis for Fault Tolerance

Possess Capability for Mixed Redundancy

Support Independence between Network and Site Failures

Possess Flexibility and Expandability

Use Existing Standards

Support Heterogeneous Computational Platforms

Support Heterogeneous Networks

Support Variable Integrity and Security Requirements

Current inter-computer architectures which support some of these attributes do so by means of a tightly synchronous triple layer network where redundant copies of messages are voted by receiving sites. This approach requires that there be some means for synchronizing multiple media layers to one another as well as to the processing sites. Furthermore, the multiple layers are used strictly for fault tolerance and cannot be used for bandwidth enhancement. The inherent limitations imposed by the theoretical requirements for system which does not use authentication has made the implementation of the such a network costly in terms of complexity and performance. Moreover, difficulties of communication and fault identification in networks arise primarily because the sender of a transmission cannot be identified with certainty, an intermediate node can corrupt a message without certainty of detection, and a babbling node cannot be identified and silenced without lengthy diagnosis and reconfiguration.

Authentication Protocols use digital signature techniques to verify the authenticity of messages with high probability. Such protocols appear to provide an efficient solution to many of these problems. The objective of this program is to develop, demonstrate, and evaluate inter-computer communication architectures which employ authentication. As a context for the evaluation, the authentication protocol-based communication concept was demonstrated under this program by hosting a distributed, real-time, flight-critical Advanced Guidance, Navigation, and Control (AGN&C) application on a distributed, heterogeneous, mixed redundancy system of workstations and embedded fault tolerant computers.

This page intentionally left blank.

Table of Contents

List of Tables

This page intentionally left blank.

List of Figures

This page intentionally left blank.

1. Introduction

Real-time distributed systems for mission- and life-critical applications typically employ processing sites of various levels of redundancy which communicate with one another as well as with I/O devices by means of fault tolerant intercomputer and I/O networks. Figure 1 is an abstract representation of such a system.

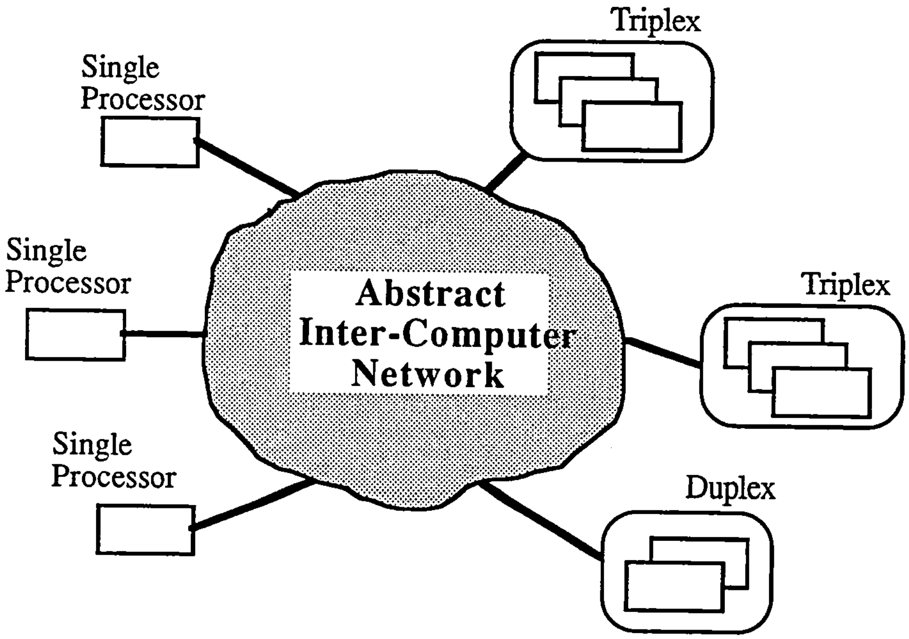


Figure 1. Generic Mixed-Redundancy Distributed System

The Charles Stark Draper Laboratory has developed and implemented a number of reconfigurable circuit-switched networks for input/output and intercomputer communication. Such networks currently are considered fundamental building blocks for Draper designed distributed fault tolerant systems such as the Advanced Information Processing System (AIPS). They appear as various I/O networks and as the Intercomputer (IC) network for the AIPS system.

The design of a network for this type of mixed redundancy distributed system poses numerous challenges. The following are the principal attributes desired in such an inter-computer network:

> Possess Theoretically Sound Basis for Fault Tolerance - The network should support the theoretically sound basis for fault tolerance known as Byzantine resilience; that is, the capability to operate correctly in the presence of a defined number of arbitrary faults.

Possess Capability for Mixed Redundancy - The network should provide communication between sites of varying levels of redundancy without jeopardizing the more reliable sites.

Support Independence between Network and Site Failures - Network and site/processor failures should be independent of one another. Thus any number of site failures (simplex, duplex, or triplex) should not affect communication between non-failed sites. In addition, any single failure in the network should not prevent non-faulty sites from communicating with one another nor should it affect the reliability of communication between different sites.

Possess Flexibility and Expandability - The network should be expandable in terms of adding additional sites regardless of redundancy level, adding additional fault tolerance to the network or any site, and providing increased bandwidth. The network should allow a simple interface for non-redundant sites and provide an interface which is independent of the degree of synchronization within the redundant elements of any single site.

Use Existing Standards - The network should make maximum use of existing industry and military standard network topologies, protocols, and physical media.

Support Heterogeneous Computational Platforms - The network should support interoperability of disparate computational platforms (workstations, embedded computers), operating systems (Unix, LynxOS, Ada Run Time Systems), and programming languages (C, Ada).

Support Heterogeneous Networks - The network should support interoperability of disparate network topologies, protocols, and physical media, such as Ethernet, FDDI, ATM, MIL-STD 1553, etc.

Support Variable Integrity and Security Requirements - The data integrity and security requirements supported by the network should range from data integrity but no security in the presence of random malicious faults to data integrity and security in the presence of crypto-malicous faults.

Current inter-computer architectures which support some of these attributes, such as the AIPS system [Lala84][Lala87], do so by means of a tightly synchronous triple layer network where redundant copies of messages are voted by receiving sites. This approach requires that there be some means for synchronizing multiple media layers to one another as well as to the processing sites. Furthermore, the multiple layers are used strictly for fault tolerance and cannot be used for bandwidth enhancement. The inherent limitations imposed by the theoretical requirements for system which does not use authentication has made the implementation of the such a network costly in terms of complexity and performance. Moreover, difficulties of communication and fault identification in networks arise primarily because the sender of a transmission cannot be identified with certainty, an intermediate node can corrupt a message without certainty of detection, and a babbling node cannot be identified and silenced without lengthy diagnosis and reconfiguration.

Authentication Protocols use digital signature techniques to verify the authenticity of messages with high probability. Such protocols appear to provide an efficient solution to many of these problems.

The purpose of this program is to develop, demonstrate, and evaluate inter-computer communication architectures which employ authentication. As a context for the evaluation, the authentication protocol-based communication concept will be demonstrated by hosting a distributed, real-time, flight-critical Advanced Guidance, Navigation, and Control (AGN&C) application on a distributed, heterogeneous, mixed redundancy system of workstations and embedded fault tolerant computers.

This page intentionally left blank.

2. Technical Approach and Theory

2.1. Architectural Concepts using Authentication Protocols

Authentication Protocols are constructed using data integrity properties bestowed by digital signaturing techniques. The correct use of digital signatures allows a recipient to verify the authenticity of received messages with high probability. Figure 2 shows the use of secret-key authentication for communication between two nodes in an arbitrary network. The sender, node A, signs his message M using the signaturing function F and the secret key k. He then sends the message-signature pair <M, S> to B through the communication network. When B receives the message he computes the signature on the message M using the signaturing function F and the secret key k. If the computed signature matches the signature sent with the message then B is guaranteed that the message is from the sender A and that the message is uncorrupted. Note that we assume in this example that the network has sufficient connectivity to guarantee that at least one uncorrupted copy of the message gets to B (hence B may get several copies), that messages contain sequence numbers to prevent malicious nodes from resequencing messages, and that each pair of nodes share a pairwise common private key. The issue of private versus public key-based authentication will be discussed below.

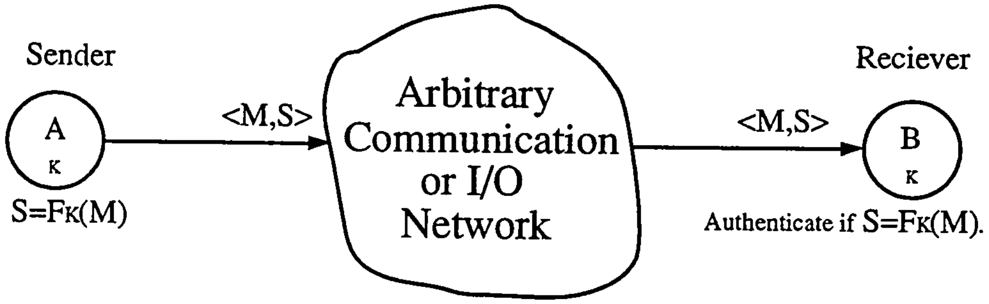


Figure 2. Secret-Key Authentication

A communications protocol which uses digital signatures to achieve fault tolerance must meet several requirements. First, signatures must be message-specific in order to prevent an intervening node from corrupting a message and then appending a correct signature to it. In the flawed example shown below, node A sends "A says go" and signs it with Sa. If the signatures were not message-specific, node B could corrupt the message to "A says stop," append Sa to it, and C would accept the corrupted message as authentic.

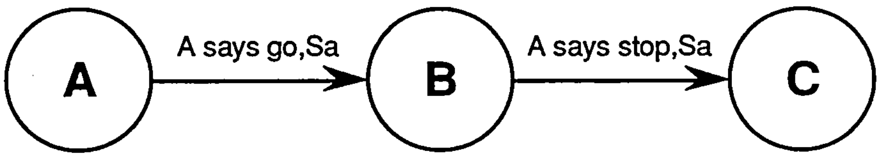


Figure 3. Message Corruption Without Authentication

Message-specific signatures keep a node from copying another's signature and appending it to a corrupted message. Replaying the above scenario with message-specific signatures, if B corrupts the message from "A says go" to "A says stop" and attempts to append Sa to it, C will detect that the signature and the message do not match and discard it.

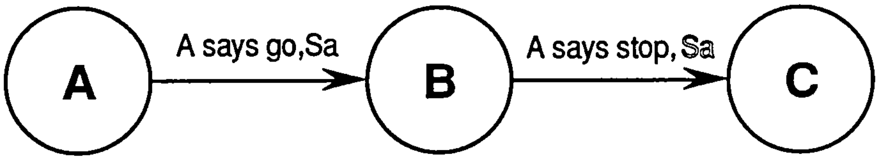


Figure 4. Use of Authentication to Protect Against Message Corruption

However, even with message-specific signatures, an intermediate node can still erroneously and undetectably repeat a message/signature pair. In the flawed example shown below, node A sends "A says go" followed by "A says stop." Node B can erroneously save the first message and transmit it twice, absorbing the second message without C being any the wiser, since both messages received by C authenticate.

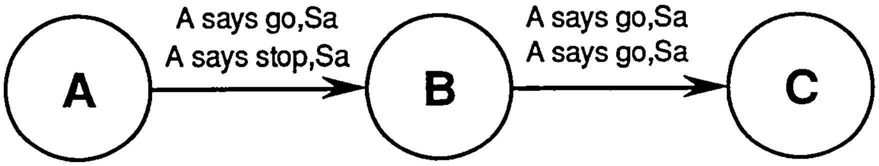


Figure 5. Undetected Message Replay

To eliminate this possibility, a monotonically increasing sequence number must be attached to each message by the sender. The sequence number introduces a varying message component which ensures that a relaying node cannot undetectably replay a saved copy of a message. Such replays would be rejected by the receiver because they have identical sequence numbers. Sequence numbers also ensure that, when a source intentionally transmits two identical messages, the signatures will differ because the signature, which is calculated with respect to the message and the sequence number, has a spectrally white dependence on both the message and the sequence number.

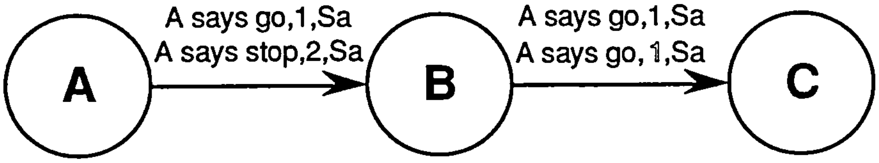


Figure 6. Use of Sequence Numbers to Protect Against Undetected Message Replay

The above figures show a single path from node A to node C, passing through node B. This connectivity is inadequate to achieve fault masking communication, since, even with the precautions outlined above, a single node can absorb a message, as shown below.

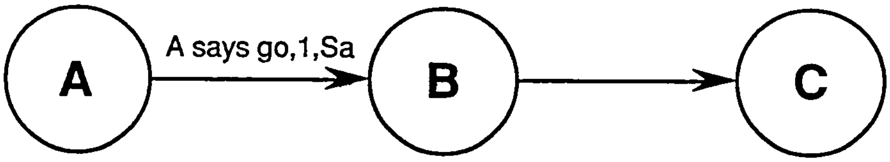


Figure 7. Undetected Message Absorption

Therefore dual, mutually exclusive paths from each possible source node to each possible destination node are necessary to ensure delivery of at least one correct copy of a given message without retry. In this document, each path is designated a Media Layer.

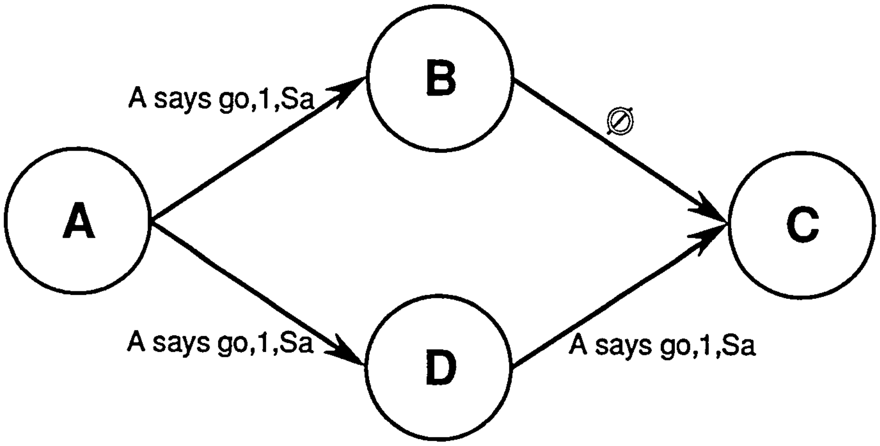


Figure 8. Use of Biconnected Graph to Protect Against Undetected Message Absorption

The use of authentication significantly reduces the theoretical requirements necessary for providing Byzantine resilience (the ability to tolerate arbitrary faults with 100% coverage) and provides an increase in the ability to diagnose faults. Table 1 below summarizes the theoretical requirements necessary to provide f-Byzantine resilience as reported in [Pease80, Lamport82, Dolev82, Dolev83]. The use of authentication decreases the required number of participants and the required network connectivity such that Byzantine resilient protocols may be embedded into common communication and I/O network topologies to provide unity fault survival coverage. These topologies include the ring, redundant contention bus, cube/hypercube, multiple bus, torus, and braided mesh. In addition to the reduction in connectivity, the use of authentication provides a dramatic increase in the efficiency of Byzantine resilient protocols since authentication protocols provide a reduction in the number of required messages from $O(n^f)$ to $O(nf)$. With the ability to embed Byzantine protocols into I/O and communications networks one can make rigorous statements about the reliability of communication in those networks similar to the rigorous statements we make about the reliability of Draper FTPs.

	Protocols Without Authentication (Previous Draper Systems)	Authentication Protocols
Participants	3f+1	2f+1
Connectivity	2f+1	f+1
Rounds of Communication	f+1	f+1
Messages	$O(n^f)$	O(nf)

Table 1. Metrics for Byzantine Agreement

With the reduction in theoretical requirements through the use of authentication one can develop system architectures which exploit these bounds. Towards this end we have developed two types of network architectures which are classified in terms of the connectivity between processors and media layers. The first architecture type is denoted "Cross-Strapped" since the redundant processor channels are cross-strapped with media layers, and the second type is denoted "Non-Cross-Strapped" since it employs no cross-strapping between processor and media layers.

2.1.1. Cross-Strapped Network Architecture

Figure 9 shows the basic scheme for the communication between two triply redundant processing sites in a cross-strapped authentication inter-computer network. The network contains two disjoint media layers, denoted Network L and Network M. Communication proceeds in three basic phases.

In the first step of the protocol, each processor in the sending site signs its local copy of the message to be sent. In the cross-strapped design, all members of the sending site generate identical signatures in the absence of faults.

In the second step, redundant copies of the signed message are voted and transmitted on one or both outgoing paths. Any single failure of a sending processor or failure to correctly sign any single local message copy will be masked by the voter on each outgoing layer. A voter failure may affect at most one media layer, and hence if messages are sent on both layers we are guaranteed that at least one uncorrupted, correctly signed message enters one of the media layers. If a message is sent on only a single layer then end-to-end handshaking must be performed to insure that if one layer is not working properly communication is switched to the alternate layer.

In step 3, upon receiving a message from one or both of the communication paths, the message is authenticated locally by each processor in the receiving site. Each processor discards any messages which are determined to be not authentic. Additionally, messages which arrive and are duplicates of authentic messages already received (either a redundant message from the alternate layer or a message which was incorrectly sent twice) are discarded.

The use of multiple media layers is to guarantee that at least one fault-free path exists between any non-faulty pair of sites. The simplest method for ensuring separation of the multiple paths is to make the media layers completely disjoint (i.e. two separate busses or rings); however, numerous other multi-path communication network architectures could be used. Since no voting of received messages is performed, multiple media layers need not be synchronized and do not necessarily need to be used redundantly (carrying redundant copies of messages). A processing site may choose to use one media layer as its primary communication path, sending all its messages on only this path, and designate the other layer as an alternate to be used if proper handshaking from destination sites is not forthcoming.

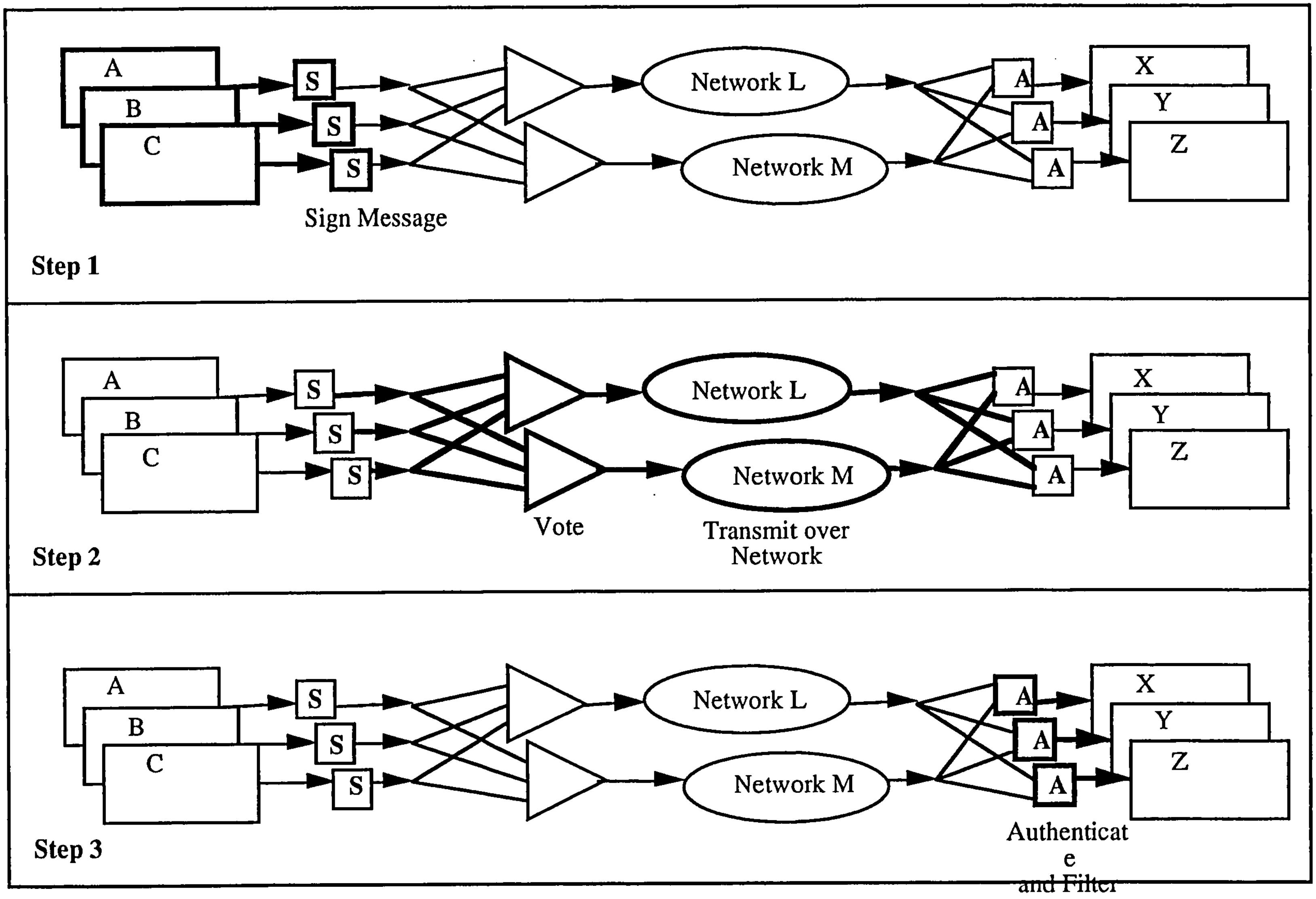


Figure 9. Communication in a Cross-Strapped Authentication Inter-Computer Network

The signature generation and authentication functions must be considered part of the processor Fault Containment Region (FCR). These functions along with the necessary bus interface logic are grouped into a functional unit denoted as the Processor Interface Unit (PIU). In addition each voter must be grouped with its respective layer of the network and the voter and other network interface logic make up a functional unit denoted as the Network Interface Unit (NIU). The voters are isolated from the processing site such that any processor site failure will not affect voter operation.

Figure 4a shows the motivation behind associating the signature generator with each processor rather than placing it after the voter on each media layer. If the processor simply sent data to the network interface unit, which first voted and then signed the data, a single voter fault could cause a corrupted but correctly signed message to be sent on one of the layers. Thus a receiving site could receive conflicting but authentic messages. Figure 10 shows the rationale for grouping the authenticator with each receiving processor rather than placing an authenticator on each receive layer. If placed on a receive layer, an authenticator could send conflicting messages to the different redundant channels of the processing site.

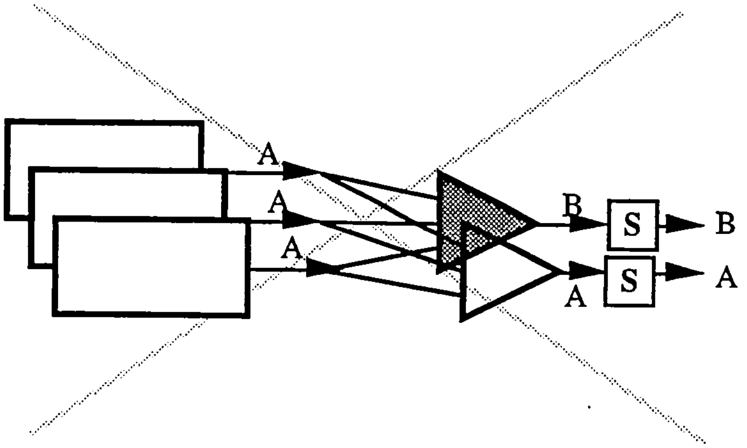


Figure 10a. Incorrect Configuration of Voters

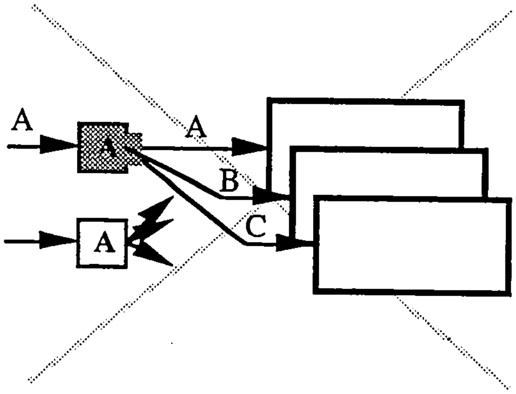


Figure 10b. Incorrect Configuration of Signature Unit

A unique feature of this type of authentication network is that simplex (non-redundant) processors are able to connect to both layers. The motivation behind this is to provide all non-faulty simplex processors with the capability to communicate even in the presence of network faults. A faulty simplex processor is not capable of impersonating any other processing site (simplex/duplex/triplex) since it is incapable, even when faulty, of forging messages. However we must prevent the failure of a simplex processor from corrupting good messages on the network. The method for doing this is shown in Figure 11. The PIU interfaces to the media layers through the NIU which is considered part of the media layer fault containment region. The NIU is responsible for network interfacing, voting of multiple copies of messages from redundant PIUs (only for duplex and triplex sites), network protocol maintenance, etc. The PIU simply queues signed messages to be sent to particular layers. A PIU/processor failure affects only the messages going in or coming out of the queues to the NIU and an NIU failure could take down a whole layer but cannot affect PIU operation to alternate NIUs. The NIU simply acquires the bus/network and sends the message. However, to prevent a faulty simplex from saturating one or more layers with excessive message traffic (thereby degrading network performance) the NIU performs round robin prioritized bus contention as well as acting as a throttle for processor message traffic.

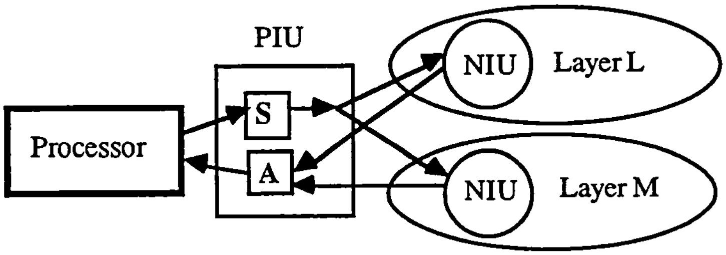


Figure 11. Dual Attached Simplex Processor

2.1.1.1. Dual Token Ring Network Architecture

Figure 9 shows an authentication cross-strapped communication architecture that uses a dual fiber-optic token ring network as a dual media layer. The topology consists of two unidirectional rings consisting of nodes, Network Interface Units, connected with point-to-point uni-directional fiber optic links. Two media layers are provided to insure that in the presence of an arbitrary failure one ring layer will still be usable. Figure 12 also shows the use of a spare link to provide fault tolerance within a single media layer. This would be used to bypass sections of the ring which were faulty. Unlike the AIPS Inter-Computer Network [Lala87], where media layers and network sites were synchronized, the dual layers are not synchronized with one another and operate independently.

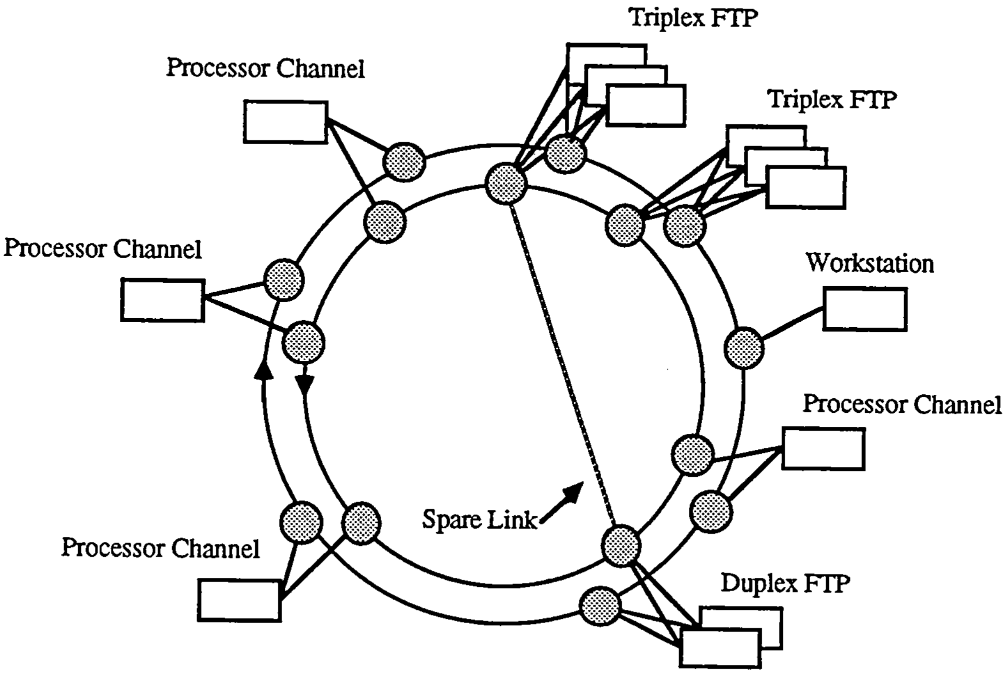


Figure 12. Authentication Dual-Ring Network Architecture

2.1.1.1.1. Network Interface Unit

Figure 10 shows the architecture of the ring Network Interface Unit (NIU). The NIUs each have one incoming port and one outgoing port which are connected to one another to form a unidirectional ring.

The NIUs use a traditional prioritized token passing protocol to perform communication. As a frame of data arrives at the NIU the initial bytes of the frame are buffered in a local FIFO while the Node Controller determines the action to be performed on the frame. Depending upon the frame header the frame may be delivered to the processor attached to the NIU and/or may be forwarded to the next node in the ring. Note that the local frame buffer need only buffer enough of the message for the decision to be made.

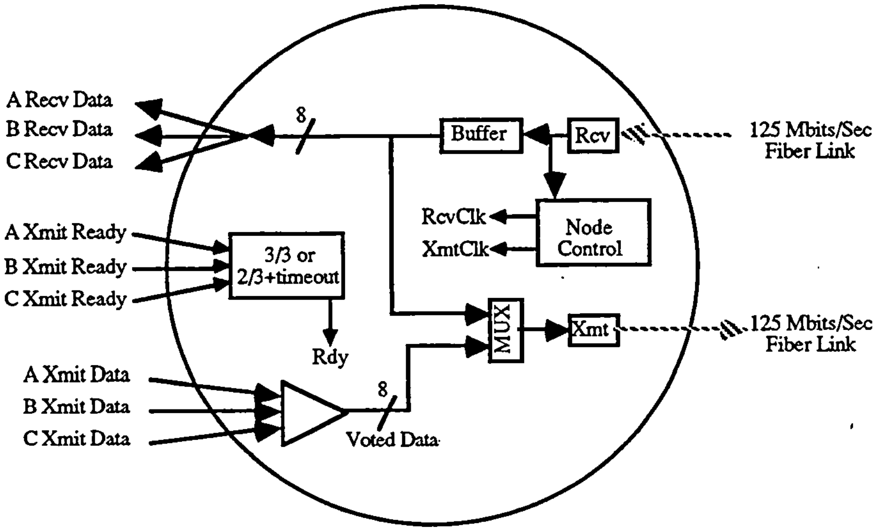


Figure 13. Network Interface Unit

The *xmit data* and *xmit ready* inputs to the NIU are connected to a message queue in the PIU of each redundant processor. The NIU performs a vote/timeout on the message ready signals from each processor. The message ready signals should go true within the maximum skew between the actions of nonfaulty processors, and a timeout needs to be applied in case a processor has failed. Once all of the message ready signals are true or a majority are true and the timeout has expired, the *msg ready* signal informs the node control block that a message is ready to be sent. The node controller then attempts to capture the next transmit token to arrive. Without a prioritized token passing scheme, each site, regardless of redundancy level, would have even access to the ring. A prioritized token passing scheme would be implemented to insure that simplex sites would only have access to the ring if no duplex/triplex sites wished to transmit. Once the NIU has captured the transmit token it clocks data out of the redundant PIU buffers, through the voter and onto the ring. Once the message has been sent, the NIU generates a new transmit token and forwards it onto the ring. Note that the operation of the NIU is independent of processor/PIU failures. In the case of triplex sites any single processor/PIU failure is masked and we are guaranteed that correct data comes out of the voter in each NIU. For simplex sites, a fault in the processor/PIU may cause excessive traffic on the ring. However, a simplex site cannot disrupt communication .

2.1.1.1.2. Processor Interface Unit

The Processor Interface Unit (PIU) is the interface between a processor and the Network Interface Units on each layer. The architecture of the PIU is shown in Figure 14. The PIU consists of a Bus Interface Unit (BIU) and several Authentication Engines (AEs) which act to either sign or authenticate messages on the fly. The BIU consists of a variable set of ring buffers which are used to buffer data between the network and one or more processors on the processor bus. Several transmit ring buffers may exist to allow multiple processor transmit queues as well as multiple priority levels for transmitted data. For received data, ring buffers may be specified to buffer data according to message destination, message priority and source redundancy level.

The PIU contains a single transmit Authentication Engine. For the cross-strapped authenticated architecture the PIU transmits data to one/both NIUs and a selector circuit after the AE determines to which NIU/layer the message is sent. In the case of the non-cross-strapped architecture the PIU may connect directly to a media layer through a Network Media Controller contained within the PIU. The PIU contains one or two receive Authentication Engines. Depending on the type of architecture used, data may be received from an NIU on each layer, from a Network Media Controller within the PIU and from a redundant PIU.

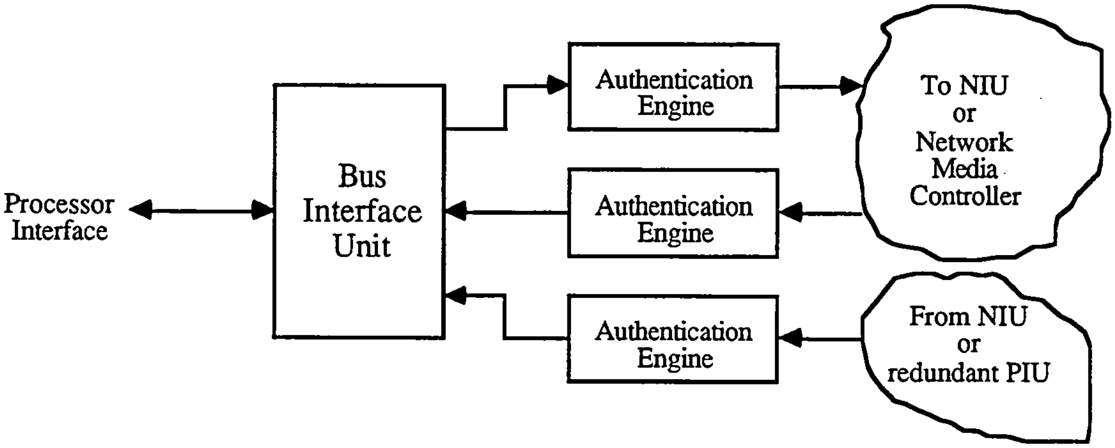


Figure 14. The Processor Interface Unit

2.1.2. Non-Cross-Strapped Network Architecture

The second type of authentication system architecture we have developed requires no cross-strapping between the redundant processing channels and media layers. This provides several advantages over the previously mentioned architecture which are as follows.

(1) The most important implication of eliminating cross-strapping is that we are able to reduce the number of connections required to implement the architecture. This is particularly critical for several reasons. When implementing cross-strapping one is connecting three to five FCRs together. The underlying assumption behind any Byzantine Resilient architecture is that faults will not propagate from one FCR to another. In order to accomplish this careful consideration must be paid to maintain the isolation/integrity between FCRs by

providing both physical and logical barriers against the propagation of faults. This is a costly process due to the complexity of both design and implementation.

(2) The use of a voter in the NIU implies that data received from the multiple PIUs is synchronous such that redundant data values can be voted directly, therefore some mechanism must exist to synchronize redundant PIUs to one another.

(3) In the event of a failure one would like to mask voter inputs so that an erroneous channel's data is no longer voted against good data. (This is typically used to provide the capability to tolerate a subsequent failure.) This requires a means for the redundant channels to communicate a mask value to the NIU as well as a mechanism for voting the masks before they are applied to the voter.

(4) The use of a cross-strapped architecture prevents the use of off-the-shelf communication devices which attach directly to a processor bus. However, with the non-cross-strapped architecture we could use the signature/authentication mechanisms as a "co-processor" for signing and authenticating message with processor communicating directly with the Network I/O device.

Figure 15 below shows a "Non-Cross-Strapped" authentication network architecture and the sequence of steps required to send a message using this architecture.

The source processor(s) (in this case a triplex, each channel of which has a copy of the message to send) sign their local copy of the message with their channel-specific signature.

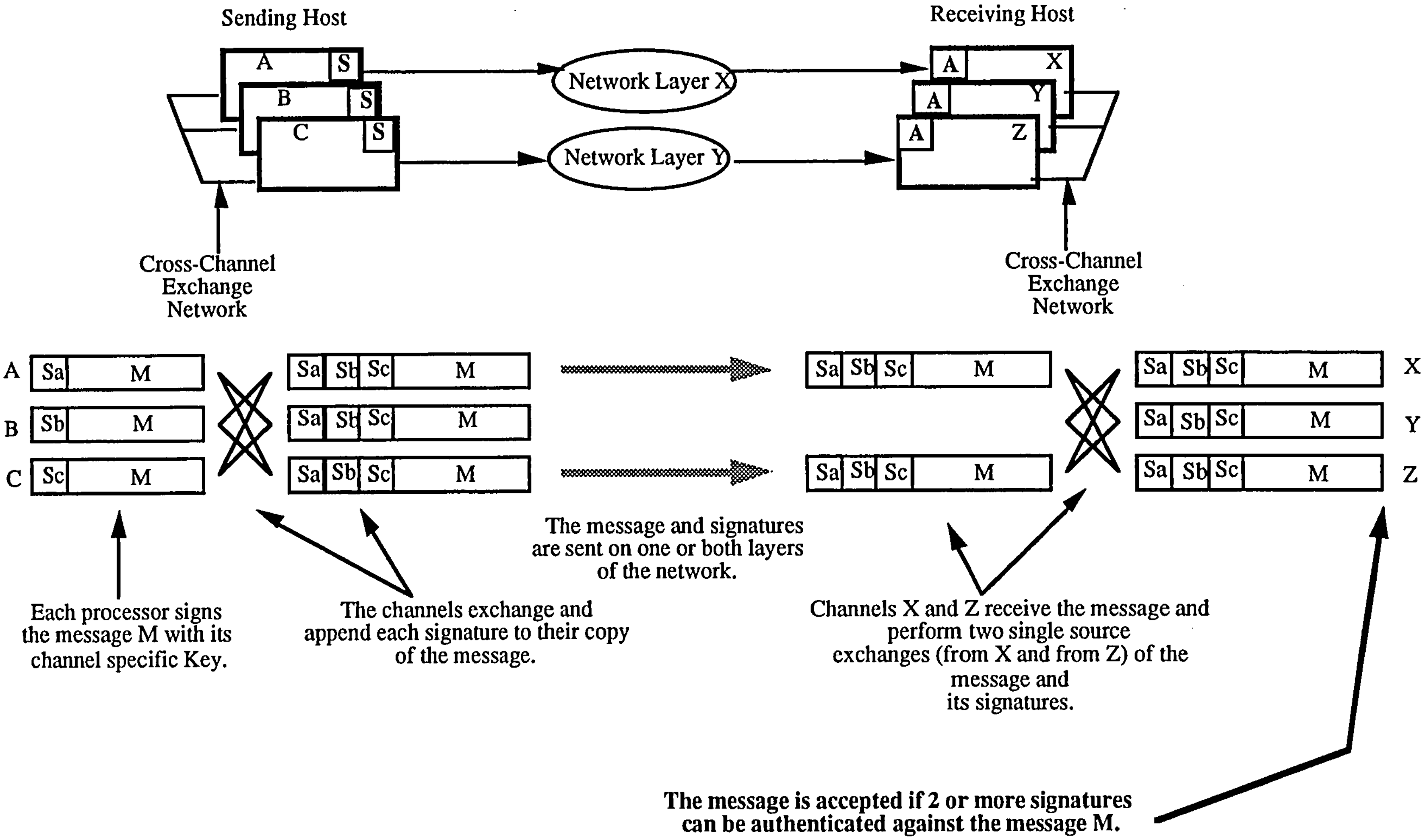
The three channels then exchange their respective signatures (using the cross-channel source congruency exchange mechanism within the triplex) and append each signature to the message to be sent.

The message is then sent to the receiving site on one or more layers of the network. If the message is sent on only one layer then end-to-end handshaking and retry must be used.

When a processor receives a message on either media layer it uses the cross-channel exchange mechanism to exchange the message and signatures. At this point each channel in the redundant receiving site has an identical copy of the message and signatures received over that media layer. Each channel then authenticates each of the three signatures against the message and accepts the message as valid if at least two of the three signatures authenticate against the message. Similar authentication rules apply for quadruplex, duplex, and simplex senders. The duplicate message which may arrive over the other media layer may be unambiguously identified as a duplicate using its sequence number and subsequently discarded.

There are several performance implications for the non-cross-strapped architecture, as compared to the cross-strapped architecture. While the cross-strapped architecture requires little to no use of the sender/receiver's intra-computer exchange mechanism to effect communication, the non-cross-strapped architecture requires the sending channels to exchange their respective signatures (8 bytes per channel for a total of 24 bytes for a triplex) and append the signatures to each message. This process does not incur a large amount of overhead since it is independent of the message size and can be accomplished very quickly. However, a receiving channel must exchange the message received on its media layer through the intra-computer exchange mechanism and then all channels of a redundant recipient must authenticate each redundant signature against the received message. The latter part of the process is message length independent and can be accomplished efficiently, however the single source exchange of the received message is dependent on the message length and therefore the rate at which the processors can exchange data becomes the limiting factor. (It also affects processor bandwidth if the processor must actively exchange the data). Current Draper FTP's can exchange data at a rate of 400KBytes/sec and would hence limit their end-to-end communications bandwidth to this rate (since all communications must pass through this exchange network, sometimes twice), although media layer communications could still proceed at much higher speeds.

The non-cross-strapped architecture also imposes additional requirements of the signature and authentication mechanisms. The signature hardware must now be able to sign a message, return the value to the processor, then receive and append additional signatures from the processor. The authentication mechanism must be able to authenticate three signatures against the received message and indicate the majority result to the processor. However, these additional overheads need not increase the transport delay of the system if signing/authenticating can be overlaid (pipelined) with the reception of the current or next message.



Each processor signs the message M with its channel specific Key.

The channels exchange and append each signature to their copy of the message.

The message and signatures are sent on one or both layers of the network.

Channels X and Z receive the message and perform two single source exchanges (from X and from Z) of the message and its signatures.

The message is accepted if 2 or more signatures can be authenticated against the message M.

Figure 15. Non-Cross-Strapped Authentication Architecture

2.2. Authentication Concepts

The use of authentication protocols presumes the ability to generate a signature-message pair with the following properties:

A1. The receiver can readily verify the authenticity of a sender's signature-message pair.

A2. The sender's signature cannot be forged.

A3. Any alteration of the sender's message can be detected.

Assumption A2 can never be guaranteed since a signature is simply a digital sequence of bits and could be randomly generated by a processor. However, by making the number of bits in the signature sufficiently large we can make the probability of a processor forging a sender's signature by random attempts extremely small. For example, if we assume that a processor generates messages at the rate of 10^6 per/hour, and a probability of system failure of 10^{-10} per hour is desired, then the probability of forging a signature needs to be 10^{-16} per message. The signature would need to consist of at least 53 bits to provide a probability of forgery of 10^{-16}. This is only a lower bound for the number of bits in the key since this assumes random attempts to forge a signature. Assuring unforgeability to malicious attempts may require a greater signature length.

Similarly, assumption A3 cannot be guaranteed with unity probability since the generation of an n bit signature for a k bit message where $n<k$ implies that there exists a signature which corresponds to at least two messages. Thus a processor may change the content of a message with a finite probability that the changed message's signature is identical to the original message's signature. If we assume that the signaturing function uniformly distributes messages over the signature space, that is for all signatures v_i in the signature space Q, v_i is the signature for exactly 2^{k-n} messages, we can bound the probability that a processor can undetectably corrupt a message by guessing its corresponding signature to be 2^{-n}. To provide a probability of undetectable corruption of 10^{-16} per message would again necessitate a key of 53 bits and a “spectrally white” signaturing function.

For many applications we can assume that faulty processors may exhibit malicious behavior, but not to the extent of a malicious cryptoanalyst, particularly in cases in which we are using authentication to protect against random hardware failures in fast hard real-time applications. Hence our authenticator need not be cryptographically secure but only robust against randomly malicious behavior. Due to the malicious nature of some hardware failures we still need to exercise some caution in determining a signaturing scheme. To this end both private key and public key authenticators have been examined.

2.2.1. Private Key Authentication

The approach for an authenticator using private (secret) keys uses a signaturing function $v=S_k(M)$ which generates a signature v for message M based upon key k. Each participant which wishes to authenticate messages from a sender must possess k, the key the sender used to sign the message. A receiver i verifies a message from a sender j by computing $S_k(M)$ using the sender's key and comparing that signature with the signature sent with the message. The problem with private key authentication is that the authentication key must be identical to the key used to sign the message. Hence a receiver which is able to authenticate a message from a sender j will also be able to forge outgoing messages with j's signature. Two methods may be used to prevent this scenario from occurring.

The first method is to use pair-wise common keys. Each sender j has a different key for each other node in the system. A receiver i has the key that each sender uses to sign messages sent to i. (See Figure 16.) Thus in the worst case, a faulty node can only forge messages to himself. This solution creates a number of key management problems and precludes the ability to broadcast information to all participants without attaching a separate signature for each potential recipient.

The second method to prevent a receiver from forging a message is to *compartmentalize* the receiver from the transmitter. (See Figure 16.) By this we mean that one would insure that no propagation of key information from receiver to transmitter would be possible through the use of hardware isolation/protection mechanisms. One method for doing this might be to only allow the transmit key to be set on power-up. Again, this method creates a number of key management problems; however unlike the previous method it does allow a node to broadcast messages without attaching a separate signature for each potential recipient.

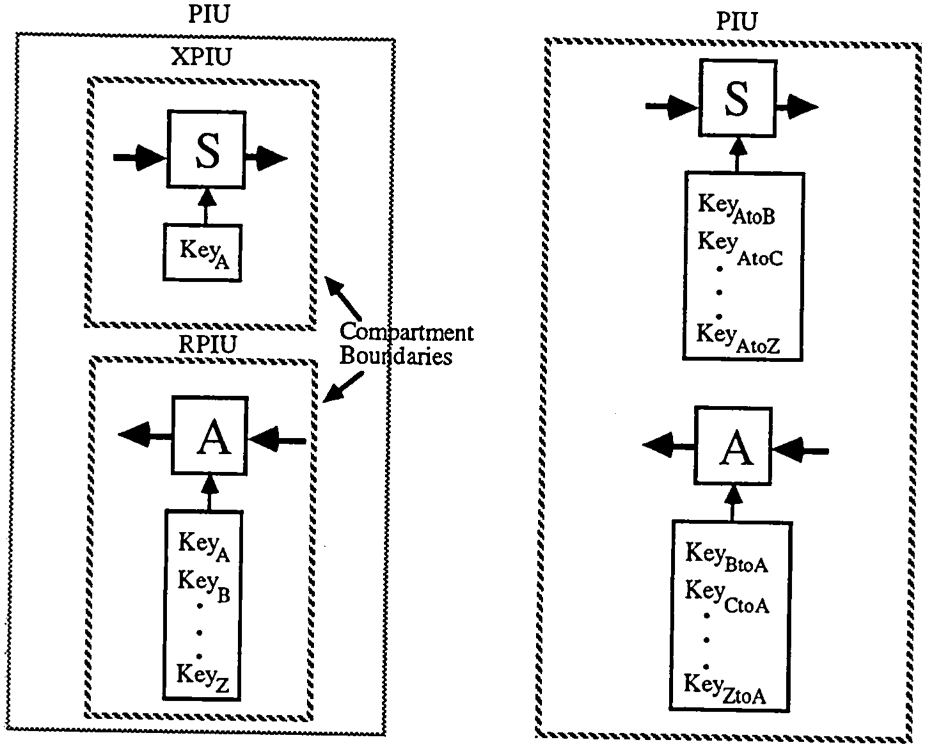


Figure 16. Compartmentalized Single Key vs. Pairwise Secret Key System

2.2.2. Private-Key Implementation Method

An approach for developing a private key authenticator that is secure against randomly malicious faults is to use a signaturing method based upon the Cyclic Redundancy Check (CRC). Most communications systems use the CRC as a method for detecting errors in the communications medium. At transmission time a CRC of the message to be sent is computed and attached to the end of the message. Upon reception, the receiver compares the CRC value it received from the sender with the CRC computed on the received data. If a mismatch occurs then the message has been corrupted. Most currently-used CRC schemes employ a 16 bit CRC check. From the evaluation discussed previously the number of bits necessary to make the probability of forgery adequate for our applications would necessitate a CRC of about 64 bits.

One method for generating simple but cryptographically insecure signatures is to use a modification of the CRC error checking technique. This method would be to have the n bit binary key k[1..n] specify the generator polynomial G for the CRC as follows:

$$G(X)= k[0] + k[1]*X + k[2]*X^2 + k[3]*X^3 + \ldots + k[n-1]*X^n$$

In this manner the signature (or CRC) of a message would be different for nodes with different keys. Some polynomials are better than others for generating signatures for use in digital signaturing schemes [Galetti90].

2.2.3. Public-Key Authentication

In public-key cryptosystems each participant i has an encryption function Ei and a decryption function Di with the following properties:

B1. $E_i(D_i(\text{message}))$ = message

B2. Both E_i(message) and D_i(message) are easy to compute

B3. D_i cannot not be inferred from knowledge of E_i with any reasonable effort

To apply the public-key cryptosystem scheme to a public key authentication system sender i encodes the message M_{text} such that M# is a n bit number representative of M_{text}. The spectrally white encoding function is common knowledge; computation of a CRC over the message is an acceptable algorithm. The encryption procedure E_i for all i is also common knowledge.

For A to send a message to B, the following steps are taken:

1. A computes M# = CRC(message)
2. A computes $S_A = D_A(M\#)$
3. A sends $<M,S_A>$ to B
4. B computes M# = CRC(message)
5. B strips off S_A and computes $E_A(S_A)$
6. If $E_A(S_A) = M\#$ then the message is authentic

While public-key signature systems do not require secret keys, and broadcasts are possible, they do require the availability of suitable functions E and D which possess properties B1-B3. The public-key authentication methods eliminate the problems associated with key management and do allow the efficient use of broadcasts.

2.2.4. Public Key Authentication Method

Current implementations of crypto-secure public-key authenticators are very time consuming and can process data at only a rate of 100-300 KBits/Sec. However, fast hard real-time embedded applications will require an eventual throughput of 1 GBit/Sec but unlike current public-key crypto-systems need only to protect against randomly malicious faults. Therefore we can use a public-key authentication scheme based on modular inverses as suggested in [Lamport82]. This scheme uses two numbers P and P^{-1} (P inverse) for which $P \bullet P^{-1} \bmod N = 1$, where N is a very large number (in our case 2^{64}). With this scheme P is the private signature key and P^{-1} is the public authentication key. Several simple examples of P and P^{-1}'s are shown below.

$13 \times 5 \bmod 2^5 = 1$

$1033 \times 569 \bmod 2^{11} = 1$

$9294586028090793467 \times 3509695877449905l5 \bmod 2^{64} = 1$

To sign a message, the message is first encoded as a 64-bit number with a 64-bit CRC function similar to that discussed in the previous section. Thus, for A to send a message to B using CRC message encoding and modular inverses P_A and P_A^{-1} over $N = 2^{64}$, the following steps are taken:

1. A computes M# = CRC(message)
2. A computes $S_A = D_A(M\#) = P_A \bullet M\# \bmod N$
3. A sends $<M,S_A>$ to B
4. B computes M# = CRC(message)
5. B strips off S_A and computes $E_A(S_A) = S_A \bullet P_A^{-1} \bmod N$
6. If $E_A(S_A) = M\#$ then the message is authentic

This is true since modular multiplication is commutative and associative:

$$(P_A \bullet M\#) \bullet P_A^{-1} = (P_A \bullet P_A^{-1}) \bullet M\# = (1) \bullet M\#$$

The cost of providing public-key authentication using modular inverses is reasonable. To sign a message the sender calculates the 64-bit CRC of a message and performs a single 64-bit multiply. To authenticate a message the receiver calculates the CRC of a received message, multiplies the received signature with the public-key, and compares the result to the CRC calculated on the received message. Note that in this case all nodes may use the same generator polynomial for the CRC function.

2.2.5. DoD Secure Authenticators

Authentication Protocols are able to provide security as well as fault tolerance. In secure applications an authenticator must provide signatures which are cryptographically secure - the data itself may or may not need to be encrypted. Because current applications primarily benefit from the fault tolerance aspects of Authentication Protocols rather than crypto-secure aspects, we have not pursued the use of crypto-secure authentication beyond this point. However, our approach in the development of both the systems architectures and hardware authenticators has been modular. A transition to a crypto-secure environment would only entail the replacement of the authenticator module, and, if encryption is required, the use of the cross-strapped architecture.

2.3. Authentication Implementation Issues

2.3.1. Authentication Engine

The authentication network architectures discussed previously have specified the signature and authentication functions as distinct functional blocks. However, the process of authenticating a message is almost identical to that for signing a message and hence we have chosen to design a single "Authentication Engine" which is capable of either signing messages or authenticating received messages. The Authentication Engine (AE) functional block is pictured in Figure 17. The implementation of the AE requires 14 chips, provides a sustained throughput of 12.5 MBytes/Sec and a transport lag of less than 10 μsec. In transmit mode the authentication engine signs outgoing messages on-the-fly with a key written into the AE by the processor at start-up. In receive mode the AE authenticates received messages on-the-fly and performs a process called "source throttling." The authentication process' receipt of a message is somewhat more complex since the public key for the sender must be looked up in the receive key table according to a source ID contained in the message. Source throttling controls the rate at which messages from specific processors are forwarded through the AE to prevent a faulty simplex processor (which floods the network with messages) from overflowing the receive buffers of other non-faulty processors. For example a simplex processor which exceeded the specified message rate would have its messages marked as rejected (the messages would not be placed in the receiver's buffer).

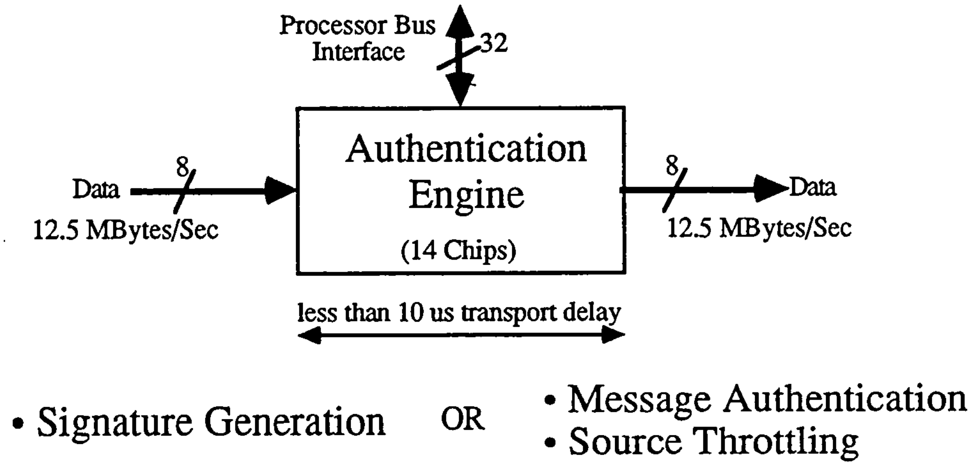


Figure 17. Authentication Engine

The internal components of the Authentication Engine are shown in Figure 18. The input and output data paths are buffered through synchronizing FIFO's to allow data to arrive and leave the AE asynchronously. In transmit mode the AE signs a message by clocking arriving data out of the synchronizing FIFO into the packet FIFO and into the CRC-64 function. When all the data in the packet have been clocked out of the FIFO, the 64-bit CRC is clocked into the MAC (Multiply-Accumulator) where it is multiplied with the private key which was pre-loaded into the MAC at startup. The resulting signature is then clocked out of the MAC through the CRC-64 mechanism (which now acts a 32 bit to 8 bit bus converter) and into the outgoing synchronizing FIFO along with a preset source ID (this places the signature and source ID on the head of the message).

In receive mode, the source ID is first clocked out of the Synchronizing FIFO and into the Source Throttle and CRC-64 functions. The CRC-64 block performs a key lookup from the source ID and loads the source's public key into the MAC. The signature is then clocked out of the Synchronizing FIFO, through the CRC-64 mechanism and into the MAC where it is multiplied with the public key. The remaining data in the packet is then clocked out of the Synchronizing FIFO and into the packet FIFO and CRC-64 mechanism. The 64-bit CRC computed on the packet data is then compared against the result stored in the MAC and if not identical the A-reject line is set true. If the Source Throttle determines that the source has exceeded its specified message rate it sets the T-reject line to true. If either T-reject or A-reject is true the buffering logic (external to the AE) does not buffer the packet but instead places the packet into a diagnostic buffer for examination by fault identification software.

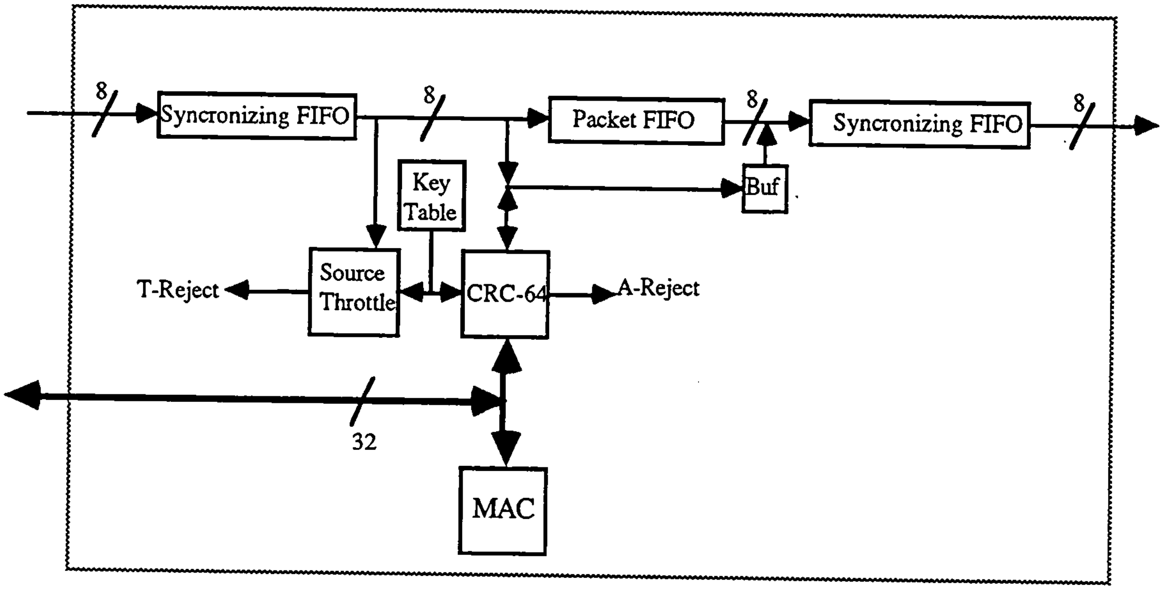


Figure 18. Internals of Authentication Engine

The functions required to implement the AE are either off-the-shelf parts (e.g. the FIFOs, RAMs, MAC) or easily implemented in programmable logic. The CRC-64 function uses a simple byte-wide table lookup algorithm which has been modified to minimize the data path widths as shown in Figure 19. This logic minimizes the required data path widths (from 64 to 32) by multiplexing the RAMs and exclusive OR logic to provide an implementation requiring four RAMs and four programmable logic devices. The look-up table (in the RAMs) is pre-loaded by the processor on start-up for a specific CRC polynomial.

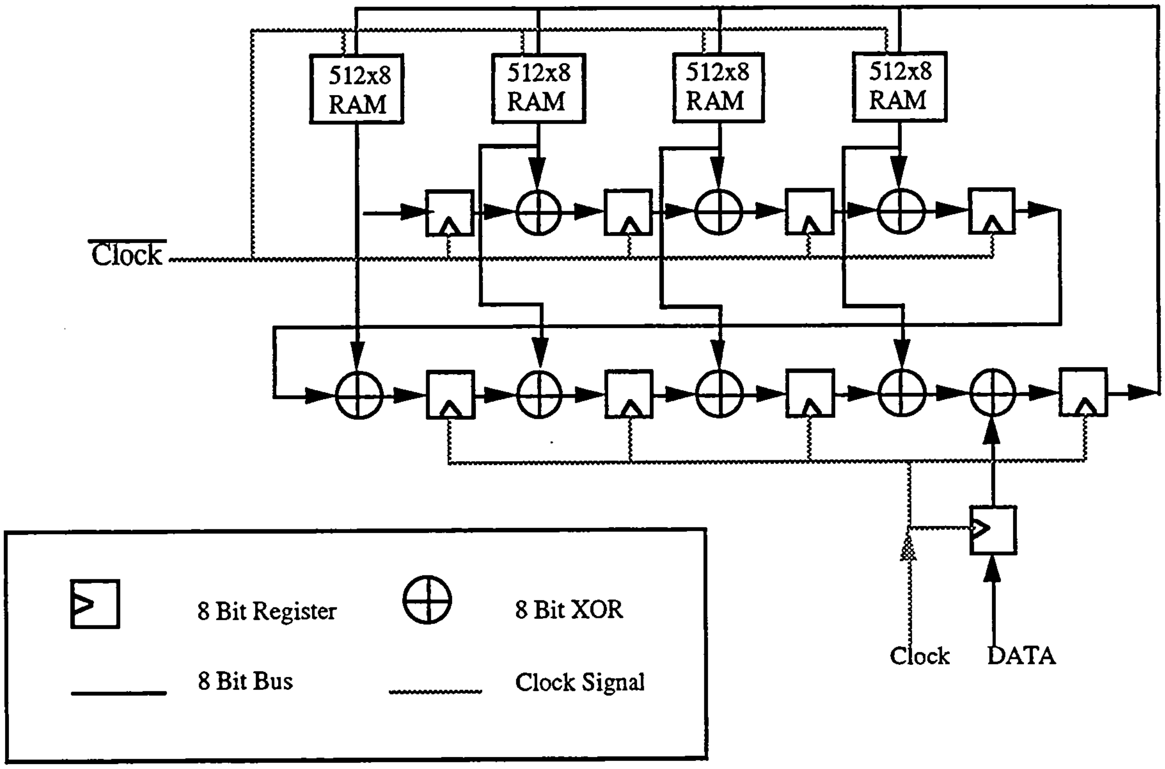


Figure 19. Multiplexed CRC-64 Block Diagram

2.3.2. Crypto-Secure Authenticator

Authentication can be used to provide a Byzantine Resilient communication system which is cryptographically secure for all levels of DoD secure information. The complexities of implementation and the difficulty of receiving NSA endorsement for embedded encryption devices motivate the use of an off-the-shelf encryption device. The Ultron Crypto-Engine[Ultron88] is a near optimal device due to its high throughput and functionality. It is also attractive since the device itself is unclassified and the information regarding its operation is readily available. The Crypto-Engine (CE) is a single chip embeddable device which provides up to 40 Mbits/sec throughput and is NSA-endorsed for all levels of classified data. It has a Message Authentication Code (MAC) mode for generating message specific signatures. In systems which use signatures without encryption, the MAC mode would be used to provide signatures although the device could also be used to provide encrypted DoD secure communication along with authentication. The block diagram of the CE is shown in Figure 20. The CE uses a dual-key system with red and black keys. The red keys are classified at the same level as the data being transferred and the black keys are unclassified and are themselves an encrypted form of the key and are red key specific. To operate, the CE needs a single red key and a black key for every node it wishes to communicate with. Both keys' types have special properties and must be obtained directly from the NSA.

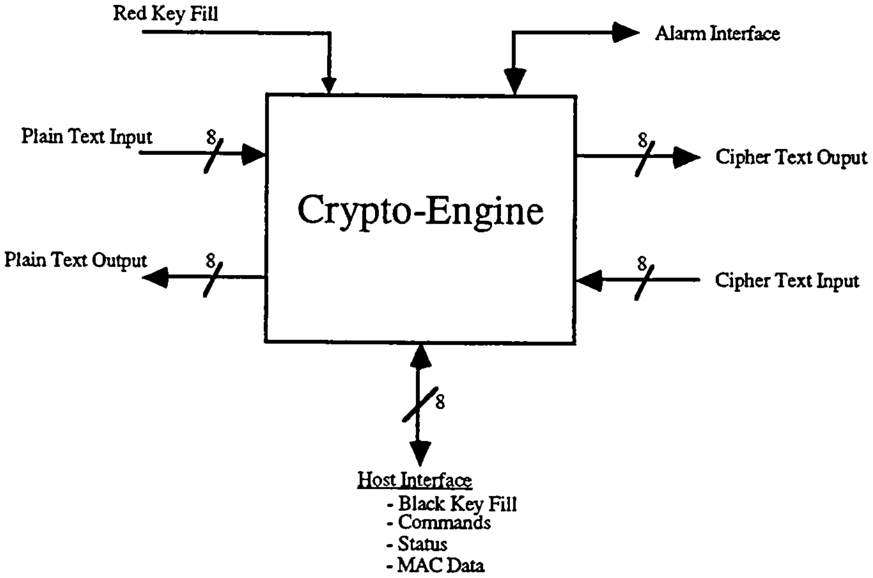


Figure 20. Block Diagram of the Ultron Crypto-Engine

2.3.3. Authenticator Implementation Benchmarks

In order to provide a low cost non-real time alternative to hardware implemented authenticators as well as to provide a means for testing our prototype hardware we have developed and benchmarked software implementations for CRC generation, signature generation, message authentication and an authenticated *vote* operation. These routines have been coded as hand optimized assembly language routines for maximum performance. The benchmarks measured on a 25 MHz 68030 processor are given in Table 2 along with estimates for the time required for equivalent operations when implemented in hardware. Note that except for CRC generation, the times required to perform the functions are independent of message length.

	Software (25 MHz 68030)	Hardware (Estimated)
CRC Generation	6.6 µsec + 2.1 µsec/byte	0 (on-the-fly)
Sign	0.3 µsec/message	700 nsec/message
Authenticate	11.1 µsec/message	800 nsec/message
Auth. "Vote"	8.2 µsec/message	500 nsec/message

Table 2. Benchmarks of Authentication Mechanisms

This page intentionally left blank.

3. Detailed Design of Authentication Protocol System

3.1. Overview

An Authentication Protocol (AP) inter-computer communication system was implemented and demonstrated under this program. The purpose of this implementation is to provide a feasibility demonstration and performance evaluation of authentication-based fault tolerant network communication between a number of mixed redundancy heterogeneous hosts (the Fault Tolerant Parallel Processor Cluster 3 (FTPP C3), the Fault Tolerant Parallel Processor Cluster 2 (FTPP C2), Sun, and 68030-based Unix workstation), operating systems (SunOS, LynxOS, Ada Run Time Systems), and programming languages (Ada, C). The open system attributes of the design were to be demonstrated by layering the AP design on an industry-standard network such as Ethernet and User Datagram Protocol (UDP). The resultant implementation was then to be demonstrated while the various hosts were executing a real-time distributed launch vehicle Advanced Guidance, Navigation, and Control processing demonstration which includes ASTER-generated code, and performance data were to be taken while executing this application. These objectives have been met.

The AP implementation uses the non-cross-strapped architecture and CRC / modular inverse-based authentication schemes described earlier. The protocol architecture is modular to allow any network to be accommodated by simply changing or adding network interface driver code to the bottom of the protocol stack. The protocol architecture is also designed to allow multiple, possibly disparate, networks to be interconnected as required by the application, with support for AP-based communication being provided to all hosts which require the level of fault tolerance and security provided by AP. The protocol architecture's modularity extends to allow the use of various authentication schemes ranging from those which provide data integrity in the presence of random-malicious faults to those which provide cryptographically sound data integrity and security in the presence of malicious faults.

Figure 21 shows the main components comprising the AP implementation and AGN&C demonstration. The configuration consists of five hosts: the quadruply redundant FTPP Cluster 3 (also known as the Army Fault Tolerant Architecture, or AFTA), the quadruply redundant FTPP Cluster 2 (an earlier version of the FTPP), a Sun SPARC 10 (hostname "Einstein"), a Sun SPARC 2 (hostname "Feynman"), and a Motorola MVME147 (68030)-based workstation (hostname "Dirac").

The FTPP C3 executes the AFTA XDAda Run Time System, the FTPP C2 executes the C2 XDAda Run Time System, the Sun workstations execute SunOS, and the MVME147 workstation executes LynxOS, a POSIX-compliant kernel.

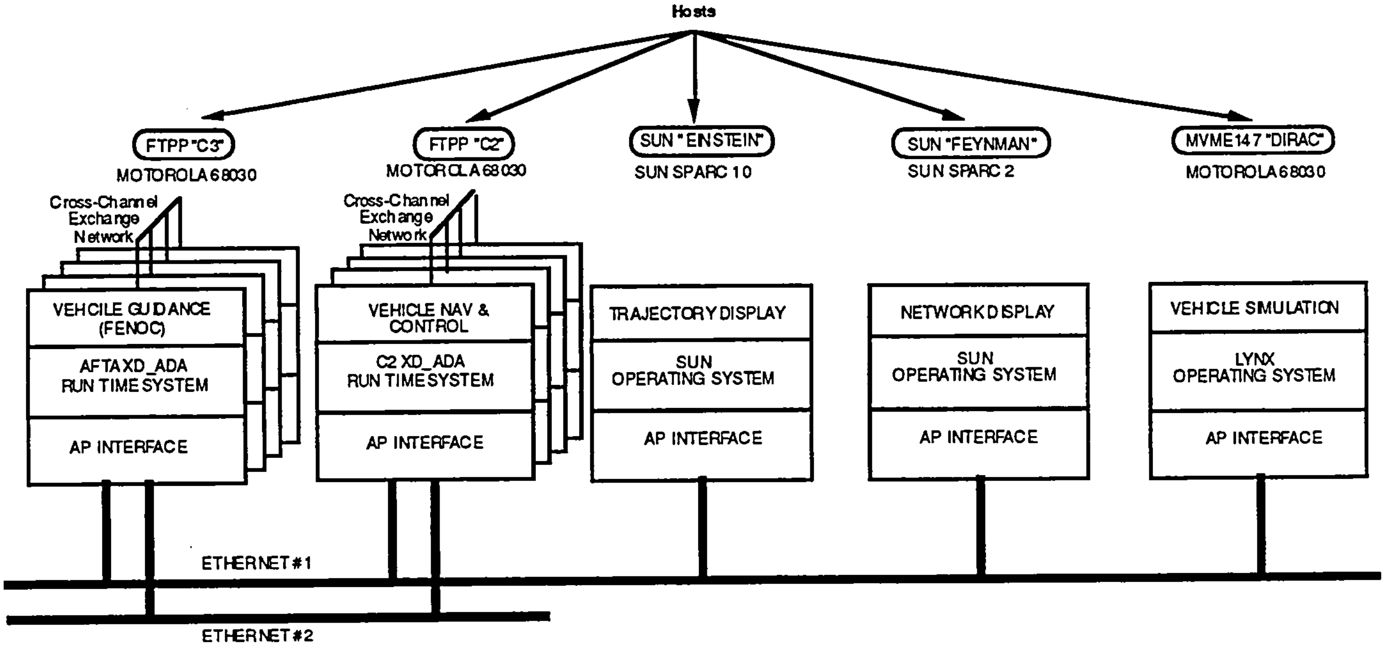


Figure 21. Architecture of AGN&C Demonstration

The Authentication Protocol interface software executes on each host. This software provides the application message-passing interface, signing, exchanging, routing, media layer interface, and authentication functions described below.

The redundant hosts possess internal cross-channel exchange networks which are implemented using FTPP Network Elements. The media layers are implemented using the Draper Laboratory-wide Ethernet contention network. Each host must possess at least one Ethernet connection to communicate over AP (Virtual Groups inside the FTPP C3 need not possess Ethernet connections to communicate over AP. In their case AP traffic would be routed through the Network Elements to the Ethernet Gateway Virtual Group.) The redundant hosts may possess two or more Ethernet connections comprising additional media layers. However, two are required for Byzantine resilient communication to occur.

The FTPP C3 executes Ada launch vehicle trajectory generation code which was automatically generated using the Draper ASTER software engineering tool. The FTPP C2 executes Ada launch vehicle control code which was also ASTER-generated. The Sun Einstein executes a C-based graphical interface which displays important application-related parameters such as trajectory, and allows a user to control the entire distributed system via a set of mouse-operated push buttons. The Sun Feynman executes a C-based graphical display of the fault status of the hosts and the AP system, including fault injection via mouse-downs on various parts of the display. Finally, the MVME147 executes the C-based real-time simulation of the launch vehicle dynamics. All communication between applications is through the AP's message-passing services.

The data communication patterns and frequencies for the AGN&C application are depicted in Figure 22. All messages are 256 bytes in length.

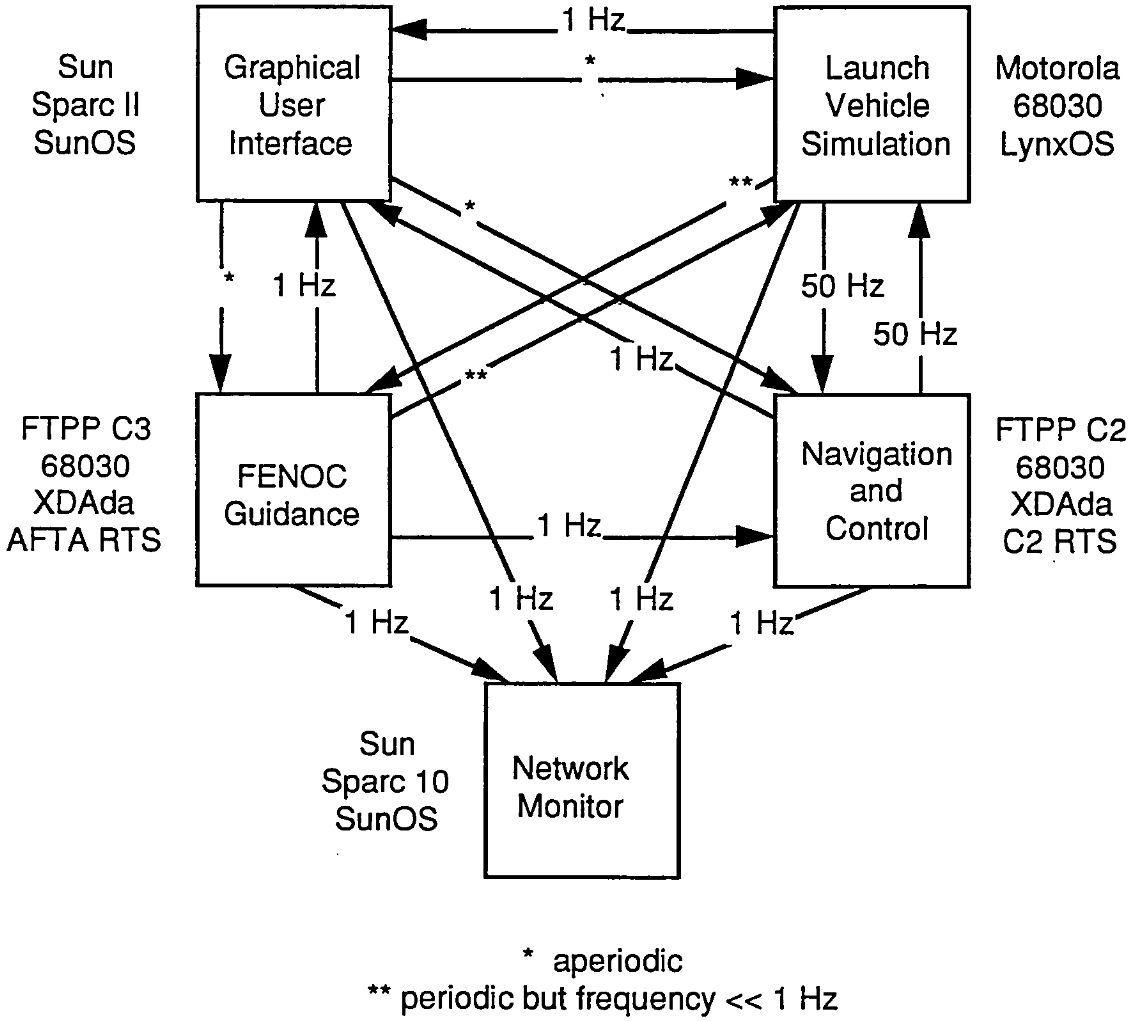


Figure 22. AGN&C Demonstration Configuration and Inter-Host Communication Requirements

3.2. Major Components of AP Processing Suite

The major components required to execute AP on a single host are shown in Figure 23. Two sets of tasks are required for a host to participate in AP communication: the application tasks and the AP Send-Recv task. The following sections describe those tasks and other interfaces, data structures, and algorithms used to implement AP.

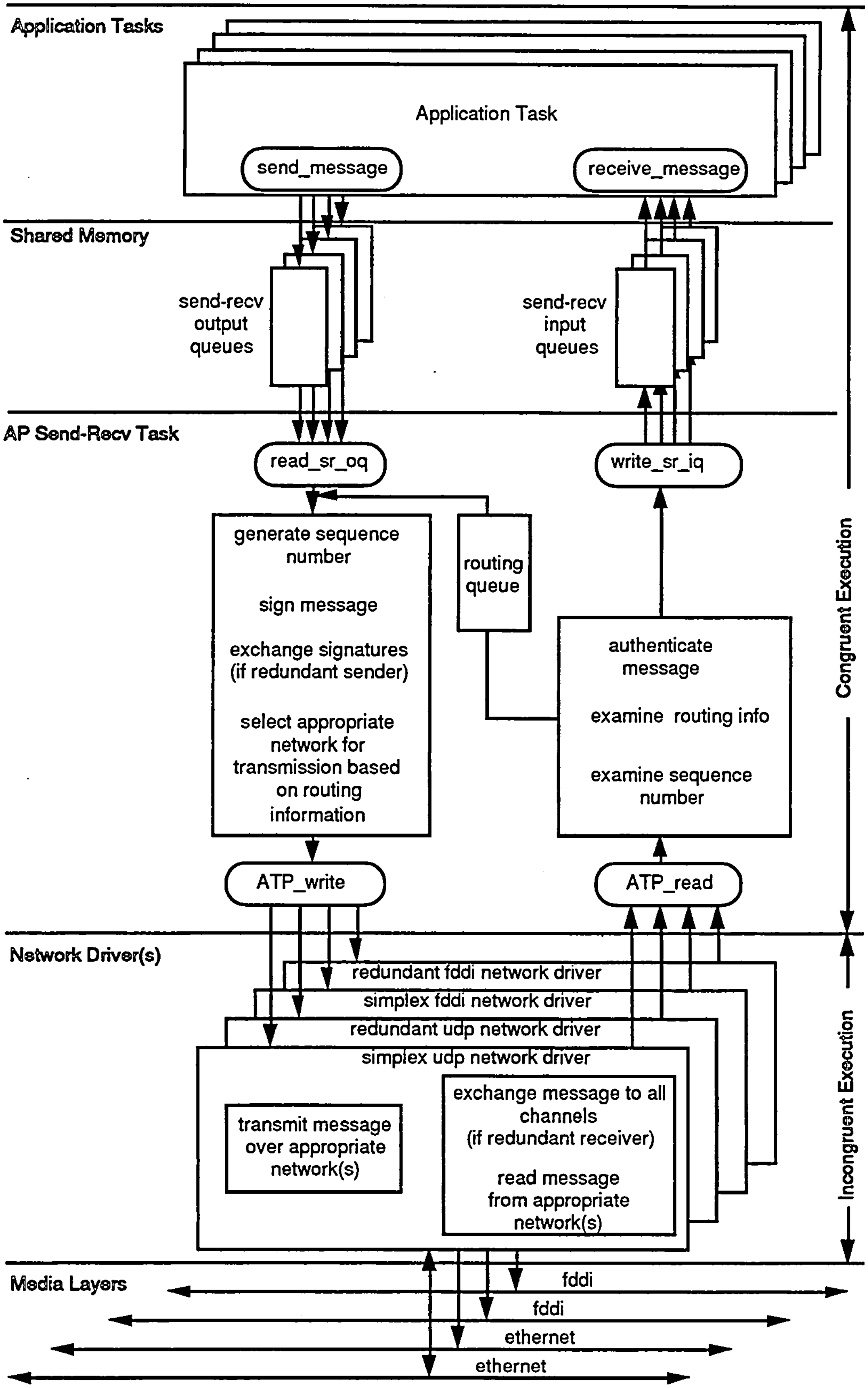


Figure 23. Authentication Protocols Software Architecture - Single Host

3.2.1. Application Task

An arbitrary number of application tasks may reside on any given host, subject to memory and performance limitations. Application tasks communicate with each other by sending and receiving messages using the calls defined in section 3.2.5.1.1. To set up for communication, a task must first establish a persistent *port.* During the port creation process input and/or output queues are created; the dimensions of these queues are specified in the operation of opening a port. An application task may create up to 256 ports, any of which may be either for input, output or bi-directional communication. Messages which are sent by the application are deposited in a Send-Recv Output queue, and messages which are destined for the task are read from a Send-Recv Input queue.

3.2.2. Authentication Protocols (AP) Send-Recv Task

The AP Send-Recv Task is responsible for the execution of the authentication-based communication protocol. Output messages emanating from application tasks are taken singly off a Send-Recv Output Queue chosen by the AP Send-Recv Task's Queue Selection Logic and copied into the Host Output Buffer. The task looks up the routing information necessary to send the message to the correct destination. An authentication trailer is generated containing a monotonically increasing sequence number and digital signatures (CRC-based in the current implementation). The routing information determines the Network Driver ID and the next node that the message needs to be sent to. Once this has been determined, the AP Send-Recv Task writes the message to the appropriate Network Driver and is ready to either process another outgoing or incoming message.

For input messages, the AP Send-Recv Task reads a messages from a specific Network Driver determined by the AP Send-Recv Task's internal Network Selection Logic. The sequence number is checked to see if the message is either a duplicate or out of order. If the message is neither, the AP Send-Recv Task examines the message routing information to determine whether the message should be delivered to the host or routed to another host. If the message is to be routed, it is placed into the routing queue for forwarding by the output side of the AP Send-Recv Task. If the message is destined for a task resident on the local host, the AP Send-Recv Task checks the message for authenticity. If the message is valid, AP Send-Recv Task copies it onto the correct Send-Recv Input Queue.

3.2.3. Network Driver

The Network Driver function resides at the bottom of the protocol stack and is responsible for relaying messages between the AP Send-Recv task and the media layer(s) to which the host is attached. A given host may be attached to several media layers, which may vary arbitrarily in redundancy level, type, and technology. Each host possesses a Network Driver for each media layer to which it is attached.

For outgoing messages, each Network Driver forwards the message from the AP Send-Recv Task to the media layer(s) controlled by that Network Driver. For incoming messages, each Network Driver determines whether a message has arrived on its relevant media layer(s) and forwards that message up to the AP Send-Recv Task.

Note that, in the case of a redundant host, all members may not possess interfaces to media layers. The Network Driver is responsible for hiding this from the AP Send-Recv task and all higher layers of the protocol, allowing them to perform their functions in bitwise exact match (also known as congruent) execution. It only transmits outgoing messages on media layers to which the host is attached. It also performs appropriate source congruency (interactive consistency) on incoming messages to ensure that all members of a redundant host have identical views of the incoming message stream, regardless of the redundancy level of the controlled media layer.

3.2.4. Network Monitor

The Network Monitor (Netmon) program provides a convenient graphical display of the activity within the authentication protocols network. The Netmon program has two windows: the virtual window and the physical window. The virtual window shows the distributed virtual computation sites, the applications mapped to each site, and the virtual AP network topology. The physical window shows the individual elements that make up the virtual configuration.

The virtual window display is shown in Figure 24. The current configuration includes 5 systems: Einstein, a simplex Sun workstation; Feynman, another simplex Sun workstation; C2, a quadruplex FTPP; C3, a quintuplex FTPP; and Dirac, a simplex LynxOS system. Each system is shown as a virtual simplex system since this is the application programmer's model, even for the systems that are redundant. The redundant systems are shown with shadow processors to indicate their redundancy; however, the application code on each redundant processor is identical.

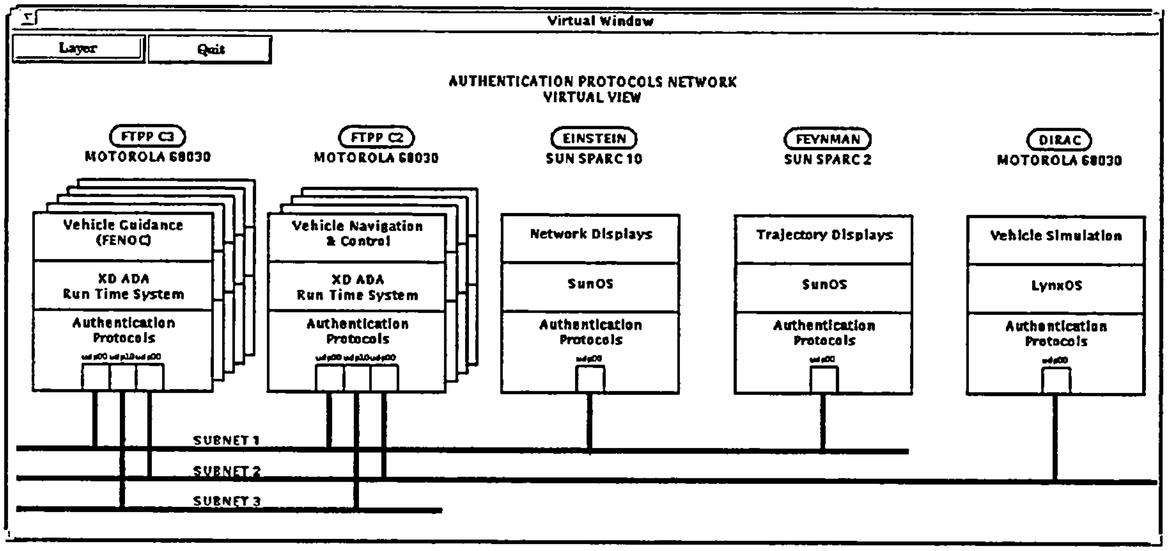


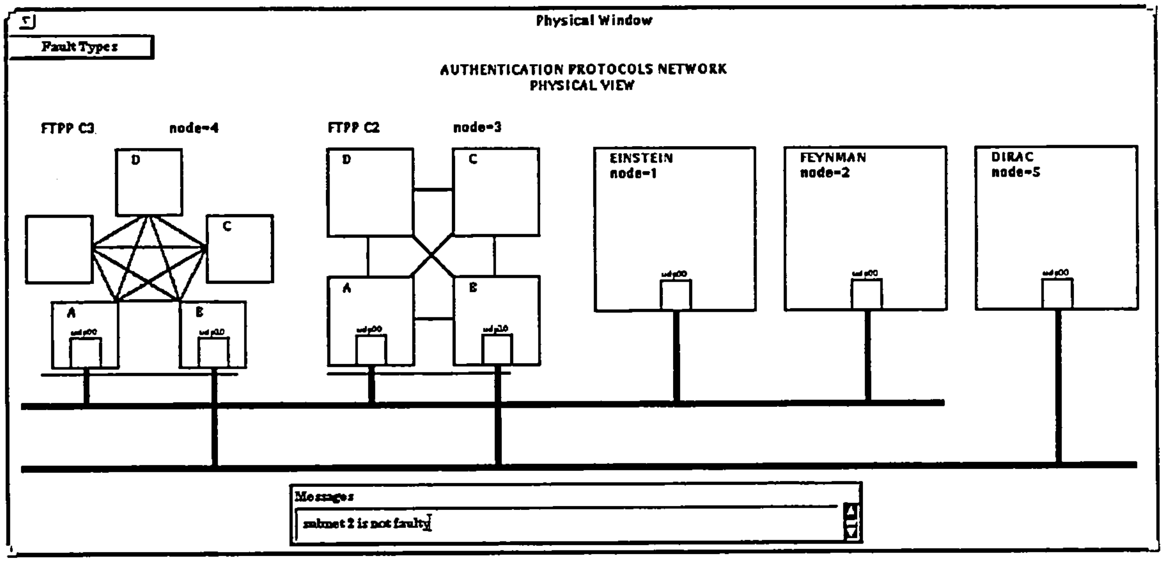
Figure 24. Network Monitor Virtual Window

Each of the virtual sites in the virtual configuration are connected together by one or more AP subnets. An AP subnet is a virtual communication channel and may be simplex or redundant. Redundant subnets are composed of multiple simplex subnets . The current virtual configuration includes two simplex subnets and one virtual subnet. Only redundant processing sites can be connected to a redundant subnet.

The virtual display provides two menu options. The first menu option allows the user to selectively display the mapping of application and operating system software onto the virtual configuration. Selecting "Node Name" causes only the name of each system to be displayed. Selecting "Processor" adds the type of CPU to each system. The "Application" options displays the application software executing on each platform. Selecting "Operating System" shows the type of operating system in use on each site. Finally, "Network" shows the network layer protocols used by each site to communicate with the other sites. For all systems in the current virtual configuration, the network layer is the AP network. The only difference between systems at the network layer is whether each system has simplex or redundant interfaces to the network.

The second menu option allows the user to terminate Netmon. Selecting "Exit" causes the program to display a confirmation box. If the exit is confirmed, Netmon will terminate. If the exit is not confirmed, the program will return to normal operation.

The physical window display is shown in Figure 25. Each virtual site from the virtual window is represented in the physical window in its true configuration. Simplex sites are shown nearly identical to their virtual display. The FTPP sites, however, are displayed as redundant processors interconnected by internal communication links. The current physical configuration has two redundant FTPPs, C2 and C3. The network is also shown in its true physical configuration. There are two simplex media layers in the physical network topology. The simplex sites are only connected to one of the simplex layers. The redundant sites are connected to both simplex layers. However, a different channel is used to connect to each layer, thus preventing a single FTPP channel from disrupting both simplex layers. For redundant communication, the redundant processing sites can transmit on both media layers simultaneously. This mode of operation is implemented through the virtual redundant subnet shown in the virtual configuration.



subnet 2 is not faulty

Figure 25. Network Monitor Physical Window

The physical window shows the current fault status of each channel, device, or subnet in the system. It also allows the user to inject software level faults into the AP network software to simulate real faults in the communication system. These simulated faults are limited to the network software and do not affect the applications or operating system running on each machine.

To inject a fault, the user first selects the type of fault to be injected from the menu item. Fault types include "Babble", which causes the element to emit random data at random intervals; "Silent", which causes the element to cease operation altogether; "Corrupt Message", which causes part of the message (possibly including the header, data, sequence number, and/or signature) to be modified in a random fashion; and "Repeat Message", which causes a complete message, including sequence number and signatures, to be repeated at a random point in time. The user may also select "Fix Fault", which causes a previously failed element to be "repaired." After selecting a fault to inject or repair, the user clicks on an element, which could be a processor channel, a network interface device, or a network subnet. When the element is clicked on, a message is sent to that element to cause it to change its software fault state. The element will send a message back to Netmon to report its new fault state. Netmon will then redraw the element in red if the element is failed, or in its natural color if it has been repaired.

The physical window also contains a message box for displaying messages that arrive at Netmon. These messages include the status messages mentioned above and fault detection messages. For example, if a system detects s faulty signature, a message is sent to Netmon and displayed at the bottom of the physical window. A more sophisticated version of Netmon could use the information contained in the message to diagnose and reconfigure around faults in the system.

3.2.5. Important Data Structures, Interfaces, and Algorithms

3.2.5.1. Send-Recv Input/Output Queues

The Send-Recv input/output queues represent the communication ports through which application tasks communicate over AP. Application tasks create queues, and send and receive messages via the queues.

The Send-Recv input and output queues are the interface buffers between the application tasks and the AP Send-Recv task. Multiple application tasks can deposit or retrieve messages from a queue. However, a single AP Send-Recv task on each computer system processes messages from and to these queues. The queues serve to decouple the application tasks from the authentication protocol send-receive task. Application tasks perform "send message" operations which save messages in their output send-recv queue. The authentication protocol send-receive task consumes messages from all application tasks' output queues, signs them and deposits them in the network output queue. Incoming messages are authenticated and stored into the appropriate destination input send-recv queues which are consumed by each destination application task by the "receive message" operation.

A queue is associated with a task by its port number. This port number is incorporated into the source and destination addresses. Therefore, a task which sends a message to another task must be cognizant of the destination's task port address.

Each application task must establish at least 1 queue for communication over AP. Each task may create a maximum of 256 message queues for either the input or output of messages. The implementation of multiple queues enables a task to maintain multiple ports through which it may send or receive messages. This queue architecture enables a task to sort incoming messages by publishing a specific port for reception of certain types of messages. In addition, if the appropriate scheduling mechanisms are implemented, these queue may be perceived as priority queues.

The send-recv input and output queues consist of data structures for the maintenance of global host information for accessing task queues, of data structures for the maintenance of queue information and of buffer space for the messages themselves. Additionally, the send-recv input and output queues include the access operations which decompose the buffer management of these queues into numerous functions. There are well defined interfaces to the global host information as well as to the queue methods which manipulate the queue data. There are essentially 2 types of tasks which interface to the send-recv input/output queues--the application tasks and the authentication protocol Send-Recv task. The application tasks interact with the queues by means of 5 interface calls -- open_port, configure_port, close_port, send_message and receive_message. The AP Send-Recv task interfaces to the send-recv input/output queues by invoking write_sr_oq and read_sr_iq, by initializing the global data structures for the host computer via the bs_create_sm_buffer, qm_create_port_table, qm_create_lock calls and by searching the port table directly by accessing qm_access_port_table. These data structures as well as the operations accessing these structures are shown hierarchically by the following diagram. The shadowed areas represent the data structures and the arrows represent routine invocations.

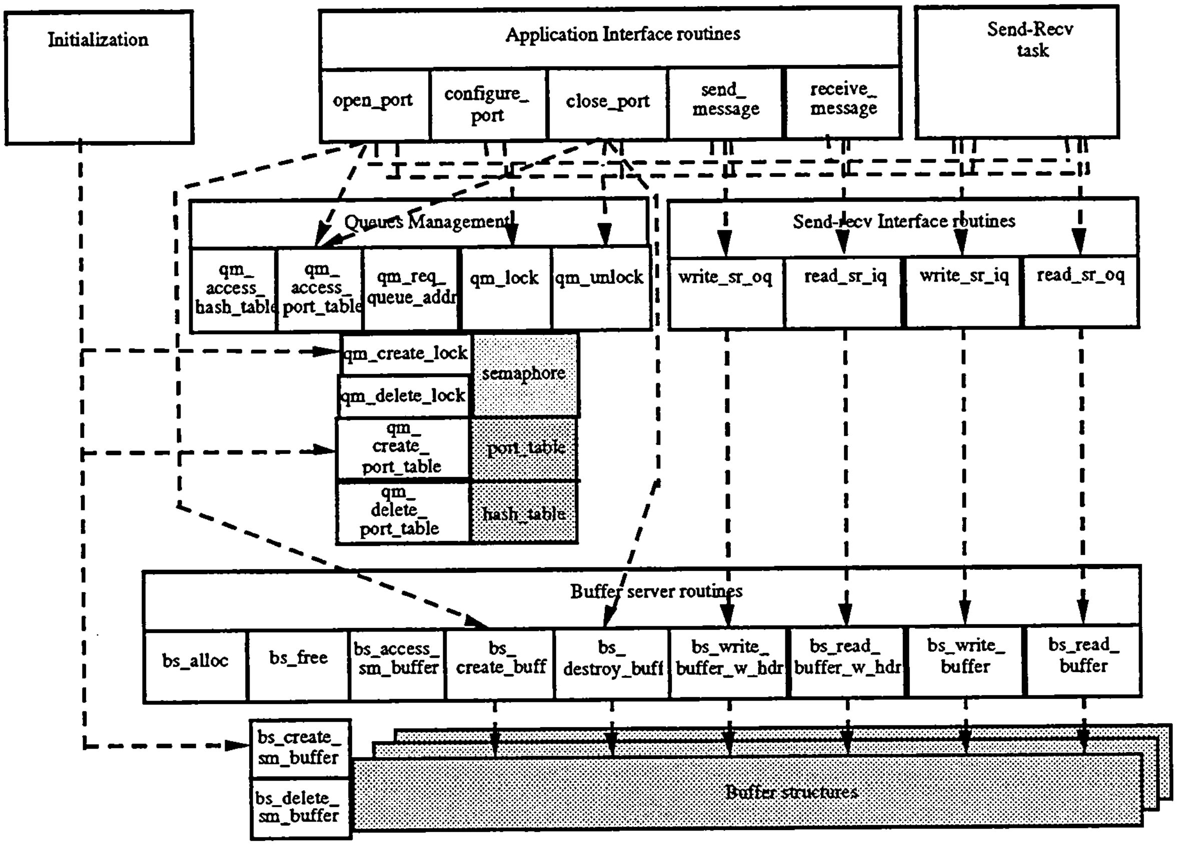


Figure 26. Queue Hierarchy Diagram

3.2.5.1.1. Application Task Interface with Queues

Application tasks manipulate the input and output queues by means of the application interface calls. These procedures enable the user to create a communication port for input or output and with certain size dimensions, to read or write messages via a certain port or to modify the port configuration. The following procedures comprise the application interface:

1) open_port creates a port and queue
2) close_port closes a port
3) configure_port reconfigures a port's parameters
4) send_message sends a message through an AP port
5) receive_message reads a message from an AP port

Because these procedures comprise a major interface in the protocol stack, the procedures, parameters, and usage are more fully described in Appendix A, "Man Pages."

3.2.5.1.1.1. open_port

The open_port call creates a port for communication using authentication protocol (AP). A task uses this call to initialize a queue and buffer pool pair corresponding to a certain port number. These port numbers are made up of a task ID in the upper byte and a queue ID in the lower byte. A task is only allowed to open ports with either its own task ID for the upper byte or certain protocol-specified ports with zero as the upper byte. The caller configures the port for either unidirectional or bi-directional message passing.

The open_port routine updates the port table to add this port request. The open_port routine locks and unlocks the port table to ensure that the port table is not accessed during the open port call. The open_port modifies the port table by adding entries to it. It defines a port for input, output or both input and output.

The open_port facility also enables a previously opened input (output) port to be subsequently reopened for the complementary output (input) operation. If the port has already been opened for both output (input) then an error message is returned.

The underlying buffer pool is dynamically allocated to provide the requested application message passing buffers. The size of the buffer space is estimated using both the in_queue and in_bufpool_size parameters (or alternatively, out_queue and out_bufpool_size). The in_queue parameter is used as an estimation of the aggregate number of messages expected within a frame and the in_bufpool_size is used as an estimation of the total number of bytes of these messages. The total estimate is inflated by 25% and is longword aligned. bs_create_buff actually allocated memory returning the address of where the buffer information resides. This address is saved in the port table.

3.2.5.1.1.2. close_port

The close_port call closes a port being used for communication using authentication protocol (AP). A task uses this call to end communications using this port. All structures associated with the port (queues and buffer pools) are deallocated and all messages remaining in the queues are deleted.

The close_port facility updates the port table to delete the indicated port. close_port deletes the port for both input and output. Additionally, the underlying buffer pools are dynamically deallocated and the port table updated to delete the entry. The close_port procedure must lock and unlock the port table to prevent simultaneous modifications to the port table by different procedures.

3.2.5.1.1.3. configure_port

The configure_port call is used for control and status operations on open ports. The configure_port procedures updates the port table to modify the selected port's configuration. Configure_port interprets the command and takes appropriate action. Because it updates the port table information the configure_port facility must lock and unlock this table.

3.2.5.1.1.4. send_message

The send_message procedure copies the requested message to the designated Send-Recv Output Queue via a call to the write_sr_oq facility. It returns immediately after the copy. To maintain consistency the source network address in the header information is updated from the port number. The send_message facility must perform a lock prior to invoking write_sr_oq and to perform an unlock afterward.

3.2.5.1.1.5. receive_message

The receive_message call retrieves the oldest message from the caller-specified Send-Recv Input Queue and places it into a buffer provided by the application programmer. It strips the network header (size, message type, destination, source) and authentication trailer (sequence number, digital signature) from the message and places them into appropriate data structures.

3.2.5.1.2. AP Send-Recv Task Interface with Queues

The AP send-recv task is decoupled from the application task by the Send-recv queue structures. The AP send-recv task scans the output queues for the defined ports and extracts outgoing messages. Likewise, the AP send-recv task deposits authenticated messages in the destination input queue. These 2 functions are facilitated by the invocation of the read_sr_oq and write_sr_iq. These comprise 2 of the functions defined in the Send-Recv interface. The other 2 functions (read_sr_iq and write_sr_oq) are invoked by the application task's interface routines (receive_message and send_message).

The send-recv interface calls are responsible for the interface between the AP send-recv task and the buffer server or between the application interface and the buffer server. It determines which queue should be accessed given the port number and the specific operation requested. It interfaces with the queue manager to locate the buffer and requests the appropriate operations of the buffer server.

The 4 send-recv interface calls:

1) to write a message to the output queue (interface with the application task) - write_sr_oq,
2) to read a message from the input queue (interface with the application task) - read_sr_iq,
3) to read a message from the output queue (interface with the send-recv task) - read_sr_oq,
4) to write a message to the input queue (interface with the send-recv task) - write_sr_iq.

3.2.5.1.2.1. write_sr_oq

The write_sr_oq procedure is invoked by the application task's send_message call. It requests the address of the output queue for the specified portnum from the queue manager and invokes the bs_write_buffer_w_hdr routine. The msg_size parameter is assumed to represent the size of the message (exclusive of the header).

3.2.5.1.2.2. read_sr_iq

The read_sr_iq procedure is invoked by application task's read_message call. It requests the address of the input queue for the specified portnum from the queue manager and invokes the bs_read_buffer_w_hdr to copy header, message and trailer into application buffers from the specified portnum queue.

3.2.5.1.2.3. read_sr_oq

The read_sr_oq routine is invoked by the AP Send-Recv Task. It requests the address of the output queue for the specified portnum from the queue manager and invokes the bs_read_buffer routine to copy one message out of a specific Send-Recv Output Queue and increment the Send-Recv Output Queue's pointers.

3.2.5.1.2.4. write_sr_iq

The write_sr_iq routine is invoked by the AP Send-Recv Task. It requests the address of the input queue for the specified portnum from the queue manager and invokes the bs_write_buffer routine to copy one message into a specific Send-Recv Input Queue and increment the Send-Recv Input Queue's pointers.

3.2.5.1.3. Buffer Server

The buffer server is responsible for managing the storage of message and header data. The buffer server manipulates a buffer pool which is preallocated by the application task via an invocation to a buffer server function. Because these interfaces are internal to the AP protocol stack, they are not documented in Appendix A, "Man Pages."

The buffer server provides the following facilities:

1) to create a shared memory segment from which buffer pools are allocated

```
int bs_create_sm_buffer();
```

2) to delete the shared memory segment

```
int bs_delete_sm_buffer();
```

3) to retrieve the address of the shared memory segment

```
int bs_access_sm_buffer();
```

4) to allocated and free memory blocks from the shared memory segment

```
char *bs_alloc(int size);
        int bs_free(char *);
```

5) to create a buffer pool

```
int bs_create_buff(int bufpool_size,
                        buff_struct **buff_addr);
```

6) to destroy a buffer pool

```
int bs_destroy_buff(buff_struct *buff_addr);
```

7) to write a header and its associated message

```
int bs_write_buffer_w_hdr(buff_struct *buff_addr,
                        char *message,
                        int msg_size,
                        header_struct *header);
```

8) to read a header and its associated message

```
int bs_read_buffer_w_hdr(buff_struct *buff_addr,
                                char *message,
                                int max_msg_size,
                                header_struct *header,
                                trailer_struct *trailer);
```

9) to write a message

```
int bs_write_buffer(buff_struct *buff_addr,
                                char *message,
                                int msg_size);
```

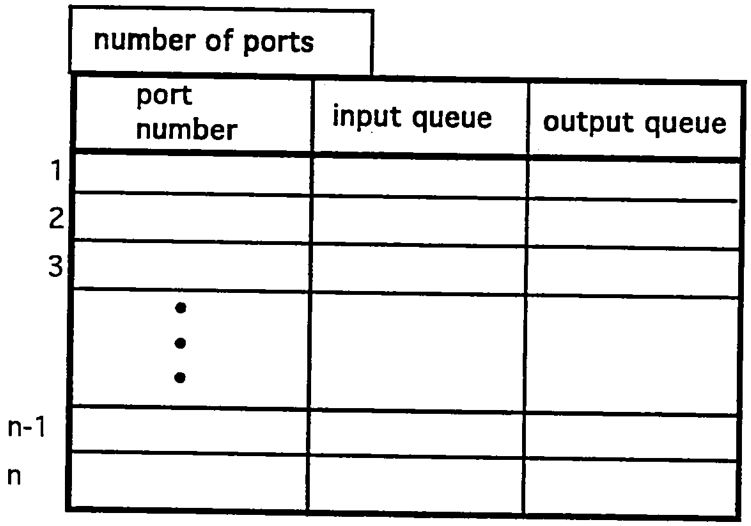
10) to read a message

```
int bs_read_buffer(buff_struct *buff_addr,
                                        char *message,
                                        int max_msg_size);
```

3.2.5.1.4. Queue Management

A management facility must maintain information regarding these queues on a single host. Specifically, a facility must maintain the number and locations of queues, whether a queue is input or output and which port is associated with each queue.

The port table defines the mappings of port to the input and/or output queues. Furthermore, the port table provides accessibility to all send-recv input/output queues within a host computer system. For each port on the host computer it defines the locations of input and output queues. A num_of_ports variable defines the number of assigned ports; it represents the number of legitimate entries in the port table. The format of the port table is shown in Figure 27.



number of ports

	port number	input queue	output queue
1			
2			
3			
	• • •		
n-1			
n			

Figure 27. Port_table

Additionally, the queue management is responsible for the creation and deletion of the port table, a hash table, semaphores, and a shared buffer. Because the send-recv queues are shared between an application task and a send-recv task, a host computer-wide data space must be created and access primitives provided.

The open_port application interface call interfaces directly with the port table in the creation of a queue. The application task specifies the buffer pool size, and whether the queue is designated input, output or both. The close_port application interface call similarly operates on the port table.

In addition, the queue management function must respond to requests to translate (portnum, input/output flag) to a specific buffer pool address. As part of this request it must ensure that a buffer pool for the portnum exists and that it is designated for input or output as requested by the invocation. The encapsulation of an input/output flag within the global queue management will enable the use of the same network address for both input and output.

The queue management function also provides the mechanisms for creation and deletion of a semaphore and for locking and unlocking that semaphore.

The following routines are defined in the queue management:

1) to create a shared memory segment for the port table and task hash table.

```
int qm_create_port_table();
```

2) to delete the shared memory segment for the port table and task hash table

```
int qm_delete_port_table();
```

3) to enable a process to access the port table in shared memory

```
int qm_access_port_table(port_table_struct **port_table_ptr);
```

4) to enable a process to access the hash_table in shared memory

```
int qm_access_hash_table(int **hash_table_ptr);
```

5) to retrieve the queue address associated with the specified portnum

```
int qm_req_queue_addr(short portnum,
 boolean in_out_flag,
 buff_struct **queue_ptr);
```

6) to create and to delete a semaphore

```
int qm_create_lock();
int qm_delete_lock();
```

7) to perform P and V operations on the semaphore

```
int qm_lock();
int qm_unlock();
```

Because these interfaces are internal to the AP protocol stack, they are not documented in Appendix A, “Man Pages.”

3.2.5.1.5. Buffer Queue

The port table guides the location of the buffer queue for the storage of messages. A buffer queue consists of a buffer header and a contiguous memory area where messages are stored for either input or output. A single buffer queue (also referred to as buffer pool) is associated with each queue. An example is shown below:

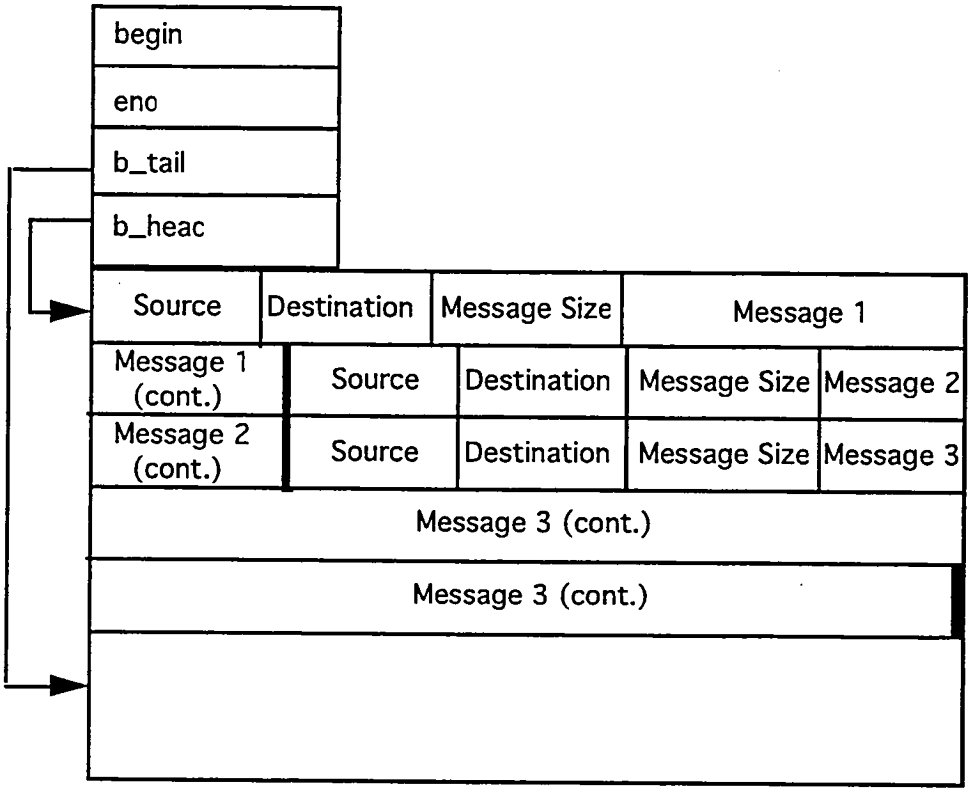


Figure 28. Structure of Buffer Queue

The fields in this buffer area are as follows:

begin: First address of the buffer area (a constant).
end: Address of the 1st byte beyond buffer area (a constant).
b_tail : Address of the next available location where a message can be added.
b_head: Address of the oldest message in the buffer.

When a port is opened by the application task, the buffer pool is allocated according to the dimensions derived from the queue_size and buffpool_size parameters on the open port invocation. The buffer pool is maintained by the buffer server as a first-in-first-out circular queue. When a message is sent via a port, the buffer server copies a contiguous message from the application task address space and stores it within the buffer pool. If necessary, the message will be segmented by the bottom of the buffer pool; the remainder will be saved at the beginning of the buffer pool. Retrieved messages will be reconstructed into a contiguous message.

A message can be saved in the queue with or without its header; this depends upon the buffer server invocation. When the header is saved, the fixed length header consisting of a source address, destination address and a message size is prepended to the message as it is stored in the buffer area. For efficiency reasons, the headers and messages will always be longword aligned. The messages themselves, however, are not required to be an integral multiple of a longword in size.

Because of issues of sharing the buffer pool between 2 tasks, the buffer pools were allocated from a shared memory area which is controlled by the queue management function. Using a shared memory methodology the queues are addressable by either the application task or the AP Send-Recv task. While the accessibility issue is not a problem for an embedded computer system such as the C2 or C3 FTPP OS, it does pose a significant problem for the UNIX operating system which isolates tasks from each other. For this reason, buffer server primitives have been established for accessing the shared memory buffer space from which the buffer queues are allocated. These routines are bs_create_sm_buffer, bs_delete_sm_buffer and bs_access_sm_buffer.

The port table and a buffer pool are related as shown in the following diagram.

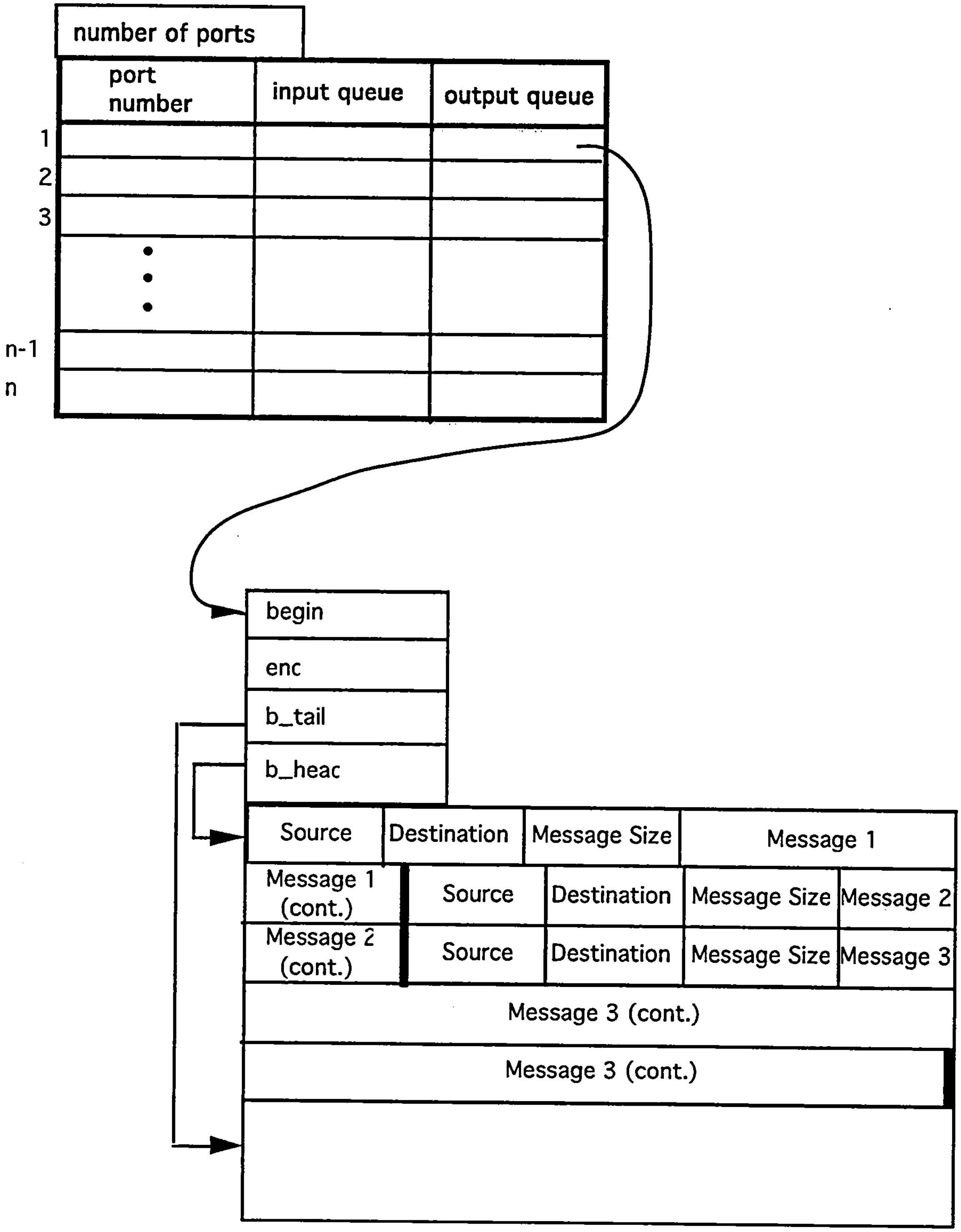


Figure 29. Port_table and buffer pool interaction

3.2.5.2. AP Send-Recv Task Interface to Network Driver

The protocol stack on a given host contains one Network Driver per subnet connected to that host.

The AP Send-Recv Task interface calls to the Network Driver are outlined below. Because this is a major interface in the protocol stack, these calls, parameters, and usage are more fully described in Appendix A, "Man Pages."

3.2.5.2.1. ATP_open

The ATP_open call initializes the data structures and hardware necessary to provide communications over a specific device.

3.2.5.2.2. ATP_close

ATP_close call discontinues communications using the specified device.

3.2.5.2.3. ATP_read

ATP_read call checks a specific device and returns a message if any present.

3.2.5.2.4. ATP_write

ATP_write call sends a message out through the specified device.

3.2.5.2.5. ATP_ioctl

ATP_ioctl call provides various control and status operations on an open device.

3.2.6. Routing and Authentication

Routing is performed in both the AP Send-Recv task and the Network Driver, while authentication is performed solely within the AP Send-Recv task. Routing in the AP Send-Recv task is responsible for a lookup based on source host and destination subnet that determines whether there is a direct connection between the source and destination host or whether a gateway is needed and, in either case, which Network Driver to use to transmit the message. If an authentication trailer is needed, the task appends it to the end of the message. The send_message_network interface call is then used to pass the authenticated message, the required Device ID, and the host number of the next machine to receive the message to the correct Network Driver. The Network Driver looks up in the ARP table the needed physical information on the destination and sends one message for each media layer it is responsible for.

3.2.6.1. Network Addressing

All application tasks which use AP to communicate must use a network address to designate the source and destination of any given message. Network addresses uniquely identify a subnet, host, and port. The network address consists of three 16-bit quantities: the subnet ID, the host, and the port.

```
typedef          unsigned short       network_address[3];
network_address[0] --> Network ID (mode and subnet)
```

bits 0..13:	Subnet ID		
bits 14..15:	Mode =>	00	Point to Point
		01	Multicast (Not implemented)
		11	Broadcast (Not implemented)

network_address[1] --> Host/VID ID

network_address[2] --> Port number, which may be further broken down into Task ID and Queue ID (this provides capability for application-defined prioritized queues)

bits 0..7:	Queue ID
bits 8..15:	Task ID

3.2.6.2. Routing

The routing function of the AP uses the following 2 tables:

1) A routing table is indexed by host and scanned by the routing function until the correct subnet ID is found, containing connection information (direct vs. gateway), gateway ID's (if necessary), and interface drivers (IPR0, IP11, etc.).

2) An Address Resolution Protocol (ARP) table is used, whose format is dependent on the interface driver supported by the overall network. The ARP table contains the addressing information of all machines in the network. For UDP drivers, this includes IP address of host and port and socket numbers.

To illustrate routing, the following figure shows three hosts connected over six virtual subnets. The subnets are overlaid over four media layers: two media layers between Host 1 and Host 2, and two media layers between host 2 and Host 3. Two subnets (1 and 4) are dual-redundant, and four subnets (2, 3, 5, and 6) are nonredundant.

The subnet identification includes its redundancy. In this example, the two media layers between Hosts 1 and 2 can be used to construct two nonredundant subnets as well as a duplex redundant subnet, both of which can be simultaneously in use.

Network Driver IPR0 (Internet Protocol, Redundant, Subnet 1) is used by Host 1 and Host 2 to reliably communicate over redundant subnet 1, while Network Driver IPR1 (Internet Procotol, Redundant, Subnet 4) is used by Host 2 and Host 3 to reliably communicate over redundant subnet 4 . Host 1 and Host 2 may also unreliably communicate over nonredundant subnets 2 and 3 using Network Driver IP00 and IP01, respectively. Host 2 and Host 3 may unreliably communicate over nonredundant subnets 5 and 6 using Network Driver IP10 and IP11, respectively.

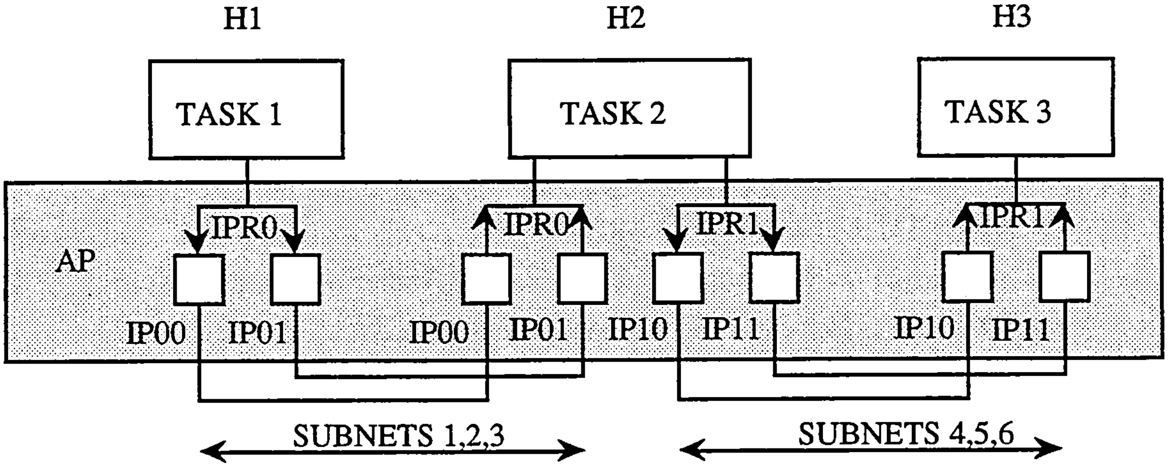


Figure 30. Routing Example

3.2.6.2.1. Routing Table

If Host 1 wishes to reliably send a message to Host 3, that message must be routed through Host 2, which serves as a gateway between subnets 1, 2, and 3 and subnets 4, 5, and 6. To determine where to send a given message, each host uses its routing table. Given a destination subnet (derived from the subnet field of the destination host's network address), the routing table informs the source host whether that subnet is directly connected to the source host (D) or whether the message must be sent through an intermediate Gateway (G) host. In either case, the routing table informs the source host which Network Driver ID should be used to send the message.

The routing table is organized as a list indexed by source host and destination subnet. Each source host has an entry, and each entry is of the following structure:

```
struct Routing_Table_Entry
{
unsigned long                  Subnet_ID;
unsigned long                  connection;
unsigned long                  GW_id;
unsigned long                  Device_id;
}
struct Routing_Table_Entry     Routing_Table[NUM_HOSTS]
```

For example, the routing table for H1 in the configuration above would be as follows:

Destination Subnet ID	Type of Connection	Gateway ID	Device ID
1	Direct	none	IPR0
2	Direct	none	IP00
3	Direct	none	IP01
4	Gateway	H2	IPR0
5	Gateway	H2	IPR0
6	Gateway	H2	IPR0

Table 3. Routing Table Example

3.2.6.2.2. ARP Table

The ARP table contains the physical addressing information for all of the hosts in the network. The Network Driver will only look at the fields necessary for communications. Currently, the table only supports Ethernet connections (both single and virtual dual). The table entry is defined as:

```
struct ARP_Table_Entry
{
unsigned long                    host_address;
unsigned long                    send_IP0;
unsigned long                    send_IP1;
unsigned long                    read_IP0;
unsigned long                    read_IP1;
unsigned long                    send_port0;
unsigned long                    send_port1;
unsigned long                    read_port0;
unsigned long                    read_port1; }
struct ARP_Table_Entry           ARP_Table[NUM_HOST];
```

3.2.6.2.3. Routing Header

The AP Send-Recv task uses the information in the routing header to determine the Network Driver and local entities to which the message should be delivered. The routing header is generated within the send_message interface call before the message is placed in a Send-Recv Output Queue. The information in the routing header remains constant for the lifetime of the message.

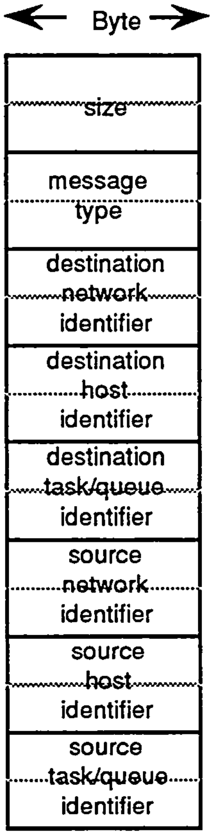

Byte
size
message type
destination network identifier
destination host identifier
destination task/queue identifier
source network identifier
source host identifier
source task/queue identifier

Figure 31. Routing Header

The size specifies the size of the user message, in bytes, which immediately follows the routing header.

The message type field is used by the AP Send-Rec task to determine which of multiple transport layer protocols is to receive the message. The most significant bit indicates whether or not the authentication trailer is present. If the authentication trailer is present, the MSB is set, otherwise the MSB is cleared. This field may also be used for FDIR to force each node to append an authentication trailer.

The destination network identifier specifies the subnet to which the message is to be delivered. The AP Send-Recv task uses this entry each time a new message enters the routing layer to determine what to do with the message.

The destination host identifier indicates the host on the specified subnet to which the message is to be delivered. The AP Send-Recv task uses this entry to determine if it is an intermediate node or if the message has reached its final destination.

The destination task/queue identifier indicates which application and queue the message is intended for. The AP Send-Recv task uses this entry to determine which specific Send-Recv Input queue to put the message in.

The source network identifier specifies the subnet from which the message originated.

The source host and task/queue identifier, in conjunction with the subnet ID, indicates the entity from which the message originated. This field never indicates a multicast group.

3.2.6.3. Authentication

3.2.6.3.1. Authentication Trailer

If a message is to be transmitted over a subnet requiring authentication, an authentication trailer must be attached to the message. If an incoming message already has an authentication trailer attached, that trailer is used. Otherwise, the AP Send-Recv task requests the authentication protocol to generate one.

When the message has reached its final destination, the AP Send-Recv task requests the authentication protocol to test the authenticity of a message based on the contents of the attached authentication trailer.

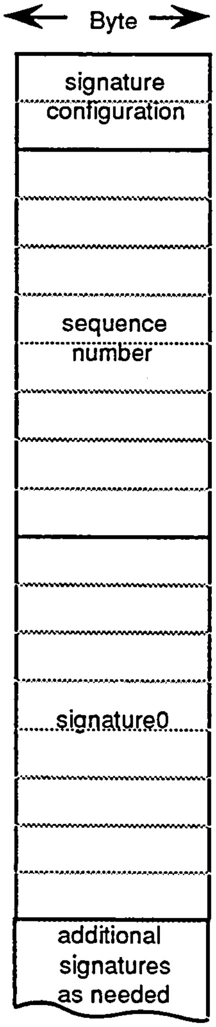


Figure 32. Authentication Trailer

The signature configuration indicates the number and identity of the signing members of the authentication trailer. A message may be signed by one to four individual members, with each using a different signature key. The sigconfig field is divided into four fields of 4 bits each. Each four-bit field indicates the key that was used to generate each corresponding signature. A field of all 0s indicates no signature. If a field indicates no signature for a given signature N, no signatures beyond N-1 may be included in the trailer, and any configurations specified in the sigconfig field will be ignored. Thus, if the sigconfig field is 1010 0101 0000 1011, only two signatures would be expected in the authentication trailer.

15 12 8 4 0

sig0 key	sig1 key	sig2 key	sig3 key

Figure 33. Signature Configuration Field

The netid and netext fields in the authentication trailer specify the agent which signed the message. The authenticating agent may or may not be the source of the message specified in the routing header, thus the need for a separate specification. The form of the authentication agent is the same as for the source address. The authentication agent address is used to determine which set of authentication keys and which sequence number to use in checking the authenticity of a message.

The sequence number fields and signature fields are defined by the authentication protocol. Signature fields cover the entire user message, the routing header, and all of the authentication trailer except for the signatures.

3.2.6.3.2. Keys

Verification is carried out using pairs of keys: host-specific private keys to sign the message and globally-known public keys to authenticate the message. Each host capable of sending and/or receiving Authentication Protocol messages is assigned one of these pairs when it is added to the network. If a redundant host is responsible for a message (i.e., C3's VIDs), each member will sign the message.

3.2.6.3.3. Private Keys

Private keys are stored in local memory on each individual host. Each processor has a unique private key. A triplex virtual group containing three processors would therefore possess three private keys, one for each member. No member of a virtual group possesses the private key of another member.

3.2.6.3.4. Public Keys

Public keys are stored in a global table which must be updated any time the network is reconfigured (i.e., reconfiguration of VID groups). The table has an entry for each host which contains all of the keys needed to authenticate a message. For example, in the table below, C3, VID 1 is configured as a triplex and the entry must provide three public keys, one for each PE in the unit. The keys (denoted # in the table) are placed into the table contiguously, with the remaining fields containing zeros.

Host	Public Key 1	Public Key 2	Public Key 3	Public Key 4
H1	#	0	0	0
H2	#	0	0	0
C3, VID 1	#	#	#	0
C3, VID 2	#	#	#	#
Einstein	#	0	0	0

Table 4. Public Key Table

3.2.6.3.5. Key Pair Generation

As mentioned earlier, a public-key authentication scheme used for the current protocol implementation is based on modular inverses. This scheme uses two numbers P and P^{-1} (P inverse) for which $P \bullet P^{-1} \bmod N = 1$, where N is a very large number (in our case 2^{64}). P is the private signature key and P^{-1} is the public authentication key. At system startup or compile time the key pairs P and P^{-1} must be generated for each authenticating node in the system. The key pairs are generated using an extended version of Euclid's algorithm (see page 302 in Knuth's The Art of Computer Programming, Volume 2 / Seminumerical Algorithms). To use this algorithm, we note that if

$$P \bullet P^{-1} \bmod N = 1 \tag{1}$$

then

$$P \bullet P^{-1} = N \bullet v + 1 \tag{2}$$

$$(\text{or } P \bullet P^{-1} + N \bullet v = 1) \tag{2'}$$

since $N \bullet v = 0$ for any v, $0 \le v < N$. If equation (2) is true, then it is also true that

$$\gcd(P, N) = 1 \tag{3}$$

Given N and P, where P is a prospective 64-bit private key, perhaps chosen at random, the Extended Euclid's algorithm can be used to calculate P^{-1} and gcd(P, N) (v is of no interest). If $\gcd(P, N) \neq 1$ (not every number P, $0 \le P < N$, has a solution in equation (2)), then P is not co-prime with respect to N and hence is not a suitable private key. If gcd (P,N) = 1, then P and P^{-1} are modular inverses and are suitable for use as a key pair.

3.3. AP Send-Recv Task Algorithm

Pseudocode for application tasks, the AP Send-Recv Task, and the Network Drivers is presented below.

3.3.1. AP Send-Recv Task Initialization

```
Each task copy, i (Congruent Operations on Congruent Data):
   AP Send-Recv Task:
      Check Routing Table:
      Scan for all direct connections
      Initialize a device for each Device ID which is directly connected to the host
      Determine number and location of all Send-Recv Application Output / Input Queues, set up system Send-Recv Output / Input Queues
```

Figure 32. Pseudocode of Authentication Protocols Send-Recv Task Initialization

3.3.2. Application Task Output

```
Each task copy, i (Congruent Operations on Congruent Data):
  AP Application Interface:
    Send message(i) to correct Send-Recv Output Queue
```

3.3.3. AP Send-Recv Task Output

```
   AP Send-Recv Task (Congruent Operations on Congruent Data) :
      Copy message from Send-Recv Output Queues into Host Output Buffer using Queue Selection Logic which:
         Decide which application to process
         Decide which Send-Recv Output Queues to access and in what order
         Decide how many messages from each queue are processed this round and in what way (i.e., all messages from 1 queue, one message from each queue, 10 messages from one and 5 from the next)
         Decide how many message are processed (by frame time or number of messages?)
      Examine routing information
         Extract destination network address from network header
         Use destination[0] to determine destination subnet
         Scan Routing Table for destination subnet
            Check connection field -> Direct or Gateway
            Save the Host ID of the next node (destination[1] if it is a direct connection, or the Gateway_ID)
            Save Device ID
         Extract Source network address from network header
         Compare source[1] (Host ID) with current host
            If (source[1] = Host) or (Message_type says to authenticate),
               append trailer
                  Append Signature Configuration to end of message
                  Generate Sequence number
                     Must be monotonically increasing
                     Add to trailer
                     Increment sequence number
                  Generate Signature(i) (Congruent Operations on Non-Congruent Data)
                     Currently CRC-based, using private key
                  From_i : Signature(i)
                  Append all signatures to end of sequence number
      Send message(i) using Network Send-Recv call
Each task copy, j, with Media Layer Interface:
   Network Driver (Incongruent Operations on Congruent Data) :
      Transmit message(j)
```

Figure 33. Pseudocode of Authentication Protocols Output Message Processing

3.3.4. AP Send-Recv Task Input

```
Each AP Send-Recv task copy, i:
  Poll Network Driver (Incongruent Operations on Incongruent Data) :
    Read any messages coming in over the networks
    Retrieve message from Network Send-Recv calls into Input Host
    Buffer using Network Selection Logic which decides:
      Which network to check first
      How many message from each network to take
      How many iterations to run
    If message present,
      From_i : Message(i)
    (Congruent Operations, Congruent Data)
    Check Sequence number and source to see if message is new If message is old,
      Stop processing and get next message
    Each task copy, j (Congruent Operations, Congruent Data) :
      Examine Routing Information
        Extract destination and source network addresses from network header
        Compare destination[1] to My_Host_ID
        If destination[1] = My_Host_ID,
          Use public key for source[1] (Host) to authenticate message
            If invalid,
              Log message
            Else
              Use destination[2] to distribute to correct Send-Recv Input Queue
            Else
              Put in Gateway Queue
```

Figure 34. Pseudocode of Authentication Protocols Input Message Processing

3.4. AP System Performance

The following performance figures were taken on the AFTA's 25 MHz 68030 PEs while executing the AGN&C application. All messages are 256 bytes in length.

Authentication Protocols Function	Execution Time, milliseconds
send_message (application call)	0.430
read_message (application call)	0.510
sign and cross-channel exchange four channels' signatures	3.890
authenticate four channels' signatures	2.850

Table 5. Performance of Selected AP Functionality

3.5 Lessons Learned

In the course of implementing and integrating the Authentication Protocols and the Advanced Guidance, Navigation, and Controls systems, several lessons were learned. It should be noted that few are related to the basic AP theory of operation or underlying fault tolerance of the FTPPs themselves. Instead they are more symptomatic of the immaturity of the state of the art of developing a complex distributed system on heterogeneous execution platforms.

The software for the AP system was implemented in C and developed on Sun workstations using the GCC compiler. This included code for the Suns themselves, the Dirac LynxOS workstation, and the FTPPs' XDAda Run Time System. The FTPP XDAda code was developed on a VAX. Conditional compilation switches were used for including host-specific operating system calls and other features necessitated by the heterogeneous operating environment. A complete AP system (not including the AGN&C code) comprises about 96 source and header files, for a total of 18,000 lines of C code (including comments). At any given time, four to five programmers were working on developing the system, often working on the same file concurrently. Extreme cost and schedule pressures were in effect throughout the project.

In retrospect, successful completion of this project would not have been possible had not several key decisions been made.

Modular Design and Implementation: A significant amount of up-front work and debate was expended in attempting to clearly define the desired capabilities of AP, modularizing these capabilities, defining the interfaces between the modules, and assigning developers to the modules. While module capabilities, interfaces, and "ownership" changed (sometimes significantly) as the implementation and integration proceeded, this initial effort was crucial to the successful implementation of this complex design.

RCS: The large number of files and concurrently working programmers had the potential for creating a configuration management (CM) nightmare. Early on in the implementation a configuration management system based on the Unix RCS version management system was implemented for the AP code. (Code that was not included in the AP CM system was a continual source of difficulty.) While configuration management was still difficult for the AP code, failure to implement a rigorous CM program at such an early stage would have made it impossible. In addition, the constant documentation encouraged by RCS allowed a programmer to understand and debug another's code without the presence of the other programmer being required. Notwithstanding, towards the end of the program we instituted a rule that no one could leave the building while they had an RCS source file "checked out."

NFS: All source files under CM were resident on a single Sun workstation and exported to other environments for final linking and downloading. The ability to easily share the large repository of AP source files in real time without lengthy file copying made updating (and diagnosing) the entire set of load modules relatively straightforward.

Windows: The full-up AP / AGN&C demonstration exercises code running on about a dozen processors, with a total of about fifty concurrently active tasks. The multi-windows capability of the Sun workstations allowed a developer / operator to simultaneously view and control a carefully selected subset of these concurrent actions on one screen. This capability is now thought to be absolutely essential when developing any type of distributed system.

Logic Analyzer: A multichannel logic analyzer was connected to the FTPP processors' parallel output ports. The processors could then be programmed to write selected patterns to their output ports at selected points during the execution of code. By capturing and viewing on the logic analyzer the sequence of patterns emanating from all processors of interest, the redundant system's real time behavior could be conveniently and unobtrusively observed.

Some design decisions and development / operating environment constraints adversely impacted the AP design's ease of implementation and performance.

Distributed System Debugging: Debugging the code running on the Sun and LynxOS systems was relatively straightforward using the GDB debugging tool. However, it was not possible to globally set and clear breakpoints, single step execution, etc., throughout the distributed system due to the lack of a distributed system debugger. In addition, the Ada code running on the redundant FTPPs had no debugger which would work in the real time, redundant, distributed, synchronous environment. The programmer in this case had to resort to programming screen print outs and examining memory dumps from halted processors.

XDAda Run Time System: As mentioned before, the AP system was implemented in the C programming language. For commonality, the same source code was compiled for use on all hosts running AP, including the FTPPs. However, the XDAda Run Time System running on the FTPPs at the time has no provision for interfacing to code written in any other high order language. Therefore the AP code had to be downloaded separately to a "safe" area of memory, and, when an Ada task invoked an AP call, a "jump" to the appropriate C code was forced via an assembly language Pragma Interface. This was implemented with surprisingly little effort and caused no obvious problems. However, it seems to violate the principles of good software engineering practice.

Mutual Exclusion on Shared Memory: The AP system is designed to allow a large number of application processes on any given host to utilize AP communication services. On the Unix / LynxOS hosts, where processes may be scheduled arbitrarily, this necessitated the enforcement of mutual exclusion on data structures which are shared among application processes and the AP Send-Recv Task. (On the FTPPs' Run Time Systems, mutual exclusion is easily enforced via precisely scheduling the various tasks.) Mutual exclusion was obtained using Unix system calls which, in conjunction with the latency required to schedule two tasks to send a message (the application task as well as the AP Send-Recv Task), proved to severely slow down the execution of application / AP task suites compared to simple socket-based communication. Tuning was required to achieve real time performance, and in.some cases real time performance could not be achieved at all. In addition, necessary mutual exclusion was occasionally and erroneously removed in an attempt to improve performance, resulting in hard-to-track spurious errors. In retrospect it is clear that insufficient attention was paid to this part of the implementation.

4. References

[Diffie76] Diffie, W., Hellman, M.E., "New Directions in Cryptography", *IEEE Trans. Inf. Theory* , IT-22 (Nov. 1976), 644-654.

[Lala84] Lala, J.H., "Advanced Information Processing System", 6th Digital Avionics Systems Conference (DASC), Baltimore, MD, December 1984, pp. 199-210.

[Lala87] Lala, J.H., Adams, S.J., "Inter-Computer Communication Architecture for a Mixed Redundancy Distributed System", AIAA Guidance Navigation, and Control Conference, Monterey, CA, August 1987, pp. 1571-1578.

[Lamport82] Lamport L., Shostak, R., Pease, M., "The Byzantine Generals Problem" *ACM Transactions on Programming Languages and Systems*, Vol. 4, No. 3, July 1982, pp. 382-401.

[Rivest78] Rivest, R.L., Shamir, A., Adleman, L., "A Method for Obtaining Digital Signatures and Public Key Cryptosystems" *Communications of the ACM* Vol. 21-2 (Feb. 1978), 120-126.

[Fischer82] Fischer, M. J., Lynch, N. A., "A Lower Bound for the Time to Assure Interactive Consistency," *Information Processing Letters*, Vol. 14, No. 4, 13 June 1982, pp. 183-186.

[Dolev82] Dolev, D., "The Byzantine Generals Strike Again," *Journal of Algorithms*, Vol. 3, 1982, pp. 14-30.

[Dolev83] Dolev, D., Strong H., "Authenticated Algorithms for Byzantine Agreement," *SIAM Journal of Computing*, Vol. 12, No. 4, November 1982, pp. 656-666.

[Dolev84] Dolev, D., Dwork, C., Stockmeyer, L., "On the Minimal Synchronism Needed for Distributed Consensus," IBM Research Report RJ 4292 (46990), 5/8/84.

[DZ83] J.D. Day, H. Zimmerman, "The OSI Reference Model," Proceedings of the IEEE, VOL.. C-12, December 1983.

[Galetti89] Galetti, R., "The CSDL Keyed Polynomial Authenticator - Serial Implementation," Draper IR&D Internal Project Report 278-2, May 1989.

[Galetti90] Galetti, R., "Real-Time Digital Signatures and Authentication Protocols," Masters Thesis , Massachusetts Institute of Technology, June 1990.

[JEM77] McNamara, John E., Technical Aspects of Data Communication, Digital Equipment Corporation, Bedford, MA, 1977.

[MK88] Miller, Stuart E., and Kaminow, Ivan P. eds., Optical Fiber Telecommunications II, Harcourt Brace Jovanovich, Boston, MA, 1988.

[MM78] Mallela, S., Masson, G., "Diagnosable Systems for Intermittent Faults", *IEEE Trans. on Computers*, Vol. C-27, No. 6, June 1978, pp.560-566.

[Pease80] Pease, M., Shostak, R., Lamport, L., "Reaching Agreement in the Presence of Faults," *Journal of the ACM*, Vol. 27, No. 2, April 1980, pp. 228-234.

[PMC67] Preparata, F., Gernot, M., Chien, R., "On the Connection Assignment Problem of Diagnosable Systems," *IEEE Trans. on Electronic Computers*, Vol. EC-16, No. 6, Dec. 1967, pp.848-854.

[Stein88] Steiner, J., Neuman, C., Schiller, J., "Kerberos: An Authentication Service for Open Network Systems," USENIX 88.

[Sundstrom74] Sundstrom, R. J., "On-Line Diagnosis of Sequential Systems," PhD Thesis, University of Michigan, 1974.

[Ultron88] "Crypto-Engine Interface Specification," Ultron Labs, Document No. 0N397096, San Jose CA, 1988.

Appendix A. Man Pages

APPLICATION INTERFACE

The application interface is defined in section 1. This section is useful for applications programmers who need to know details of the system calls for accessing ATP communication ports.

CONTENTS

NAME

open_port - open an ATP port for unidirectional or bidirectional communication

SYNOPSIS

```
#include "appl_interface.h"
int open_port(uword16 portnum, port_config_struct *config);
```

DESCRIPTION

open_port() creates a port for communication using the authentication protocol (ATP). The port may be either unidirectional or bidirectional.

The desired port number is indicated by *portnum*, a 16-bit quantity which defines the last short integer in the network address. Mapping of ports to processes or tasks is defined by the implementation (see getmyport(1)).

The port is configured according to the parameters defined in the *config* structure.

The port is opened for output if the *config->open_for_output* parameter is set to TRUE. If the port uses a static queue structure, the queue is allocated to handle *config->out_queue_size* messages. If the port uses a static buffer pool, a pool of *config->out_bufpool_size* bytes is allocated. If either of these structures is non-existent or dynamically allocated, the corresponding parameter is ignored.

The port is opened for input if the *config->open_for_input* parameter is set to TRUE. If the port uses a static queue structure, the queue is allocated to handle *config->in_queue_size* messages. If the port uses a static buffer pool, a pool of *config->in_bufpool_size* bytes is allocated. If either of these structures is non-existent or dynamically allocated, the corresponding parameter is ignored.

A port is never (even partially) opened if any error occurs.

RETURN VALUES

ATPSUCCESS
: success

ATPEOPEN
: The requested port is already open

ATPEBADPORT
: The requesting task has asked for an illegal port number, or a port which it does not have permission to open.

ATPENOOUTSPACE
: There is not sufficient memory to open the port with the requested output port configuration. It might be possible to open the port by reducing the size of the output buffers.

ATPENOINSPACE
: There is not sufficient memory to open the port with the requested input port configuration. It might be possible to open the port by reducing the size of either the output buffers, the input buffers, or both.

SEE ALSO

close_port(1), configure_port(1), send_message(1), receive_message(1)

NAME

close_port - close an ATP port.

SYNOPSIS

```
#include "appl_interface.h"
int close_port(uword16 portnum);
```

DESCRIPTION

ATP ports are closed using close_port(). If the port requested by *portnum* is open, all structures associated with the port are deallocated and no further communication with the port is allowed unless it is reopened. Any messages remaining in the queues belonging to the port are lost.

RETURN VALUES

ATPSUCCESS
success

ATPENOTOPEN
The requested port is not open

ATPEBADPORT
The requesting task has asked for an illegal port number, or a port which it does not have permission to close.

SEE ALSO

open_port(1), configure_port(1), send_message(1), receive_message(1)

NAME

configure_port() - modify or obtain status information on an ATP port

SYNOPSIS

```
#include "appl_interface.h"
int configure_port(uword16 portnum, int command,
    port_config_struct *config);
```

DESCRIPTION

The implementation of configure_port() is currently undefined. Eventually, it will be used for various control and status operations on open ports, including (possibly) changing queue and buffer allocation, determining queue status, receiving accumulated error information, etc.

Some examples:
get message lost counter
reset message lost counter
reallocate queue and/or buffer size
flush queue and/or buffer

RETURN VALUES

ATPSUCCESS

success

ATPENOTOPEN

The requested port is not open

ATPEBADPORT

The requesting task has asked for an illegal port number, or a port which it does not have permission to configure.

ATPEILLEGAL

The requested *command* is not a legal operation for this port.

ATPEINVALID

The parameter structure *config* contains an invalid entry.

SEE ALSO

open_port(1), close_port(1), send_message(1), receive_message(1)

NAME

send_message() - send a message over the authentication network

SYNOPSIS

```
#include "appl_interface.h"
int send_message(uword16 portnum,
    char *message, header_struct *header,
    flags_type flags);
```

DESCRIPTION

The send_message() function is used to transmit a message using the authentication protocol out ATP port *portnum* . The message is a contiguous stream of bytes pointed to by *message*. All characteristics of the message, including message length, message type, source address, and destination address are defined by **header*, except the port part of the source address, which is specified by *portnum*. The send_message() call does not fill in the information in the header, except for the source port. The user is expected to format the packet header correctly before transmitting the message.

The *flags* are used to request special operations. Currently, the definition of the flags operand is undefined.

RETURN VALUES

On success, send_message() returns the actual number of data bytes transmitted (not including the header). This should be the length of the message specified in the header structure. On error, send_message() returns a negative number indicating one of the following errors:

ATPENOTOPEN
: The requested port is not open for writing.

ATPEBADPORT
: The requesting task has asked for an illegal port number, or a port which it does not have permission to transmit on.

ATPEQFULL
: The output queue for the requested port are full.

ATPEBUFFULL
: The output buffer pool is incapable of holding a message of the requested length.

SEE ALSO

open_port(1), close_port(1), configure_port(1), receive_message(1), getmysubnet(1), getmynode(1)

NAME

receive_message() - receive a message from the authentication network

SYNOPSIS

```
#include "appl_interface.h"
int receive_message(uword16 portnum,
    char *message,
    int max_msg_size,
    header_struct *header,
    trailer_struct *trailer,
    flags_type flags);
```

DESCRIPTION

Attempts to read a message from the ATP port requested by *portnum*. If there is a ready message on the port, and the user buffer is large enough, the message will be copied to the location indicated by *message*. The size of this buffer is indicated by *max_msg_size*; the receive_message() call will never return a message which is longer than *max_msg_size* (not including the header or trailer).

Only the data part of the message is copied to *message*. The header (and optionally, the trailer) are copied to the structures *header* and *trailer*, respectively. If the *trailer* argument points to NULL, the trailer is not returned. It is an error for either *header* or *message* to point to NULL.

The *flags* are used to request special operations. Currently, the definition of the flags operand is undefined.

RETURN VALUES

On success, receive_message() returns the actual number of data bytes read (not including the header or the trailer). This should be the length of the message specified in the header structure. On error, receive_message() returns a negative number indicating one of the following errors:

ATPENOTOPEN
The requested port is not open for reading.

ATPEBADPORT
The requesting task has asked for an illegal port number, or a port which it does not have permission to receive on.

ATPETOOSMALL
The message buffer *message* of length *max_msg_size* is too small to contain the next message from the port.

ATPEBADPARAM
Either the *message* or *header* parameters did not point to a valid memory location.

ATPENOMSG
There were no message waiting to be received at the specified port.

ATPEQFULL

The input queue for the requested port was full and at least one incoming message was discarded.

ATPEBUFFULL
The input buffer pool was incapable of holding at least one incoming message. This(ese) message(s) was(were) discarded.

SEE ALSO

open_port(1), close_port(1), configure_port(1), send_message(1)

NAME

getmysubnet, getmynode, getmyport, getmyportmask - get information about the local host

SYNOPSIS

```
#include "appl_interface.h"
uword16 getmysubnet(int interface);
uword16 getmynode(int interface);
#include "atp_groups.h"
uword16 getmyport(int group);
uword16 getmyportmask(int group);
```

DESCRIPTION

getmysubnet() obtains the subnet identifier to which the network interface specified by *interface* is connected. If the parameter *interface* is -1, the host's "preferred" subnet identifier is returned. The preferred subnet is generally the highest reliable subnet to which the host is connected.

getmynode() obtains the node identifier assigned to the specified network interface. If the *interface* parameter is -1, the host's "preferred" node identifier is returned. The preferred node generally corresponds to the highest reliable interface on the host. The return value of getmynode(-1) is consistent with the return value of getmysubnet(-1), i.e. the combined result can be used to create the source address for a message header.

getmyport() and getmyportmask() are used to determine a valid port name for the task. The *group* parameter is used to identify certain system groups to which the task may belong. The default group is USER. Other groups include:

FDIR
: reserved for fault detection, isolation, and reconfiguration software.

DIRECTORY
: directory assistance

NETMON
: network monitor

GATEWAY_PORT
: reserved for ATP internal use

EXP0-EXP3
: these four groups are available for experimentation when the use of dynamic port numbers and directory assistance is not desired. Since the number of fixed ports is limited, the use of these ports should generally be replaced by dynamic port allocation and directory assistance after preliminary testing is complete.

The value returned by getmyport() may represent a range of valid port names. The mask value returned by getmyportmask() indicates which bits of the port name range may be modified by the task. If a bit is set in the mask, the corresponding bit will be cleared in the port name. However, the task may modify this bit when selecting a port name. If getmyportmask() returns 0, the task must use the exact value returned by getmyport().

getmyport() and getmyportmask() do not actually test the permission of the task to open a specific port; they return the appropriate port number assuming the task has the appropriate permission. Thus, getmyport() and getmyportmask() can be used to determine the local port address for communicating with certain system tasks.

For example, a newly awakened task can use getmyport() and getmyportmask() to determine the appropriate local address of the Directory Assistance (group DIRECTORY) task.

EXAMPLES

```
#include "header.h"
#include "atp_groups.h"
#include "appl_interface.h"
header_struct myheader;
myheader.src_addr[0] = getmysubnet(0);
myheader.src_addr[1] = getmynode(0);
/* using the convention <15:8> == taskID and */
/* <7:0> == queueID, set queueID to 37 */
myheader.src_addr[2] = getmyport(USER) |
     (getmyportmask(USER) & 37);
```

RETURN VALUES

getmysubnet() and getmynode() return the appropriate result if the specified interface exists. If the specified interface does not exist, or is not configured, getmysubnet() and getmynode() both return 0.

getmyport() and getmyportmask() return a valid value under all conditions. If the parameter to either getmyport or getmyportmask is not specified or corresponds to a non-existent group the assumed value of the parameter is USER.

SEE ALSO

configure_port(1), send_message(1)

NETWORK DEVICE DRIVER INTERFACE

The network device driver interface is defined in section 2. This section is of interest primarily to those writing either new network device drivers or those implementing the application interface described in section 1.

CONTENTS

NAME

atp_init - initialize each authentication protocol interface driver

SYNOPSIS

```
#include "atp_driver.h"
atp_init(driver_table_struct *master_driver_table[]);
```

DESCRIPTION

Drivers for ATP interfaces are initialized at startup by atp_init(). The argument *master_driver_table* specifies a hierarchical initialization structure for configuration of drivers and devices. The *master_driver_table* is an array of pointers to driver_table_struct's. The size of *master_driver_table* is MAXNUMDRIVERS. The atp_init() function searches the table starting with entry 0. For each valid entry, corresponding to a driver to be initialized, atp_init() will call the individual driver initialization routine. At the first invalid entry, indicated either by a pointer to NULL in *master_driver_table* or by scanning past the (MAXNUMDRIVERS-1)'th entry, initialization is terminated.

Initialization results in allocation and initialization of local data structures. It may also generate network traffic.

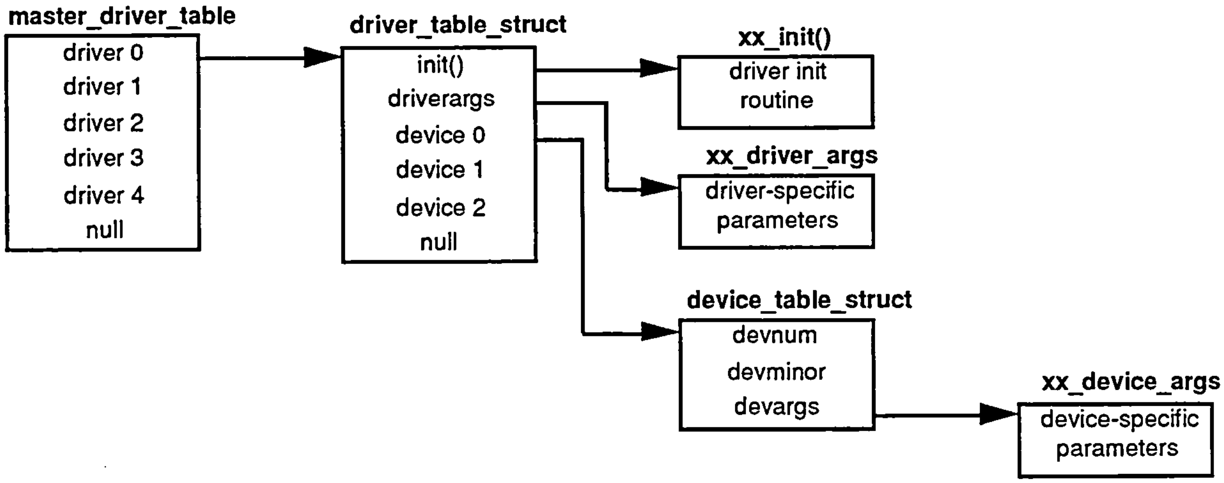
RETURN VALUES

ATPEINVALID

The parameter *arg* contains an invalid quantity.

EXAMPLES

The figure below demonstrates the hierarchy of structures passed to the atp_init routine, including the driver- and device-specific structures for a hypothetical driver named xx.



Initialization Structure Hierarchy

The following example code demonstrates how to configure the pointers to construct a hierarchy similar to the one shown above. Initialization of structure elements besides pointers is not shown.

```
/* configure one loopback device, two udp devices, */
/* and one udpr device */
#include "atp_driver.h"
#include "lb.h"
#include "udp.h"
#include "udpr.h"

driver_table_struct *master[MAXNUMDRIVERS];
driver_table_struct lb_driver;
driver_table_struct udp_driver;
driver_table_struct udpr_driver;

device_table_struct lb_device;
device_table_struct udp_device[2];
device_table_struct udpr_device;

lb_driver_args lb_drv;
lb_device_args lb_dev;

udp_driver_args udp_drv;
udp_device_args udp_dev[2];

udpr_driver_args udpr_drv;
udpr_device_args udpr_dev;

master[0] = &lb_driver;
master[1] = &udp_driver;
master[2] = &udpr_driver;
master[3] = NULL;

lb_driver.driverargs = (char *)&lb_drv;
lb_driver.device_table[0] = &lb_device;
lb_driver.device_table[1] = NULL;
lb_device.devargs = (char *)&lb_dev;
/* configure lb driver in lb_drv, configure the single */
/* lb device in lb_dev */

udp_driver.driverargs = (char *)&udp_drv;
udp_driver.device_table[0] = &udp_device[0];
udp_driver.device_table[1] = &udp_device[1];
udp_driver.device_table[2] = NULL;
udp_device[0].devargs = (char *)&udp_dev[0];
udp_device[1].devargs = (char *)&udp_dev[1];
/* configure udp driver in udp_drv, configure the two */
/* udp devices in udp_dev[0] and udp_dev[1] */

udpr_driver.driverargs = (char *)&udpr_drv;
udpr_driver.device_table[0] = &udpr_device;
udpr_driver.device_table[1] = NULL;
udpr_device.devargs = (char *)&udpr_dev;
/* configure udpr driver in udpr_drv, configure the */
/* single udpr device in udpr_dev */
```

SEE ALSO

atp_open(2), atp_close(2), atp_read(2), atp_write(2), atp_ioctl(2), driver_table_struct(3), device_table_struct(3), lb(4), udp(4), udpr(4)

NAME

atp_open - open an interface to the ATP network

SYNOPSIS

```
int atp_open(int device, int flags);
```

DESCRIPTION

atp_open opens the interface specified by *device* for sending and receiving messages over the authentication protocol network.

The atp_open() call behaves similarly to the Unix open(2) system call. However, instead of passing a character string filename parameter to indicate the desired device, atp_open takes the device's integer name. This integer name is also used by the other atp_ calls. For consistency with open(2), atp_open() returns the device integer name on success.

A device that is marked DOWN may be opened, but no data can be sent or received by that device until it is configured UP by an atp_ioctl() call.

The atp_open() call supports the following flags:

O_RDWR Open device for read and write. Interface devices in ATP are always opened for bidirectional access.

RETURN VALUES

On success, atp_open() returns a positive value indicating the opened device. Currently, the returned value will be the same as the first parameter passed to atp_open.

On failure, atp_open() will return one of the following errors.

ATPEOPEN
The named device is already opened and the device requires exclusive access.

ATPENOEXIST
The named device does not exist.

ATPEINTERNAL
An internal protocol error occurred within the driver.

SEE ALSO

atp_init(2), atp_close(2), atp_read(2), atp_write(2), atp_ioctl(2)

NAME

atp_close - close an ATP interface

SYNOPSIS

atp_close(int device);

DESCRIPTION

The atp_close() call is used to close an interface. Interfaces are marked DOWN before they are closed. An interface may be opened later by this or another process.

RETURN VALUES

atp_close() will return one of the following values.

ATPSUCCESS
: device was successfully closed.

ATPENOTOPEN
: The named device was not opened by this process.

ATPENOEXIST
: The named device does not exist.

ATPEINTERNAL
: An internal protocol error occurred within the driver.

SEE ALSO

atp_init(2), atp_open(2), atp_read(2), atp_write(2), atp_ioctl(2)

BUGS

Since ATP device names are not stored in the standard file descriptor table, ATP devices are not closed automatically upon exit.

NAME

atp_read - read from an ATP interface

SYNOPSIS

```
atp_read(int device, char *buf, int nbyte);
```

DESCRIPTION

atp_read() is used to get input from the interface device specified by *device.* The atp_read() call behaves in a similar manner to the read(2) system call.

The atp_read() call preserves message boundaries. If *nbyte* is smaller than the length of the next message segment remaining, exactly *nbyte*s of the message are returned in *buf*, and the remaining bytes are left on the device queue to be read by subsequent calls to atp_read() (this behavior is similar to the STREAMS message-nondiscard mode). If *nbyte* is larger than or equal to the length of the next message segment, the complete message segment is returned in *buf* and the number of characters in the message segment is returned by atp_read().

If *nbyte* < 0, atp_read() does nothing and returns 0.

Following a call to atp_read(), the source address of the message can be retrieved using an atp_ioctl() call to GETSRCPHYS.

Calls to atp_read() are always non-blocking, so the error ATPENOMSG is returned when there is no data to be read on the named device.

RETURN VALUES

On success, atp_read() returns the actual number of characters returned in *buf.* Note that zero-length messages are valid in ATP.

On failure, atp_read() returns one of the following errors.

ATPENOTOPEN
The named device is not opened by this process.

ATPENOEXIST
The named device does not exist.

ATPEINTERNAL
An internal protocol error occurred within the driver.

ATPENOMSG
There is no data waiting to be read on the named device.

ATPENOPHYS
The physical address of the device has not been configured.

SEE ALSO

atp_init(2), atp_open(2), atp_close(2), atp_write(2), atp_ioctl(2)

NAME

atp_write - write to an ATP interface

SYNOPSIS

atp_write(int device, char *buf, int nbyte);

DESCRIPTION

atp_write() is used to send a message on the interface device specified by *device*. The atp_write() call behaves in a similar manner to the write(2) system call.

The atp_write() call preserves message boundaries. If the device is incapable of accepting a message of length *nbyte*, the atp_write() call will return an ATPENOSPACE error.

If *nbyte* < 0, atp_write() does nothing and returns 0.

Before calling atp_write(), the destination address must be configured using an atp_ioctl() call to SETDESTPHYS.

Calls to atp_write() are always non-blocking, so the error ATPENOSPACE is always returned if the device cannot accept *nbyte*s of data.

RETURN VALUES

On success, atp_write() returns the actual number of characters transmitted from *buf.* This value will always be exactly equal to *nbyte.*

On failure, atp_read() returns one of the following errors.

ATPENOTOPEN
The named device is not opened by this process.

ATPENOEXIST
The named device does not exist.

ATPEINTERNAL
An internal protocol error occurred within the driver.

ATPENOSPACE
The named device is temporarily incapable of accepting a message of the requested length.

ATPETOOLARGE
The specified message is too large for transmission by the specified device.

ATPENOPHYS
The destination physical address has not been configured.

SEE ALSO

atp_init(2), atp_open(2), atp_close(2), atp_read(2), atp_ioctl(2)

NAME

atp_ioctl - control an ATP interface

SYNOPSIS

```
#include "atp_driver.h"
#include "phys_addr.h"
#include "arp.h"
atp_ioctl(int device, int request, char *arg);
```

DESCRIPTION

The following ioctl calls use *arg* as a pointer to a physaddr_struct structure.

SETMYPHYS - sets the physical address of the device. This operation is not allowed on some types of devices. The type of physical layer is specified in *arg->phys_type* as follows:

ATP_UNK - unknown physical type.
ATP_UDP - physical layer is UDP
ATP_ETHER - physical layer is Ethernet
ATP_FDDI - physical layer is FDDI (either single or dual attach).
ATP_LTALK - physical layer is LocalTalk
ATP_1553 - physical layer is MIL-STD-1553.
ATP_NE - physical layer is the FTPP (a.k.a. AFTA) network element.

The number of distinct physical addresses is specified by *arg->phys_count.* Currently, only one or two physical addresses are supported. Multiple physical addresses are necessary for certain types of redundant network connections with different addresses on each media layer. The order in which multiple physical addresses are specified is significant.
The physical address to be assigned to the device is contained in *arg->phys_addr*. The interpretation of this union is dependent on the actual physical type as indicated above. Each union element may be an array containing one or more physical addresses, one for each media layer to which the device is connected.

GETMYPHYS - returns the physical address of the underlying physical layer device.

SETDESTPHYS - sets the destination physical address to be attached to outgoing packets to that specified by *arg*. The destination physical address is the address to which all packets sent via atp_write() will be transmitted.

GETDESTPHYS - returns the current destination physical address configured for the device in *arg*.

GETSRCPHYS - returns the source physical address of the most recently received packet. This information is not normally required by the upper layer. The source physical address will be retained by the device driver, to be read with this ioctl call, until a subsequent atp_read() call occurs.

The following ioctl calls use *arg* as a pointer to an arp_struct structure. The physaddr_struct structure in the arp_struct structure is suitable for use with the preceeding ioctl calls that interpret *arg* as a pointer to a physaddr_struct structure.

ARPADD - add an entry to the ARP table. *arg* indicates an arp_struct. *arg->phys_addr* indicates the physical address to be added. *arg->net_addr[3]* is the network address to be matched to the physical address. Only the first two values in the network address are used by the ARP table.

ARPDELETE - delete an entry from the ARP table. *arg* indicates an arp_struct. The *phys_addr* element of *arg* is ignored by ARPDELETE.

ARPLOOKUP - look up an entry in the ARP table. A network address is presented in *arg->net_addr* and the resulting physical address, if found, is returned in *arg->phys_addr.* An error of ATPENOADDR is returned if no entry was found corresponding to the specified network address.

The following ioctl calls require no input or output parameter.

SETDEVICEUP, SETDEVICEDOWN - Enables or disables the device, respectively.

RETURN VALUES

atp_ioctl() returns ATPSUCCESS upon successful completion, or one of the following errors:

ATPENOTOPEN
The named device is not opened by this process.

ATPENOEXIST
The named device does not exist.

ATPEINTERNAL
An internal protocol error occurred within the driver.

ATPENOTFOUND
no entry could be found in the ARP table for the requested network address.

ATPEARP
An error or inconsistency exists in the ARP table.

ATPEILLEGAL
The requested operation is not permitted on the specified device.

ATPEINVALID
The parameter *arg* contains an invalid quantity.

ATPENOPHYS
The requested physical address has not yet been assigned. This error will occur on a GETSRCPHYS call if no messages have been received on the indicated device since it was opened.

SEE ALSO

atp_init(2), atp_open(2), atp_close(2), atp_read(2), atp_write(2), physaddr_struct(3), arp_struct(3)

INCLUDE FILES

atp_ioctl.h - defines valid ioctl calls for atp_ioctl()
phys_addr.h - defines structures for various physical address forms, and structures for manipulating the ARP tables.

BUGS

The routines for manipulating the routing tables (ARPADD and ARPDELETE) are not currently implemented.

TYPE DEFINITIONS

Several compound types are defined in section 3. These types have application throughout the ATP network system. The header_struct and trailer_struct are primarily of interest to application programmers and to those implementing the application interface. The physaddr_struct and arp_struct are used in the application interface and in the device drivers. The driver_table_struct and device_table_struct are of interest to those implementing device drivers and for those who are configuring devices in a system.

CONTENTS

NAME

physaddr_struct - structure to contain physical addresses

SYNOPSIS

#include "phys_addr.h"
physaddr_struct phys_addr;

DESCRIPTION

The physaddr_struct type is a generic structure for containing an arbitrary physical address. The definition of physaddr_struct is intended to be independent of the device-specifics associated with different types of network devices. By using physaddr_struct structures wherever a representation of a physical address is required, the user can manupulate physical addresses without knowing the detailed physical address representation.

The definition of the physaddr_struct structure is as follows:

```
typedef struct
{
  uword16 phys_type;
  uword16 phys_count;
  union
  {
    char unknown[64];
    udp_addr udp[2];
    ether_addr ether[2];
    fddi_addr fddi[2];
    ltalk_addr ltalk[2];
    m1553_addr m1553[2];
    ne_addr ne;
  } phys_addr;
} physaddr_struct;
```

The *phys_type* element indicates the type of physical address contained in *phys_addr*. The type indicates how the *phys_addr* union is to be interpreted. Predefined types in phys_layer.h include:

ATP_UNK - unknown physical type.
ATP_UDP - physical layer is UDP
ATP_ETHER - physical layer is Ethernet
ATP_FDDI - physical layer is FDDI (either single or dual attach).
ATP_LTALK - physical layer is LocalTalk
ATP_1553 - physical layer is MIL-STD-1553.
ATP_NE - physical layer is the FTPP (a.k.a. AFTA) network element.
ATP_UNDEFINED - physical address is undefined. This is different than the unknown physical type.

The type ATP_UNK can be used to represent physical addresses with no representation in the *phys_addr* union. The type ATP_UNDEFINED is used specifically to indicate that the physical address represented by *phys_addr* is not defined.

The number of distinct physical addresses is specified by *phys_count*. Currently, only one or two physical addresses are supported. With certain types of redundant network connections, such as redundant FDDI or redundant Ethernet, each copy of a message to be transmitted on multiple media layers will use different destination physical addresses. Other types of redundant connections, such as the network element, do not require different physical addresses. Thus, *phys_count* does not necessarily indicate the redundancy level of the interface, but simply how many different physical addresses of the indicated type are needed.

For all devices which require multiple physical addresses, the order of the physical addresses is important. The true physical address for such a redundant device is always a complete, ordered list. The driver must always treat this list, and its own internal list of physical connections, in a consistent manner (i.e., always in the same order).

The physical address is contained in *phys_addr*. The interpretation of this union is dependent on the *phys_type* indicated above. Each union element is an array containing one or more physical addresses as specified by *phys_count*. Unless otherwise noted, the formatting and byte ordering of these physical addresses is defined by the relevant standard.

The physical address *unknown* is used for new types of interfaces which are not currently defined. It is simply a string of 64 bytes which can be interpreted in any way. Since it also represents the maximum size of the *phys_addr* union, it can also be used to access the byte image of any other type of physical address. However, due to machine dependencies, this access mode should be minimized, and should never be used for inter-processor communication.

The UDP address represents one or more UDP ports for communication over a simulated authentication protocol network. In this case, UDP is used as a physical layer to the ATP network, although it actually resides at the transport layer in the TCP/IP protocol suite (an example of recursive protocol stacks). The defintion of *udp_addr* is as follows:

```
typedef struct
{
  uword32 ipaddr;
  uword16 port;
} udp_addr;
```

The UDP address for a UDP interface specifies, for each media layer, an IP network address (*ipaddr*) and a UDP port number (*port*).

Ethernet addresses are represented by the *ether_addr* structure as follows:

```
typedef struct
{
  uword8 ether[6];
  llc_s llc_addr;
} ether_addr;
```

The 6-byte element *ether* represents a valid Ethernet address as defined by IEEE 802.3. The Ethernet address structure also includes the logical-link address, as defined by IEEE 802.2 and IEEE 802-1990.

Fiber Distributed Data Interface (FDDI) addresses are represented by the *fddi_addr* structure as follows:

```
typedef struct
{
  uword16 fddi_addr_type;
  union
```

```
          {
            uword8 long[6];
            uword8 short[2];
          } fddi;
          llc_s llc_addr;
        } fddi_addr;
```

Either short or long addresses are permitted. The type of address contained in *fddi* is indicated by *fddi_addr_type*, and is either FDDI_SHORT_ADDR or FDDI_LONG_ADDR. The format for short and long addresses is defined in ANSI X3.139-1987. The *fddi_addr* structure, like the *ether_addr* structure, contains entries for the IEEE logical-link control layer.

LocalTalk network addresses are represented by the *ltalk_addr* struct. The use of LocalTalk as the physical layer for ATP networks is currently undefined.

Addresses for the MIL-STD-1553B are represented by the *m1553_addr* structure. The use of MIL-STD-1553B for ATP networks is currently undefined.

The Fault-Tolerant Parallel Processor (FTPP) network element addressing is defined by the *ne_addr* structure as follows:

```
      typedef struct
      {
        uword8 fromvid;
        uword8 tovid;
        uword8 class;
        uword8 user;

        uword8 votesyn;
        uword8 clkerr;
        uword8 linkerr;
        uword8 unused1;

        uword8 obne_to;
        uword8 ibnf_to;
        uword8 score_votesyn;
        uword8 unused2;

        uword32 timestamp;
      } ne_addr;
```

The *ne_addr* structure is designed to closely resemble the format of the network element info blocks (defined in the *AFTA Network Element Hardware Documentation*, "Programmer's Reference") to optimize communication with the network element. For this reason, not all fields are defined in all circumstances. For example, if a destination physical address is represented by *ne_addr*, the syndrome, timeout, and timestamp fields will be meaningless. Although these fields have nothing to do with the physical address, they are represented here so that the higher layer protocols have a mechanism to obtain this useful information from the network element device driver. Typically, the higher layer will obtain the information through a GETSRCPHYS ioctl call to the network element device.

Although the network element, when used as a subnet of the ATP network, represents a redundant, Byzantine resilient interconnect, multiple physical addresses are not required, as they are on other redundant connections, such as Ethernet.

NAME

arp_struct - structure for address resolution protocol (ARP) requests

SYNOPSIS

#include "arp.h"
arp_struct arpreq;

DESCRIPTION

Each device driver is responsible for manipulating address resolution tables for the address resolution protocol (ARP). Each device may have its own unique table. Containing these tables in the device ensures that any device-specific aspects of the physical address are handled inside the driver, freeing the higher-layer protocols from having to deal with physical addresses at anything but an abstract level.

ARP operations are performed on the ARP tables through special ioctl calls defined by atp_ioctl(2). These ioctl calls use an arp_struct structure as the argument. The *arp_struct* structure is defined as follows:

```
typedef struct
{
  physaddr_struct phys_addr;
  uword16 net_addr[3];
} arp_struct;
```

The *phys_addr* is a representation of a physical address as described by physaddr_struct(3). *net_addr* is the network address corresponding to *phys_addr*. Only the first two bytes of *net_addr* are used in mapping between physical and network addresses.

SEE ALSO

atp_ioctl(2), physaddr_struct(3)

NAME

header_struct - ATP packet header structure

SYNOPSIS

#include "header.h"
header_struct header;

DESCRIPTION

The header of an ATP packet is described by the *header_struct* structure as follows:

```
typedef struct
{
  uword16 msg_size;
  uword16 msg_type;
  uword16 dest_addr[3];
  uword16 src_addr[3];
} header_struct;
```

The length of the data part of the packet (ignoring the header and trailer), in bytes, is indicated by *msg_size*. The *msg_type* field is user defined, except for the MSB. The MSB of *msg_type* is used to indicate the presence (1) or absence (0) of the authentication trailer.

The *dest_addr* and *src_addr* fields indicate the destination and source network addresses, respectively, of the packet. For both network addresses, the first byte, *xx_addr[0]*, is the subnet ID, the second byte, *xx_addr[1]* is the node ID, and the third byte, *xx_addr[2]*, is the port ID.

The bit field <15:14> of the subnet ID indicates the mode of the network address as follows:

0 - point-to-point
1 - multicast
2 - undefined/reserved
3 - broadcast

The remaining bits in the subnet ID are used to identify the subnet using a system-wide unique identifier. (Note: the use of multicasts and broadcasts in the ATP network is currently undefined).

For point-to-point communication, the node ID indicates a single physical processing site (which, particularly in the FTPP, could be a redundant virtual group). For multicasts, the node ID indicates a multicast group encompassing zero or more processing sites.

The port ID indicates a software address. A process may have multiple ports of communication. By convention, ports 0-255 are reserved for fixed system applications. Also by convention, the remaining ports are divided such that the bit field <15:8> indicates a unique task ID, and the bit field <7:0> indicates a unique socket ID for the indicated task. Thus, any given task can have a maximum of 256 ports, in addition to any of the reserved ports the task is allocated.

SEE ALSO

trailer_struct(3)

NAME

trailer_struct - ATP packet trailer structure

SYNOPSIS

#include "trailer.h"
trailer_struct trailer;

DESCRIPTION

The trailer of an ATP packet is defined by the *trailer_struct* structure. Presence of the trailer is indicated by the msg_type field in the message header. If the trailer is not present, the message terminates after the last data byte in the message. If the trailer is present, the first byte of the trailer immediately follows the last data byte in the message during message transmission. For alignment reasons, the trailer may have to be moved when stored in memory.

The *trailer_struct* structure is defined as:

```
typedef struct
{
  uword8 sig_config[4];
  uword64 seq_num;
  struct
  {
    uword64 sig;
  } sigs[4];
} trailer_struct;
```

The *sig_config* array indicates one of 255 possible signatures used to generate each of the one to four signatures in the trailer. Each element of *sig_config* corresponds to a signature in the *sigs* array. If an element in *sig_config* is 0, the corresponding signature, and all following signatures, do not exist.

The *seq_num* is the authentication sequence number attached to this packet. The sequence number is paired with the source network address (subnet and node IDs only) and should be greater than the sequence number attached to the most recently accepted packet from that host. If the sequence number is less than or equal to the sequence number on the last packet accepted from the host, the newest packet should be rejected.

Following the sequence number appears from 1 to 4 64-bit signatures. The existence of signature N ($0 \leq N \leq 3$) is indicated by non-zero entries in all elements of the *sig_config* array from 0 to N. Although the *trailer_struct* structure defines all four signatures, some of these signatures may not actually exist by the above rules. The programmer must make sure that a program does not attempt to access a signature that does not exist.

The *trailer_struct* structure contains a number of 64-bit quantities defined as uword64. These entities are actually 2-element arrays of 32 bit unsigned integers. Thus, to access an entire 64-bit quantity requires two distinct accesses to each 32-bit element (see the example below).

EXAMPLES

To read signature i:

```
trailer_struct mytrailer;
uword32 upper,lower;
```

```
int sigexist;

if (mytrailer.sig_config[i])
{
  upper = mytrailer.sigs[i].sig[0];
  lower = mytrailer.sigs[i].sig[1];
  sigexist = YES;
}
else
  sigexist = NO;
```

SEE ALSO

header_struct(3)

NAME

driver_table_struct - Table entry for configuring ATP device drivers

SYNOPSIS

#include "atp_driver.h"
driver_table_struct drivetab;

DESCRIPTION

A device driver for ATP devices is configured using a driver_table_struct. An array of pointers to driver_table_struct's is passed to the atp_init() call for configuring all devices for ATP. Each structure indicated by the array is targeted for a different driver.

The driver_table_struct is defined as:

```
     typedef struct driver_table_s
     {
       int (*init)();
       char *driverargs;
       device_table_struct *device_table[MAXNUMDEVICES];
     } driver_table_struct;
```

The *init* member indicates the initialization subroutine for the driver. The *driverargs* pointer indicates a driver-specific structure containing additional parameters for configuring the driver.

The *device_table* array is an array of pointers to device_table_struct's. These individual structs are used for configuring the individual devices under the control of the driver. If there are fewer than MAXNUMDEVICES associated with this driver, the active devices should be listed in order starting with *device_table[0]*. The last device in the list should be followed by a pointer to NULL.

SEE ALSO

atp_init(), device_table_struct(),

NAME

device_table_struct - Table entry for configuring ATP devices

SYNOPSIS

```
#include "atp_driver.h"
device_table_struct devtab;
```

DESCRIPTION

An ATP device is configured using a device_table_struct. An array of pointers to device_table_struct's is passed to the device driver in the driver's driver_table_struct. Each structure indicated by the array is targeted for a different device.

The device_table_struct is defined as:

```
typedef struct device_table_s
{
  int devnum;
  int devminor;
  char *devargs;
} device_table_struct;
```

The *devnum* parameter indicates the public name of the device; this name is generally reflected in the file devicenames.h. The *devminor* parameter indicates the private name of the device, used only within the atp_ device driver and the individual device driver. The minor device number is used so that, while the public names of devices in a system may be non-contiguous, the internal names can be contiguous. Contiguous internal names makes device array manipulation much easier.

The *devargs* parameter points to a device-specific structure containing additional parameters for configuring the device.

SEE ALSO

atp_init(), driver_table_struct()

DEVICE DRIVERS

The current selection of device drivers is defined in section 4. This section is useful for those implementing new device drivers and for those who are configuring devices in a system.

CONTENTS

NAME

lb - loopback device driver

SYNOPSIS

#include "lb.h"

DESCRIPTION

The loopback driver is used to implement a simple device for testing read and write procedures. The driver implements a single loopback device, lb0, which places any messages written to it using lb_write() into a queue to be read using lb_read().

The driver is configured using the lb_driver_args structure. The device is configured using the lb_device_args structure. Currently, these structures are empty, as there is nothing to be configured on either the driver or the device.

The lb driver responds to all standard ioctl() calls, although they are all ignored. There is no physical address associated with the loopback device. The atp_ioctl call always returns ATPSUCCESS, provided the specified device exists and is opened, and the request argument specifies a legal operation.

SEE ALSO

atp_ioctl()

NAME

udp - UDP (User Datagram Protocol) device driver

SYNOPSIS

#include "udp.h"

DESCRIPTION

The udp device driver implements a simplex subnet connecting an individual UDP socket to any other UDP socket accessible on the underlying TCP/IP network. Since the IP layer below UDP generally provides routing services, the device can communicate with other devices located on physically independent Ethernets (or whatever the underlying physical layer is).

Although the UDP protocol is normally used as a transport layer protocol, with underlying protocols for network, data-link, and physical layers, it is used as a virtual physical layer within ATP. The physical address of a udp device is the IP address and port number of the socket the device uses to communicate through UDP.

The udp driver is configured using the udp_driver_args structure, defined as:

```
typedef struct
{
  arp_struct *arp_table[MAXARPTABLELEN];
} udp_driver_args;
```

The only parameter to the driver is the initial ARP table. All udp devices share the same ARP table. The table is specified as an array of pointers to arp_struct's. The driver scans this array beginning with *arp_table[0]*, copying each entry to its internal ARP table. If the driver encounters either a null pointer in the array, or the (MAXARPTABLELEN-1)'th entry, the ARP table copy is terminated.

The individual udp devices are configured using the udp_device_args structure, defined as:

```
typedef struct
{
  udp_addr udp;
} udp_device_args;
```

The only parameter to the device is the UDP address (IP address plus UDP port number) to attach to the device. The definition of the udp_addr structure is defined in the physaddr.h file. Since the udp devices represent simplex subnets, there is only one UDP address per device.

The udp device takes its physical address at initiailization time from the udp_device_args structure. The physical address is static, and thus cannot be changed with a SETMYPHYS ioctl(). Such a call will result in a return of ATPEILLEGAL.

For now, the only way to define the ARP table for the udp driver is through the udp_driver_args structure. The ARP table cannot be manipulated after initialization. A call to ioctl() with a request of ARPADD or ARPDELETE will result in a return of ATPEILLEGAL. However, ARP table searches using ARPLOOKUP are honored by the udp driver.

SEE ALSO

atp_ioctl()

BUGS

The ARP table should be dynamically alterable.

NAME

udpr - Redundant UDP (User Datagram Protocol) device driver

SYNOPSIS

#include "udpr.h"

DESCRIPTION

The udpr device driver implements a redundant (duplex) subnet connecting two UDP sockets to other UDP sockets accessible on the underlying TCP/IP network. Since the IP layer below UDP generally provides routing services, the device can communicate with other devices located on physically independent Ethernets (or whatever the underlying physical layer is).

Although the UDP protocol is a transport layer protocol, with underlying protocols for network, data-link, and physical layers, it is used as a virtual physical layer within ATP. The physical address of a udpr device is the IP address and port number of each of the two sockets the device uses to communicate through UDP.

The udpr driver is configured using the udpr_driver_args structure, defined as:

```
typedef struct
{
  arp_struct *arp_table[MAXARPTABLELEN];
} udpr_driver_args;
```

The only parameter to the driver is the initial ARP table. All udpr devices share the same ARP table. The table is specified as an array of pointers to arp_struct's. The driver scans this array beginning with *arp_table[0]*, copying each entry to its internal ARP table. If the driver encounters either a null pointer in the array, or the (MAXARPTABLELEN-1)'th entry, the ARP table copy is terminated.

The individual udpr devices are configured using the udpr_device_args structure, defined as:

```
typedef struct
{
  udp_addr udp[2];
} udpr_device_args;
```

The only parameter to the device is the two UDP addresses (IP address plus UDP port number) to attach to the device. The definition of the udp_addr structure is defined in the physaddr.h file. Since the udpr devices represent duplex subnets, there are two UDP addresses per device. The order in which these addresses is specified is significant.

The udpr device takes its physical address at initiailization time from the udpr_device_args structure. The physical address is static, and thus cannot be changed with a SETMYPHYS ioctl(). Such a call will result in a return of ATPEILLEGAL.

For now, the only way to define the ARP table for the udpr driver is through the udpr_driver_args structure. The ARP table cannot be manipulated after initialization. A call to ioctl() with a request of ARPADD or ARPDELETE will result in a return of ATPEILLEGAL. However, ARP table searches using ARPLOOKUP are honored by the udpr driver.

SEE ALSO

atp_ioctl()

BUGS

The ARP table should be dynamically alterable.

REPORT DOCUMENTATION PAGE

Form Approved
OMB No. 0704-0188

1. AGENCY USE ONLY (*Leave blank*)

2. REPORT DATE
June 1994

3. REPORT TYPE AND DATES COVERED
Contractor Report

4. TITLE AND SUBTITLE
Advanced Information Processing System: Authentication Protocols for Network Communication

5. FUNDING NUMBERS
C NAS1-18565
WU 505-64-52-53

6. AUTHOR(S)
Richard E. Harper, Stuart J.Adams, Carol A. Babikyan,
Bryan P. Butler, Anne L. Clark, and Jaynarayan H. Lala

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)
The Charles Stark Draper Laboratory, Inc.
Cambridge, MA 02139

8. PERFORMING ORGANIZATION REPORT NUMBER

9. SPONSORING / MONITORING AGENCY NAME(S) AND ADDRESS(ES)
National Aeronautics and Space Administration
Langley Research Center
Hampton, VA 23681-0001

10. SPONSORING / MONITORING AGENCY REPORT NUMBER
NASA CR-194923

11. SUPPLEMENTARY NOTES
Langley Technical Monitor: Carl R. Elks
Final Report

12a. DISTRIBUTION / AVAILABILITY STATEMENT
Unclassified-Unlimited

Subject Category 62

12b. DISTRIBUTION CODE

13. ABSTRACT *(Maximum 200 words)*

In safety critical I/O and intercomputer communication networks, reliable message transmission is an important concern. Difficulties of communication and fault identification in networks arise primarily because the sender of a transmission cannot be identified with certainty, an intermediate node can corrupt a message without certainty of detection, and a babbling node cannot be identified and silenced without lengthy diagnosis and reconfiguration.

Authentication Protocols use digital signature techniques to verify the authenticity of messages with high probability. Such protocols appear to provide an efficient solution to many of these problems. The objective of this program is to develop, demonstrate, and evaluate inter-computer communication architectures which employ authentication. As a context for the evaluation, the authentication protocol-based communication concept was demonstrated under this program by hosting a real-time flight critical Guidance, Navigation and Control algorithm on a distributed, heterogeneous, mixed redundancy system of workstations and embedded fault-tolerant computers.

14. SUBJECT TERMS
Fault-Tolerant, Real-Time, Computer Networks

15. NUMBER OF PAGES
102

16. PRICE CODE

17. SECURITY CLASSIFICATION OF REPORT
Unclassified

18. SECURITY CLASSIFICATION OF THIS PAGE
Unclassified

19. SECURITY CLASSIFICATION OF ABSTRACT

20. LIMITATION OF ABSTRACT