FLIP: an Internetwork Protocol for Supporting Distributed Systems

M. Frans Kaashoek
*Robbert van Renesse**
Hans van Staveren
Andrew S. Tanenbaum

Vrije Universiteit
Amsterdam, The Netherlands

ABSTRACT

Most modern network protocols give adequate support for traditional applications such as file transfer and remote login. Distributed applications, however, have different requirements (e.g., efficient at-most-once remote procedure call even in the face of processor failures). Instead of using ad-hoc protocols to meet each of the new requirements, we have designed a new protocol, called the Fast Local Internet Protocol (FLIP), that provides a clean and simple integrated approach to these new requirements. FLIP is an unreliable message protocol that provides both point-to-point communication and multicast communication, and requires almost no network management. Furthermore, by using FLIP we have simplified higher-level protocols such as remote procedure call and group communication, and enhanced support for process migration and security. A prototype implementation of FLIP has been built as part of the new kernel for the Amoeba distributed operating system, and is in daily use. Measurements of its performance are presented.

1. INTRODUCTION

Most network protocols are designed to support a reliable bit stream between a single sender and a single receiver. For applications such as remote login sessions or bulk file transfer these protocols are adequate. However, distributed operating systems have special requirements such as achieving transparency, specific remote procedure

* Current address: Dept. of Computer Science, Upson Hall, Cornell University, Ithaca, New York.

call (RPC) semantics even in the face of processor crashes, group communication, security, network management, and wide-area networking. Furthermore, applications on distributed operating systems often use a complex local internetwork of communication subsystems including Ethernets, high-speed multiprocessor buses, hypercubes, and optical fibers. These kinds of communication are not well supported by protocols such as TCP/IP, X.25, and OSI TP4.

As part of our ongoing research on the Amoeba distributed operating system, we have designed, implemented, and evaluated a new internet protocol that, in many respects, is better suited for distributed computing than existing protocols. This new protocol, called FLIP (Fast Local Internet Protocol), is the subject of this paper.

Although the ISO OSI protocols are not widely used, the OSI model is convenient for describing where functionality can be put in a protocol hierarchy [Zimmerman 1980]. In Figure 1, we show the OSI model, along with TCP/IP and FLIP protocol hierarchies. Very briefly, FLIP is a connectionless (datagram) protocol, roughly analogous to IP, but with increased functionality and specifically designed to support a high-performance RPC protocol rather than a byte-stream protocol like TCP or OSI TP4.

Level	**OSI**	**TCP/IP**	**FLIP**
7	Application	User-defined	User-defined
6	Presentation	User-defined	Amoeba Interface Language (AIL)
5	Session	Not used	RPC and Group communication
4	Transport	TCP or UDP	Not needed
3	Network	IP	FLIP
2	Data Link	E.g., Ethernet	E.g., Ethernet
1	Physical	E.g., Coaxial cable	E.g., Coaxial cable

Fig. 1. Layers of functionality in OSI, TCP/IP, and FLIP.

The outline of the rest of this paper is as follows. In Section 2 we will describe the requirements that a distributed operating system places on the underlying protocol. In Section 3 we will discuss the FLIP service definition; that is, what FLIP provides. In Section 4 we will discuss the interface between FLIP and higher layers. In Section 5 we will discuss the protocol itself. In Section 6 we will discuss how FLIP can be implemented. In Section 7 we present measurements we have made of its performance. In Section 8 we will compare it to related work. Finally, in Section 9 we will draw our conclusions. The appendix describes the protocol itself in detail.

2. DISTRIBUTED SYSTEM REQUIREMENTS

Distributed systems place different requirements on the operating system than do traditional network systems. Network systems run all of a user's applications on a single workstation. Workstations run a copy of the complete operating system; the only thing that is shared is the file system. Applications are sequential; they make no use of any available parallelism. In such an environment, file transfer and remote login are the two basic applications that the communication mechanisms in the operating system must support. In a distributed system the situation is radically different. A user process may run anywhere in the system, to allow efficient sharing of computing cycles. Applications are rewritten to take advantage of the available parallelism. For example, distributed systems can provide a version of the UNIX† *make* program that allows compilations to run in parallel. Other applications may be rewritten to provide fault tolerance by using the redundancy of hardware. In such an environment file transfer is only one of the many applications that depend on the communication mechanisms provided by the operating system.

In this section, we will investigate the requirements for communication in a distributed system and outline the approach taken by FLIP. We identify six requirements: transparency, remote procedure call, group communication, security, network management, and wide-area networking. We discuss each of these requirements in turn. It should be noted that many existing network and distributed systems meet all or a subset of the requirements, but in this paper we argue that the implementation of these systems can often be simplified by using a better network protocol.

Transparency

An important goal for distributed systems, such as Amoeba [Tanenbaum et al. 1990; Mullender et al. 1990], Chorus [Rozier et al. 1988], Clouds [Dasgupta et al. 1991], Sprite [Ousterhout et al. 1988], and V [Cheriton 1988b], is transparency. Distributed systems are built from a large number of processors connected by LANs, buses, and other communication media. No matter where a process runs, it should be able to communicate with any other process in the system using a single mechanism that is independent of where the processes are located. The communication system must be able to route messages along the ‘‘best’’ route from one process to another. For example, if two processes can reach each other through a LAN and high-speed bus, the communication system should use the bus. The users, however, should not have to specify which route is taken.

Most communication protocols do not provide the transparency that is required

† UNIX is a registered trademark of UNIX System Laboratories, Inc.

by applications running on a distributed system. Addresses in these protocols identify a host instead of a process. Once a process is started on a machine, it is tied to that machine. For example, if the process is migrated to another processor, the process has to inform its communication partners that it has moved. To overcome such problems, distributed systems require that an address identifies a *process*, not a *host*.

Remote Procedure Call

Distributed operating systems are typically structured around the client-server paradigm. In this model, a user process, called the *client*, requests another user process, called the *server*, to perform some work for it by sending the server a message and then blocking until the server sends back a reply. The communication mechanism used to implement the client-server model is called RPC [Birrell and Nelson 1984].

The RPC abstraction lets the programmer think in terms of normal procedure calls, which are well understood and have been around for a long time. This is in sharp contrast with, for example, the ISO OSI model. In this model, communication is treated as an input/output device, with user primitives for sending messages and getting indications of message arrivals. Many people think that input/output should not be the central abstraction of a modern programming language. Therefore, most distributed system builders, language designers, and programmers prefer RPC.

Group Communication

Although RPC is a good abstraction for the request/reply type of communications, there is a large body of applications that require a group of several processes to interact closely. Group communication allows a message to be sent reliably from 1 sender to *n* receivers. Many applications profit from such a communication primitive. For example, applications may replicate data to achieve fault tolerance. Such applications can profit from group communication to keep the replicated data consistent [Birman and Joseph 87]. Another way of using group communication is in building efficient distributed shared memory [Tanenbaum et al. 1992]. Interestingly enough, many networks provide mechanisms to do broadcast or multicast at the data-link layer. For example, Ethernet and some token rings, two commonly used LANs, both provide broadcast and multicast. Future networks, like Gigabit LANs, are also likely to implement multicasting or broadcasting to support high-performance applications such as multimedia [Kung 1992]. Communication protocols, however, often hide these useful capabilities from the applications. Although broadcast can be done by sending *n* point-to-point messages and waiting for *n* acknowledgements, this algorithm is inefficient and wastes bandwidth. Therefore many researchers have proposed other algorithms that use data-link broadcast to implement reliable broadcast efficiently.

One of the difficulties in making a protocol that allows user applications to use the data-link broadcast or multicast capability of a network is routing. A group address has to be mapped on one or more data-link addresses, possibly on different networks. The protocol has to make sure that messages will not loop and that a minimum number of messages are used to transmit user data to the group. Groups may change over time, so routing tables have to be dynamically updated. Furthermore, to achieve good performance, the routing protocol should use a data-link multicast address to send a message to a number of receivers whenever possible.

Security

Although security cannot be provided by a communication protocol alone, a good protocol can provide mechanisms to build a secure, yet efficient distributed system. With current protocols, addresses can often be faked, making it possible for a process to impersonate an important service. For example, in many systems a user process can impersonate the file server once it knows the address of the file server (which is typically public knowledge). Most protocols do not provide any support for encryption of data. Users must decide whether or not to use encryption. Once they have decided to do so, they have to encrypt every message, even if both source and destination are located in the same secure room. A protocol provides much better performance by avoiding encryption if it knows a network is trusted, and using encryption if the network is not trusted.

Network Management

In an environment with many processors and networks, it often happens that a processor has to be taken down for maintenance or a network has to be reconfigured. With current software, reconfiguring a network typically requires manual intervention by a system administrator to assign new network numbers and to update the configuration files. Furthermore, taking some machines down often introduces communication failures for the rest of the machines. Ideally, a protocol makes it possible that network management can be done without any manual intervention.

Wide-Area Networking

Most processes in a distributed system communicate with services that are located nearby. For example, to read a file, users normally do an RPC with their local file server and not with a file server in another domain on another continent. Although communication with another domain must be possible, it should not introduce a performance loss for the more common, local case.

Why a New Protocol?

None of the current protocols addresses the requirements for distributed systems and applications adequately. The TCP and OSI protocols are connection-oriented and require a setup before any message can be sent. In a distributed system, processes are often short-lived and perform mostly small RPCs. In such an environment the time spent in setting up a connection is wasted. Indeed, almost none of the current RPC implementations are based on connections. Although IP is a connectionless protocol, it still has some serious disadvantages. Because addresses in IP identify hosts instead of processes, systems based on IP are less transparent, making certain functionality, such as process migration, harder to implement.

To meet the distributed system requirements using IP, a new protocol was invented for each subset of requirements. For example, The Internet Control Message Protocol (ICMP) has been introduced to implement dynamic routing and to cope partially with network changes [Postel 1981b]. The Address Resolution Protocol (ARP) has been introduced to map IP addresses on data-link addresses [Plummer 1982]. The Reverse Address Resolution Protocol (RARP) has been introduced to acquire an IP address [Finlayson et al. 1984]. Internet Group Management Protocol (IGMP) has been introduced to implement group communication [Deering 1988]. The Versatile Message Transport Protocol (VMTP) has been introduced to meet the requirements for group communication and a secure, efficient, and at-most-once RPC protocol [Cheriton 1986].

A big advantage of this approach is that one can adjust to new requirements without throwing away existing software. However, it is sometimes better to start from scratch. The main contribution of this paper is a protocol (FLIP) that addresses these requirements in a clean, simple, and integrated way. The following FLIP properties allow us to achieve the requirements:

1. FLIP identifies entities with a location-independent 64-bit identifier. An entity can, for example, be a process.
2. FLIP uses a one-way mapping between the ‘‘private’’ address, used to register an endpoint of a network connection, and the ‘‘public’’ address used to advertise the endpoint.
3. FLIP routes messages based on the 64-bit identifier.
4. FLIP discovers routes on demand.
5. FLIP uses a bit in the message header to request transmission of sensitive messages across trusted networks.

In the next sections we will present FLIP, discuss our experience using it, and its performance in the Amoeba distributed system. FLIP is the basis for all communication within Amoeba and is in day to day use.

3. FLIP SERVICE DEFINITION

FLIP is a connectionless protocol that is designed to support transparency, efficient RPC, group communication, secure communication, and easy network management. This section describes the services that FLIP delivers.

Communication takes place between *Network Service Access Points* (NSAPs), which are addressed by 64-bit numbers. NSAPs are location-independent, and can move from one node to another (possibly on different physical networks), taking their addresses with them. Nodes on an internetwork can have more than one NSAP, typically one or more for each entity (e.g., process). FLIP ensures that this is transparent to its users. FLIP messages are transmitted unreliably between NSAPs and may be lost, damaged, or reordered. The maximum size of a FLIP message is $2^{32}-1$ bytes. As with many other protocols, if a message is too large for a particular network, it will be fragmented into smaller chunks, called *fragment*s. A fragment typically fits in a single network *packet*. The reverse operation, re-assembly, is (theoretically) possible, but receiving entities have to be able to deal with fragmented messages.

The address space for NSAPs is subdivided into 256 56-bit address spaces, requiring 64 bits in all. The null address is reserved as the broadcast address. In this paper we will define the semantics of only one of the address spaces, called the *standard space*, and leave the others undefined. Later these other address spaces may be used to add additional services.

The entities choose their own NSAP addresses at random (i.e., stochastically) from the standard space for four reasons. First, it makes it exceedingly improbable that an address is already in use by another, independent NSAP, providing a very high probability of uniqueness. (The probability of two NSAPs generating the same address is much lower than the probability of a person configuring two machines with the same address by accident.) Second, if an entity crashes and restarts, it chooses a new NSAP address, avoiding problems with distinguishing reincarnations (which, for example, is needed to implement at-most-once RPC semantics). Third, forging an address is hard, which, as we will see, is useful for security. Finally, an NSAP address is location-independent, and a migrating entity can use the same address on a new processor as on the old one.

Each physical machine is connected to the internetwork by a *FLIP box*. The FLIP box can either be a software layer in the operating system of the host, or be run on a separate communications processor. A FLIP box consists of several modules. An example of a FLIP box is shown in Figure 2.

The *packet switch* is the heart of the FLIP box. It transfers FLIP fragments in

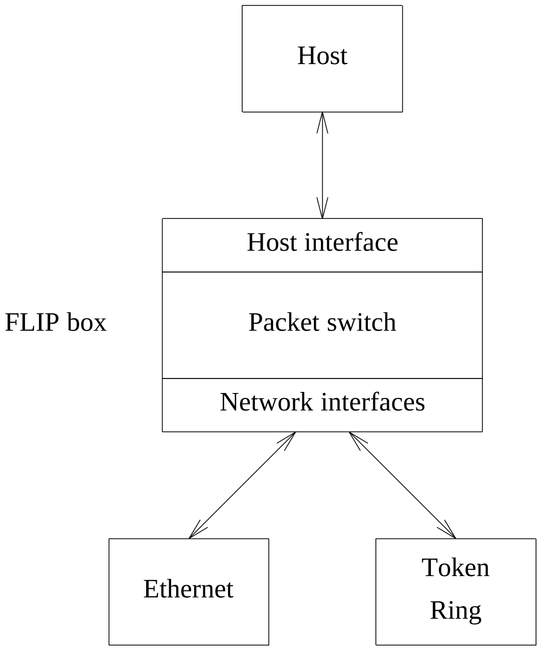


Fig. 2. A FLIP box consists of an host interface, packet switch, and network interfaces.

packets between physical networks, and between the host and the networks. It maintains a dynamic hint cache mapping NSAP addresses on data-link addresses, called the *routing table*, which it uses for routing fragments. As far as the packet switch is concerned, the attached host is just another network. The *host interface* module provides the interface between the FLIP box and the attached host (if any). A FLIP box with one physical network and an interface module can be viewed as a traditional network interface. A FLIP box with more than one physical network and no interface module is a router in the traditional sense.

4. THE HOST INTERFACE

In principle, the interface between a host and a FLIP box can be independent of the FLIP protocol, but for efficiency and simplicity, we have designed an interface that is based on the FLIP protocol itself. The interface consists of seven downcalls (for outgoing traffic) and two upcalls (for incoming traffic), as shown in Figure 3.

An entity allocates an entry in the interface by calling *flip_init*. The call allocates an entry in a table and stores the pointers for the two upcalls in this table. Furthermore, it stores an identifier used by higher layers. An allocated interface is removed by calling *flip_end.*

By calling *flip_register* one or more times, an entity registers NSAP addresses

Routine	**Description**
Flip_init(ident, receive, notdeliver) → ifno	Allocate an entry in the interface
Flip_end(ifno)	Close entry in the interface
Flip_register(ifno, Private-Address) → EP	Listen to address
Flip_unregister(ifno, EP)	Remove address
Flip_unicast(ifno, msg, flags, dst, EP, length)	Send a message to *dst*
Flip_multicast(ifno, msg, flags, dst, EP, length, ndst)	Send a multicast message
Flip_broadcast(ifno, msg, EP, length, hopcnt)	Broadcast *msg* hopcnt hops
Receive(ident, fragment description)	Fragment received
Notdeliver(ident, fragment description)	Undelivered fragment received

Fig. 3. Interface between host and packet switch. A fragment description contains the data, destination and source, message identifier, offset, fragment length, total length, and flags of a received fragment (see next section).

with the interface. An entity can register more than one address with the interface (e.g., its own address to receive messages directed to the entity itself and the null address to receive broadcast messages). The address specified, the *Private-Address,* is not the (public) address that is used by another entity as the destination of a FLIP message. However, public and private addresses are related using the following function on the low-order 56 bits:

Public-Address = One-Way-Encryption(Private-Address)

The One-Way-Encryption function generates the Public-Address from the Private-Address in such a way that one cannot deduce the Private-Address from the Public-Address. Entities that know the (public) address of an NSAP (because they have communicated with it) are not able to receive messages on that address, because they do not know the corresponding private address. Because of the special function of the null address, the following property is needed:

One-Way-Encryption(Address) = 0 if and only if Address = 0

The One-Way-Encryption function is currently defined using DES [National Bureau of Standards 1977]. If the 56 lower bits of the Private-Address are null, the Public-Address is defined to be null as well. The null address is used for broadcasting, and need not be encrypted. Otherwise, the 56 lower bits of the Private-Address are used as a DES key to crypt a 64-bit null block. If the result happens to be null, the result is again encrypted, effectively swapping the result of the encrypted null address with the encrypted address that results in the null address. The remaining 8 bits of the Private-Address, concatenated with the 56 lower bits of the result, form the Public-Address.

Flip_register encrypts a Private-Address and stores the corresponding Public-Address in the routing table of the packet switch. A special flag in the entry of the routing table signifies that the address is local, and may not be removed (as we will see in Section 5). A small EP-identifier (End Point Identifier) for the entry is returned. Calling *flip_unregister* removes the specified entry from the routing table.

There are three calls to send an arbitrary-length message to a Public-Address. They differ in the number of destinations to which *msg* is sent. None of them guarantee delivery. *Flip_unicast* tries to send a message point-to-point to one NSAP. *Flip_multicast* tries to send a message to at least *ndst* NSAPs. *Flip_broadcast* tries to send a message to all NSAPs within a virtual distance *hopcnt*. If a message is passed to the interface, the interface first checks if the destination address is present in the routing table and if it thinks enough NSAPs are listening to the destination address. If so, the interface prepends a FLIP header to the message and sends it off. Otherwise, the interface tries to locate the destination address by broadcasting a LOCATE message, as explained in the next section. If sufficient NSAPs have responded to the LOCATE message, the message is sent away. If not, the upcall *notdeliver* will be called to inform the entity that the destination could not be located. When calling one of the send routines, an entity can also set a bit in *flags* that specifies that the destination address should be located, even if it is in the routing table. This can be useful, for example, if the RPC layer already knows that the destination NSAP has moved. Using the *flags* parameter the user can also specify that security is necessary.

When a fragment of a message arrives at the interface, it is passed to the appropriate entity using the upcall *receive*.

This interface delivers the bare bones services that are needed to build higher-level protocols, such as RPC. Given the current low error-rates of networks, we decided not to guarantee reliable communication at the network level, to avoid duplication of work at higher levels [Saltzer et al. 1986]. Higher-level protocols, such as RPC, send acknowledgement messages anyway, so given the fact that networks are very reliable it is a waste of bandwidth to send acknowledgement messages at the FLIP level as well. Furthermore, users will never call the interface directly, but use RPC or group communication.

5. THE FLIP PROTOCOL

A FLIP box implements unreliable message communication between NSAPs by exchanging FLIP fragments and by updating the routing table when a fragment arrives. In this section, we will describe the layout of a FLIP fragment and tell how the routing table is managed.

5.1. The FLIP Fragment Format

Similar to fragments in many other protocols, a FLIP fragment is made up of two parts: the FLIP header and the data. The general format of a FLIP header is depicted in Figure 4. A header consists of a 40-byte fixed part and a variable part. The fixed part of the header contains general information about the fragment. The *Actual Hop Count* contains the weight of the path from the source. It is incremented at each FLIP box with the weight of the network over which the fragment will be routed. If the *Actual Hop Count* exceeds the *Maximum Hop Count*, the fragment will be discarded. The *Reserved* (*Res*.) field is reserved for future use.

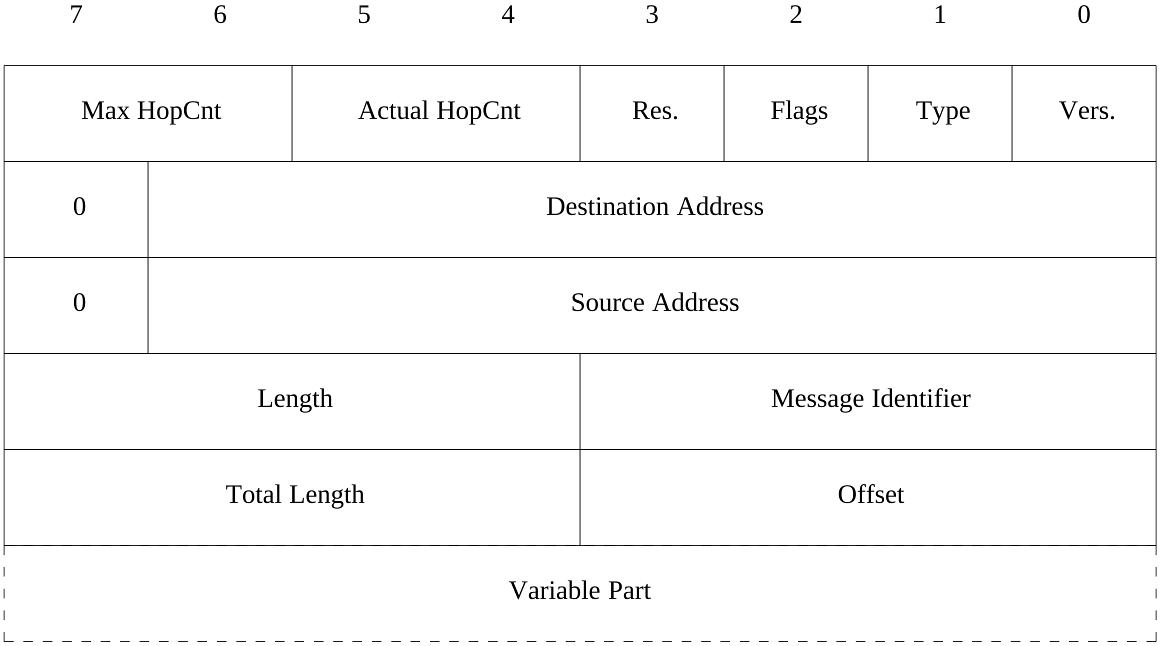


Fig. 4. General format of a FLIP fragment.

The *Flags* field contains administrative information about the fragment (see Fig. 5). Bits 0, 1, and 2 are specified by the sender. If bit 0 is set in *Flags*, the integer fields (hop counts, lengths, Message Identifier, Offset) are encoded in big endian (most significant byte first), otherwise in little endian [Cohen 1981]. If bit 1 is set in *Flags*, there is an additional section right after the header. This *Variable Part* contains *parameters* that may be used as hints to improve routing, end-to-end flow control, encryption, or other, but is never necessary for the correct working of the protocol. Bit 2 indicates that the fragment must not be routed over untrusted networks. If fragments only travel over trusted networks, the contents need not be encrypted. Each system administrator can switch his own network interfaces from trusted to untrusted or the other way around.

Bit	Name	Cleared	Set
0	Endian	Little endian	Big endian
1	Variable Part	Absent	Present
2	Security	Not required	Don’t route over untrusted networks
3	Reserved		
4	Unreachable	Location unknown	Can’t route over trusted networks only
5	Unsafe	Safe	Routed over untrusted network(s)
6	Reserved		
7	Reserved		

Fig. 5. Bits (4 input and 4 output) in the *Flags* field.

Bits 4 and 5 are set by the FLIP boxes (but never cleared). Bit 4 is set if a fragment that is not to be routed over untrusted networks (bit 2 is set) is returned because no trusted network was available for transmission. Bit 5 is set if a fragment was routed over an untrusted network (this can only happen if the *Security* bit, bit 2, was not set). Using bits 2, 4, and 5 in the *Flags* field, FLIP can efficiently send messages over trusted networks, because it knows that encryption of messages is not needed.

The *Type* field in the FLIP header describes which of the (six) messages types this is (see below). The *Version* field describes the version of the FLIP protocol; the version described here is 1. The *Destination Address* and the *Source Address* are addresses from the standard space and identify, respectively, the destination and source NSAPs. The null *Destination Address* is the broadcast address; it maps to all addresses. The *Length* field describes the total length in bytes of the fragment excluding the FLIP header. The *Message Identifier* is used to keep multiple fragments of a message together, as well as to identify retransmissions if necessary. *Total Length* is the total length in bytes of the message of which this fragment is a part, with *Offset* the byte offset in the message. If the message fits in a single fragment, *Total length* is equal to *Length* and *Offset* is equal to zero.

The *Variable Part* consists of the number of bytes in the *Variable Part* and a list of parameters. The parameters are coded as byte (octet) strings as follows:

Bytes	0	1	2	…	Size+1
	Code	Size			

The (non-zero) *Code* field gives the type of the parameter. The *Size* field gives the size of the data in this parameter. Parameters are concatenated to form the complete *Variable Part*. The total length of the *Variable Part* must be a multiple of four bytes, if necessary by padding with null bytes.

5.2. The FLIP Routing Protocol

The basic function of the FLIP protocol is to route an arbitrary-length message from the source NSAP to the destination NSAP. In an internetwork, destinations are reachable through any one of several routes. Some of these routes may be more desirable than others. For example, some of them may be faster, or more secure, than others. To be able to select a route, each FLIP box has information about the networks it is connected to.

In the current implementation of FLIP, the routing information of each network connected to the FLIP box is coded in a *network weight* and a *secure flag*. A low network weight means that the network is desirable to forward a fragment on. The network weight can be based, for example, on the physical properties of the network such as bandwidth and delay. Each time a fragment makes a hop from one FLIP box to another FLIP box its *Actual Hop Count* is increased with the weight of the network over which it is routed (or it is discarded if its *Actual Hop Count* becomes greater than its *Maximum Hop Count*)†. A more sophisticated network weight can be based on the type of the fragment, which may be described in the *Variable Part* of the header. The *secure* flag indicates whether sensitive data can be sent unencrypted over the network or not.

At each FLIP box a message is routed using information stored in the routing table. The routing table is a cache of hints of the form:

(Address, Network, Location, Hop Count, Trusted, Age, Local)

Address identifies one or more NSAPs. *Network* is the hardware-dependent network interface on which *Address* can be reached (e.g., Ethernet interface). *Location* is the data-link address of the next hop (e.g., the Ethernet address of the next hop). *Hop*

† *Hop Count* is a misnomer, but it is maintained for historical reasons.

Count is the weight of the route to *Address*. *Trusted* indicates whether this is a secure route towards the destination, that is, sensitive data can be transmitted unencrypted. *Age* gives the age of the tuple, which is periodically increased by the FLIP box. Each time a fragment from *Address* is received, the *Age* field is set to 0. *Local* indicates if the address is registered locally by the host interface. If the *Age* field reaches a certain value and the address is not local, the entry is removed. This allows the routing table to forget routes and to accommodate network topology changes. The *Age* field is also used to decide which entries can be purged, if the routing table fills up.

The FLIP protocol makes it possible for routing tables to automatically adapt to changes in the network topology. The protocol is based on six message types (see Fig. 6). The precise protocol is given in the Appendix; here we will give a short description. If a host wants to send a message to a FLIP address that is not in its routing table, it tries to locate the destination by broadcasting a LOCATE message‡. LOCATE messages are propagated to all FLIP boxes until the *Actual Hop Count* becomes larger than the *Maximum Hop Count*. If a FLIP box has the destination address in its routing table, it sends back an HEREIS message in response to the LOCATE. User data is transmitted in UNIDATA or in MULTIDATA messages. UNIDATA messages are used for point-to-point communication and are forwarded through one route to the destination. MULTIDATA messages are used for multicast communication and are forwarded through routes to all the destinations. If a network supports a multicast facility, FLIP will send one message for all destinations that are located on the same network. Otherwise, it will make a copy for each location in the routing table and send point-to-point messages.

If a FLIP box receives a UNIDATA message with an unknown destination, it turns the message into a NOTHERE message and sends it back to the source. If a FLIP box receives a UNIDATA message that should not be routed over untrusted networks (as indicated by the *Security* bit), and that cannot be routed over trusted networks, it turns the message into an UNTRUSTED message and sends it back to the source just like a NOTHERE message. Moreover, it sets the *Unreachable* bit in the message (regardless of its current value). For a message of any other type, including a MULTIDATA message, if the *Security* bit is set, and the message cannot be routed over trusted networks, it is simply dropped. If, for a NOTHERE or a UNTRUSTED message, a FLIP box on the way back knows an alternative route, it turns the message back into a UNIDATA message and sends it along the alternative route. If, for a NOTHERE message, no FLIP box knows an alternative route, the message is returned to the source NSAP and each FLIP box removes information about this route from the routing table.

LOCATE messages must be used with care. They should be started with a *Maximum Hop Count* of one, and incremented each time a new locate is done. This limits

‡ We assume that a network has a broadcast facility. For networks that do not have such a facility, we are considering adding a name server.

Type	**Function**
LOCATE	Find network location of NSAP
HEREIS	Reply on LOCATE
UNIDATA	Send a fragment point-to-point
MULTIDATA	Multicast a fragment
NOTHERE	Destination NSAP is unknown
UNTRUSTED	Destination NSAP cannot be reached over trusted networks

Fig. 6. FLIP message types.

the volume of broadcasts needed to locate the destination. Even though the hop counts are a powerful mechanism for locating a destination and for finding the best route, if routing tables become inconsistent, LOCATE messages may flood the internetwork (e.g., if a loop exists in the information stored in the routing tables in the internetwork). To avoid this situation, each FLIP box maintains, in addition to its routing table, a cache of (Source Address, Message Identifier, Offset, Destination Network, Location) tuples, with a standard timeout on each entry. For each received broadcast message, after updating the routing table, it checks whether the tuple is already in the cache. If not, it is stored there. Otherwise, the timeout is reset and the message is discarded. This avoids broadcast messages flooding the network if there is a loop in the network topology.

To illustrate how the FLIP box works, let us look at an example of how the RPC layer sends a point-to-point message to another process in the network topology depicted in Figure 7. The topology consists of three machines. It is not very realistic, but it allows us to explain some important properties of FLIP using a very simple internetwork. When a FLIP box boots, it reads information about its configuration from a table (i.e., the type of networks it is connected to and information about these networks, such as the maximum packet size). This information tells the machine how many interfaces it has, the type of the interfaces, and some network dependent information, such as the weight of the network and whether the network has a multicast facility. After a FLIP box is initialized, it starts running with an empty routing table.

The example network topology contains two network types: a VME-bus and an Ethernet. Because a VME-bus is faster than an Ethernet, the weight given to the VME-bus is lower than the weight given to the Ethernet. Every FLIP box is reachable from another host through different routes. There is, for example, a path of weight 1 from *A* to *B*, but also a path of weight 4 (from *A* to *C* over the Ethernet and then from *C* to *B* over the VME-bus).

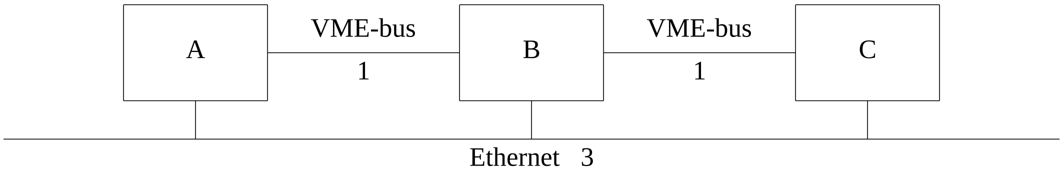


Fig. 7. An example network topology. FLIP box *A* and *C* both have two network interfaces: one for the VME-bus and one for the Ethernet. FLIP box *B* has 3 network interfaces: two to the VME-bus and one to the Ethernet. The VME-bus has weight 1 and the Ethernet has weight 3.

Let us now consider the case that the RPC layer sends a message from process P_1 on host A to process P_2 running on host *B*. When both processes start, the RPC layers register the FLIP addresses for the processes with their own FLIP box. The RPC layer of P_1 sends a message by calling *flip_unicast* with the public address of P_2 as the destination address (we assume that P_1 knows the public address of P_2). Because the address of P_2 is not initially present in the routing table of *A*, *A* buffers the message and starts to locate P_2 by sending a LOCATE message with *Max Hop Count* set to 1. *A*'s FLIP box will forward this message on the VME-bus, but not on the Ethernet, because to forward a message across the Ethernet the *Maximum Hop Count* must be at least 3. When the LOCATE message arrives at *B*, the FLIP address of P_1 will be entered in *B*'s routing table along with the weight of the route to P_1, the VME-bus address of *A*, and the network interface on which *A* is reachable. Because the public address of P_2 is registered with *B*'s routing table, *B* will return an HEREIS message. When the HEREIS message arrives at *A*, *A* enters P_2's public address in its routing table and sends the message that is waiting to be sent to P_2. Lower layers in the FLIP box will cut the message in fragments, if necessary. *B* receives the message for P_2 from the VME-bus and will forward it to the RPC layer of P_2 by calling *receive*. From now on, the routes to both P_1 and P_2 are known to *A* and *B*, so they can exchange messages without having to locate each other.

Now, assume that P_2 migrates to host *C*. The RPC layer unregisters the address at host *B* and registers it at host *C*. Thus, P_2 has removed its address from *B*'s routing table and has registered it with C's routing table. The next FLIP UNIDATA message of a message that arrives at *B* from *A*, will be returned to *A* as a FLIP NOTHERE message, because the address of P_2 is not present in *B*'s routing table. When *A* receives the NOTHERE message, it will invalidate the route to P_2. As *A* does not know an alternative route with the same or less weight to P_2, it will pass the NOTHERE message to the interface. The interface forwards the message to P_1's RPC layer by calling *notdeliver*. P_1's RPC layer can now retransmit the message by calling *flip_unicast* again. As the route to P_2 has been invalidated, the interface will buffer the message and start by locating P_2 with *Max Hop Count* set to 1. After a timeout it will locate with *Max Hop Count* set to 2. Then, it will find a route to P_2: a hop across the VME-bus to *B* and another hop across the VME-bus from *B* to *C*. It will enter this new route with weight 2 in its routing table and forward the message across the VME-bus to *B*. When *B* receives the message, it will forward the message to *C*. From then on, P_1 and P_2 can exchange messages without locating each other again.

If the topology changes and, for example, *A* is disconnected from the VME-bus, the route to P_2 in *A*'s routing table will be removed after a period of time, because no messages will arrive via the VME-bus and therefore the age field of the entry in the routing table will reach the value that causes it to be removed. If P_1 then tries to send a message to P_2, the interface will again start by locating P_2's public address. This time it will find a route with weight 3; one hop across the Ethernet. If P_1 sends a new message before the route to P_2 is purged from *A*'s routing table, A will forward the message across the VME-bus and the message will be lost (assuming that the driver for the VME-bus does not return any error). In this case, the RPC layer has to tell the interface explicitly (using the *flags* parameter of flip_unicast) to purge the routing table entry. It does this, for example, if it did not receive an acknowledgement after a number of retrials.

Finally, assume that instead of *A*, *B* is disconnected from the VME-bus. A will first use its route with weight 2 and send the message to B across the VME-bus. If B does not know yet that the route over the VME-bus to C disappeared, it will forward the message over the VME-bus and the message will be lost. Otherwise, it will send the message as a NOTHERE message back to A, because the *Max Hop Count* is set by *A* to 2. In both cases, *A* will send the message to *C* using the Ethernet, possibly after doing another locate.

6. USING FLIP UNDER AMOEBA

FLIP is the basis for all communication within the Amoeba distributed system [Tanenbaum et al. 1990; Mullender et al. 1990]. The configuration at the Vrije Universiteit is depicted in Figure 8. The pool processors, the I80486 router, and the specialized servers run the Amoeba kernel. The workstations and the SPARC router run either Amoeba or a version of UNIX containing a FLIP driver, so UNIX and Amoeba processes can communicate transparently. All the 70 machines are connected through 3 Ethernets and the processors in the MC68030 pool are also connected by VME-buses. We also implemented FLIP accross TCP/IP and UDP/IP, so that we can use TCP/IP connections as a data link. This implementation is the basis for a small scale WAN project that connects multiple sites in The Netherlands, and has been tested across the Atlantic as well.

The Amoeba software consists of two pieces: a microkernel, which runs on every

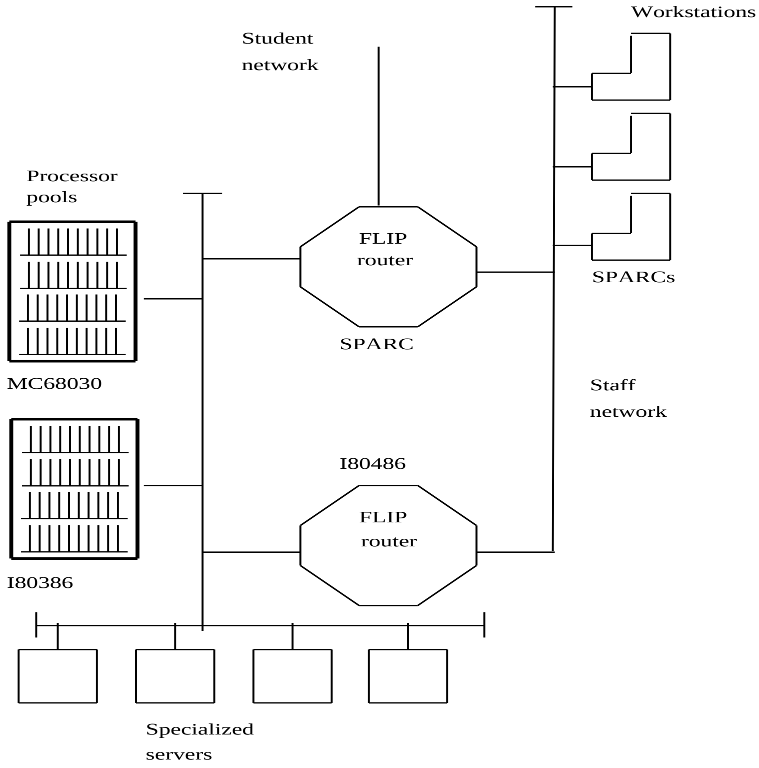


Fig. 8. The FLIP internetwork at the Vrije Universiteit. It contains three different machine architectures with different endianness and two types of networks: Ethernet and VME-buses. On average ten people are using the Amoeba system every day to develop system software, and distributed and parallel applications.

processor, and a collection of servers that provide most of the traditional operating system functionality. Besides process management, memory management, and low-level I/O, the Amoeba kernel provides interfaces for two communication protocols: RPC and group communication (see Fig. 9) [Kaashoek and Tanenbaum 1991]. Both protocols use the FLIP box interface to send and receive messages. We will now describe how FLIP meets each of the distributed system requirements listed in Section 2.

Transparency

The primary goal of Amoeba is to build a transparent distributed operating system. To the average user, Amoeba looks like a traditional timesharing system. The difference is that each command typed by the user makes use of multiple machines spread around the network. The machines include process servers, file servers, directory servers, compute servers, and others, but the user is not aware of any of this. At the terminal, it just looks like an ordinary time sharing system.

An important distinction between Amoeba and other distributed systems is that

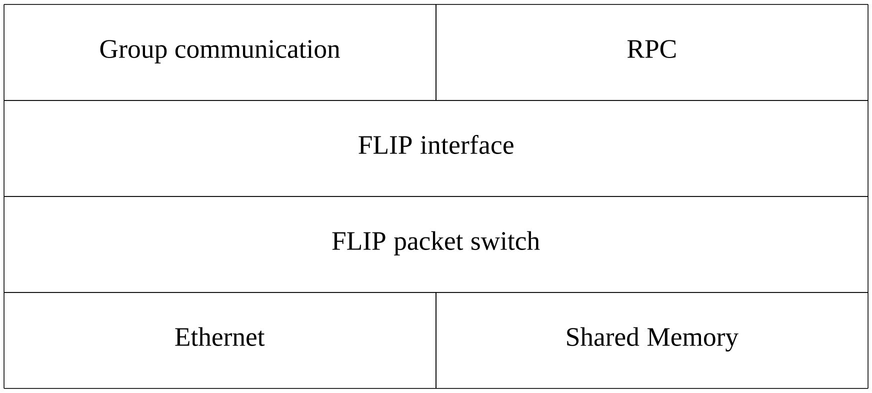


Fig. 9. Communication layers in the Amoeba kernel for a pool processor.

Amoeba is not based on the workstation model of distributed computing, but on a processor pool model. When a users logs in, it is to the system as a whole, not to a specific machine. Machines do not have owners. As commands are started up, in general they do not run on the same machine as the shell. Instead the system automatically looks around for approximately the most lightly loaded host to run each new command on. Thus, all resources belong to the system as a whole, and are managed by it. They are not dedicated to specific users, except for short periods of time to run individual processes. This model attempts to give the user complete transparency

To achieve this degree of transparency a two-level naming scheme is used: capabilities and FLIP addresses. Each object (e.g., a file) is named by a *capability* [Tanenbaum et al. 1986]. Associated with each object type is a service (a single process or a group of processes) that manages the object. When a client wants to perform an operation on an object, it sends a request message to the service that manages the object. The service is addressed by a *port* that is part of the capability. In short, capabilities are persistent names that identify objects.

To make capabilities easy to use, users can register them with the directory service. The directory service allows users to register capabilities under an ASCII string. Futhermore, it also implements a UNIX-like access protection scheme.

Within the kernel, ports are mapped onto one or more FLIP addresses, one for each server. When a client wants to perform an operation on an object, it provides the kernel with the capability of the object. The kernel extracts the port from the capability and looks in its port cache for a FLIP address that listens to the port. Using the FLIP address, the kernel sends messages, relying on the FLIP box to deliver the messages to the right location. If there is no mapping from port to FLIP address in the cache, the kernel uses *flip_broadcast* to locate the port. The FLIP addresses of the responders to the LOCATE request are stored with the port in the port cache to avoid future locates.

This locate procedure has the important side effect that at the same time the FLIP boxes build up their routing tables, so a second locate at the FLIP level is avoided. In the common case that networks do not change rapidly and processes migrate infrequently, no LOCATE messages are sent.

At-Most-Once RPC

The RPC layer in the Amoeba kernel provides an interface for at-most-once RPC, so when the RPC returns the invoker knows whether (1) it was executed exactly once, or (2) it was not executed at all, or (3) it arrived at one server before contact was lost due to communication errors or a crash. One of the problems in achieving at-most-once semantics is deciding if a new incoming request has been executed or not. With FLIP, this problem is easily solved. Each time a server is started, the server chooses a new FLIP address. Thus, all requests sent to a crashed server will fail automatically, because the old FLIP address is unknown. During one incarnation of the server, the server can decide, based on sequence numbers in the message, whether the request was executed or not.

Our implementation of RPC is very similar to Birrell and Nelson's [Birrell and Nelson 1984], except for two important differences. First, because FLIP addresses are 64-bit large and location-independent, our implementation has no need for a unique identifier; the FLIP address is the unique identifier. Second, our implementation does not use the next request as an acknowledgement for the last reply. Instead, our implementation sends an explicit acknowledgement when the reply is received. This simplifies the implementation of the RPC layer. Furthermore, sending the acknowledgement is not in the critical path of an RPC (see the next section).

Group Communication

Group communication in Amoeba is based on the protocols described in [Kaashoek et al. 1989; Kaashoek and Tanenbaum 1991]. Amoeba provides a primitive to send a message to a group of processes reliably. Furthermore, this primitive guarantees that all broadcast messages within a group are totally ordered. The group communication protocols make heavy use of *flip_multicast*. This has the advantage that a group of *n* processes can be addressed using one FLIP address, even if they are located on multiple networks.

As explained in Section 5.2, we treat the ability of a network to send multicast messages as an optimization over sending *n* separate point-to-point messages. If the FLIP box discovers that a FLIP address is routed to *n* locations on the same network, it asks the network dependent layer to return a multicast address for the *n* locations. It is then up to the network layer to create such a multicast address and to make sure that the *n* locations will listen to it. After the network layer has done so, it returns to the packet switch a multicast address and a list of locations that listen to the multicast address.

From then on, the packet switch can use the multicast address. The implementation of the Ethernet layer does this as soon as the FLIP box maps an address on two locations onto the same network.

Thus, the FLIP protocol does all the routing of multicast messages, and recognizes when a data-link multicast could be used to reduce the number of messages. Once it recognizes the possibility of optimization, it leaves it up to a network dependent layer to perform it. The reason that FLIP itself cannot perform the optimization is that FLIP does not know about the data link addresses for multicast.

Security

Security in Amoeba is implemented using the FLIP support for security. Although FLIP does not encrypt messages itself, it provides two mechanisms for supporting security. First, messages can be marked sensitive by the sender (using the *Security* bit), so that they will not be routed over untrusted networks. Second, messages going through FLIP may be marked unsafe (using the *Unsafe* bit), so that the receiver can tell whether or not there is a safe route to the sender. If, based on this information, a process thinks there is a safe route to the destination, it can try to send secure messages unencrypted, but with the *Security* bit set. If this message is bounced with the *Unreachable* bit set, no trusted path exists after all. This can only happen due to configuration changes. The process then encrypts the message, and retransmits it with the *Security* bit cleared.

Our implementation of secure RPC is in an experimental phase and is not yet in day to day use; we are still studying how to do secure group communication. Like many secure systems, Amoeba secure RPCs are based on a shared key between the client and the server and its implementation is roughly similar to Birrell's [Birrell 1985]. The main difference is that our implementation uses FLIP's knowledge about trusted and untrusted networks. The Amoeba processor pools and specialized servers are located in one single room and together form a trusted network. Thus, all communication between processes in the processor pool and, for example, the file service does not have to be encrypted. However, as soon as a FLIP message leaves this network, it is guaranteed to be encrypted (if it is part of a secure RPC). This encryption is transparent to the user. Our expectation is that we can build a complete secure system with acceptable performance, because the common case does not require encryption. Furthermore, it is not necessary that all processors be equipped with encryption hardware.

Network Management

Little network management is required in Amoeba. FLIP can deal automatically with network changes: we add machines, networks, or reconfigure our systems just by plugging or unplugging cables. When a machine comes up, it does not have to send out

ARP or RARP requests and wait until a server responds; instead it can be used as soon as it is plugged into the network.

The only network management that is required has to do with trusted and untrusted networks. FLIP relies on the system administrator to mark a network interface as ‘‘trusted’’ or ‘‘untrusted,’’ because FLIP itself cannot determine if a network can be considered trusted. In our implementation only the system administrator can toggle this property.

Wide-Area Communication

Although FLIP has been used successfully in small WANs, it does not scale well enough to be used as the WAN communication protocol in a large WAN. Addresses form a flat name space that is not large enough to address all the machines in the world and still ‘‘guarantee’’ uniqueness. Furthermore, the way FLIP uses broadcast makes it less suitable for a WAN. We traded scalability for functionality. Moreover, we believe that WAN communication should not be done at the network layer, but at a higher layer in the protocol hierarchy.

There are three reasons for doing so. First, most communication is local within one domain†. Thus, we decided we did not want to give up on flexibility and performance, because a message could go to a remote domain.

A second reason to make a distinction between a local and remote domain is that protocols on a WAN link differ from protocols used in a distributed system. WAN links are mostly owned by phone companies that are not interested in fast RPCs. Furthermore, different protocols on WANs are used to cope with the higher error rates and the lower bandwidth of WAN links. Thus, making a protocol suitable for WAN communication at the network layer could very well turn out to be a bad design decision, because at the boundary of a domain the messages may have to be converted to the protocols that are used on the WAN link.

The third reason has to do with administration of domains. WAN communication typically costs more money than communicating across a LAN. Transparently paying large amounts of money is unacceptable for most people. Furthermore, even if there is no boundary at the network layer, there is still a logical boundary. Administrators control domains independently and they like to have control over what traffic is leaving and entering their domain. An administrator might want to keep ‘‘dangerous messages’’ out of his domain. If communication is transparent at the network layer, this is hard to achieve, as recently demonstrated by the worm on the Internet [Spafford 1989].

In the Amoeba system we have implemented WAN communication above the RPC layer [Van Renesse et al. 1987]. If a client wants to access a service in another

† Measurements taken at our department show that 80% of all IP messages are destined for a host on the same network, 12% stay within the department, and that 8% are destined for some other IP site.

domain, it does an RPC to a *server agent* in its domain. The server agent sends the RPC to the WAN server, which forwards the RPC to the WAN service in the server's domain using the appropriate protocol for the WAN link. The WAN service in the server's domain creates a *client agent* that executes the same RPC and it will find the server.

7. PERFORMANCE OF FLIP

An important measure of success for any protocol is its performance. We have compared the performance of Amoeba 5.0 RPC (with FLIP) with Amoeba 4.0 RPC (pre-FLIP version) and with other RPC implementations on identical hardware. The delay was measured by performing 10,000 0-byte RPCs. The throughput was measured by sending maximum-size RPCs. In Amoeba 4.0 this is measured by sending 30,000-byte RPCs; in Amoeba 5.0 this is measured by sending 100,000-byte RPCs (which is still smaller than the maximum possible size); in SunOS using 8-Kbyte RPCs; in Sprite using 16-Kbyte RPCs; and in Peregrine using 48,000-byte RPCs. To make direct comparisons possible we also measured Amoeba 5.0 RPC with the sizes used for the other systems. All measurements were made on Sun3/60s and a 10 Mbit/s Ethernet.

The first row in the table in Figure 10 gives the performance of RPC using the protocols in Amoeba 4.0. The second row in the table gives the performance for the new RPC implementation on top of FLIP. The delay in Amoeba 4.0 is lower than in Amoeba 5.0, because Amoeba 4.0 RPC is implemented over bare Ethernet and requires all machines in a domain to be on one network, so it does not have to do routing and the implementation can be tuned for the case of one network interface. In spite of the overhead for routing, the throughput in Amoeba 5.0 is 30% higher, largely because Amoeba 4.0 uses a stop-and-wait protocol, while Amoeba 5.0 uses a blast protocol [Zwaenepoel 1985] to send large messages. This enables user processes in Amoeba 5.0 RPC to get 87% of the total physical bandwidth of an Ethernet (the FLIP and RPC protocols including headers use 90% of the total bandwidth).

For comparison, the delay of a 0-byte RPC in SunOS is 6.7 msec and the bandwidth for an 8-Kbyte RPC is 325 Kbyte/s (the maximum RPC size for SunOS is 8 Kbyte). This is due to the fact that SunOS copies each message several times before it is given to the network driver, due to its implementation on UDP/IP, and due to the higher cost for context switching. In Sprite, the delay is 2.0 msec and the maximum throughput is 821 Kbyte/s (these numbers are measured kernel to kernel). Although Sprite's kernel-to-kernel RPC does not do routing, the delay of the null RPC is almost the same as the delay for Amoeba 5.0, while Amoeba's delay is measured user-to-user. Sprite also uses a blast protocol for large messages, but its throughput is still less than the throughput achieved by Amoeba 5.0. This can be explained by the fact that Amoe-

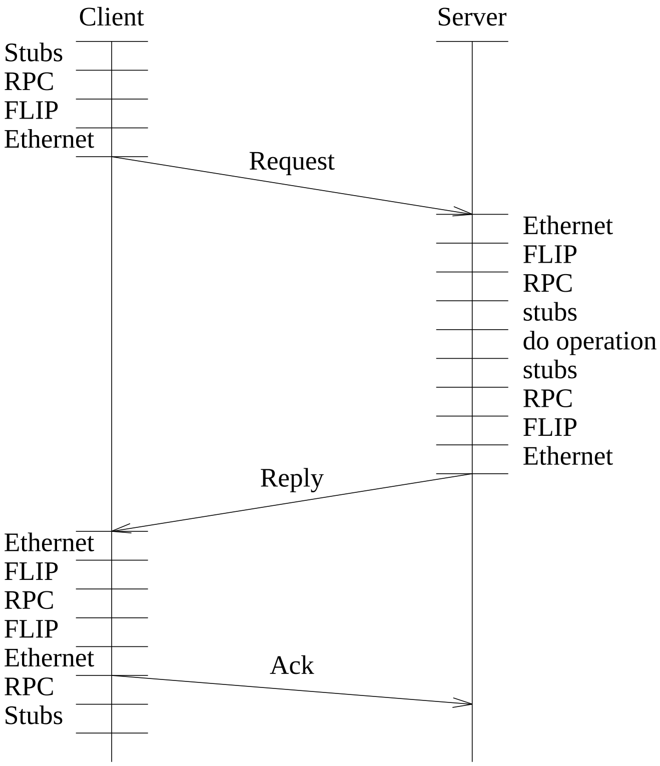
RPC implementation	Delay (msec)	Maximum Bandwidth (Kbyte/s)	Amoeba 5.0 Bandwidth (Kbyte/s)
Amoeba 4.0 RPC	1.1	814	993
Amoeba 5.0 RPC (FLIP)	2.1	1061	1061
Sprite RPC	2.0	820	884
Sun RPC	6.7	325	755
Peregrine RPC	0.6	1139	1001

Fig. 10. Performance numbers for different RPC implementations on Sun3/60s. The Sprite numbers are measured from kernel to kernel. The others are measured from user to user. The fourth column gives the bandwidth for Amoeba 5.0 RPC using the data size for the system measured in each row.

ba keeps its buffer contiguously in memory and that it has a much better context switching time [Douglis et al. 1991].

Compared to Peregrine's RPC [Johnson and Zwaenepoel 1991], Amoeba's delay for a 0-byte RPC is high and Amoeba's maximum throughput is low. Peregrine achieves on identical hardware a delay of 589 μsec and a bandwidth of 1139 Kbyte/s. Peregrine's performance for the null RPC is only 289 μsec above the minimum possible hardware latency. Peregrine achieves this performance by directly remapping the Ethernet receive buffer in the server machine to become the new thread's stack and by using preallocated and initialized message headers. Furthermore, Peregrine uses a two-message RPC protocol, while Amoeba is using a three-message RPC protocol, although the third message is only partly in the critical path. Peregrine achieves a high throughput by overlapping the copying of data from a packet with the transmission of the next packet. The last packet is, like the single-packet case, directly remapped, avoiding the copying of data. We believe that we can apply many of Peregrine's optimizations in Amoeba, which will probably result in a similar performance as Peregrine's. For more performance numbers on these and other RPC implementations see [Tanenbaum et al. 1990]. Thus, in addition to providing more functionality, FLIP makes it also possible to achieve very good performance.

To determine the overhead in FLIP, we measured the time spent in each layer during a null RPC (see Fig. 11). The overhead due to FLIP is 21% of the total delay for a null RPC. From the numbers given one can also compute what the costs are if the server and client were located on different networks. Each additional hop over another Ethernet increases the delay of a null RPC by 975 μsecs.



(a)

Layer	**Time (μsec)**
User	100
RPC	840
FLIP	450
Ethernet	750

(b)

Fig. 11. (a) The Amoeba RPC protocol and (b) the time spent in the critical path of each layer for a null RPC. The RPC protocol for a small RPC uses three messages: 1) a request message from the client to the server; 2) a reply message that acknowledges the request and unblocks the client; 3) an acknowledgement for the reply, so that the server can clean up its state. The acknowledgement is only for a small part in the critical path. The Ethernet time is the time spent on the wire plus the time spent in the driver and taking the interrupt.

The delay for sending a null broadcast reliably and totally ordered for varying group sizes is depicted in Figure 12. The experiment measures the delay seen by the

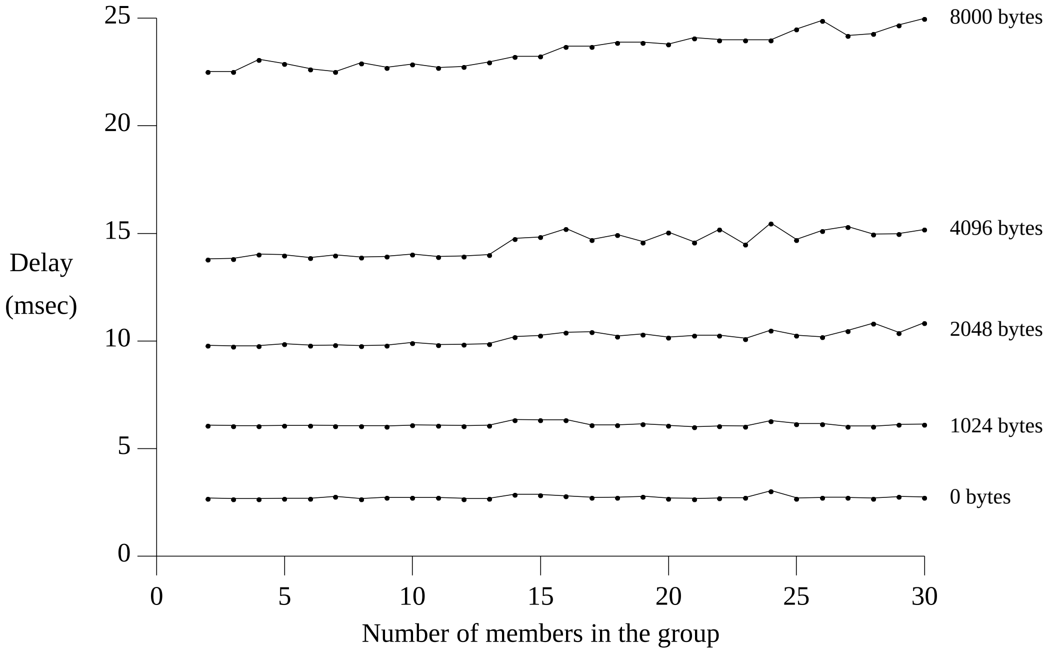


Fig. 12. Delay for sending a broadcast to a group varying from 1 to 30 members.

sender when sending a message to a group of receivers varying in size from 1 to 30 members. The experiment was done on slightly different hardware, a collection of 20Mhz MC68030 boards, as we do not have 30 Sun3/60s. (The null RPC time on these boards is 2.5 msec.) The delay measured is independent of the number of receivers, because FLIP dynamically switches from using point-to-point to using the hardware multicast facility provided by the Ethernet.

8. DISCUSSION AND COMPARISON

Many communication protocols have been introduced in the last decade. Some of them are accepted as official standards or are used by a large user community, such as X.25 [Zimmerman 1980] and IP; others are tailored to specific applications, such as the Express Transfer Protocol (XTP) [Saunders and Weaver 1990]. In a distributed system like Amoeba many entities are short-lived and send small messages. In such an environment, setting up connections would be a waste of time and resources. We therefore decided to make FLIP a connectionless protocol. In this section, we compare FLIP to other connectionless protocols and discuss the advantages and disadvantages of FLIP over these other protocols.

However, before comparing FLIP to other connectionless protocols, we first summarize the requirements that distributed computing imposes on the communication system and the support that FLIP offers to meet these requirements (see Fig. 13).

Requirements	**FLIP support**
Transparency	• FLIP addresses are location-independent • A FLIP box performs routing • Messages can be as large as $2^{32}-1$ bytes. • Routing tables change automatically.
Efficient at-most-once RPC	• FLIP is an unreliable message protocol • FLIP can use a blast protocol for large messages • A process uses a new FLIP address after it crashes
Group Communication	• A FLIP address may identify a number of processes • Routing tables change dynamically • FLIP uses data-link multicast, if possible • FLIP also works if multicast is not available
Security	• Addresses are hard to forge • FLIP takes advantage of trusted networks
Network Management	• Every machine is a router • Routing tables are dynamically updated
WAN	• Works for small WAN-based projects

Fig. 13. How FLIP meets distributed systems requirements discussed in section 2.

8.1. Discussion

The main property of FLIP that gives good support for distributed computing is a combination of dynamic routing and the fact that FLIP addresses identify logical entities (processes or groups) rather than machines. Dynamic routing is done in a way roughly similar to transparent bridges [Backes 1988]. Each FLIP box keeps a cache of hints that is dynamically updated by FLIP messages. To keep routing tables up-to-date with the network topology, FLIP headers have a type field and include hop counts. The combination of dynamic routing tables and communication between entities simplifies the implementation of higher-level protocols such as RPC and group communication and gives enhanced support for process migration. Furthermore, little network management is required.

The only requirement for which FLIP does not have full support is wide-area networking. We think, however, that wide-area communication should not be done at the network layer, but in higher layers.

The costs for the functionality of FLIP can be divided roughly into 3 areas: limited scalability, costs for broadcast, and memory for routing tables. By using a flat name space we lose on scalability, but gain the ability to make addresses location-independent and on the ability to do routing on a per-entity basis. One could envision adding a domain identifier to the FLIP header, so that FLIP would scale to larger internetworks. Using a domain identifier, all the good properties of FLIP would exist in a single domain, but not between two domains.

A danger in our current implementation of FLIP is that addresses might clash. Two processes could accidentally register the same FLIP address. In this case, messages sent to process A may end up at process B. However, as long as the same process is not talking to A and B at the same time and routes to A and B do not intersect, most of the messages will still be delivered correctly†. In the current situation with a good random generator and seed, clashes of FLIP addresses do not occur. Of course, if the number of entities increases enormously, the chance of clashes increases.

By using locate messages we have the ability to reconfigure networks dynamically and move processes around. The costs are that FLIP will generate more broadcasts than a protocol like IP and that there is a startup cost involved in locating a destination. Furthermore, there is a danger that FLIP could cause a flood of broadcasts. To avoid this we have introduced a hop count in the header, kept state (1 Kbyte) in each kernel to break loops, and limited the number of broadcasts per second that a FLIP box can forward. The net result is that Amoeba in the environment depicted in Figure 8 (which contains loops) on average generates only 1.6 broadcasts per second to locate ports and 0.1 broadcasts per seccond to locate FLIP addresses (measured over a 60 hour time period: 3 working days and two nights). Given the fact that it takes approximately 500 μsec to process a broadcast, we are paying only 0.1% of the CPU for dealing with broadcasts. We find this a good tradeoff.

We locate destinations by expanding the scope of each broadcast. This has the disadvantage that networks close by will receive more broadcasts than networks further away. Furthermore, it introduces a potentially longer delay for destinations far away or destinations that do not exist. Because the RPC implementation caches mappings of ports to FLIP addresses and the FLIP implementation caches the mapping of FLIP addresses to locations, very little locating takes place, so the number of broadcasts is low. Most of the broadcasts are due to attempts to locate services that no longer exist. In a large internetwork the number of broadcasts could be too high and the delay too long. In such an environment, one could implement a scheme which caches unreachable

† As soon as we started running FLIP on all our machines, we came across this problem, because many of the pseudo-random generators were at that time fed with the same seed.

ports and FLIP addresses to reduce the number of broadcasts for non-existing services. This scheme is, however, not trivial to implement correctly.

By using routing tables in each kernel, we can do routing on a per-process basis. The cost for doing this is that each kernel must keep such a table. In our current environment, we are using tables that can store 100 FLIP addresses; this requires only 6 Kbyte of storage.

8.2. Comparison

The rest of this section discusses alternative solutions for communication in distributed systems. One of the most widely used internet protocols is IP [Postel 1981a; Comer 1992]. In IP, an address identifies a host. Thus, if a process is migrated from one host to another host, it must change its IP address and tell other processes that it did so. Because IP uses a hierarchical address space, machines cannot be disconnected from one network and connected to another network without changing their IP addresses, although a new extension to IP has been proposed to deal with mobile computers [Ioannidis et al. 1991]. FLIP's flat address space also has some disadvantages. Routing tables are larger. Instead of having one entry for a collection of addresses on one network, FLIP needs a separate entry for every address. With the flat address space, FLIP also scales less well to wide-area communication. Another fundamental difference between IP and FLIP is IP's limit to the size of a message (64 Kbyte). Higher-level protocols have to break messages in 64 Kbyte fragments and reassemble them at the other side. As a result, IP does not benefit from communication links that allow packets larger than 64 Kbyte. A final fundamental difference is that IP provides limited support for secure communication. For example, the standard IP specification does not provide secure routing.

Besides these fundamental differences, there are also a number of differences that are dependent on the IP implementation and routing protocol used. The Internet Control Message Protocol improves end-to-end flow control and routing [Postel 1981b]. However, there are still many problems. For example, many IP implementations make a distinction between a router and a host. A router does routing, and a host runs processes and does not do routing. If the network topology changes, it often happens that machines have to be restarted or reconfigured manually. Furthermore, all ongoing communication between a machine that is about to be moved and other machines will have to be aborted. As most departments own a large number of machines and many networks, these changes need to be done more often than any system administrator cares for. FLIP eliminates almost all need for network management; system administrators can install, move, or remove machines without making changes to the configuration tables.

Another protocol that has been especially designed for distributed operating systems is the Versatile Message Transaction Protocol (VMTP) [Cheriton 1986, 1988a].

Like FLIP, VMTP provides a base to build higher-level protocols, and has been used for the protocols in the V distributed system [Cheriton 1988b]. Unlike FLIP, VMTP is a transport protocol, which relies on an internet protocol for routing. Therefore VMTP may be implemented on top of FLIP, providing the VMTP abstraction with the advantages of FLIP.

Three types of addresses exist in VMTP. They differ in the time that they are usable. T-stable addresses, for example, are guaranteed not to be reused for at least T seconds after they become invalid. This allows a timer-based implementation of at-most-once Remote Procedure Call. If one were to run VMTP on FLIP, such timed addresses would not be needed, because the 56 bits of an address would almost certainly be unique and an entity can pick a new address at any time. VMTP is a reliable transport protocol, and uses a single mechanism for fragmentation and flow control on all network types. To be able to implement this protocol efficiently, the designers also put an artificial upper bound on the size of a network message. Due to this artificial upper bound, and the fact that networks differ greatly in their physical properties, VMTP may perform well on one network and less well on another.

The routing algorithm that FLIP uses for MULTIDATA packets is similar to the single-spanning-tree multicast routing algorithm discussed by Deering and Cheriton [Deering and Cheriton 1990]. In the same paper, the authors also discuss more sophisticated multicast routing algorithms. These algorithms could be implemented in FLIP using the *Variable Part* of the header.

9. CONCLUSION

In this paper we have discussed protocol requirements for distributed systems and proposed a new protocol that meets them. Current internet protocols do not address various problems, leaving the solution to higher-level protocols. This leads to more complex protocols, that cannot perform well, because they cannot take advantage of hardware support. We presented the FLIP protocol that supports many of the requirements of distributed systems in an integrated way. FLIP addresses management of internetworks, efficient and secure communication, and transparency of location and migration.

FLIP is used in the Amoeba 5.0 distributed operating system to implement RPC and group communication over a collection of different networks. The advantages over Amoeba 4.0 include better scaling, easier management, and higher bandwidth. Round-trip delay is currently higher, but this can probably be improved by careful coding and tuning.

There is more work to be done. For example, we have no experience with large networks containing thousands of subnets. However, since Amoeba implements wide-area communication transparently in user space, using X.25 or TCP links between Amoeba sites, this is at least conceivable. Additionally, locating endpoints with location-independent addresses can be a problem, and we are currently considering a location service for a possibly large network of subnets that may or may not support hardware multicast.

ACKNOWLEDGEMENTS

Henri Bal, Brian Bershad, Leendert van Doorn, Fred Douglis, Philip Homburg, Wiebren de Jonge, Sape Mullender, Greg Sharp, Mark Wood, and Willy Zwaenepoel provided comments on drafts of this paper, which improved its content and presentation substantially. Wiebren de Jonge also suggested a clean and nice improvement to the one-way-function used in the host interface. We would also wish to thank the referees for their input, which further helped to improve the paper.

REFERENCES

Backes, F., “Transparent Bridges for Interconnection of IEEE 802 LANs,” *IEEE Network*, Vol. 2, No. 1, pp. 5-9, Jan. 1988.

Birman, K. P. and Joseph, T. A., “Exploiting Virtual Synchrony in Distributed Systems,” *Proc. Eleventh ACM Symposium on Operating Systems Principles*, pp. 123-138, Austin, TX, Nov. 87.

Birrell, A. D., “Secure Communication Using Remote Procedure Calls,” *ACM Trans. Comp. Syst.*, Vol. 3, No. 1, pp. 1-14, Feb. 1985.

Birrell, A. D. and Nelson, B. J., “Implementing Remote Procedure Calls,” *ACM Trans. Comp. Syst.*, Vol. 2, No. 1, pp. 39-59, Feb. 1984.

Cheriton, D. R., “VMTP: A Transport Protocol for the Next Generation of Communication Systems,” *Proc. SIGCOMM 86*, pp. 406-415, Stowe, VT, Aug. 1986.

Cheriton, D. R., “VMTP: Versatile Message Transaction Protocol,” RFC 1045, SRI Network Information Center, Feb. 1988a.

Cheriton, D. R., “The V Distributed System,” *Commun. ACM*, Vol. 31, No. 3, pp. 314-333, Mar. 1988b.

Cohen, D., “On Holy Wars and a Plea for Peace,” *IEEE Computer*, Vol. 14, pp. 48-54, Oct. 1981.

Comer, D. E., “Internetworking with TCP/IP 2nd ed.,” Prentice-Hall, Englewood Cliffs, NJ, 1992.

Dasgupta, P., Leblanc, R. J., Ahamad, M., and Ramachandran, U., “The Clouds Distributed Operating System,” *IEEE Computer*, Vol. 24, No. 11, pp. 34-44, Nov. 1991.

Deering, S. E., “Host Extensions for IP Multicasting,” RFC 1112, SRI Network Information Center, Aug. 1988.

Deering, S. E. and Cheriton, D. R., “Multicast Routing in Datagram Internetworks and Extended LANs,” *ACM Trans. Comp. Syst.*, Vol. 8, No. 2, pp. 85-110, May 1990.

Douglis, F., Kaashoek, M. F., Tanenbaum, A. S., and Ousterhout, J. K., “A Comparison of Two Distributed Systems: Amoeba and Sprite,” *Computing Systems*, Vol. 4, No. 4, pp. 353-384, 1991.

Finlayson, R., Mann, T., Mogul, J., and Theimer, M., “A Reverse Address Resolution Protocol,” RFC 903, SRI Network Information Center, June 1984.

Ioannidis, J., Duchamp, D., and Maguire Jr., G. Q., “IP-based Protocols for Mobile Internetworking,” *Proc. SIGCOMM 91 Conference on Communications Architectures and Protocols*, pp. 235-245, Zürich, Zwitserland, Sep. 1991.

Johnson, D. B. and Zwaenepoel, W., “The Peregrine High-Performance RPC System,” TR91-151, Rice University, Mar. 1991.

Kaashoek, M. F. and Tanenbaum, A. S., “Group Communication in the Amoeba Distributed Operating System,” *Proc. Eleventh International Conference on Distributed Computing Systems*, pp. 222-230, Arlington, TX, May 1991.

Kaashoek, M. F., Tanenbaum, A. S., Flynn Hummel, S., and Bal, H. E., “An Efficient Reliable Broadcast Protocol,” *Operating Systems Review*, Vol. 23, No. 4, pp. 5-20, Oct. 1989.

Kung, H. T., “Gigabit Local Area Networks: a Systems Perspective,” *IEEE Communications Magazine*, Vol. 30, No. 4, pp. 79-89, Apr. 1992.

Mullender, S. J., Van Rossum, G., Tanenbaum, A. S., Van Renesse, R., and Van Staveren, H., “Amoeba: A Distributed Operating System for the 1990s,” *IEEE Computer*, Vol. 23, No. 5, pp. 44-53, May 1990.

National Bureau of Standards, “Data Encryption Standard,” Fed. Inf. Process. Stand. Publ. 46, Jan. 1977.

Ousterhout, J. K., Cherenson, A. R., Douglis, F., Nelson, M. N., and Welch, B. B., “The Sprite Network Operating System,” *IEEE Computer*, pp. 23-36, Feb. 1988.

Plummer, D. C., “An Ethernet Address Resolution Protocol,” RFC 826, SRI Network Information Center, Nov. 1982.

Postel, J., “Internet Protocol,” RFC 791, SRI Network Information Center, Sep. 1981a.

Postel, J., “Internet Control Message Protocol,” RFC 792, SRI Network Information Center, Sep. 1981b.

Rozier, M., Abrossimov, V., Armand, F., Boule, I., Gien, M., Guillemont, M., Herrmann, F., Kaiser, C., Langlois, S., Leonard, P., and Neuhauser, W., “Chorus Distributed Operating System,” *Computing Systems*, Vol. 1, No. 4, pp. 305-370, 1988.

Saltzer, J. H., Reed, D. P., and Clark, D. D., “End-to-End Arguments in System Design,” *ACM Trans. Comp. Syst.*, Vol. 2, No. 4, pp. 277-288, Nov. 1986.

Saunders, R. M. and Weaver, A. C., “The Xpress Transfer Protocol (XTP) - A Tutorial,” *Computer Communication Review*, Vol. 20, No. 5, pp. 67-80, Oct. 1990.

Spafford, E. H., “The Internet Worm: Crisis and Aftermath,” *Commun. ACM*, Vol. 32, No. 6, pp. 678-688, June 1989.

Tanenbaum, A. S., Kaashoek, M. F., and Bal, H. E., “Parallel Programming Using Shared Objects and Broadcasting,” *IEEE Computer*, Vol. 25, pp. 10-19, Aug. 1992.

Tanenbaum, A. S., Mullender, S. J., and Van Renesse, R., “Using Sparse Capabilities in a Distributed Operating System,” *Proc. Sixth International Conference on Distributed Computing Systems*, pp. 558-563, Cambridge, MA, May 1986.

Tanenbaum, A. S., Van Renesse, R., Van Staveren, H., Sharp, G., Mullender, S. J., Jansen, A., and Van Rossum, G., “Experiences with the Amoeba Distributed Operating System,” *Commun. ACM*, Vol. 33, No. 12, pp. 46-63, Dec. 1990.

Van Renesse, R., Tanenbaum, A. S., Van Staveren, H., and Hall, J., “Connecting RPC-based Distributed Systems Using Wide-area Networks,” *Proc. Seventh International Conference on Distributed Computing Systems*, pp. 28-34, Berlin, Sep. 1987.

Zimmerman, H., “OSI Reference Model - The ISO Model of Architecture for Open Systems Interconnection,” *IEEE Trans. on Communications*, Vol. 28, pp. 425-432, Apr. 1980.

Zwaenepoel, W., “Protocols for Large Data Transfers over Local Networks,” *Proc. Ninth Data Communications Symposium*, pp. 22-32, Whistler Mountain, Canada, Sep. 1985.

APPENDIX

The FLIP protocol makes it possible that routing tables automatically adapt to changes in the network topology. The protocol is based on 6 message types. We will discuss in detail the actions undertaken by the packet switch when receiving a FLIP fragment.

If a fragment arrives over an untrusted network, the *Unsafe* bit in the *Flags* field is set. This also happens when a fragment is sent over an untrusted network. Furthermore, FLIP refuses to send a fragment with the *Security* bit set over an untrusted network. We have omitted these details from the protocol description below to make it easier to understand.

```
                                LOCATE
/* Remember that source can be reached through network ntw on location loc. */
UpdateRoutingTable(pkt→source, ntw, loc, pkt→act_hop, pkt→flags & UNSAFE);
if (pkt→act_hop == pkt→max_hop and
        lookup(pkt→destination, &dsthop, ntw, pkt→flags & SECURITY)) {
    /* Destination is known; send HEREIS message back. */
    pkt→type = HEREIS;
    pkt→max_hop = pkt→act_hop + dsthop;
    pkt→act_hop −= Networkweight[ntw];
    pkt_send(pkt, ntw, loc);
} else { /* destination is unknown or incorrect */
    /* Forget all routes to destination, except those on ntw */
    RemoveFromRoutingTable(pkt→destination, ALLNTW, ALLLOC, ntw);
    /* Forward pkt on all other networks, if the hop count and security allow it */
    pkt_broadcast(pkt, ntw, pkt→max_hop − pkt→act_hop, pkt→flags & SECURITY);
}
```

Fig. 14. The protocol for LOCATE messages. A LOCATE message is broadcast, while the route to the source address is remembered.

The LOCATE message is to find the network location of a NSAP (see Fig. 14). It is broadcast to all FLIP boxes. If a FLIP box receives a LOCATE message, it stores the tuple (Source Address, Network, Location, Actual Hop Count, Flags & UNSAFE) in its routing table, so that a reply to the LOCATE message can find its way back. If the *Actual Hop Count* in the LOCATE message is equal to the *Maximum Hop Count*, the *Destination Address* is in the routing table, the destination network is safe (if necessary), and the destination network is not equal to the source network, the LOCATE message is turned into an HEREIS message and sent back to the *Source Address*. The *Maximum Hop Count* of the HEREIS message is set to the *Actual Hop Count* of the LOCATE message plus the hop count in the routing table. If the *Actual Hop Count* in the LO-

CATE message is less than the *Maximum Hop Count*, the entries for *Destination Address* in the routing table are removed, except for the entries that route the address to the network on which the LOCATE arrived, and the message is broadcast on the other networks.

It is important that the packet switch only sends an HEREIS message back if the *Actual Hop Count* of the LOCATE message is *equal* to the *Maximum Hop Count*. By using a large *Maximum Hop Count* the sender of the LOCATE message can force an interface module to respond instead of a packet switch and at the same time invalidate any old routing information for the address that is located. If the address to be located is registered at an interface on a distance smaller than *Maximum Hop Count*, this scheme works correctly, because an interface always sends an HEREIS back for an address that is registered with it.

```
                                HEREIS
hops = pkt→max_hop − pkt→act_hop;
AddToRoutingTable(pkt→destination, ntw, loc, hops, pkt→flags & UNSAFE);
if (route(pkt→source, &dstntw, &dstloc, ntw, pkt→act_hop, pkt→flags & SECURITY)) {
   /* A network is found that is different from the network on which
    * pkt arrived and on which the destination is reachable. */
   pkt→act_hop −= Networkweight[dstntw];
   pkt_send(pkt, dstntw, dstloc);
} else discard(pkt);  /* Source is unknown, too far away, or unsafe. */
```

Fig. 15. The protocol for HEREIS messages. HEREIS messages are returned to the source in response to LOCATE messages, while the route to the destination (of the LOCATE message) is remembered.

An HEREIS message is sent as a reply to a LOCATE message (see Fig. 15). If an HEREIS message arrives, the tuple (Destination Address, Network, Location, Actual Hop Count, Flags & UNSAFE) is added to the routing table. If the *Source Address* is in the routing table and the network on which the source can be reached is not equal to the network on which the message arrived and the incremented *Actual Hop Count* does not exceed the *Maximum Hop Count*, the message is forwarded. Otherwise, the message is discarded. If the destination network is equal to the source network, *route()* will return false; the message is discarded.

UNIDATA messages are used to transfer fragments of a message between two NSAPs. When such a message arrives, the tuple (Source Address, Network, Location, Actual Hop Count, Flags & UNSAFE) is stored in the routing table (see Fig. 16). If the Destination Address is in the routing table, the destination network is not equal to the source network, the incremented *Actual Hop Count* does not exceed the *Maximum Hop Count*, and the destination network is safe, the message is fragmented (if needed), and

```
UNIDATA
UpdateRoutingTable(pkt→source, ntw, loc, pkt→act_hop, pkt→flags & UNSAFE);
hops = pkt→max_hop − pkt→act_hop;
switch (route(pkt→destination, &dstntw, &dstloc, ntw, hops, pkt→flags & SECURITY)) {
  case OK:   /* forward message */
    pkt→act_hop += Networkweight[dstntw];
    pkt_send(pkt, dstntw, dstloc);
    break;
  case TooFarAway:   /* send pkt back to source. */
    pkt→type = NOTHERE;
    pkt→acthop −= Networkweight[ntw];
    pkt_send(pkt, ntw, loc);
    break;
  case Unsafe:   /* send pkt back to source. */
    pkt→type = UNTRUSTED;
    pkt→flags |= UNREACHABLE;
    pkt→acthop −= Networkweight[ntw];
    pkt_send(pkt, ntw, loc);
    break;
  }
}
```

Fig. 16. The protocol for UNIDATA messages. If the destination is known and, if necessary, safe, the message is forwarded. If the destination is unknown, the message is returned as a NOTHERE message. If the message can only be transfered over trusted networks, and the destination network is untrusted, the message is returned as an UNTRUSTED message.

each fragment is forwarded. If there are multiple choices in the routing table, one is chosen, based on an implementation-defined heuristic, such as the safety or the minimum number of hops. The null destination address maps to all networks and locations.

If the *Destination Address* of a UNIDATA message is not in the routing table, or the destination network is unsafe, the message is transformed into a NOTHERE message by setting the *Type* to NOTHERE, and is returned to the *Source Address*. The data in the message is *not* discarded, unless the decremented *Actual Hop Count* was zero.

If the *Maximum Hop Count* minus the *Actual Hop Count* of a UNIDATA message is less than the Hop Count stored in the routing table, the implementor can decide to send a NOTHERE message back to the sender. Chances are that the message would not have reached its destination. A new locate of the *Destination Address* will re-establish the route, and update the routing tables. If the destination network is untrusted, then the *Unreachable* bit is set, and the message is returned as an UNTRUSTED message.

If a NOTHERE message arrives at a FLIP box, the corresponding entry in the routing table is invalidated (see Fig.17). If another route is present in the routing table, the

```
NOTHERE
RemoveFromRoutingTable(pkt→destination, ntw, loc, NONTW);
hops = pkt→max_hop − pkt→act_hop;
if (route(pkt→destination, &dstntw, &dstloc, ntw, hops, pkt→flags & SECURITY) {
    /* There is another route to destination; use it. */
    pkt→type = UNIDATA;
    pkt→act_hop += Networkweight[dstntw];
    pkt_send(pkt, dstntw, dstloc);
} else if (route(pkt→source, &dstntw, &dstloc, ntw, hops, pkt→flags & SECURITY)) {
    /* Forward to original source. */
    pkt→act_hop −= Networkweight[dstntw];
    pkt_send(pkt, dstntw, dstloc);
} else discard(pkt);   /* Source is unknown, too far away, or untrusted. */
```

Fig. 17. The protocol for NOTHERE messages. If there is an alternative route, try that one. Otherwise forward back to the original source.

Type field is set back to UNIDATA. Now operation continues as if a UNIDATA message arrived, except that the routing table operation is skipped. This way an alternate route, if available, will be tried automatically. If not, the NOTHERE is forwarded to its source (if still safe).

```
UNTRUSTED
UpdateRoutingTable(pkt→destination, ntw, loc, pkt→act_hop, UNSAFE);
hops = pkt→max_hop − pkt→act_hop;
if (route(pkt→destination, &dstntw, &dstloc, ntw, hops, SECURE) {
    /* There is another safe route to destination; use it. */
    pkt→type = UNIDATA;
    pkt→act_hop += Networkweight[dstntw];
    pkt_send(pkt, dstntw, dstloc);
} else if (route(pkt→source, &dstntw, &dstloc, ntw, hops, SECURE)) {
    /* Return to source (if still safe). */
    pkt→act_hop −= Networkweight[dstntw];
    pkt_send(pkt, dstntw, dstloc);
} else discard(pkt);   /* Source is unknown, too far away, or untrusted. */
```

Fig. 18. The protocol for an UNTRUSTED message. If there is an alternative safe route, try that one. Otherwise return to its original source.

If an UNTRUSTED message arrives at a FLIP box (see Fig. 18), the route in the routing table is updated, and a new safe route, if present, is tried. If there is no such route, the message is forwarded back to the original source (but only if there exists a route back that is safe).

A MULTIDATA message is transfered like a UNIDATA message (see Fig. 19).

MULTIDATA

```
UpdateRoutingTable(pkt→source, ntw, loc, pkt→act_hop, pkt→flags & UNSAFE);
/* See if there are any known (and safe) destinations. */
if (list = lookup(dstaddr, &dsthop, ntw, pkt→flags & SECURITY)) {
    /* Send message to all locations on list, if the hop count allows it. */
    pkt_multicast(list, pkt→max_hop – pkt→act_hop);
} else discard(pkt);
```

Fig. 19. The protocol for MULTIDATA messages. Forward to all known destinations.

However, if there are multiple entries of the *Destination Address* in the routing table, the message is forwarded to all destinations instead of just one. If there is no entry for the *Destination Address*, or the destination network is unsafe, the message is discarded and not returned as a NOTHERE message. FLIP does not assume that a network has support for multicast. If a network has such a capability, FLIP will try to take advantage of it. If not, the message is sent point-to-point to all destinations on the network.