A HOMOGENEOUS NETWORK FOR DATA SHARING - COMMUNICATIONS

by
Eric Manning and Richard Peebles

Research Report CS-77-07

Department of Computer Science
and
Computer Communications Networks Group
University of Waterloo
Waterloo, Ontario, Canada

March 1977

ABSTRACT

The communications aspects of a distributed architecture for transaction processing are described. The architecture is aimed at transaction processing on physically distributed data bases, where most of the hits on a given component of the data base come from a single geographic region. The architecture is physically based on a homogeneous set of host minicomputers, a message-switched communications subnetwork (loop or packet-switched), and a set of network interface processors which connect the hosts to the communications subnetwork. It is logically based on two primitives; all data objects (including messages) are segments and all control objects (including files) are tasks. Each task runs in a private virtual space and all inter-task communication is done by passing message segments. Segment passing is done by a single message-switching task in each host, assisted by the interface processors and communications subnetwork where necessary. The message-switching task also enforces protection rules without the need for special hardware.

A two-host implementation of the logical architecture is operational. It is based on PDP-11 minicomputers and a non-switched wire pair subnetwork. The companion paper describes modelling studies of the architecture, using simulation and queueing-theoretic techniques.

KEY WORDS

distributed processing

distributed data bases

message-switched operating systems

inter-process communication

network-wide addresses

homogeneous computer networks

transaction processing

Footnotes

1. The small numbers and static character of Software Machines are more typical of hardware modules than software. Hence our choice of name. In fact, implementation of some SMs in hardware is an obvious possibility and the logical architecture is structured so as to facilitate this.

2. In the prototype implementation (see Section V) we have used the UNIX [21] operating system as a "software factory" to build and test SMs. UNIX has excellent software development and testing facilities and this approach allows us to separate the problems of building code from the problems of executing it.

3. Access to stored files of course provides a second form of inter-task communication, but it is an indirect one.

4. In the case where the machines use the UNIBUS addressing structure; otherwise, only primary stores are spanned.

I. INTRODUCTION

A prime motivation for computer networking is resource sharing, resources being central processors, peripheral I/O and storage devices, programs and data. The leading example of an operational resource sharing network is ARPANET [1,2]; it has inspired a number of closely related networks which are in various stages of design or in operation [3,4,5]. ARPANET consists of a communications subnetwork, analogous to the switched voice network, and a set of host computers, analogous to voice network subscribers. The communications subnet was designed independently of the hosts and is prepared to support any host able to meet minimal hardware and software interfacing requirements. Hence a very wide variety of hosts can be (and have been) attached to ARPANET, most of which were designed without networking requirements as a criterion. This approach is reflected in the management of ARPANET; the communications subnetwork is managed by the network organization whereas the hosts remain under the control of individual subscribers. We refer to such networks as heterogeneous networks.

Heterogeneous networking is required if a variety of qualitatively different resources are to be made available for sharing. One thinks of jobs being prepared on an interactive system such as a PDP-10, which then invokes other hosts to do appropriate parts of a computation, such as a CDC 7600 for numeric calculations. However, other types of resource sharing - peripheral device sharing, load balancing and, particularly, data sharing - do not require such diversity. Moreover, the diversity of heterogeneous

networking as exemplified by ARPANET has created technical problems ranging from incompatibilities in word length and codes, to substantial problems caused by the interconnection of hosts (hardware and operating systems) which were not designed to be interconnected. There are also network management problems ranging from lack of control over a user's physical site to the need for extensive user support for N different hosts at each of N sites. Finally, it appears that jobs which freely draw on multiple host types to optimize efficiency are fairly rare in ARPANET, owing to the absence of appropriate support mechanisms.

A glance at potential network applications, moreover, reveals many which primarily require data sharing, with a lesser demand for peripheral device sharing (central file storage and remote access to expensive or highly-specialized I/O devices), but little or no need for access to multiple host types.

Commercial applications such as banking and retail credit-card sales involve the processing of transactions to update a data base. These data bases often exhibit geographic locality of reference: they can be split into regional components such that most of the "hits" on a particular component come from "its" city or region. Each regional component of such a data base can and should be kept in a host computer located in the appropriate region; this approach offers lower communications costs, more organizational flexibility, and potentially greater reliability than the current practise of holding the entire data base in a single, large computer. It also opens the way to distributed processing, allowing us to exploit the power of mini-

computers (which are often more cost-effective than their larger brethren), as the processing load is distributed among many processors. In addition, parallelism is achieved easily and naturally, as transactions which hit different sites are processed simultaneously. However, geographic locality of reference is a statistical property of data bases, and is therefore subject to exceptions. (For example, a motorist who lives in City X will usually buy his gasoline in City X, but he may occasionally buy gasoline in City Y during trips. The oil company's data base of customer sales records exhibits geographic locality of reference nonetheless.) For this reason, and because managerial queries for information involving the entire data base will have to be processed, the regional hosts must be interconnected to form a network.

A second type of locality of reference is functional locality of reference. Here, the data base can be split into components based on function rather than geography. Terminals can be grouped into classes by function, and most of the transactions from terminals in functional class X produce hits on functional component X of the data base. The classical example of functional locality of reference is health care, where the information about each patient can be subdivided into clinical data, pharmaceutical data, hospital data and so forth. Most of the transactions generated by a pharmacy terminal would hit the pharmaceutical component of the data base, and similarly for the other functional types. Here geography is no longer a factor and all of the hosts of the network might well reside in a single room; the other comments about geographic locality of reference remain true. We use the generic term locality of reference to refer to

either or both of geographic locality of reference and functional locality of reference. The notion of locality of reference leads directly to the idea of homogeneous networks for data sharing.

A homogeneous network for data sharing is a communications subnetwork plus a set of attached hosts. It is intended for transaction processing on distributed data bases which exhibit locality of reference. Because data is the resource to be shared, rather than different types of CPU power, the hosts are assumed to have a common hardware and operating system architecture. Differences in amounts and types of storage and I/O equipment, and of course in data stored, are permitted among the hosts. A homogeneous network can and should be designed, installed and managed as a single "package system", obviating many of the management problems mentioned above and permitting research into the problems of structuring host hardware, host software and the communications subnetwork as a single, integrated entity. Among the potential advantages of this design freedom we may hope for improved efficiency due to:

- the absence of low-level interfacing problems (codes and word lengths),
- the ability to specify a specialized communcations subnet tailored to very specific host needs,
- simpler protocol, as the host operating systems work in a co-operative rather than unknown environment (they are copies of one another).

We also expect improved reliability, due to the ability to freely move data

from host to host and to overcome troubles at one host with the aid of its neighbours (down loading of software and remote diagnosis of hardware faults). The price paid for this is, of course, the inability to call on a multiplicity of different host architectures.

The notion of homogeneous networks is not new; the Integrated Computer Network System [6] provides an early example. However, the ICNS was based on large, general-purpose computers and existing operating systems, and it provided conventional, general-purpose computing facilities such as program preparation, compilation and execution. The idea of networks of mini-computers is also well-known; the Distributed Computing System of Farber et al. [7,8], the KOCOS network [9] and the SPIDER-based network of Fraser [10] all provide examples. However, these networks all attempt to provide general-purpose facilities for interactive and batch computing in academic or research environments, and all are heterogeneous in nature. Our work, on the other hand, is oriented exclusively towards transaction processing on physically distributed data bases in a commercial environment, and uses homogeneous networks.

The new ideas presented in this paper include the ideas of Real and Virtual Network-wide address spaces for inter-processor communication, the use of multiple small virtual spaces within a single application program to provide protection (as an alternative to procedures and capability hardware [11]), the use of only two primitives - segments and tasks - to express all data and control constructs, and the definition of efficient inter-process communication algorithms based on the ideas just mentioned.

Finally, we describe an implementation of the inter-process communication facility (called the Communications Nucleus) which is now operational. A companion paper [12] describes analytic and simulation models of the structures presented in this paper.

II. THE PROBLEM

The problem is to design a logical architecture oriented to transaction processing on distributed data bases exhibiting locality of reference. The architecture -- which we call the MININET logical architecture -- must be able to provide efficient data sharing among hosts, with minimum host and communications overheads. (The need for minimum overheads hardly needs comment, especially in view of the postulated mini-scale CPUs.) Finally, one would like to have an architecture which provides a reasonably close fit to available hardware, to permit a trial implementation without major hardware design and construction efforts.

Two design issues of importance are layering and protocol uniformity. Layering has been explained by Scantlebury and Wilkinson [13] as a technique to ensure logical independence of the modules of a network; it has been observed in the design of the ARPANET and CYCLADES protocols (op.cit.) and the System Network Architecture of IBM. We regard it as essential, so that different communications subnets, inter-process communication designs and data management modules can be "tried out" with minimum disruption. However, strict observance of layering results in an explosion of control fields belonging to the various layers and a drastic increase in communications overhead. We have therefore violated the layering

principle in one instance for the sake of efficiency (between the Switch and Communications Device layers, defined later).

Protocol uniformity is not the same as layering, and represents homogeneity in another sense. By it we mean that the communications facility should handle inter-host and intra-host communications in a uniform manner. That is, a pair of communicating tasks resident in two different hosts should be able to use the same protocol to communicate as do a pair of tasks resident in a single host. We have incorporated protocol uniformity in the MININET logical architecture. The main features of the architecture are described next, in Section III.

III. The MININET Logical Architecture - Concepts and Key Design Features

3.1 Introduction

This section presents the concepts and key design features of the MININET logical architecture. The presentation does not refer to any particular computer or computers, reflecting the fact that the logical architecture is essentially machine-independent. (A particular implementation is described in Section V.)

The logical architecture is organized by layers of abstraction and this Section describes the major layers. We begin at the top, with a graph model of transaction processing, in Section 3.2. We describe a second-level model for the graph's nodes in Section 3.3 and for the graph's edges in Section 3.4.

3.2 A Model of Transaction Processing

A transaction is a short request which generates a rapid response, plus side effects such as the updating of a data base. (Transactions are the analog of jobs in conventional systems.) We represent the processing of a transaction T by a rooted, directed graph G(T). As shown in Figure III.1, all of the tasks which participate in the processing of T are represented by nodes of the transaction graph G(T), and message flows between pairs of tasks are represented by directed edges of G. The user at his terminal is represented by the root node of G, which spawns the initial request-message.

It is necessary to provide unique names or labels for transaction graphs and the nodes of graphs and this is done as follows. As each new transaction enters the network it is given a unique label. The number of labels available for labelling transactions is bounded by a system parameter; this has the effect of limiting the number of transactions which can be "in flight" at any given time, and thereby provides one form of congestion control. The transaction's label serves to uniquely identify the transaction graph which is spawned to process it.

In addition to unique labelling of transactions and their graphs, we need to uniquely label the nodes of a given graph to provide task identifiers. This is done as follows. Transaction graphs correspond one-to-one with transactions "in flight", so the label of transaction T forms the prefix of each node label of the graph G(T). The remainder of the node label is

computed as follows:

1) label the root node .00

2) the children of a node whose label is .L are labelled .L.00, .L.01, .L.02,... .

If directed edges from more than one ancestor node are incident on a child node, the ancestor which created the child is considered to be its father, and the father's label is used to compute the child's label. Thus the name of a parent task determines the names of its children.

Finally, we mention two additional properties of transaction graphs. First, a transaction graph can span two or more hosts; this happens whenever a transaction violates locality of reference. (The node labels of G(T) are prefixed with host identifiers to allow this situation to be managed simply.) Secondly, transaction graphs are dynamically created and destroyed. At no point in time does the entire graph exist; nodes spawn successors as needed and predecessor nodes destroy themselves when their work is done. This approach tends to minimize primary store requirements, at the expense of CPU cycles to perform the creation and destruction of nodes. It can be exploited to reduce required primary store, up to the point where real-time response becomes unacceptable owing to CPU load.

This completes our description of the top layer of abstraction. Next, we must explain how the objects of the transaction graph model -- tasks and message flows between tasks -- are constructed from simpler objects. This is the purpose of the second-level model which is described below. Tasks

are constructed from software machines which are constructed from segments; this is described in Section 3.3. Message flows between tasks are carried out by an object called the Communications Nucleus; it is described in Section 3.4 and Sevtion IV.

3.3 Segments, Software Machines and Tasks - The Nodes of G(T)

The two primitive objects of the MININET logical architecture are the segment and the task. Tasks are built from entities called Software Machines which in turn are built from segments. Hence the segment is the natural place to begin in describing these constructs.

3.3.1 Segments

A segment is a contiguous array of linearly-addressed storage cells. It is defined by a Segment Table Entry or STE which gives its starting address and length. (Figure III.2). The STE also specifies the access control attributes of a segment, namely the read-permission, write-permission, execute-permission and message attributes. If the message attribute is TRUE then the segment is of type message (is a message-segment) and can be sent from one task to another; the other attributes have their usual meanings. All of the STEs resident in a host are stored in a single table called the Master Segment Table.

3.3.2 Software Machines

A Software Machine or SM is a re-entrant program module whose inputs and outputs are message-segments. Each SM has a single, simple

function to perform and is a small, self-contained unit of code.[1] The intent is to decompose the processing of a transaction into a set of small steps, each represented by one SM. Each transaction type is processed by a subset of the set of SMs installed at a host. (We have analyzed several examples and have always found it possible to make such a decomposition.) For a typical transaction, the set of SMs includes a Terminal Machine to mother terminals, a Login Machine to validate users, a Command Machine to decode user requests, a set of File Machines to retrieve customer records, and a Disc Machine to mother secondary storage devices.

Each SM is realized as a set of execute-only segments. This set of segments is called a code module and is the internal representation of the Software Machine. The STEs of a code module are chained together within the Master Segment Table and are called a code panel. SMs are installed at the factory; they are not written by MININET users.[2]

3.3.3 Tasks

A task is created from a given Software Machine to process a particular message-segment or set of message-segments. Thus an abstract

definition of a task is a pair (SM, message). When a command is entered from a terminal it is stored in a message-segment. This message-segment is passed from one task to another, each processing it in some way that leads ultimately to the generation of a response message. (This is the basis of the task graph model described in Section 3.2.) Because SMs are re-entrant, a single SM can be used to spawn many tasks, by binding many data modules to it. (Each data module contains one or more message-segments to be processed by the task, plus additional segments for working storage; the STEs of these segments are chained together and called a data panel.) A task is defined within the MININET logical architecture by a task queue entry, which specifies the message-segment(s) to be processed by the task and the SM used to process it (them).

The format of a task queue entry or tqe is shown in Figure III.3; it contains the task identifier which is the name of the node in the task graph which the task occupies, a pointer to a code panel in the MST, a pointer to a data panel in the MST, a status field and a virtual program counter value. As shown in Figure III.4, the code panel contains pointers to the segments of the Software Machine (code module), and the data panel contains pointers to the message-segments and scratch segments (data module). The status field specifies the status of the task - ready to run or blocked - and the program counter value records the location in virtual memory of the last instruction executed.

Much of the above material can be viewed as a natural extension of Brinch Hansen's [14] development of message-switched operating systems;

the major novelty lies in the graph model of transaction processing, and in the use of segments to represent messages. The use of tasks in the MININET architecture to enforce modularity and protection is also new, and is described next.

The key to our treatment of modularity and protection is the decision to provide each task with its own, private virtual space, disjoint from every other task's virtual space. The sole means of direct inter-task communication[3] is via the passing of message-segments, and this has several implications.

The traditional approach to modularity and protection assumes that an entire program occupies a single real or virtual space. Separately compiled subroutines, procedures with restricted scope of variables, and, most recently, capabilities enforced by special hardware, have been introduced to provide "compartments" within the single space. The MININET approach is different. Each MININET task corresponds in size and function to a procedure or subroutine of the traditional approach and each MININET task lives in a private virtual space. There is, therefore, complete separation of information between tasks, as standard memory-mapping hardware serves to prevent access by a task to any items except those which are in its own virtual space. By the same token, the need to introduce special hardware such as capability registers to enforce protection rules has been avoided.

The passing of control from one procedure to another in conventional systems has been replaced in MININET by context switching from one task's virtual space to another's, and the passing of data by procedure parameters and shared variables has been replaced by the sending of message-segments between tasks. These functions are carried out by a task in each host called the message-switching task or Switch; it is described in the next subsection. The Switch is able to alter the access control attributes of a segment as it passes from task to task; this is the mechanism used for capability administration. Finally, the placing of tasks in separate, disjoint virtual spaces opens the way to placing them in separate, disjoint real spaces, i.e., microprocessors. Hence the MININET architecture facilitates the further distribution of processing functions at a host site, by allowing us to easily split off groups of tasks and install them in separate microprocessors.

3.4 The Communications Nucleus - The Edges of G(T)

We have modelled transaction processing by the directed graph G(T), whose nodes are tasks and whose edges are message flows. We have described the structure of tasks, which are the nodes of G(T), and have stressed that tasks run in private virtual spaces. Hence the edges of G(T) are significant, because they represent the only means of direct communications between tasks. The edges of G(T) are implemented by the communications nucleus of the MININET architecture, which is described here.

The communications nucleus is the MININET entity which transmits

message-segments between tasks. It consists of

i) a task in each host called its Switch Task or Switch,

ii) a processor attached to each host called its Communications Device, and

iii) a packet-transport subnetwork,

as shown in Figure III.5.

The Switch Task or Switch is a unique task in each host which is created at system startup time and is never destroyed. All message transmissions are carried out by invoking the Switch, which is therefore of central importance. A task asks the Switch to send one of its message-segments to another task by executing the call

SEND(FROM_NAME, TO_NAME, SM_NAME, MSG_NAME, SØ, RØ)

where the values of FROM_NAME and TO_NAME are task identifiers (labels of nodes in G(T)), SM_NAME is the name of the Software Machine associated with the receiving task, and MSG_NAME specifies which data segment of the sender is to be sent. SØ and RØ are the Sending and Receiving Options, and they provide for the creation and destruction of nodes in G(T) as the processing of a transaction progresses. Namely,

$$S\emptyset \in \{\text{preserve, destroy}\}$$

and specifies that the sending task is to be preserved or destroyed after the message has been transmitted.

$$R\emptyset \in \{\text{create, append}\}$$

and specifies that the receiving task is to be created (using the software machine specified and labelled with the node-label provided), or that the message is to be appended to an existing task having the given name. These four option values, together with the possible responses transmission successful and transmission blocked, are provided to control the creation and destruction of transaction graph nodes and thereby to allow the tasks of a transaction graph to synchronize themselves. Thus the communications nucleus does not synchronize tasks; it takes the view that the writers of transaction processing code are better able to implement the synchronization of their code and it provides facilities to allow them to do so.

As Figure III.5 suggests, message transmission between a pair of tasks living in a single host (intra-host transmission) is done by that host's Switch alone. However, if the tasks live in different hosts the other components of the Communications Nucleus are called into play. The other components are a packet transport network, which is any facility able to transport addressed packets from one host site to another, and the Communications Devices, which are small interface processors interposed between each host and the packet transport network. The CDs assist the Switches in the sending and receiving hosts to move message-segments; the algorithms used to do so are described in Section IV. Suitable packet transport networks would include the DATAPAC [15], ARPANET [1] or TELENET [16] networks for long-haul applications, or the Newhall Loop [17] network for networks contained in a single building. (The use of the Newhall Loop was assumed in the modelling study described in the companion paper [12]).

3.5 Summary

We have presented the MININET logical architecture in terms of two layers of abstraction. The top layer is a model of transaction processing based on the transaction graph G(T). G(T), like any graph, consists of nodes and edges, and the second layer of abstraction provided the construction of these from simpler objects. For the nodes of G(T) - tasks - the simpler objects were segments used to construct Software Machines and then tasks. For the edges of G(T) - message flows - the simpler object was the Communications Nucleus.

Although we described the components of the nucleus, we have not explained the algorithms which they execute and the protocols governing their interactions. This is the subject of Section IV.

IV. Protocols and Algorithms for the Communications Nucleus

4.1 Introduction

This section explains how the Communications Nucleus carries out the transmission of message-segments between tasks. It is, therefore, a description of a third layer of abstraction, underlying the edges of G(T). The logical structure of the nucleus is shown in Figures IV.1 and IV.2; the CDs and packet transport network disappear in Figure IV.2 because they are not invoked in intra-host transfers. Figures IV.1 and IV.2 are protocol diagrams as used, for example, by Walden [18] to describe ARPANET protocols; the circles are functional modules and the lines represent communications paths, for which protocols are required. The dashed lines represent

"logical paths", which are implemented by passing messages along the solid lines or "physical paths." (For example, the Switches of Figure IV.1 cannot communicate directly; they lie in different hosts and there is no physical path between them. Instead, they must communicate via the CDs and the packet transport network.)

This section first discusses an abstraction -- network wide addressing -- which figures prominently in the nucleus protocols. It then explains the protocols associated with the lines of Figure IV.1, several of which are formulated in terms of network-wide addressing.

4.2 Network-wide Address Spaces - A Format for Protocol Messages

The notion of a single address space spanning several hosts of a network was suggested by D. Mills [19] in 1972. By extension of standard addressing ideas one is led to consider both virtual and real network addresses; these are examined below.

A naive notion of a network-wide virtual address space would suggest a single (large!) virtual space containing all active tasks of all hosts. This would be difficult to implement and its value is hard to see, so it was quickly discarded. Instead, the following approach was tried and adopted.

A virtual address issued by one machine and referring to another raises two questions. First, where is it to be translated - at the sending or the receiving machine? Secondly, to which virtual space at the receiving machine does it refer? (Virtual addresses do not belong to machines; they belong to the virtual space of some task resident in a machine.) These

questions suggest the format for network-wide addresses of Figure IV.3. The machine id designates the machine to which the address refers and the address field contains the address-within-machine. The two mode bits represent the four possible addressing modes, which can occur when addresses generated by machine i are transmitted to machine j. These are as follows:

a) Real Mode:

An address issued by Mi is translated neither at Mi nor Mj. This provides real network addresses as described below.

b) Virtual Local Mode:

An address issued by Mi is interpreted as a virtual address in the space of some task of Mi. Hence it is mapped at Mi but not at Mj.

c) Virtual Remote Mode:

An address issued by Mi is interpreted as belonging to the virtual space of some task of Mj. This provides a mechanism for remote access to data objects, and the address is mapped at Mj but not Mi. This mode is in fact used in the message transfer protocol described in Section V.

d) Double Virtual Mode:

An address issued by Mi is interpreted as belonging to the virtual space of some task of Mi. The result of mapping is regarded as virtual address in some space of Mj. Hence the address is mapped both at Mi and at Mj. No obvious application of this mode is known to us; it is included for completeness.

The only virtual addressing mode presently used by the MININET nucleus is virtual remote. Specifically, a Switch Task is able to issue virtual remote addresses, which are interpreted as belonging to the virtual space of the Switch Task at another host. (Put another way, a Switch Task is able to address objects in the virtual spaces of other Switch Tasks by issuing network virtual addresses.) This use of virtual network addresses is in connection with the allocation request message of the segment transfer algorithms, and is explained below.

The real mode provides a form of real address space spanning the network. It can be thought of as an extension of Bell's UNIBUS construct [20], where the conventional linear address space is extended to span the data and control registers of all peripheral devices (the UNIBUS I/O page) as well as the cells of primary store. Real mode network addressing takes this notion one step further and spans the primary stores and I/O pages of all machines of the network, in a single space.[4] Needless to say, access to Real Network Address Space (RNAS) by application tasks is forbidden; real network addresses are used only in the Communications Nucleus and are used by the CDs to move segments from primary store of a sender to primary store of a receiver. The details are explained fully in connection with the transmission protocols, which are described below.

4.3 Protocols and Algorithms

This subsection describes the protocols which govern the communication paths (lines) of Figures IV.1 and IV.2, and the algorithms which the modules (circles) execute. The purpose of the whole structure is to move a message-segment from one task to another.

4.3.1 Task Protocols

As Figure IV.1 suggests, there are two types of protocol associated with tasks; the task-to-task protocols and the task-to-switch protocol. Task-to-task protocols are not the business of the nucleus; it simply provides facilities for inter-task communication (segment transmission, and a SIGNAL facility as in UNIX [21] to alert the receiver that a message has arrived), but makes no assumptions as to how these are used by client tasks. The task-to-switch protocol is built around the SEND call described in Section III; the Switch provides return codes to the calling task to indicate "transmission successful" or "transmission blocked because of XXX." Note that the format of the SEND call is identical for intra-host and inter-host transmissions; hence the location of tasks is hidden and hence the basis for making the distribution of data transparent to higher level software modules.

4.3.2 Intra-host Switch Algorithms

Although the sending task uses the same call for intra and inter-host transmission, the Switch's actions are widely different for these two cases. The intra-host case is simpler and is described first.

For efficiency, intra-processor message transmission is always done by switching a pointer to the message-segment; the physical moving of segments is therefore avoided. Also, if the switch owns a dedicated set of CPU state registers, the overhead of context switching can be reduced. The four possible modes of intra-host message transmission are carried out by the following algorithms.

i) SØ = destroy, RØ = create

The Switch scans the task queue (the task queue and Master Segment Table exist within the Switch's virtual space) to locate the TQE of the sending task S. It changes the code panel pointer of the TQE to point to the code panel of the receiving SM, sets the virtual program pointer to the initial entry value for that SM, and frees the data segments pointed to by the data panel (except for the message-segment and the initial working storage for the new task). The task id field is changed to the name of the new task R.

ii) SØ = destroy, RØ = append

The Switch scans the task queue to locate R's TQE (error return if it does not exist) and hence the STE of one of R's available message-segments (transmission blocked if there is none). The segment pointer of this STE is changed to point to the message-segment coming from S. The TQE of S is then put on a list of free TQEs and the data segments

it points to are marked free. Flow control is exercised by receiving tasks in this manner; a task can dynamically control the maximum number of messages it wishes to receive by varying its number of empty message-segments.

iii) SØ = preserve, RØ = create

The Switch moves an unused TQE from the free list to the task queue and sets its pointers to point to the code panel of the receiving SM and a new data panel. One STE of this data panel is pointed to the message-segment; the corresponding pointer of the sending task's data panel is erased.

iv) SØ = preserve, RØ = append

The Switch locates R's TQE and hence its data panel. It searches the panel for the STE of an empty message-segment. If there are none the Switch returns the "transmission blocked" code to S; otherwise the STE found is made to point to the message-segment. The STE of the message-segment in S is modified by erasing the segment pointer and setting its length field to zero.

In summary, this scheme seeks to minimize the excessive overhead in message passing which has plagued other message-switched operating systems, through the exclusive use of pointers to pass messages and efficient context-switching to the switch task. (The use of pointers to avoid this overhead has also been suggested by Wecker [22].) It also avoids

congestion problems due to long input queues on tasks, by allowing each task to exercise control over the number of message-segments passed to it. Synchronization is performed by the transaction tasks themselves, using the four possible calls to the switch and appropriate responses to the returns Transmission Successful or Transmission Blocked.

4.3.3 Inter-host Switch and CD Algorithms

We have just discussed the transmission of message-segments between two tasks running in the same host of the network. We now discuss the more complicated case in which the tasks run in different hosts. The task-to-switch protocol remains the same, in order to make the distribution of tasks among hosts transparent, but the other algorithms and protocols are different. Finally, the layering principle is not strictly observed between the Switch and CD layers of Figure IV.1; this violation was made for the sake of efficiency and it means that the Switch and CD algorithms have to be described together.

Briefly, the problem to be solved is the following. Using information passed to them by the two Switch tasks, the two CDs have to move a message-segment from the primary store of the sending host to the primary store of the receiving host, using the packet transport network for communications between CDs and imposing as little overhead on the host CPUs as possible. To do so, the sending CD has to determine the base address and length of the message-segment in the sending host's primary store, and the receiving CD has to be told where to put the segment in the receiving host's primary store, i.e., the location of an empty segment frame of suitable

length. Also, the unit of information or packet accepted by the packet transport network is generally of different maximum length than message-segments, so the segment has to be fragmented into packets at the sending end and re-assembled at the receiving end. A virtual circuit has to be built by the sending and receiving CDs to ensure end-to-end preservation of order, and they have to support multiple concurrent segment transfers to ensure adequate response time.

Rather than give a formal, precise description of the Switch and CD protocols, we illustrate the major ideas with an example. The example describes a message-segment transfer or call from task M1.S of sending host M1, to a task M3.R of receiving host M3. The Switch Tasks and CD of M1 are referred to as M1.Switch and M1.CD, and similarly for M3. The steps of the call are as follows.

i) M1.S creates a message segment and invokes M1.Switch with the command and parameter list

SEND(FROM_NAME, TØ_NAME, SM_NAME, MSG_NAME, SØ, RØ).

This is exactly the same command and parameters as are used in the intra-host case described above; hence the calling task M1.S need not be explicitly aware of the distribution of tasks among the hosts of the network.

ii) M1.Switch examines the task graph name TO_NAME and so determines that the receiving task is in another host; hence inter-host transmission is required. M1.Switch appends

a pair of Network Address Space Addresses (called the TO_NASA and the FROM_NASA) and the STE of the message segment to the parameter list, then passes all of these as parameters to the local CD, M1.CD. Two commands accompany these parameters. One is for the local CD; it is the command START_CALL. The other is for the remote switch; it is passed via the CDs and the packet network and is the command ALLOCATE.

The NASAs belong to the switch-switch protocol and are defined as follows. The FROM_NASA is a Real NASA (mode bits 00) and gives the starting address in M1's primary store of the message segment. The TO_NASA is a Virtual Remote NASA and refers to the virtual space of the receiving Switch M3.Switch. If task M3.R does not yet exist then M3.Switch must create it and then pass it the message segment; in this case (SØ = CREATE) the TO_NASA points to the code panel of the appropriate Software Machine in M3.Switch's virtual space. If task M3.R does exist, (SØ = APPEND) the TO_NASA points to the task queue in M3.R's virtual space.

iii) M1.CD responds to the START_CALL command as follows. It first creates a Call Control Record or CCR in its private primary store; there is one CCR for each call in progress and the CCR contains all information required to process the

call. Specifically, the CCR has a call id which uniquely identifies this call, and a call state variable, which is initially set to START and which indicates how far the call has progressed. Finally, the parameter list passed from M1.Switch is held in the CCR.

Secondly, M1.CD creates a packet whose text contains a copy of the CCR. This allocation request packet is offered to the packet network for transmission to M3; it contains a command field for M3.CD with value BEGIN_CALL.

iv) When the remote CD, M3.CD, receives the allocation request packet it acknowledges receipt, strips off the packet header and examines the command field. The command is BEGIN_CALL so M3.CD stores the rest of the text in a new Call Control Record in its primary store. The switch command (ALLOCATE) is passed on to M3.Switch; the parameters are the NASAs, the task graph names, the options SØ and RØ and the STE.

v) The remote Switch task M3.Switch treats the options RØ = CREATE and RØ = APPEND differently, If CREATE, it uses the TO_NASA (which points to the code panel of the SM to be used to create the receiving task) to create the receiving task, whose identifier is the value of TO_NAME. Storage is allocated for the newly-created task's data module, and one segment of this storage is an empty segment or segment frame of type message, which will receive the message-segment from

M1.S. If APPEND, the Switch searches the task queue for the task whose name (M3.R in this example) matches the value of TO_NAME. (The call is blocked if no match is found.) The Switch then looks for a data segment of M3.R with attribute message and length zero; if one is found it is grown to the necessary length. (The call is blocked if no such STE exists or if insufficient store is available.) In either case -- CREATE or APPEND -- the Switch builds a real NASA pointing to the empty message-segment, and passes it back to M3.CD. M3.CD stores the real NASA in its CCR, and passes it back to M1.CD which also stores it in its CCR.

vi) The two CDs now have the real primary store addresses of the message-segment in M1's primary store and of the segment frame in M3's store. M1.CD therefore initiates a sequence of packet transmissions to M3.CD; each contains the command STORE_BLOCK, the call_id, a sequence number to ensure that blocks are stored in the segment frame in the correct order, and a block of the message-segment.

The blocks are fetched by M1.CD from M1's primary store using DMA operations and are stored by M3.CD in M3's primary store by the same technique. (Hence no CPU overhead is incurred after the initial allocation request is made.) When the last block has been transmitted, stored and acknowledged, the CDs interrupt the Switch tasks with the

code "transmission successful," and destroy the Call Control Records. M1.Switch returns the code "transmission successful" to task M1.S, M3.Switch schedules the receiving task for execution, and the call is complete.

V. Implementation

This section describes a prototype implementation of the MININET logical architecture. It was undertaken to try out the ideas presented above, to obtain results on performance, and to stimulate further research. We discuss our choice of hardware, languages and methods and the techniques used to implement the MININET logical architecture, then we present some preliminary numeric data on performance.

5.1 Hardware, Languages and Methods

We wished to use available, off-the-shelf hardware in order to obtain a working prototype as quickly as possible. Almost any computer could be used in principle to implement the MININET architecture, but there are several desirable features. A suitable machine should possess segmented virtual memory (as the segment plays a central role in the architecture), a flexible bus structure, and good software development tools. It should also belong to a family of ISP-compatible processors to allow further distribution of processing at a host site. For all of these reasons, and, to be honest, because it was available to us, we chose the PDP-11 family of processors. The PDP-11 Model 45 was chosen for the hosts, and the PDP-11 Model 20 was chosen for the CDs.

The Queen's University version of the PL-11 language [23] was initially chosen as our implementation language, and the CD and Switch programs were written in QPL-11. However, the advent of the C language and a suitable C compiler under the UNIX operating system [21] changed our views. The Switch was rewritten in C and the CD program is being rewritten in C as well. All software - the Switch, CD program and Software Machines - is thus written in C and is developed and tested under UNIX, in order to exploit the powerful and pleasing environment which UNIX provides for software development. Completed and tested load modules are then shipped to MININET hosts where they are loaded into the MININET environment. (A test-bed package was written to allow testing of Software Machines under UNIX.) This approach has been very satisfactory and has drastically reduced the need for software testing and development tools under the MININET run-time environment. This is consistent with our principle, that MININET users are people who generate commercial transactions, not programmers who write and debug software. The latter class of people use UNIX as a "software factory."

5.2 Implementation of the MININET Architecture

5.2.1 Packet Transport Networks

The prototype implementation is a two-node network. In this case switching of packets is not required and the packet-transport network is unswitched. It is in fact a pair of wires terminated in asynchronous line drivers and running at 19 kbps. (A Newhall Loop is under test in our

laboratory and will be substituted for the wire pair when it is operational.) Finally, the SNAP II access protocol for the DATAPAC [15] public packet network has been implemented separately, and we hope to use DATAPAC when it is operational.

5.2.2 Hardware Interfaces

The host/CD interface is a pair of DEC DR-11A word transfer devices modified to run back-to-back. It is supported by a protection scheme which checks the access rights of a program to another program's store area. Transfer rates of 19 kbps have been obtained.

The CD/communications line interfaces are DEC DC-11 asynchronous drivers supported by standard software. Provision has been made to substitute Newhall Loop and DATAPAC software as mentioned above. The host/terminal interfaces are also DC-11s.

5.2.3 Software Constructs

MININET segments are implemented by PDP-11/45 segments. These have a maximum length of 4K words, grow in 32-word increments and are managed by the 11/45 Memory Management Unit. The STEs of active segments are held in the registers of this unit (maximum of 16) and virtual to physical address translations are done in 90 nsec. Hardware Segmentation is not obligatory; in fact, the first version of the Switch built segments in software. Software Machines and tasks are implemented from 11/45 segments in the manner described in Section III.

The Switch task occupies 11/45 Kernel Space, which has a private set of CPU registers, and it is the sole occupant of this space. The SEND

call is implemented by a trap instruction to Kernel mode with parameters passed on the system stack. In addition to the SEND call which invokes the Switch, there are a number of auxiliary calls to allow resource management, both of memory (create, shrink, grow, destroy, and activate segments) and of the CPU (task sleep and wakeup). These are handled by auxiliary system tasks which live in Supervisor Space (the 11/45 hardware provides User, Supervisor and Kernel mode spaces) and are implemented by trap instructions to Supervisor mode.

Network Address Space Addresses or NASAs are implemented purely in software, as the 11/45 lacks the necessary hardware. In the prototype implementation, therefore, NASAs are simply formats used to express Switch and CD protocol messages.

5.2.4 Performance

We have implemented a two-node MININET consisting of a two-node Communications Nucleus, terminal software and a file system (borrowed from UNIX). A suite of Software Machines to do simple transaction processing is under development. The Switch, including all auxiliary system tasks which do resource management, comprises 12K words of 11/45 code and the CD program comprises 15K words of 11/20 code. Timings for the SEND system call are given in Table V.1.

VI. Conclusions and Further Work

6.1 Conclusions

The consistent use of message switching throughout the MININET architecture -- a packet-switched communications subnetwork interconnecting

message-switched host operating systems-- has led to a uniform, elegant architecture for distributed processing. This particular approach has not been previously discussed to our knowledge [6,7,9,10]. The notions of real and virtual network address spaces have been developed to provide a useful format for communication among switch tasks, and considerable generality for applications not yet foreseen has been provided as well. The use of multiple small, disjoint virtual spaces provides a powerful mechanism to enforce the modularity required by the graph model of transaction processing, and provides a technique for controlled information sharing without capability registers. It also has opened the way to exploitation of microprocessors by enforcing a rather rigid isolation of tasks. Finally, although the prototype implementation involves a two-node homogeneous network using PDP-11 processors, the logical architecture is amenable both to other processor designs and to heterogeneous networking. The major limitation hinges on locality of reference; the design only works well if the percentage of transactions which "go remote" is small. However, many collections of data of practical interest do in fact obey locality of reference.

6.2 Further Work

The analytic and simulation modelling studies reported in the companion paper [12] were done while the MININET prototype was being implemented. Consequently, comparisons between the analytic and simulation results were made and reported, but comparisons of both with the "real thing" were not possible. As soon as the Newhall Loop is fully operational we

intend to make such comparisons.

Turning to hardware issues, we have shown that the MININET logical architecture can be implemented with good efficiency on standard PDP-11 Model 20 and 45 processors. However, a number of steps toward a better hardware architecture for MININET are now clear. These include:

1) A much larger number of segmentation registers, to permit a larger number of simultaneously active segments.
2) Use of auxiliary small processors such as the DEC LSI-11 to run certain tasks, notably the Switch and the terminal manager.
3) Introduction of a separate processor to control secondary store. This processor or Data Host would contain a copy of the Switch, and all of the file tasks; file system queries would be presented to it as messages and records would be returned. Fraser's freestanding file system [10] and a proposal by the authors [24] provide models to follow.
4) Hardware implementation of NASAs. In the present implementation, NASAs are little more than a useful abstraction or format for Switch and CD protocol messages. The possibility of embedding them into the hardware should be studied.

Finally, there are several areas for future software work, of which the largest concerns distributed data management. The present architecture provides a distributed file system which exploits locality of reference and allows for simple report generation across the file system.

In no sense does it provide data management facilities as contemplated in the relational or DBTG proposals; however, it is a plausible vehicle on which to build such facilities for data bases exhibiting locality of reference. We therefore want to answer two questions. First, what subset of the relational or DBTG facilities is really needed by commercial users? Secondly, what algorithms can be devised to provide the subset in the MININET architecture? Other areas for future software research include implementation of the MININET logical architecture in a network of heterogeneous hosts of the ARPANET or CYCLADES class, and comparisons of the efficiency of the MININET protocols for data sharing with those proposed or implemented for ARPANET or CYCLADES.

6.3 Acknowledgements

We want to acknowledge the contributions of Jon Livesey, Kelvin Delbarre, Francas DuBois and David Lamb, who implemented the MININET prototype. Thanks are also due to Eileen Gilchrist who typed this manuscript, and to Ken Thompson and Dennis Ritchie who provided such a superb environment for software development. Support of the National Research Council of Canada is gratefully acknowledged.

FIGURE CAPTIONS

TABLE CAPTIONS

REFERENCES

1. Roberts, L.G., Wessler, B.D., "Computer Network Development to Achieve Resource Sharing," AFIPS, SJCC Vol. 36, May 1970, pp. 543-549.

2. Roberts, L.G., "Network Rationale, A 5-Year Reevaluation," COMPCON 73, San Francisco, February 1973, pp. 3-5.

3. Pouzin, L., "Presentation and Major Design Aspects of the Cyclades Computer Network," DATACOM 73, ACM/IEEE 3rd Data Communications Symposium, St. Petersburg, Florida, November 1973, pp. 80-87.

4. Barber, D.L.A., "The European Computer Network Project (COST)," ICCC 72, Washington, D.C., October 1972, pp. 192-198.

5. Brunel, L., "A telecommunications Network for a Multi-campus University," ONLINE 72, Conference Proceedings, Vol. 2, September 1972, pp. 45-60.

6. Howell, R.M., "The Integrated Computer Network System," Proc. ICCC 72, October 1972, pp. 214-219.

7. Farber, D.J., Larson, K.C., "The System Architecture of the Distributed Computer System - The Communication System," Symposium on Computer-Communications Networks and Teletraffic, Polytechnic Institute of Brooklyn, New York, April 1972, pp. 21-27.

8. Farber, D.J., Feldman, J., Heinrich, F.R., Hopwood, M.D., Larson, K.C., Loomis, D.C., Rowe, L.A., "The Distributed Computing System," COMPCON 73, San Francisco, February 1973, pp. 31-34.

9. Aiso, H., Matsushita, Y., et al., "A Minicomputer Complex - KOCOS," IEEE/ACM Fourth Data Communications Symposium, Quebec City, October 1975, pp. 5-7 to 5-12.

10. Fraser, A.G., "A Virtual Channel Network," DATAMATION, February 1975, pp. 51-56.

11. Fabry, R.S., "The Case for Capability Based Computers," Proc. Fourth Symposium on Operating System Principles, Yorktown, New York, October 1973.

12. Manning, Eric, Peebles, R.W. and Labetoulle, J., "A Homogeneous Network for Data Sharing: Modelling and Analysis," submitted to Computer Networks.

13. Scantlebury, R.A., Wilkinson, P.T., "The Design of A Switching System to Allow Remote Access to Computer Services by Other Computers and Terminal Devices," ACM/IEEE Second Symposium on Problems in the Optimization of Data Communications Systems, October 1971, pp. 160-167.

14. Brinch Hansen, Per, Operating System Principles, Chapter 8, "A Case Study: RC-4000," Prentice-Hall, 1973.

15. Trans-Canada Telephone System, "DATAPAC - Standard Network Access Protocol Specification," document published by TCTS, Place Bell Canada, Ottawa, March 1976.

16. Mathison, S.L., "Telenet Inaugurates Service," Computer Communication Review, ACM, Vol. 5, Nr. 4, October 1975, pp. 24-29.

17. Farmer, W. D. and Newhall, E.E., "An Experimental Distributed Switching System to Handle Bursty Computer Traffic," Proc. ACM Conf., Pine Mountain, Georgia, October 1969.

18. Walden, David C., "Host-to-Host Protocols," Network Systems and Software, Infotech Report 24, Infotech Information Ltd., Maidenhead, Berks., U.K., pp. 287-316.

19. Mills, D.L., "Communication Software," Proc. IEEE, Vol. 60, Nr. 11, November 1972, pp. 1333-1341.

20. Bell, C.G. and A. Newell, "Computer Structures: Readings and Examples," McGraw-Hill, 1971, 668 pages.

21. Ritchie, D.M. and Thompson, K., "The UNIX Timesharing System," CACM, Vol. 17, Nr. 7, July 1974, pp. 365-375.

22. Wecker, S., "A Building Block Approach to Multi-Function Multiple Processor Operating Systems," Proc. AIAA Computer Networks and Systems Conference, Amer. Inst. Astronautes and Aeronautics, Huntsville, Alabama, U.S.A., April 1973.

23. Jardine, D.A., "QPL-11 Language Manual and Programmer's Guide," research report, Dept. Computing and Information Science, Queen's University, Kingston, Ontario, Canada, March 1974.

24. Peebles, R.W. and Manning, E.G., "A Computer Architecture for Large (Distributed) Data Bases," Proc. Conf. on Very Large Data Bases, ACM, Framingham, Mass., September 1975.

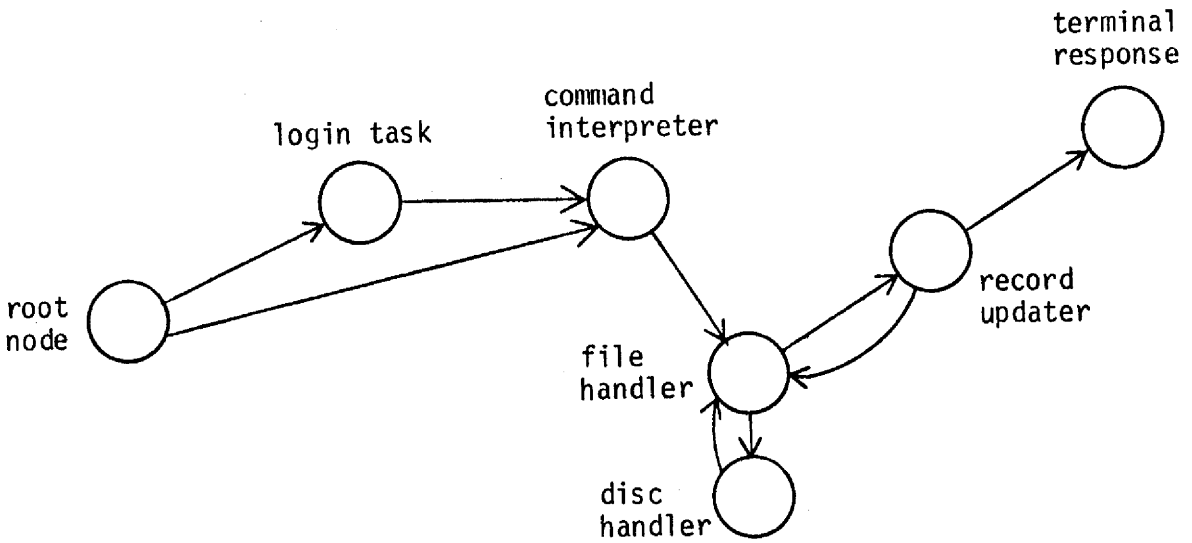

Figure III.1 - A Transaction Graph

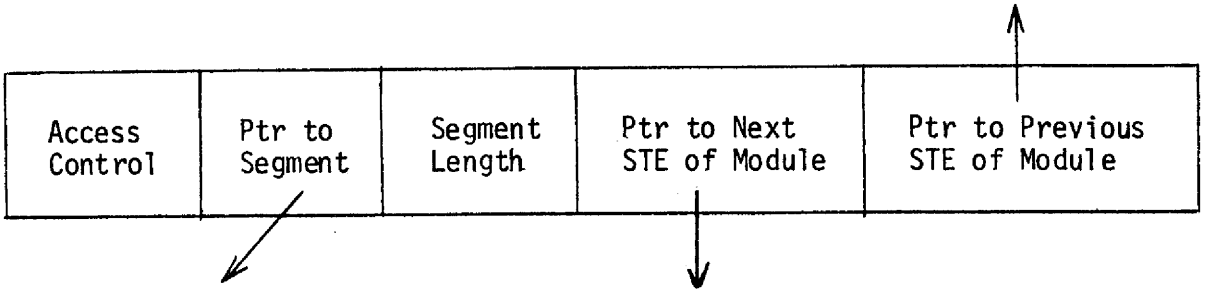

Figure III.2 - Segment Table Entry

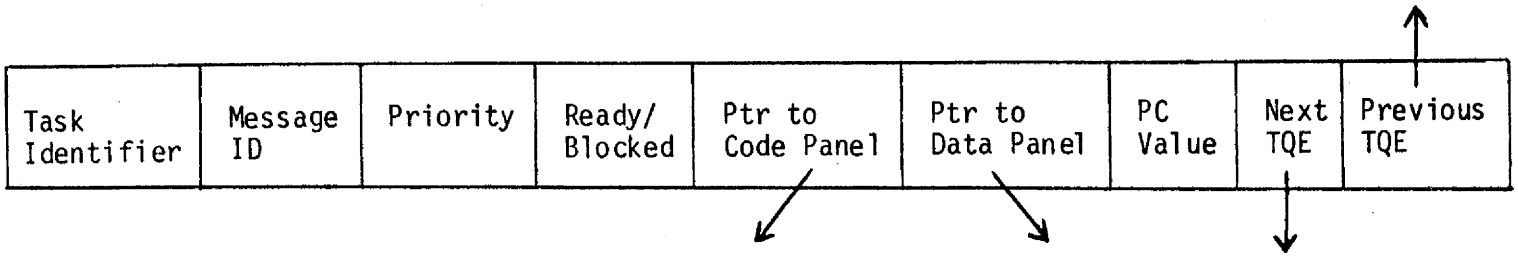

Figure III.3 - A Task Queue Entry

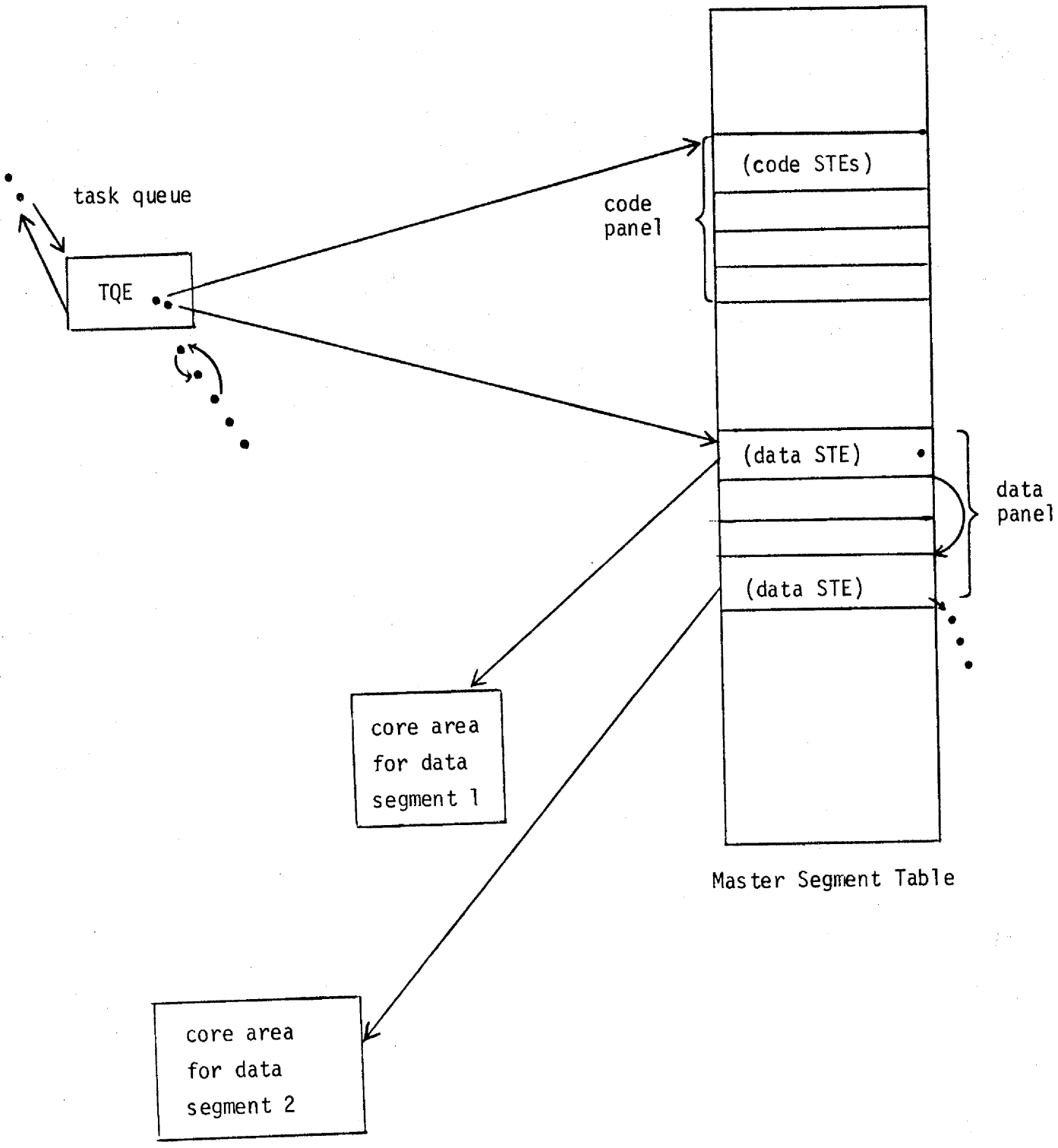

Figure III.4 - Relationships Among Task Queue and Master Segment Table

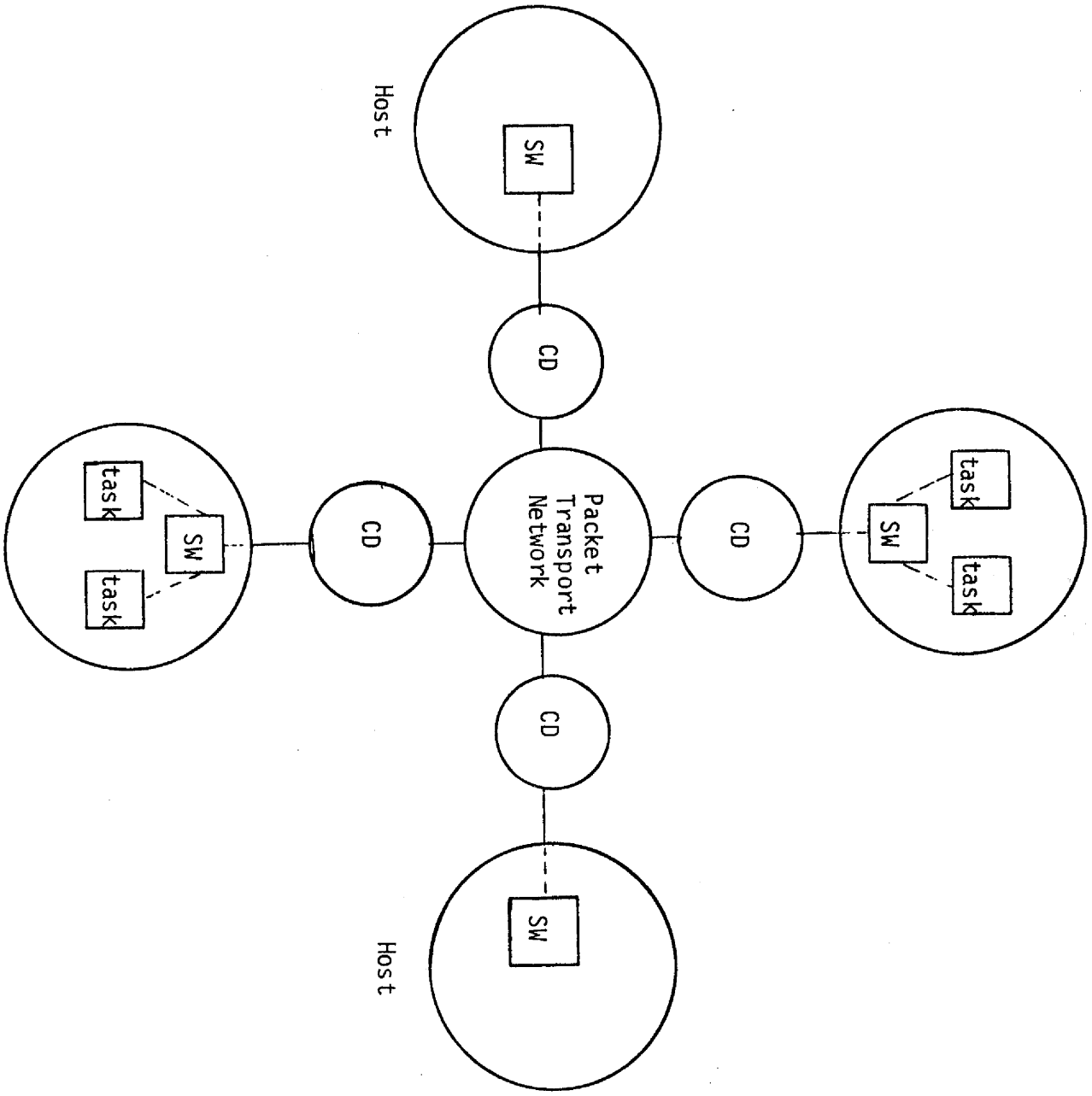

Figure III.5 - MININET Communications Nucleus

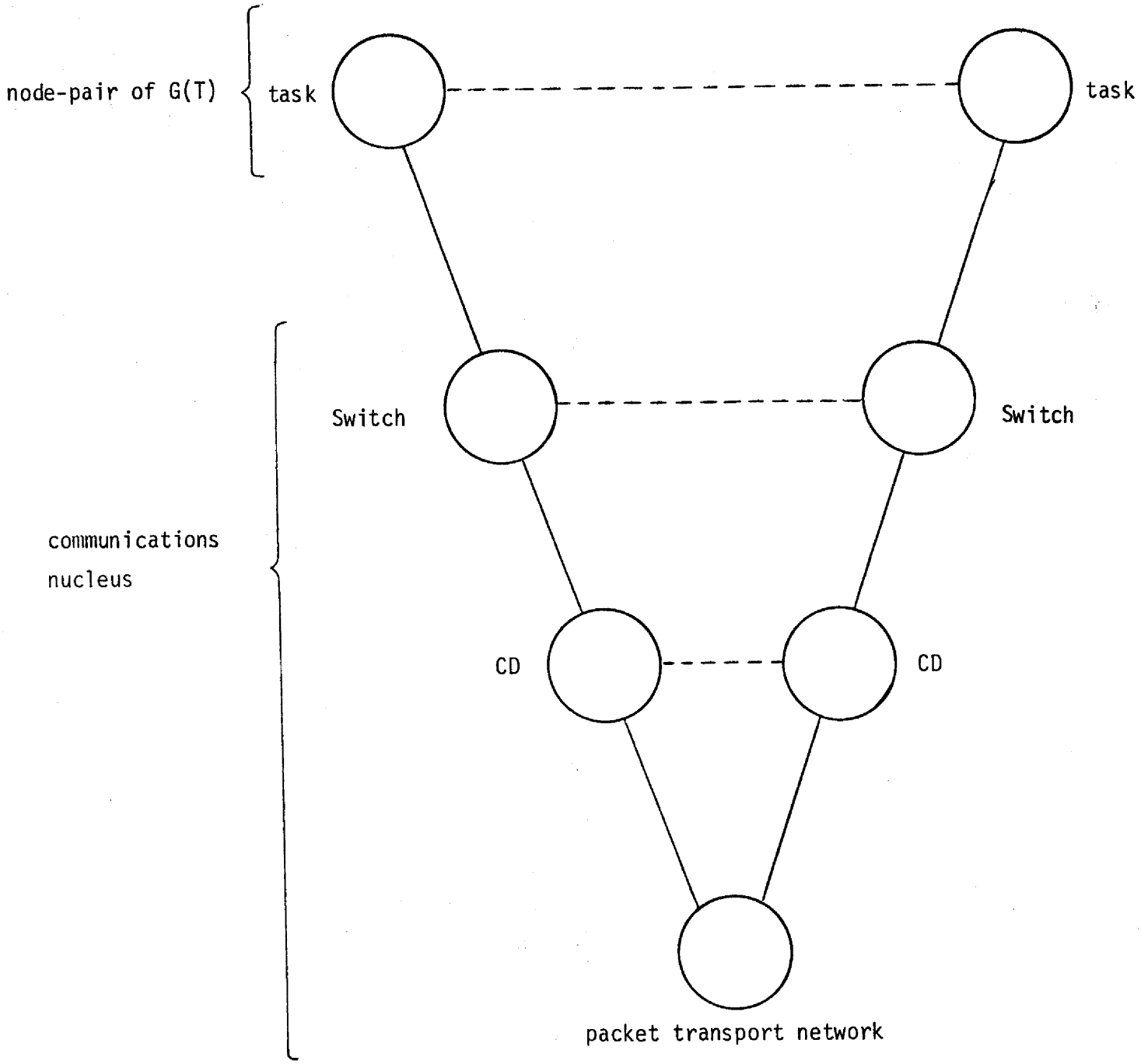

Figure IV.1 - Expansion of a Node-pair of G(T);
Inter-host Transmission

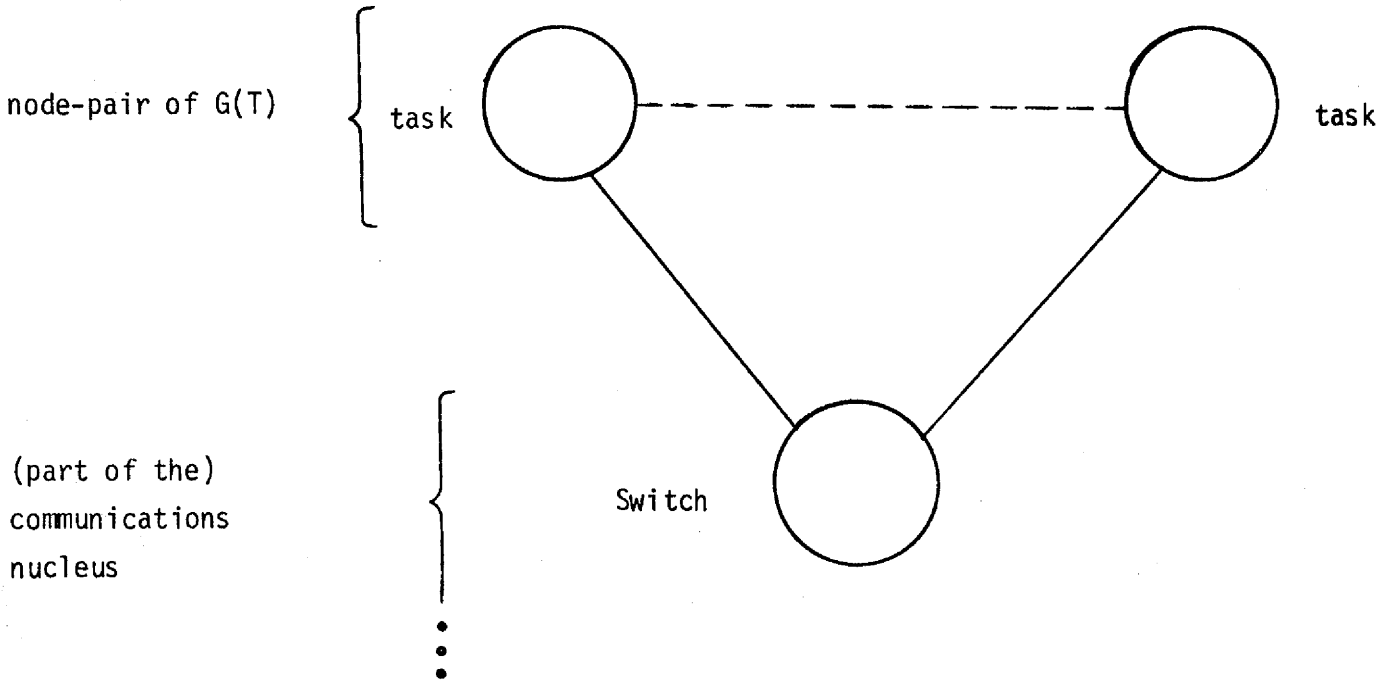

Figure IV.2 - Expansion of a Node-pair of G(T);
Intra-host Transmission

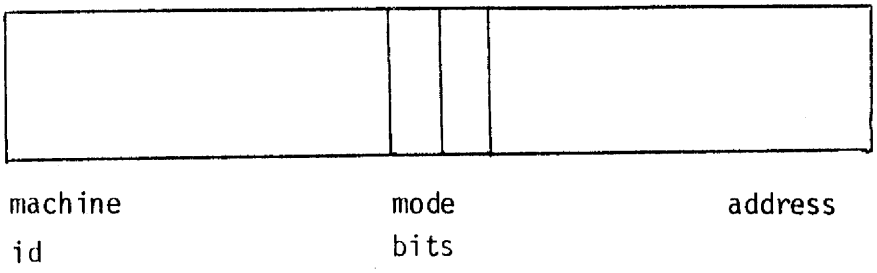

Figure IV.3 - Format for Network-wide Addresses

Options		Transmission Time in Milliseconds
SEND	RECEIVE	
preserve	append	1
destroy	append	3
preserve	create	6
destroy	create	8

(a) Intra-host segment transmission

Segment length*	Transmission Time in msec+
32 words	250

(b) Inter-host segment transmission

*transmission time is approximately a linear function of segment length

+for a line speed of 19kb/sec. measured with only one call in progress

Table V.1 - Timings for the SEND System Call