Table of Contents

1. Introduction

NetWork is a collection of software tools for experimenting with distributed computing in a network of personal workstations. It is an approach to make shared use of the computing resources in a network while respecting the absolute priority of an individual user on his/her machine.

Since a user may choose to use his machine at any time, absolute priority of individual users implies that availability of shareable computing resources cannot be guaranteed over the network. Moreover, since network access might be critical to individual users, network load due to distributed computing has to be minimized. The general ideas of NetWork are discussed in a separate article [1]. The requirements are repeated here, because they influence the design heavily and will be referenced throughout this document:

- immediate availability of any machine to its home user
- minimal interference with communication requested directly
- independence of communication system
- adaptability to heterogeneous hardware

NetWork has been designed to support distributed, asynchronous computations. However, the concepts and facilities provided are general enough to be used for other computations as well. In fact, NetWork implements *interprocess communication* using messages, but it assumes a number of special properties of these processes. These can be overridden to implement other models of cooperating processes, e.g. you can use it to implement traditional server/client concepts.

NetWork provides the following services

- idle state determination
- process management
- message transfer
- lookup of available partners
- logging

NetWork tries to isolate these individual services. Also, as long as it does not cause problems or severe performance losses, generality is preferred.

The current implementation is based on a network of Macintosh computers, but it can be used in heterogeneous networks as well. Because there are implementations of AppleTalk on platforms like VAX/VMS or Unix, the NetWork model can be ported easily to these environments. It is also possible to utilize different transport protocols, e.g. TCP/IP, DecNET, both available on the Macintosh, which may broaden the scope of NetWork.

This document serves two tasks: it provides a description of the communication model used, and it also documents the specific NetWork implementation for the Macintosh.

NetWork is a tool to be used for experimentation. It is not a final product, nor does it try to be perfect in every respect. There will be hints and suggestions for further experimentation and improvements throughout this document. Chapter 6 gives a summary of tricks we were forced to use, to work around a number of problems in the Macintosh Operating System. Apple employees should interpret this as a list of things “to do”.

2. NetWork Design Goals

The following sections each expands on one of the requirements stated above.

2.1 Immediate availability of the machine to its home user

Immediate availability of the machine to its home user requires that a machine is available for distributed processing only if the user is inactive and has no tasks executing that require the machine´s computing resources. If a home user becomes active, all processes created by other users must be terminated as fast as possible.

At process termination processes should not insist on returning results or save state, although this behaviour is not enforced by NetWork. A process that saves state or returns result uses up processor time and/or memory that the user may need. A user will expect to regain control of the machine immediately. If there are noticeable delays, he is likely to disable NetWork cooperation in the future.

One might argue that it is sufficient to lower the process priority in order to preserve the user´s right to receive prime service. However, this is generally not sufficient, at least because of two reasons: first, most operating systems lower the priority of processes that are time consuming. Eventually this may cause all user processes to be lowered in priority to the level of NetWork processes and therefore cause a delay of the user´s work. This problem may be solved by operating systems like Mach. However even Mach does not address the second problem: every active process will require memory or swap space for its state to be preserved. This memory may be required by the user.

Because NetWork processes do not exist while the user is active, there must be a mechanism that causes a process to be created, assuming the machine is available. In order to minimize communication overhead, we decided that a process is created implicitly by sending a message to it. Upon receipt of a message, NetWork will test whether the destination process exists and if it exists forward the messages to the process. If the destination process does not exist it will be created if the machine is available for use.

There is one immediate consequence: process identifications cannot rely upon the existence of the process. The identification must be known in advance and – preferably – independent of the system in use.

Another consequence is, that all communications must take place asynchronously. There is no use in waiting for a result that may never arrive. Also acknowledgments are not useful either. Even if you get an acknowledgment, you can never be sure that the user does not access the machine the very moment after you sent the acknowledgement. Therefore the only useful acknowledgment is a result.

2.2 Minimal interference with other communication requested

NetWork is supposed to cause *minimal interference with communication requested directly.* Ideally, we´d like to have a sort of “second class network access”. NetWork is supposed to utilize resources that are currently unused. Using up network bandwidth that may be required by other users is not a good idea. However, there is currently no network access mechanism that allows to do this. IP [11] supports two classes, but they are the other way round: standard and priority. Also, the distinction between priority and standard is likely to be evaluated in routers only. So here is a “to do” for system manufactures: implementation of a second class network access.

Because second class network access is not available, it is necessary to minimize the number of packets transferred. We decided to split the information load into two logical components, “priority” information and “core” information. If a message is sent, priority information is transferred at once and delivered to the destination process. Note that the use of the term “priority” is different from the one in an IP environment above. Unless there is a network access mechanism that supports a second class network access method, all packets are sent “standard”. If there was a second class network access, all information whether “priority” or “core” should be sent “2nd class”.

The destination process is given opportunity to look at the priority information and, depending on its contents, to decide, whether the remainder, the core of the message, should be transferred or not. There are several possibilities, why a process may wish to discard the information: the process may be busy or the information may be outdated or duplicated. Also, the process may not be available at all.

The separation allows to minimize the number of packets because not all of the core information is transported in these cases. The NetWork communication will pack as much of the core information into the first packet as will fit. If a message is accepted, this is the most effective technique. If a message is not accepted, just one packet has been sent.

Many datagram oriented protocols support broadcasts. Broadcasts – or multicasts – are useful to transmit data to all machines, where “all” is generally used for all machines attached to one cable. However, the data will be received by machines that do not need it, and discarding unwanted data will cost processor time. Therefore broadcasts are of questionable value. NetWork does not encourage use of broadcasts, but for those who want to experiment with broadcasts, we included the possibility to do so. Also we are considering the explicit use of broadcasts to discover idle machines. The current implementation uses AppleTalk name lookup [4] to discover idle machines, which uses broadcasts implicitly. If we make the lookup an explicit broadcast, we can obtain more information than we can obtain using AppleTalk name lookup without additional packets.

Typically session oriented protocols like TCP/IP or ADSP should be avoided, because of the cost of these protocols. In order to get a message of just a few bytes from a source to a destination process, a session oriented protocol will require about six packets, whereas a datagram oriented protocol will require just one. However this does not preclude use of session protocols, and we will outline a technique using them in chapter 7.

2.3 Independence of communication system

NetWork tries to isolate applications from the communication system in use. NetWork provides facilities to ask for the network or protocol address of other machines. A NetWork application should not assume any particular representation of addresses but depend on NetWork to learn about other machines. As far as the application is concerned, the machine´s address may be anything. Possible interpretations are:

- a 4 byte IP number
- the low two bytes of an IP number, with a port number added – assuming that the network portion is unlikely to change
- an AppleTalk internet address – which includes a socket number
- a DecNET node address,
- an index into a table of hosts.

The Macintosh implementation of NetWork supports multiple transport systems that can be active simultaneously. Currently, there are two transport systems supplied: One that provides local delivery using fast memory to memory copy, and one that supports communication using AppleTalk. A third “pseudo” transport system – called “Dispatcher” – is used to dispatch between the further two based on the destination address.

By its design criteria NetWork applications cannot assume that the transport system is reliable – messages can be lost or destroyed because a machine becomes used – but NetWork applications can assume that if they receive a message (see [2] for details), they are guaranteed that either the message is received completely, or there will be an error indication. This simplifies application design, because initialization data can be sent as one message, and the application either gets or does not get the message, but it will never be concerned with something like “I missed part 4 of 5”. While this is true conceptually, a communication system may place restrictions on the size of a message.

2.4 Portability to other systems

NetWork does not assume that all machines it is communicating with are of the same type. Although this could have been done easily using NetWork itself, NetWork does not implement code distribution because of two reasons: first a VAX will not be able to execute a Macintosh program and vice versa. Second, a code distribution technique also introduces the possibility of viri, and we did not want to implement a new super virus.

NetWork can be ported to other hardware or operating systems. If you are going to port to other systems, you will need to define and implement the following:

- availability of the machine, i.e. when to consider a machine idle,
- how to address processes, how to create and terminate processes,
- which communication protocol to use locally and for communication with other machines.

3. NetWork Terms and Concepts

NetWork implements *interprocess communication* using messages. A message can be sent from one process to another one. Messages are always sent to processes, not to machines. Addresses always consist of two parts: a machine address, usually its network (protocol) address, and a process identification, which is discussed below. Any machine participating is expected to have a NetWork communications agent, that will handle the message transfer on behalf of the processes. On the Macintosh the NetWork communications agent is called *NetWork Processor*.
NetWork does not impose any specific type of topology on the cooperating processes. Although NetWork can be used to write client/server applications, NetWork does not assume any particular relation of processes. NetWork does not maintain sessions. There is one exception to this rule which involves processes on the same machine. This is discussed in section 3.2.

3.1 Processes, Program Numbers & Signatures

As we outlined above, process identifications must be known in advance. We borrowed a concept of Sun´s Remote Procedure Call (RPC) to accomplish that [9]. Readers, who are familiar with Sun´s RPC, know that RPC uses a unique program number (32 bits) to dispatch received messages – especially those sent to the port mapper´s port. The program number must be unique and registered with Sun.

Readers, who are familiar with programming the Macintosh [3], know that programmers must register a *signature* (four characters, or 32 bits) with Apple, if they want to take advantage of the bundle concept. The bundle tells the Finder, which icons to display, and it also tells which application is to be used to open a specific type of data file. This signature must be unique and registered with Apple.

On the Macintosh, NetWork simply uses the application´s signature as the program number. Because NetWork programs are Macintosh applications as well, you will be registering your application´s signature anyway. We will be using the terms “signature”, “process identification”, and “program number” interchangeably.

Now, if NetWork receives a message for a process that is not executing, it searches for a program with the signature included in the message. If it locates one, it launches that application. If not, it discards the message.

You might want to write a process whose signature is anonymous. This might be useful, if you are writing a client that generates requests and receives results, but which does never want to receive messages by other clients. This can easily be done by using a random program number, that is – unlike signatures – not known publicly. Those processes that received messages from your application, will know the address, others do not.

Sun´s RPC uses a special range reserved for the purpose of dynamic process identification. NetWork uses a reserved range too, but because NetWork has been done outside of Apple, this reserved range is a very small one (just ten signatures), which we registered with Apple for use by NetWork. Thereby, if you ask NetWork to give you a random program number, it will simply use one of these signatures. For more details about obtaining one of these numbers, please consult [2].

Note that the use of program numbers is the only thing NetWork has in common with RPC. Unlike RPC, NetWork does not assume a typical client/server model and does not support stuff like message-ids by default. However you can implement RPC on top of NetWork easily.

3.2 Process Types and Management

A NetWork process is created either by the user or by a message being sent to a not existing process. If the machine is not idle, only messages originating on the same machine may cause creation of a new process. The exception of local messages is useful if you want to make use of local services implemented as NetWork applications. The special rules imposed on processes launched by local messages are discussed below.

Any process that wants to use NetWork, must register itself with NetWork. NetWork keeps a list of executing NetWork processes. Process registration requires two parameters: the process identification (discussed in section 3.1) and a process type.

The process type may be one of the following: *slave*, *local*, *master*, or *dynamic*. NetWork selects one of the first three types automatically if you do not specify one – if the process has been launched by NetWork, it will be *slave* or *local*, if it is launched by the user, it will be *master*. It is possible for a process to change its type at runtime, except for the last category, which must be set explicitily upon the registration with NetWork.

Slave processes are created on remote request if the system is idle. They are terminated automatically if the system´s state changes to used. *Local* processes are identical to *slave* processes, but are launched on local request and are not terminated automatically.

A *master* process is launched by the machine´s user. This is typically a client of other processes. A *dynamic* process is identical to a *master*, except that it obtains a dynamic program number (see 3.1) at startup. In the following discussion, *master* is used to refer both, *master* and *dynamic*.

If we would restricted NetWork processes to communicate with processes on other machines only, we could forget about *local* processes and not think any further. Unfortunately, local communication is useful, especially because *local* processes may be the only available partners if no other machines are available.

Local processes can have multiple local clients which are *master* processes typically. Of course, they might have remote clients too, but this is unlikely and does not contribute to the following considerations. NetWork must be able to decide, when to terminate *local* processes. Of course, NetWork cannot terminate them because of user actions, because the user will be interacting with the *master*. NetWork cannot terminate them if the *master* process, that originally caused it to be created, terminates, because other *master* processes on the same machine may still be communicating with the *local* process.

To solve this problem, NetWork maintains a list indicating which *master* used which *local* processes. Every message that is sent from a *master* to a *slave* or *local* process on the same machine, will cause this list to be updated and the process type of the receiving process to be changed to *local*. If a *master* process terminates, all references of *local* processes will be updated. If a *local* process is no longer referenced by any master, its state is changed to *slave*. This last step is likely to cause the termination of the local process.

Messages originating from *local* processes must not be used to update this information, otherwise *slave* and *local* processes could send messages to each other which would prevent them from being terminated automatically.

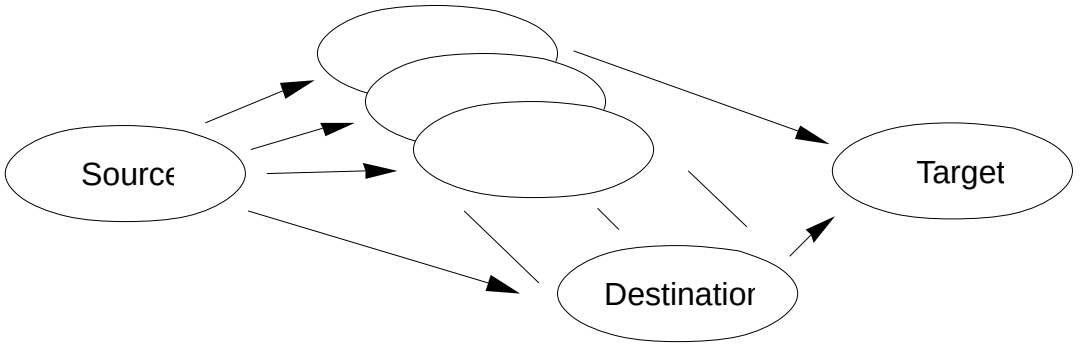
A process can ask NetWork about the *process type* of any of the NetWork processes on the same machine. A process can also ask NetWork to change its own type. This allows you to implement other session techniques than the default one outlined above. A *slave* or *local* process can change its type to *master* and terminate whenever it likes. Note: You should not use this feature to prevent your application from being terminated by NetWork because this would not be user friendly. E.g. if you think your *slave* process must return results before exiting, you should change the process type to *master*. A function in the NetWork library allows the process to learn whether the machine is still idle or not. However use of this technique is discourraged.

The actual creation and termination of processes depends on the system. One of the things to be done during termination is to tell NetWork that this process is about to die. NetWork will destroy all messages that are queued for this process. This cleanup is important, because the messages may have pointers into the process memory space associated, which is no longer valid.

3.3 NetWork Topology

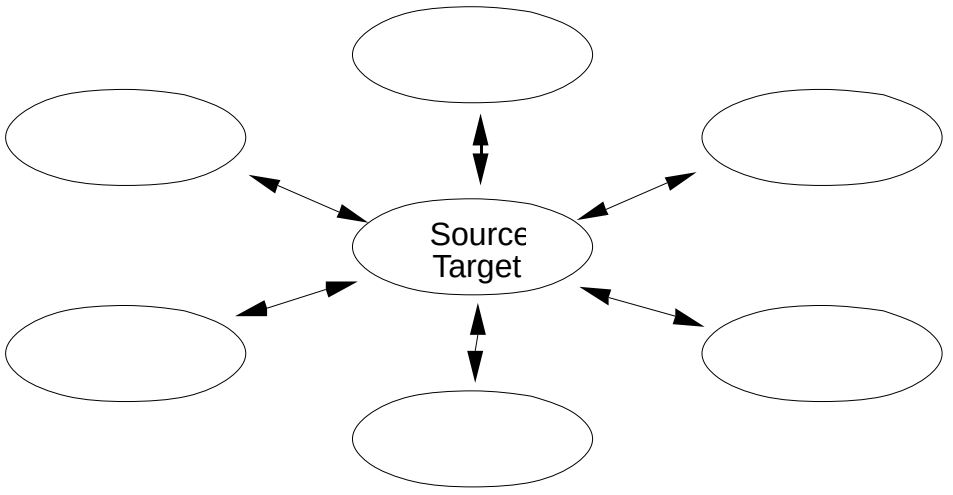
NetWork is working with cooperating processes. Typically there will be one process per participating machine but this is not a requirement – NetWork supports cooperating processes on the same machine too. The cooperating processes may be identical, as may be the machines they are running on, but this is not required. If you like, you may implement a user interface application on a Macintosh, which communicates with number crunching processes running on Crays.

A typical scenario is presented in the following figure. There is a *source* process, that generates and assigns tasks to *destination* processes, and a *target* process, that collects and processes results. Typically, the *destination* processes perform some computation.



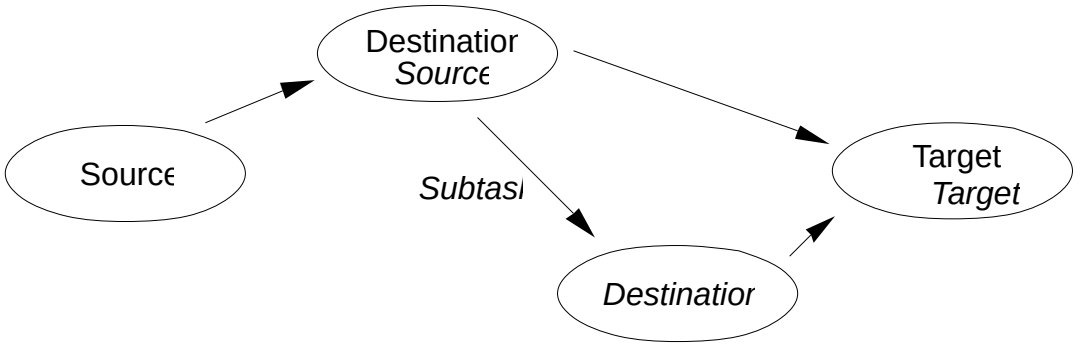
Messages are sent between *source* and *destination*, and between *destination* and *target* as well. Messages can be sent from *target* to *source* as well, to provide feedback about the progress of the computation.

Of course this feedback is easily accomplished by making *source* and *target* the same, as shown below. One might consider this as the standard scenario:



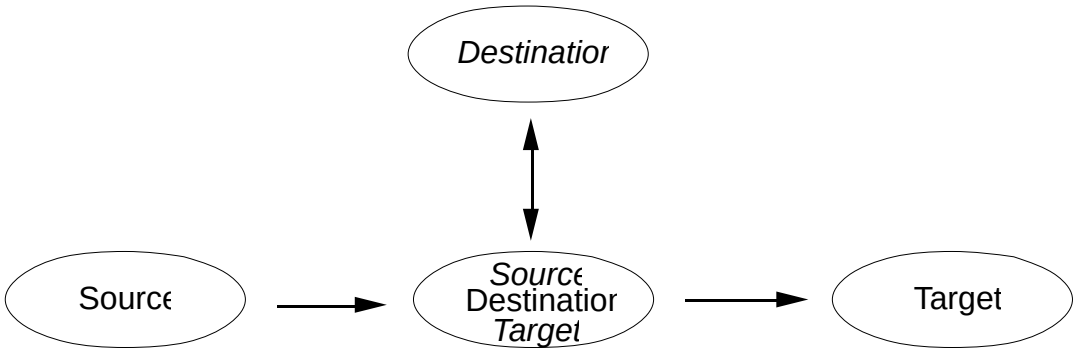
The structure of this scenario is very similar to the traditional client/server concept but it is also radically different: there is one client and any number of servers - we will call them master and slaves, respectively. NetWork is flexible enough to support the traditional multiple client / single server model as well – all you need to do is to reverse the role of *source*/*target* and *destination*.

NetWork does not assume that replies are sent to the same process that originally sent a message. The distinction of Source and Target gives you a great deal of flexibility. E.g. if you think of messages containing tasks to be performed, then *destination* process may define subtasks to be sent to other cooperating processes. Therefore the following scenario is possible too.



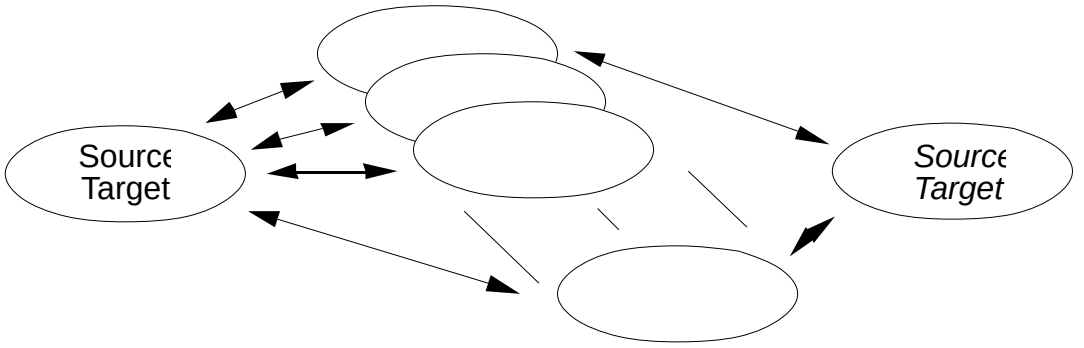
NetWork does not limit the number of times that a task can be forwarded. Unlike typical network layer protocols, NetWork does not prevent forwarded messages from circulating endlessly. It is the task of your application to do that. This can be done for example by ensuring that the complexity of a subtask is reduced every time a task is forwarded.

Yet another possible scenario:



This last scenario is of particular importance because the subtask – or task – need not be sent using the message handling system provided by NetWork; it may be any private link. The link might be a serial line, another network, or a multiprocessor bus. This allows writing gateway applications that allow utilization of computing resources like transputers or anything else across the network.

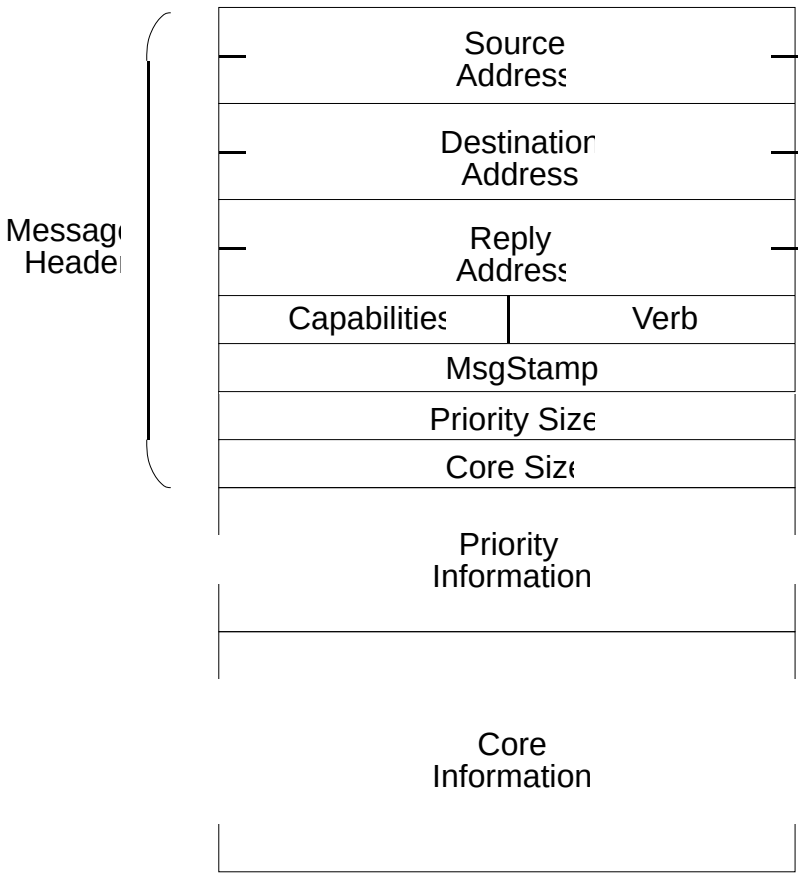
With all these nifty possibilities there are bound to be some drawbacks. The main problem is that your are not alone. In general, there will be other users. Users tend to do two things. First of all, they are going to use their machines to get their work done. The consequence is that you cannot take any of the *destination* processes for granted, except for one that possibly resides on your own machine. Second, less likely, but even worse, other users may use NetWork themselves and compete for computing resources across the network. Therefore all NetWork applications are required to support multiple clients. I.e., all NetWork applications must be written with a multiple client/multiple server model in mind.



In the simplest case a NetWork application ignores all tasks not originating from the machine that sent the initial task and exits, if it does not receive a new task for a certain time interval. Of course, you are allowed to implement better algorithms.

3.4 Message Structure

With all that said about messages, we should formally introduce them. Conceptually, messages are datagrams sent from one process to another. This does promote, but not require use of datagram oriented protocols at the transport level. The following figure illustrates the conceptual structure of a message. Note that messages are not required to be represented that way, the following representation is for illustration purpose only.



Messages consist of *header* information, *priority* information and *core* information. *Header* and *priority* information are transported and delivered to the receiving application as one unit, while delivery of the *core* information may be delayed and depends on the application actually "wanting" the information.

The idea behind this separation is to allow the application to look at the *header* and *priority* information and to have the application decide whether the information contained in the rest of the message is useful to it. If it does not want it, it tells NetWork to discard the message. The transport system does not need to transmit the bulk of the message, unless the application tells it to do. So network bandwidth will be saved.

The message *header* in more detail:

If this message defines a task to be computed – in contrast to a result message – then the naming of *source*, and *destination* is identical to the one used in the illustrations above. *Reply address* is typically a reference to *target*. However, the communications agent does not know what type of message it will be transporting, and therefore it will transport the message from the sender at *source* to a receiver at *destination*, and *reply address* is simply a – user specified – reference to some other process that knows how to interpret results.

All addresses, *source*, *destination*, and *reply address* are represented by a combination of two 32 bit words. The first of them is the machine´s network number. This can be either an AppleTalk address, an IP address, or something different, depending upon the transport systems supported by the NetWork agent on this machine (see chapters 2.3, 4.4, and 5). You should not assume a specific interpretation of this address, but use NetWork services to learn about other machines. Of course you may assume that the addresses that are part of a message you received are valid. The second one is the application´s *signature* or *program number*.

Capabilities are used to denote that certain system requirements must be matched for this message to be processed. The intended use is to have certain processes only be used on machines that match certain hardware or software requirements, e.g. “coprocessor needed”, “transputer needed”, or “system software version 7.0 needed”.

The values are interpreted as 16 individual bits, each of which must be set in the receiving machine´s system *capabilities*. NetWork will discard an incoming message with an incompatible *capabilities* value. (The computation is as follows in C notation: if ((Message´s Capabilities & System´s Capabilities) != Message´s Capabilities) discard ();)

There is already one bit defined: bit 15 (the leftmost bit), “Process must exist”. If this bit is set, the message will be forwarded to that process only if that process already exists. If the process does not exist, it will not be created. This bit should be set on result messages because it is possible that a task is finished before the last result is returned. This is possible because a client might have sent the same task to more than one machine, which may have different CPU speeds. If the bit is not set, a new process could be created inadvertently.

All other bits of the *capabilities* word are reserved for future use and must be set to 0 by default.

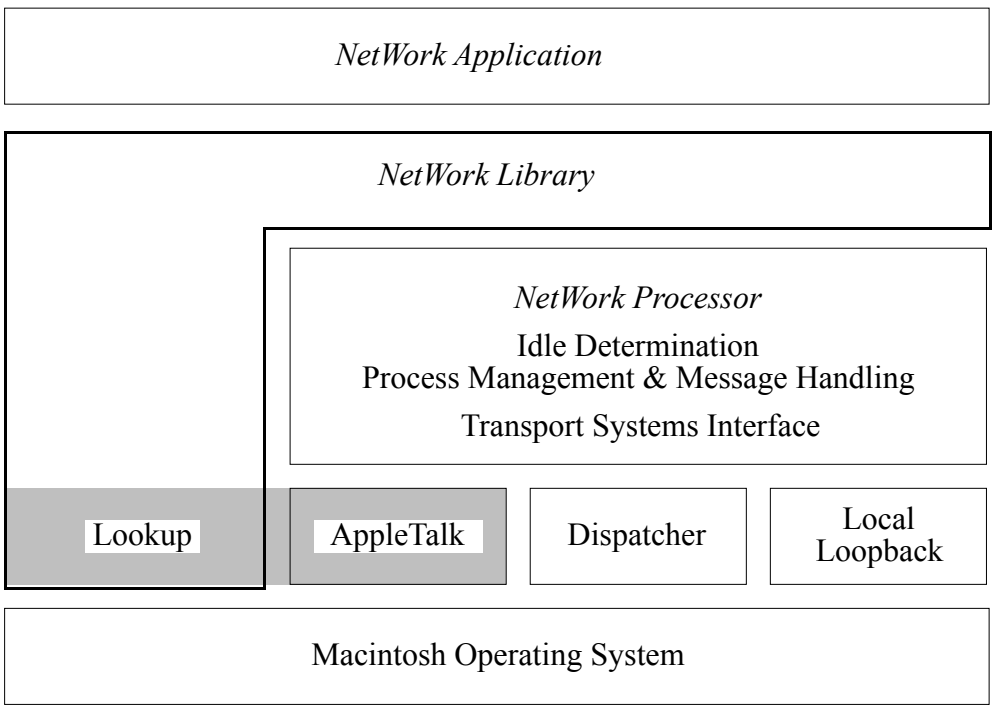
Verb is a standard place to put commands in, it is never looked at by NetWork itself. The scheduler [2] uses this, for example.

MsgStamp are 32 bits that are not interpreted by NetWork. You may use this field to sequence messages or include any other identification you like. The NetWork scheduler [2] uses this field to identify the context in which a message is to be interpreted. Hence, unlike RPC or ATP, Network does not support matching of responses received to requests sent, because it does not assume a client server model. You can add this in your own application if you want to.

There are two reasons why the sizes of *priority* and *core* information are included in the *header*. First, it is important that the receiver of a message can tell how much buffer space is required and refuse to accept the message, if it cannot allocate so much memory. Second – NetWork is an environment for experimentation – it allows new fields to be added to the *priority* information, without complicated changes in the program.

4. NetWork - The Macintosh Implementation

The Macintosh NetWork implementation consists of four major modules: idle state determination, process management, message transport, and lookup of other machines. A NetWork application generally utilizes only the latter two, but there are cases it might want to utilize process management. If you are writing a server application, message transport may be all that is needed. The following figure illustrates the organization of the NetWork modules on the Macintosh.



There is a fifth module which is not shown in the picture above: logging. Logging is utilized by all other modules in order to log important events. The log thus created simplifies debugging of NetWork applications considerably.

The Macintosh implementation has been split into two parts: the *NetWork library* and the *NetWork Processor*. The *NetWork Processor* contains only those modules that are required to be resident, whereas the library – linked to NetWork applications – implements the remainder of the functions. This separation has been done to reduce the memory requirements of NetWork. The current organization requires about 40 KB of permanently allocated memory.

On the Macintosh, the NetWork Processor is implemented as a control panel extension. Control panel extension may contain code – INIT resources – that is executed on system startup. The NetWork Processor contains an INIT executed at system startup, that loads a driver in turn. The driver contains the modules idle state determination, process management, message transport, and logging. These modules must be resident to monitor the system and to allow reception of messages. Recall that messages may cause the creation of a process. Therefore the message handling system cannot be part of the application.
The library contains an application program interface (API) or – as Macintosh programmers tend to call it – glue to the functions provided by the driver modules [2]. The library also contains the lookup module (see chapter 5). Unlike the other modules this module is not required to be resident. A NetWork application – yours – utilizes the NetWork library to learn about other machines that are

available for use, and to send messages to them. The library in turn asks the NetWork Processor to send the message to the other machines.

NetWork includes two modules – message transport and partner lookup – that depend on the transport system used. The current version of NetWork supports local message transport and AppleTalk only, but a future version might change this. The shaded area in the figure above marks the portions of NetWork that must be changed if you are going to implement another transport system. A NetWork application or the NetWork Processor do not depend on a particular transport system. The NetWork Processor utilizes code resources whenever it wants to communicate with other machines. This code resources can be changed to accommodate other transport systems. Of course, the lookup module of the library will have to be modified too. The protocol used by the “AppleTalk” transport system is documented in chapter 5. Information about transport systems in general is given below.

The other modules are described in the following sections, which will require familiarity with Macintosh programming in general, and with Multifinder in particular. For more information please consult [3], [5], and [7].

4.1 Determination of Idle Machines

One of the key concepts of NetWork is that the availability of the machine to its home user must not be questioned. NetWork maintains a variable which monitors the state of the system. A system can be either *used* or *idle*. There are two things to be done to determine the state of machines:

(1) Monitoring user activity. If the user is active, he is likely to require the computing resources (processor, memory) to get his job done, even if system load has been low recently. Any activity of the user sets the system´s state to *used*.

(2) Monitoring system load. If the system has been loaded recently, it is possibly performing a lengthy computation which will extend in the future. This could be a spreadsheet recalculation, a database query, or a compilation. We do not want these computations to be slowed down.

A system becomes *idle*, if there was no user action for a specified period of time and system load is low.

(1) Monitoring user activity

Every change of the keyboard or mouse or a new disk inserted, is an action of the user. On the Macintosh, user actions are called “events” and are entered – posted – into a queue of events. Therefore user actions – events – can be monitored by intercepting PostEvent and resetting the “time of last event” whenever an event is posted. Some of the events that can be posted are not user events, e.g. “device driver events” and “application defined events”. These event types need to be masked.

There are user actions which do not cause events to be posted: it is not an event if the mouse is moved or one of the modifier keys is pressed. Therefore these conditions are monitored too, at regular intervals, and are treated like any of the other events.

If an event is detected, the NetWork Processor updates a variable that contains the time of the last event. The user can use the control panel to set the minimum threshold of the *idle* time. If no event occurs within this interval, the system´s state may be changed to *idle*.

(2) Monitoring system load

Monitoring system load is both necessary and tricky. It is necessary to preserve the user´s priority. We cannot allow NetWork to take over a foreign machine, because it violates the imperative that this is some user´s machine. Monitoring system load is tricky, because there is no system service that allows us to measure system load. Even worse, a high system load does not necessarily imply that the system is doing useful work.

The Macintosh style of “cooperative Multitasking” allows applications – most notably the frontmost one – to use any number of time slices to do whatever they like. While most programs, written to take advantage of background processing, do only useful things while executing in the background, the frontmost application is allowed and is likely to do things that use up processor time which is not required. These operations include housekeeping, memory management (the Finder uses a very elaborate version of this) or animation - especially the type of animation that most of us will not call animation at all: mouse tracking and insertion point blinking. Therefore the frontmost application will always cause “load”.

Let us first consider background applications.

Background applications are granted time in a round-robin scheme. Most of them perform some quick tests – e.g. the Backgrounder tests if there is a spool file to print, and fires up PrintMonitor if it locates one – and exit. Some background applications perform some sort of housekeeping – the Finder tests if the contents of one of the visible folders changed, and updates its windows. Others use considerable power to do processing.

It is wellnigh impossible to detect actual background work. However there is a technique which will give a usable indication of background load. This is implemented as a module of NetWork Processor called “Background Monitor”.

The Macintosh Operating System supports timer tasks – known as “vertical blanking” or VBL Tasks – that are executed once every sixtieths of a second. Historically these were used at screen refresh time. The NetWork Processor uses this facility and asks Multifinder once every tenth of a second which program is currently executing, tests whether this application is the frontmost one, and increments a counter. The main loop of NetWork Processor computes the ratio (Multifinder + frontmost application) / (∑ of background tasks). If the ratio is less than one, the Macintosh is assumed to be busy in the background. The inclusion of Multifinder is necessary, because Multifinder serves as the “null” process of the operating system.

In order to smooth out peaks, the ratio is only recomputed if more than a specific number of samples have been taken, and the counters are not cleared after the ratio has been computed but are aged by halfing.

A number of special considerations are necessary because applications launched by NetWork Processor will affect the ratios. *Slave* (see chapter 4.2 for a definition) processes are always ignored by the background monitor. Whenever another process becomes frontmost, the counters are cleared. If a *slave* process is frontmost, then the background monitor is turned off entirely.

There is a problem caused by Multifinder 6.0. Multifinder 6.1 does not show this problem. Multifinder 6.0 saves a copy of the Quickdraw globals pointer of the current application on entry. There are a number of situations when this value is undefined. First, it may be undefined within faceless background applications. Multifinder knows how to deal with that. Second, it will be undefined during application startup. This is solved, because Multfinder updates its pointer on every entry (InitGraf and InitWindows are usually called in quick succession, and InitWindows is patched by Multifinder). Third, it is undefined in most MPW tools. This situation can be detected by checking the value of the pointer: it is nil if a MPW tool is executing that did not call InitGraf. These tools are always included in the background sum whether MPW is frontmost or not.

Of course, monitoring system load as described above does not provide an exact measure, and it does not detect all background activity. But it does detect CPU hogs like MPW or background printing. From our experience, our load indicator is usable. Unfortunately there are applications (e.g. AppleLink, ResEdit) that don´t optimize their sleep values [5], but use a small constant even if executing in the background. Therefore it may be necessary to turn off the Background Monitor in the control panel [16].

In order to have an alternative way of detecting background activity, NetWork Processor monitors calls to the IO traps Read and Write [3]. NetWork Processor assumes that if there is an application doing file IO, there must be some processing related to the file going on. This is not necessarily true, if device IO is going on, because there are applications like PrintMonitor that periodically look into a directory. But again, there is an exception to the rule: If IO is performed on the serial drivers, then the user is likely to be printing or using a terminal emulation.

Because reads of the system and the application file might happen even if the system is idle otherwise, IO to these special files is ignored. And – you might have expected that – the Finder causes an additional headache. It may access the desktop database at anytime. NetWork Processor works around this problem in the most general way: it ignores all IO to invisible files. Last but not least: all IO performed by processes known to NetWork – i.e. those launched by NetWork Processor or those that utilize them – is ignored entirely. NetWork processes are supposed to take advantage of a special call of NetWork Processor "PreventIdle" if there is a need to prevent the system from becoming idle.

While monitoring IO has been added to NetWork primarily to detect background activity if the Background Monitor fails or is turned off, it does of course detect IO that is performed by the front application. But what about other time consuming tasks performed in the foreground? The following section we will concentrate on this problem.

Simply because it is the frontmost application, it is allowed to use up any number of time slices it likes, whether it is useful or not. There are lots of applications that are still using GetNextEvent, which causes them to be called as often as possible. Others use WaitNextEvent but with small sleep values, sometimes 0, to do mouse tracking or insertion point blinking. Some of them – Finder – perform complex housekeeping tasks.

To put it simple: there is no way to tell whether the frontmost application is idle or not.

Out of luck? Wait...

There is a partial solution, based on the Macintosh´s superior user interface.

If an application is performing a lengthy calculation, it will - hopefully - tell the user what is going on. This is most often done by changing the cursor. E.g. both MPW and HyperCard use a spinning cursor to tell the user that something is going on. Other programs use other cursors to indicate that, but all of them share that the cursor shape is constantly changing. Older programs tend to use the watch cursor (part of the system file). Therefore changes in the cursor shape can be used as an indication that the machine is not idle.

Therefore, every change of the cursor shape is considered an event. Because some programs – most notably the Finder – reset the cursor once through the main event loop, we compare the new cursor to the old one and change the state only if they do not match. If the program uses the watch cursor of the system file, a flag is set to indicate this condition – the machine is considered to be busy at least until the cursor is changed again.

There are several shortcomings of the approach and its implementation:

(1) If a program uses its private version of the watch cursor (e.g. the bulldozer cursor used by MPW during garbage collection), it will not be detected. NetWork saves a copy of the watch cursor on system startup and compares the image, it does not compare handles or resource ids.

This becomes less and less important, because most of the newer programs use an animated cursor (the shape changes constantly). Also, NetWork Processor tests for *idle* only at SystemTask time – MPW doesn't call SystemTask during garbage collection. Nevertheless, there are "busy" situations NetWork Processor might not be able to detect.

(2) We do not check for colour cursors. Colour cursors are currently not supported. This is left as an exercise...

Actually we do not believe that colour cursors will be used for animation purposes, because, unlike black and white cursors, colour cursors cannot be used at interrupt conditions which prevents them from being used at VBL time.

(3) It is possible for the Finder to change its cursor because it is scanning an AppleShare volume. Actually any front application could change the cursor without a user action after having been idle for a while, but only the Finder has been observed to do it. Hopefully a future version of the Finder will use asynchronous IO to update its windows. Of course this will require a new Multifinder too which switches if an asynchronous file system request is pending.

(4) It is likely that launch of a NetWork application will change the appearance of the cursor, because most Macintosh programs do an InitCursor as part of the standard initialization sequence. If the user had been using a word processor it is likely that there will be the “I-beam” cursor. The freshly launched application is likely to change it to the watch or the arrow. Of course we could require NetWork applications not to change the cursor, but this would exclude programming frameworks like MacAPP.

To solve problem (4) and partially (3), NetWork Processor will ignore cursor changes once the system becomes “idle”. This is only a partial solution to (3), because the interval between cursor changes might be smaller than the “idle” threshold. However it will ensure that application launch will not cause an event that will kill the application immediately thereafter.

While there are problems, this implementation does work with most programs and most of the time. If we take all techniques used by NetWork Processor together, it implements sort of a superset of the Sleep Manager which is present in the Portable. Of course NetWork Processor does not put the machine to sleep. Hopefully, Apple will implement a better Sleep Manager someday, and this implementation should be available on all Macintoshes – not just the Portable. Ideally, there should be something like “Sleep Gestalt”, a system call that you can ask for the time of the last user action, the last file or device IO performed, the last cursor change, the last Quickdraw operation, background load, and that like. Then it is very easy to implement either a screensaver or NetWork on top of that. Every application might have a different set of actions it might want to consider.

There are two more problems that NetWork Processor takes care of. First, a machine is never considered idle if Multifinder is missing. The reason is obvious, if Multifinder is missing, then it is impossible to launch another application without terminating the current one. And NetWork is supposed to maintain the illusion that nothing happened during the time the machine was used by someone else – definitely not the case if the application had been changed possibly without saving changes to open documents.

The other problem is less obvious. If the system disk is offline (ejected), then launching a new application is very likely to cause the disk swap alert to occur. The disk swap dialogue is one of the very unfriendly things in the system – it stops all processing that is not done on the interrupt level. This is a problem even if the application being launched is located on another volume, because almost every application uses resources that are found in the system file. Therefore the machine is never considered idle, if the system disk is offline. Because of the same reason, NetWork Processor also checks that the folder NetWork Tools is present before attempting to search and launch a new application, and it also suspends logging whenever the system disk is ejected.

4.2 Process Creation and Termination

On the Macintosh a process can be launched on user request (e.g. double click), or it can be launched by NetWork if the destination process of a message received indicates a not existing process and the machine is idle. NetWork Processor will then search for an application matching the message´s destination signature and launch it, if it finds one.

Because an idle machine might be looked up by several machines simultaneously, and because it is not a good idea to load another program if the machine is already loaded, the “Background Monitor” described above also monitors the load caused by slave processes. If this load already exceeds 75% of total CPU time (actually samples), no more slave processes will be launched. Note that to a requesting process, this situation is identical to program not available or insufficient memory.

If the system becomes busy, NetWork Processor tells all slave processes to terminate by faking Command-Q as soon as possible. The process must clean-up and exit. This has to be an instant operation. You should not save state information or return results. If your process does not free resources quickly, the user may become unfriendly and throw away your application – or disable NetWork operation.

NetWork will call ExitToShell instead of faking Command-Q if it detects one of the following conditions: Command-Q is not supported, a modal dialogue is frontmost, or more than a tenth of a second passed since the system became busy. To protect the user even more, NetWork Processor will kill *slave* processes a tenth of a second after the system´s state changed to used. Because of limitations present in Multifinder 6.0, this is done only if Multifinder 6.1 is present.

One of the things to be done during termination of a process is to tell NetWork Processor that this process is about to die. NetWork Processor will destroy all messages that are queued for this process, thus eliminating all references to buffers within the processes address space. This operation is essential, because otherwise memory portions belonging to another application might be overwritten which might cause system operation to become unpredictable. That´s why the NetWork Processor as well as the library utilize patches in order to ensure that NetWork Processor will know when your application is about to die, even in the case of a fatal programming error.

NetWork Processor on the Macintosh allows to launch applications automatically at system startup or whenever the system´s state changes to idle. Applications located within the folder “NetWork Startup Tools” are loaded as soon as Multifinder has been launched. They are launched as processes of type master. Applications located within the folder “NetWork Idle Tools” are loaded when the system becomes idle. They are of type slave and are – unless they change their state – automatically terminated when the system becomes busy.

4.3 Message Handling

As far as the application is concerned, NetWork Processor contains all functions that are used to handle messages. The library routines simply fill in parameter blocks and call the driver. See the NetWork Programmer´s Guide [2] for details about the functions you can use.

Actually, if we consider just message handling, NetWork Processor does little more than to allocate message buffers and keep those that are active in a linked list. It generally stores parameters into message records [2] and then relies on one or more transport systems to actually transfer messages to other processes - whether they are local or remote.

NetWork Processor supports up to four transport systems simultaneously. A transport system can be part of NetWork Processor or may be installed by another application at runtime. The version of NetWork Processor distributed contains three built in transport systems, implemented as code resources, that are loaded at system startup. The transport systems are: “Dispatcher”, “Local”, and “Appletalk”. “AppleTalk” uses a protocol detailed in chapter 5.

“Local” merely copies the information from the sending process memory to the memory provided by the receiving process on the same machine. This is possible since the Macintosh OS is not protecting memory and uses one large address space for all processes. While we would like to see protected memory to be implemented in a forthcoming Macintosh OS, we think that one large address space is better suited for fast interprocess communication than any other technique. If protected memory is implemented allowing read access of the entire memory, NetWork will not require changes to account for protected memory.

In general, an application is not concerned about which transport system to use. Instead it passes a destination address when sending a message. The task of Dispatcher is to look at the address and select one of the available transport system, depending upon the destination address of the message. It does that by looking at information provided by the other transport systems [2]. Basically, every transport system has a local address of the machine, an associated broadcast address (which may be 0 if broadcasts are not supported), and a range of valid addresses. Dispatcher looks at the destination address and searches for a transport system that supports the address in question, fills in the index of the transport system, and then restarts the operation.

There is a special case that is handled jointly by NetWork Processor and Dispatcher. Any of the three addresses contained in a message might refer to the local machine, using any of the multiple local addresses. NetWork Processor looks at all addresses when sending, and replaces any reference to the local machine by zero. Dispatcher, after it selected a transport system to use, replaces all zero addresses by the local address of the selected transport system. That way, all references to the local machine are always valid on the machine that receives the message.

Unfortunately, the Macintosh OS does not support a re-entrant memory manager. Because a message may arrive anytime, the heaps may be inconsistent. Therefore, it is not possible to allocate the memory required to hold the message dynamically. Instead, NetWork Processor pre-allocates message buffers at system startup – the total number is user configurable [16]. Transport systems can ask NetWork Processor to pass them one or more of these buffers for use when a message arrives. E.g., the AppleTalk transport system asks for three buffers, which it in turn passes to the AppleTalk drivers.

Of course, a transport system is not required to use that technique, and it may change the number of buffers at any time. E.g. the Dispatcher sets the number to zero, because it never receives a message. Local sets the number to one if a local transfer is pending – no part of the message can be lost because it is held in the memory of the sending process. Last but not least, the AppleTalk transport system sets the number to zero if AppleTalk is turned off.

In order to ease implementation of a transport system, NetWork Processor performs lots of housekeeping. For every message that is pending, it periodically calls the transport system to update state information or to keep the transfer process alive. Therefore – though that might yield optimum performance – it is not necessary to write an interrupt driven transport system. In addition, NetWork Processor periodically checks for new messages or old messages that have been transferred completely, have been aborted because of an error, or have been destroyed, and it informs the application about that.

4.4 Logging

NetWork Processor allows to log all messages sent or received, all error conditions and other information to a log file. An application can ask NetWork Processor to include additional information in the log file. This can be used to trace messages and problems.

5. AppleTalk Transport and Lookup

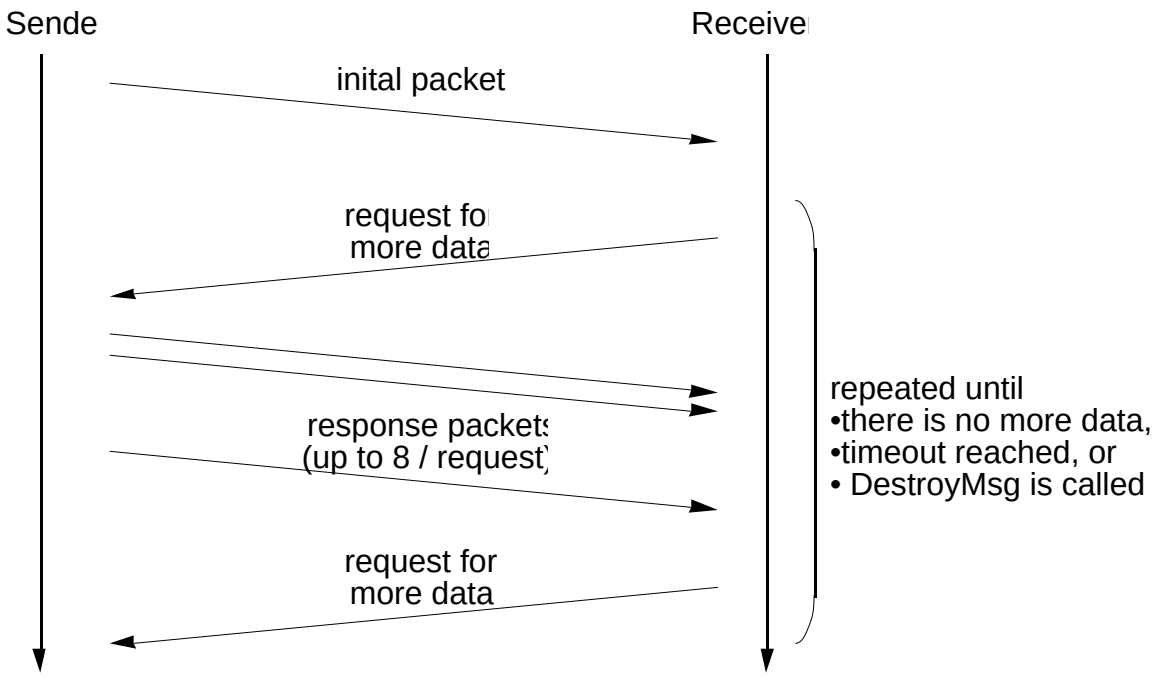
This chapter documents the AppleTalk transport and lookup procedures used by the current implementation. The AppleTalk protocol and implementation is documented not only to allow implementation of a compatible transport protocol for another platform, but also to illustrate the considerations that you should undertake when designing and implementing a new transport system.

5.1 AppleTalk Transport

One of the goals of NetWork is to minimize the impact of NetWork communications to other users´ communications. As we pointed out in chapter 2, ideally, we´d like to have a sort of second class network access. Since LocalTalk [4] allows us to monitor packet travel, it is possible to get an estimation of network load based on recent history. We could have used it to delay or even discard messages if the load exceeds a certain threshold. Also we could have used the history bytes to generate a longer wait for our messages. Because both techniques are restricted to LocalTalk connections and would break at routers, we did not implement them.

The NetWork message transport is built on top of ATP (AppleTalk Transaction Protocol) and utilizes at least once transactions [4]. We preferred ATP over DDP (Datagram Delivery Protocol) because ATP allows to write the listening functions in a highlevel language, whereas DDP requires assembly.

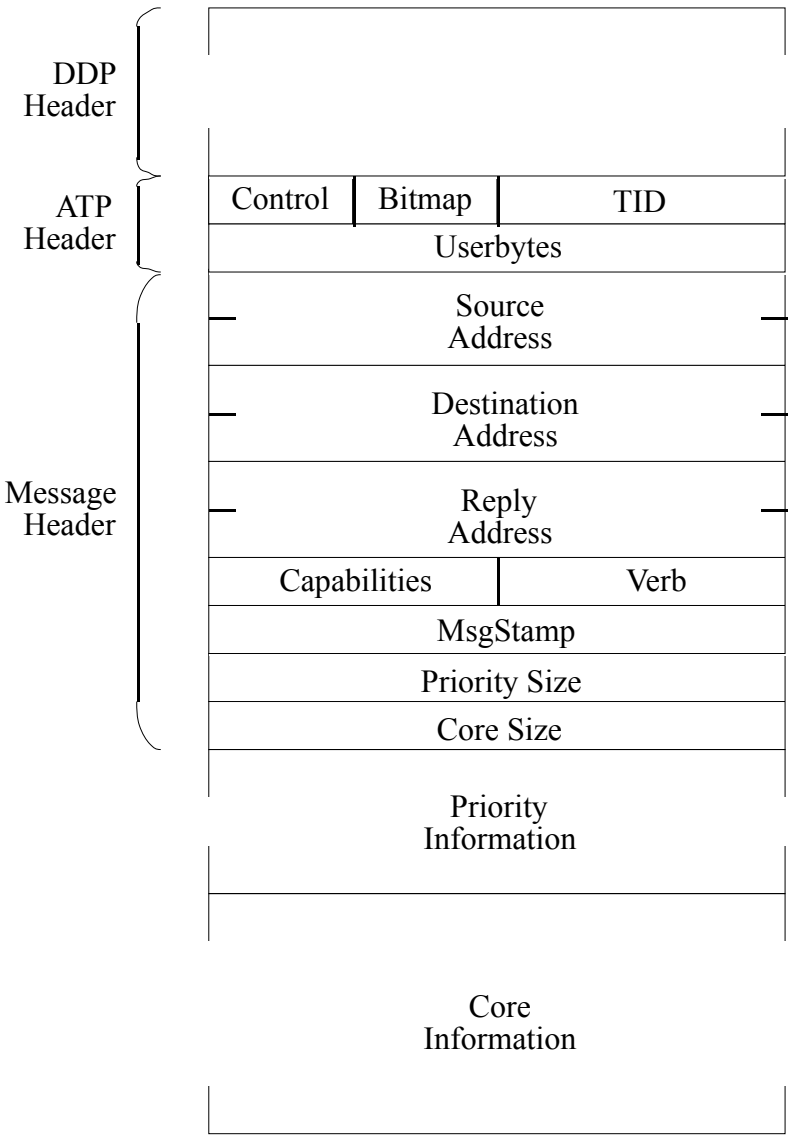
The protocol tries to minimize the number of packets transferred. It sends the *header*, the *priority* information, and as much *core* information as will fit in one packet. If there is more data than will fit in one packet, then a polling technique similar to PAP (Printer Access Protocol – PAP is built on top of ATP too) will be used. The following figure illustrates the protocol used:



The polling technique used has two great advantages over a fragmentation technique like the one used by IP: firstly, it is very unlikely to overflow the receiver or a router along the message path, because AppleTalk implementations are generally optimized to support back to back reception of at least eight packets. Because the receiver will ask for more data, flow control is built into the protocol. Also, there is no timing dependency to accommodate for different network bandwidths, the polling technique will adjust to different network speeds automatically. Secondly, only information that is actually wanted is transferred.

For all messages that are not "accepted" by an application, only the initial request is sent, all other packets will not be sent. Both the sender and the receiver can abort a message at any time. If a message is "accepted", but "destroyed" later on, while the transfer process has not yet been finished, at most 8 unused packets are sent, response packets already requested with a preceeding request for more data, but now no longer wanted.

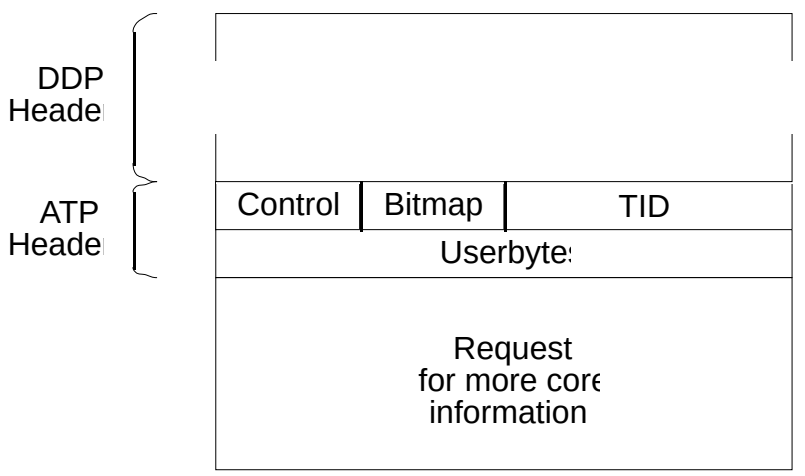
The following figure gives the packet format of the initial packet:



The high word of the userbytes is used as a protocol version number, and is set to $0003. The low word is used as a message identification and is documented below. This initial packet is sent as an ATP request.

Although the source and destination network address are part of the DDP header, they are repeated in the message header. There are two reasons behind this: First, Macintoshes running AppleTalk versions prior to version 48 – e.g. a Macintosh Plus – cannot send an ATP request through a specific socket. The source address included in the message header is always the socket the Macintosh is listening on. Second, not all operating system environments supporting AppleTalk allow different processes to share the use of one socket. The format used allows a receiving process to forward a message to another one without problems. Of course this feature might be exploited to add a general forwarding scheme in the future.

Because the priority information is never split across packet boundaries, it is limited in size to 538 bytes (578 bytes maximum ATP data size - 40 bytes of message header information). If the message contains more than 538 bytes total and if the receiving process wants the remainder of the core information, the receiver will ask the sender for more information. This is done as an ATP request too:



The low word of userbytes is set to the ATP message identification of the initial packet (the low word of the userbytes), the high word is set to 0. The message identification is generated by the originating NetWork Processor and is used to relate the request for more core to a message it knows about. Note that sixteen bits of message identification are sufficient, because it is impossible to send more than 100 messages per second, and the message lifetime is limited to about two minutes. Btw, the message identification is only required if core information does not fit into the initial packet, but the Macintosh implementation generates and sends it with every message.

The “request for more core information” contains one to eight BDS elements [3], where the pointers denote offsets into the core information. This request information is used to set up a response BDS and the core information is sent as an ATP response. The receiver of the message will ask for more core until it received all information, a packet is lost – no retries are done – or until the message is destroyed.

Because of this scheme, *priority* information is limited to 538 bytes and core information is limited by available memory and network bandwidth – the maximum size message would require over half an hour at 10 MBit/s, not accounting for arbitration and protocol delays.

There is a more realistic limitation: NetWork Processor places a limit on the maximum message lifetime. This limit is currently set to two minutes. This will effectively limit the maximum message

size that can be transferred (assuming nobody else is using the network) to an estimated 1,000,000 bytes if you are going to use LocalTalk, or to about 3,000,000 if you are using EtherTalk.

NetWork uses the static socket with the number 40 – a socket number reserved for use by Apple. We had to decide, whether not to support broadcasts and use a dynamic socket, use one of the experimental static sockets and risk conflicts with Farallon´s Timbuktu, or to use a reserved socket. We chose to use a reserved static socket.

The broadcast address -1 is translated to the AppleTalk internet address 0,255,40 (net, node, socket). Messages sent using the broadcast address must fit into one packet, i.e. they are restricted to at most 538 bytes. NetWork sends broadcasts to all machines attached to the same cable except itself (more specifically, it discards any message it received from itself).

Actually, both addresses are conventions supported by the supplied transport systems [15].

5.2 AppleTalk Partner Lookup

Identification of communication partners should be accomplished without causing undue network load. A minimum of packets should be used. We must admit, that the current implementation fails to match that goal. It may cause considerable network load.

5.2.1 The Present…

The current implementation does the following: a NetWork Processor, whose state is idle, registers its listening socket on the NetWork, using the type “NetWork Processor” [4]. If the system´s state is used, it will use the type “NetWork User” instead. The library uses NBP to search for other NetWork Processors and makes the list of possible partners available to the application. The current implementation supports one zone only, it does not search all zones.

We have chosen this approach because of four reasons. First of all, and this is the most important reason, it is very simple to implement. Second it provides an easy way to tell which machines are available. Using a network utility like InterPoll or CheckNet you can see all available NetWork Processors (busy machines which have a NetWork Processor will be listed as “NetWork User”). Third, it can be used in a server/client situation. The library allows to register the listening socket with another name in the network. A client can then be written not to search for “NetWork Processor” but to search for this new type instead. Fourth, unless you utilize the broadcast address, this technique does not depend upon the use of static sockets. There are AppleTalk implementations, e.g. the Columbia AppleTalk implementation, that do not support all static sockets.

There are two major disadvantages of our current strategy: first, every process will perform its own lookup. With NetWork applications becoming popular, this will cause unnecessary network load. Second, one cannot tell in advance, which applications are available for execution or which capabilities a system has.

We are considering an alternative strategy that will address these problems:

5.2.2 The Future…

We consider to make use of a static socket [4] and by-pass NBP altogether. A broadcast could be sent to this static socket, and a predefined process could respond with the network address of the machine, its capabilities and optionally the available programs. We could make the technique more general and allow the result of the lookup to be used by more than one machine.

Please note that NBP implicitly uses broadcasts. Therefore use of broadcasts does not cause more network load then the current technique.

The obvious limitation is that AppleTalk broadcasts are limited to one cable. A solution might be to designate one machine per cable to act as a lookup server. Worse, there is no easy technique to tell which network numbers correspond to which cable. This problem is addressed in [14], and if we assume that at least one AppleTalk internet router implements this protocol, it can be solved without user intervention.

Despite the problem of finding out network numbers and designating a server, this approach is especially promising because it returns the maximum information possible with a minimum of packets, and because it also simplifies implementation of new transport systems – you are no longer required to support both lookup and transport, instead a transport implementation that supports broadcasts is all that is required.

If we sign out a server, another technique is possible too: NetWork Processors could tell others or the server actively that their state is changing from *used* to *idle* or vice versa. As you can see, there is plenty of room for further experimentation.

A different possibility is to use enhance the Scheduler [2] to perform the lookup – using broadcasts. The Scheduler could maintain a list of processes which had been sending messages to it recently. If this list contains less than a certain – application dependent – number of machines, the Scheduler broadcasts a message directed to the specific creator and using the specific capabilities in use. This will cause a launch of slave processes on all available machines that have a corresponding image and whose capabilities match. These machines could now ask for a task to be assigned.

Note however, using broadcasts is not a cure all solution. There may be transport systems which do not support a broadcast mechanism and the process looking up partners might e.g. consult a database. At least, a future implementation of NetWork is likely to separate the lookup of communication partners to a separate process.

6. Summary of Tricks and Problems

This chapter is intended to summarize the tricks we used and problems we encountered while implementing NetWork on the Macintosh. The tricks generally consist of trap patches. The trap mechanism is documented in [3]. In this respect the Macintosh Operating System is a wonderful system for a programmer: you can change anything you like…

NetWork Processor uses quite a number of patches. In addition to the patches applied by NetWork Processor, the library also patches a number of traps. These patches are always local to the application. All but one of the patches of NetWork Processor and all of the library patches are head patches and all of them save all registers.

There are four different types of patches used: traps, that are patched (the term in brackets is the term used to mark that type of patch in the table below)

(1) at system startup (Startup)
(2) after Multifinder has been loaded (Multifinder)
(3) temporarily (Temporary). Some of these patches are patched to by-pass Multifinder. In that case, the address of the trap at system startup is saved and temporarily restored when necessary (Saved, Temporary)
(4) by the library (Library)

The following table lists all patches:

Trap	Type of Patch	Why
Close	Startup	Process Termination
Control	Startup	Idle Determination
EventAvail	MultiFinder, Library	Process Termination
ExitToShell	Library	Process Termination
GetNextEvent	MultiFinder, Library	Process Termination
HiliteMenu	Startup	Process Termination
InitGraf	Startup	Patching
Open	Saved, Temporary	Logging
OSDispatch	Temporary	Process Creation
PostEvent	Startup	Idle Determination
Read	Startup	Idle Determination
WaitNextEvent	Multifinder, Library	Process Termination
Write	Startup	Idle Determination
SetCursor	Startup	Idle Determination
SetTrapAddress	Saved, Temporary	Patching
SysError	Library	Process Termination
SystemTask	Startup	Idle Determination

The following discussion concentrates on the individual patches.

The patches to PostEvent, SetCursor, Read, and Write are used to detect user or program activities. They are documented in section 4.1. The patches to Control and SystemTask are used to detect activity caused by DAs.

The patches to Close, ExitToShell, and SysError are used to tell the termination of a process known to NetWork. The patch to Close checks if the file being closed is the current application, and if that application is known to NetWork, removes it from the list of active NetWork processes. The library patches ExitToShell and SysError are used to stop any name lookup process that may be active, and then call NetWork Processor to remove the current application from the list of active NetWork processes. The patch to HiliteMenu is used to avoid menu highlighting whenever a slave process is terminated. Because NetWork fakes a Cmd-Q, the application will highlight the File menu. However Multifinder does not shield that call and therefore the highlighting will take place in the foreground. Even worse, whatever happens to have the same menu id will be highlighted, not necessarily the File menu.

The patches to EventAvail, GetNextEvent, and WaitNextEvent are used to terminate slave processes after the system´s state changed to busy. In order to do that, NetWork must be able to find out which application is calling that trap. NetWork Processor patches these traps after Multifinder is loaded, because Multifinder completely replaces two of these traps, and calls the third one, GetNextEvent with the front application switched in – i.e., NetWork Processor could not ask a specific application to terminate if the patch were applied at startup. These patches are replicated in the library, because in an early stage of development we did not want to depend on patches to Multifinder and therefore used patches local to the application.

Because the traps EventAvail, GetNextEvent, WaitNextEvent are patched by Multifinder, Network Processor cannot patch them at system startup. Therefore, InitGraf is patched. InitGraf is one of the traps that is called at the very beginning of virtually every Macintosh application. At InitGraf time – usually during the launch of Finder – NetWork Processor tests if Multifinder is present and if it is, applies the patches to the traps listed above.

However, Multifinder keeps track of which application patched which trap and uninstalls the patches at application termination. Therefore NetWork Processor saves the address of the ROM´s SetTrapAddress trap. During InitGraf, NetWork Processor restores the ROM´s SetTrapAddress (it performs a SetTrapAddress on SetTrapAddress!) and then calls SetTrapAddress to patch the other traps behind Multifinder´s back. After that, it resets SetTrapAddress to Multifinder´s SetTrapAddress. This last step is done a second time, in order to allow Multifinder to recognize that we have undone our first patch.

NetWork Processor tries to launch applications in the background. System 7.0 will support this type of launch. In order to support that feature today, NetWork Processor patches the core of Multifinder itself: OSDispatch. The patch is applied temporarily during the launch of a new application if “Background (invisible) processes” are checked in the control panel [16]. The patch watches for the call to OSDispatch that moves the new application to the front and simply returns without passing that call to OSDispatch.

NetWork Processor also saves the address of the Open trap at system startup and restores it temporarily when it tries to open its logfile. This is done because Multifinder patches Open as well:

all files which are opened while an application is executing will be closed by Multifinder when this application dies. However, Multifinder does not detect that a driver opened the file. Therefore we by-pass Multifinder during open and Multifinder will not close our file behind our back. Multifinder should actually detect driver entry/exit and tag files appropriately.

The very same problem can occur, if an application opens a working directory pointing to the directories searched by NetWork Processor. If NetWork Processor opens a working directory, the file manager will return the reference number of the working directory opened by the application. NetWork Processor will close that working directory after use, potentially causing problems. We did not address this problem, because it is unlikely that a user will utilize this directories.

A related problem may arise in the future. NetWork Processor launches slave or local processes, but Multifinder assumes that the currently active application is the parent process, again because Multifinder does not detect driver entry/exit. Starting with Multifinder 6.1, applications can choose to receive “chield-died-events” (the only application that currently utilizes this is SADE - a debugger). Applications might become confused, if they receive events for children they do not know about.

NetWork Processor uses a number of undocumented calls to Multifinder [7]. All but one of them are included in the documentation of release 7.0 of Apple´s system software [13], but we do not know if the calling conventions will change or not in comparison to the current implementation. Also, the current Multifinder implementation uses stack frames that differ from those documented in [7,13]. Use of these calls can be disabled using the control panel.

The calls used are MFGetVersion, MFGetPID, MFGetProcInfo, and MFKill. MFGetPID is used to find out which application is currently being executed. It is used by the patches to EventAvail, GetNextEvent, WaitNextEvent, Read, Write, Close, ExitToShell, SysError, and HiliteMenu, and it is also used within the Background Monitor.

MFGetProcInfo is used to find out if a process is frontmost or not. It is also used to verify that a particular process is still alive before killing it. This test was added at a time at which NetWork Processor did not patch Close, ExitToShell, and SysError, and at which it was possible for applications to exit without NetWork Processor knowing about that. Also it is possible to use SADE (Standard Apple Debugging Environment) to kill an application. NetWork Processor compares the creator returned by MFGetProcInfo to the creator it remembered and assumes that the process exited if they do not match.

NetWork Processor uses MFGetVersion to find out the version number of Multifinder. If the version is less than 6.1, processes will never be killed. If Multifinder 6.1 or later is available, all slave processes that did not exit within 0.1 seconds after the system´s state changed to busy will be killed. If Multifinder 6.1 or later is not available, applications are not killed, but are forced to exit by calling ExitToShell from the event patch.

If a process is killed, ExitToShell will not be called. We would like Apple to provide a means that allows an application to clean-up if it gets killed. Apple system engineers may see below for a suggestion on how to do it. If a NetWork application uses name lookup, the name lookup process will not be terminated (only master processes should do name lock-ups because of that).

The NetWork library patches a total of five traps – all of these are local to the application if using Multifinder, and are restored upon program exit to ensure compatibility with Finder. As indicated above, the patches of the library repeat part of the functionality of the patches of the Network Processor. In particular the code to fake a Command-Q (Quit) and the code to clean up is almost identical. In addition to the functionality already described, the library also uses these trap patches to maintain the name lookup task if one is active. The patches to the event traps update the name table and restart the lookup process, the patches to ExitToShell and SysError will terminate a pending lookup if the program terminates. Be sure to call either ExitToShell or ExitNetWork in your program or your system is likely to crash if you launch the application without Multifinder.

The following list documents the problems left and also lists features we´d like to see in a future Macintosh OS in order to avoid some of the patches and undocumented calls.

It would be a nice feature of a future Macintosh Operating System, if there was an indication of process creation and termination. This could be implemented similar to the AppleTalk transition queue [6, TN 250] with an additional parameter being the process id. In addition to easing the housekeeping done when killing or during exit of an application, this feature could be used to abort the name lookup process of an application.

MFWakeUp, a routine that will be available with system 7.0 [13], could be used to wake up processes if messages are pending, or if they should be exited. Apple Events could be used to tell applications to exit, removing the necessity of the event patches. Taken together with the process termination notification above, the number of traps that need to be patched is reduced from 12 to 5.

The patch to HiliteMenu should be part of Multifinder, as should be patches to the traps InitCursor and ObscureCursor.

In chapter 5 we noted that we´d like to see a second class network access to be implemented. Also, we´d like to have an officially assigned static socket – see chapter 5.

The current implementation reserves a block of memory of about 1KB per message. The default number of concurrent messages is set to 20. Therefore NetWork Processor reserves about 20KB of memory in addition to the about 20KB of code it loads. We may change this in the future and allocate the memory required to send messages dynamically from the applications memory.

The reason for preallocating messages is that there is no safe technique to allocate memory at interrupt time currently. If there was one, we could allocate just one buffer for one message and allocate the next buffer if the first one has been used. Currently, we must keep multiple buffers, because more than one message may arrive within a small time, which leads us to yet another desideratum of a Macintosh OS to come: implementation of a memory manager, that can be used at interrupt time. It should be possible to modify the current memory manager to support reentrancy for different heap zones. That way an interrupt driven process may switch to a private heap and allocate memory without knowing whether the other heaps are in a consistent state. Of course it must be safe that it its own heap is not in an inconsistent state, but that is far easier to take care of then not being able to do memory allocation at all.

7. Special Topics

This section covers two topics that we thought about already:

7.1 A simple Unix implementation

The Unix operating system is not user centred like the Macintosh. Also there may be Unix systems, that do not allow interactive logins. Therefore, it is generally not easy to define when the system´s "user" is inactive.

Assuming a Unix workstation, not a multiuser system, you may use the modification time of /dev/console to detect user actions. You may have to include other ports as well, to detect other users or mouse movements. In order to detect system load, you may compare the number of time slices used by all processes to the number of time slices used by null and the *slave* processes spawned by NetWork itself. If less than a certain threshold – say 20 % – is used by user processes – as opposed to the latter two, the system´s load might be considered low.

Process addressing can be done using Apple registered signatures, but you will need to use a table that maps between the number and the program to use. The overall operation of NetWork on a Unix host may be compared to what the internet daemon (inetd) does.

A natural choice in the Unix environment is the use of TCP/IP for communication with other machines - or better UDP if you want to minimize packets - see below. For communication within the machine UDP can be used too. You will need one well known socket for use by the communications agent, and one socket for each process. Outgoing messages can be sent from the dynamic sockets. Incoming messages must be sent to the well known socket. An optimization possible is that once a process decided that it wants the core information, it makes available the socket number to the message´s sender to avoid the extra local forwarding. This is exactly, what the AppleTalk based protocol does.

7.2 Use of session protocols

Message transport is assumed to be connectionless, i.e. no session is maintained by NetWork. This does not preclude use of typically session oriented protocols (like TCP [11] or ADSP [4]), but it definitely does not encourage the use of them because of the overhead of these protocols.

If a session oriented protocol is used, the following procedure should be used (this is not a strict requirement)a open the stream to the recipient machine and send all information (header, priority, standard, in this order). Typically, the interface to the underlying transport system will buffer the information or the request (using asynchronous IO).

The receiving machine will not accept the open request if there is no receiving process and the machine is not idle (If there is a receiving process, the message might be intended for it; if the machine is idle, we might need to create a new process). After that, it will read the header and priority information into a private buffer, search for a process (possibly creating one), and pass this information to this process, which in turn decides whether to get the rest of the message or not. If it wants the information, it allocates buffer space and tells the transport system to read the information. If not it tells the transport system to discard the information which in turn will result in closing down the connection.

Because most streaming protocols use sliding window techniques, this procedure will result in the header and priority information being transferred, with as much standard information piggy backed as will fit into one packet, thus minimizing the number of packets. However the opening and closing of the stream will involve additional packets, typically on the order of 4 packets that do not contain any useful data.

One might argue, that the stream does not need to be closed, because it might be reused to send another message. Well, we encourage you to implement it if you like…

8. Open Issues

Currently NetWork does not address the problem of contention. Contention might occur, if more than one user tries to use idle machines. As long as no user is active, the effect will be a simple first come first served scheme. If users are active, the effects might be random – which is obviously a good effect. However, more messages will be sent than necessary.

A possible solution to this problem could be part of the – not yet implemented - lookup strategy discussed in chapter 5. If we are going to designate a machine to do the lookup – or act as a registration server – this machine might as well try to tell all others only part of the available machines.

We want to allow for a smooth transition once system software 7.0 is available. The NetWork Programmer ´s Guide [2] documents the event record that will be used by NetWork in the future.

References

[1] The NetWork Project, G. Sawitzki, Universität Heidelberg, 1990
[2] NetWork Programmer´s Guide, G. Sawitzki, Universität Heidelberg, 1990
[3] Inside Macintosh, Volumes 1-5, Apple Computer, Addison Wesley, 1985-87
[4] Inside AppleTalk, Apple Computer, Addison Wesley, 1989
[5] Programmer´s Guide to Multifinder, Apple Computer, APDA, 1987
[6] Macintosh Technical Notes, Apple Computer, APDA, 1984-1989
[7] Multifinder revealed, Richard Guerra, MacTech Quarterly, TechAlliance, Autumn 1989
[8] Computer Networks (2nd ed.), Andrew S. Tanenbaum, Prentice Hall, 1988
[9] RFC 1050 Remote Procedure Call, Sun Microsystems, Information Sciences Institute, 1988
[10] RFC 768 User Datagram Protocol, Jon Postel, Information Sciences Institute, 1980
[11] RFC 793 Transmisson Control Protocol, Jon Postel, Information Sciences Institute, 1981
[12] RFC 791 Internet Protocol, Jon Postel, Information Sciences Institute, 1981
[13] System Software Release 7.0 Preliminary Notes, Apple Computer, May 1989
[14] AppleTalk 2.0 Protocol Specification, Apple Computer, June 1989
[15] NetWork Transport Systems, J. Lindenberg, Universität Karlsruhe, 1990
[16] How to Use NetWork Processor, J. Lindenberg, Universität Karlsruhe, 1990

Index