8.91 4.90 11.89

NetWork Programmer's Guide

Programmer's guide to the **NetWork** model of distributed computing.
This guide is based on the Macintosh implementation of NetWork.
For use on other systems, see your system specific documentation.

For those in a hurry

Services and facilities provided by NetWork

- management of asynchronous distributed computing
- idle time computing
- inter-application communication
- remote launching
- transparent message transport
- low level log file support

System requirements and restrictions: The Macintosh version of NetWork runs on Macintosh Plus or higher, MacOs 6.03 or better with Multifinder, or System 7, or A/UX 2.0.1 or better. The basic version of NetWork can make use of any AppleTalk implementation, but only addresses in the local zone will be used.

Getting started: Move the file *NetWork Processor* and the folder *NetWork Tools* into your system folder, and into that of any Macintosh on your AppleTalk network cooperating on NetWork experiments. Restart all Macintoshes which are to cooperate. For computers other than the Macintosh, follow the installation instructions specific to that machine to install a *NetWork Processor*. You can now run the sample program *Spinning Brain*. See *How to use Spinning Brain…* for details.

For each machine, cooperation on the experiment can be restricted or disabled by using the *NetWork Processor* control panel extension. To disable experimental use of a machine, remove the *NetWork Processor* from the system folder.

Implementing NetWork software: To develop your own software to experiment with NetWork, use

unit NetWork from file NetWork.p,
unit NetWorkLookup from file NetWorkLookup.p,

and link with

NetWorkLib.o.

The default look-up system is already contained in NetWorkLib.o. Compile NetWorkLookup.p and link with NetWorkLookup.p.o only if you want to override it

You can make use of the objects supplied in

unit SchedulerUnit.p from file SchedulerUnit.p

which provide high-level access to the NetWork message system. Look at the scheduler implementation model and modify it according to your needs.

Comments are welcome, but we cannot guarantee a reply.

Internet network@statlab.uni-heidelberg.de
Bitnetj40@dhdurz1.bitnet
AppleLink: ger.xuu0003
Ordinary snail: G. Sawitzki, StatLab Heidelberg, Im Neuenheimer Feld 294, D 6900 Heidelberg, W. Germany.

Contents

What is NetWork ?

NetWork is a collection of software for experimenting with distributed computing in a network of personal workstations. Personal computing offers the advantage of reliably guarantied (personalizable) computing power for the user. In general, this advantage conflicts with the aim of making effective use of collective computing resources. NetWork is an approach to make shared use of the computing resources in a network while respecting the absolute priority of any individual user on his/her machine. NetWork tries to keep the advantage of personal computing, that is the reliably guarantied (personalizable) computing power for the user.

The goal of NetWork is to optimize the net work output of a computing system. With todays computing facilities, it is not the computing time which is critical, it is the user's time, the effective wall-clock time to completion of a task. Thus NetWork heads for short job turn-around times, and ignores cumulative computing time.

Since a user may choose to leave or access his/her machine at any time, absolute priority of individual users implies that no availability or persistency of shareable computing resources can be considered guarantied over the network. Moreover, since network access might be critical to individual users, the network load due to distributed computing has to be minimized. Hence the usual acknowledgement schemes cannot be guarantied. The general ideas of NetWork are discussed in a separate article[1]. We will often refer to this general introduction in the sequel and assume that it is known to you.

NetWork is based on networks of Macintosh computers, but it can be used in heterogeneous networks as well.

The NetWork software consists of

SchedulerUnit.p	a PASCAL unit supporting high level access to the NetWork system
NetWork.p	a PASCAL unit for low level access to the NetWork system
NetWorkLookup.p	a PASCAL unit with AppleTalk-based look-up procedures for NetWork
NetWorkLib.o	a library to link with your object code. It contains a compiled version of NetWork.p and NetWorkLookup.p, and glue code
NetWork Processor	a communication agent and idle monitor used for the NetWork

[1]G. Sawitzki: The NetWork Project. Universität Heidelberg 1991

NetWork Tools	a collection of tools and sample programs to use or experiment with, including Spinning Brain, a sample program demonstrating the use of asynchronous iterations applied to neural networks

For C programmers

NetWork.h	a C header file, equivalent to NetWork.p

is provided.

NetWork Terms and the NetWork Computation Model

A **NetWork machine** is an addressable computing facility. A NetWork machine in general will, but need not correspond to a physical machine. Each machine has an **owner** (or home user). The owner may, but need not correspond to a real user. For example, if the machine is a dedicated server, the server process can be considered the owner. The owner is the source of events which have absolute priority on the owner's machine.

A machine may be in one of two states, depending on the activity of the owner. It may be **used**. A machine is used by its owner, if the owner directly accesses the machine, or if an owner initiated process requires the resources of the machine. If a machine is not used, it is **idle**.

A machine is addressed by its **NetWork node address**. A machine may have more than one address (for instance if it can be accessed with various transport systems using different address conventions). NetWork does not address users.

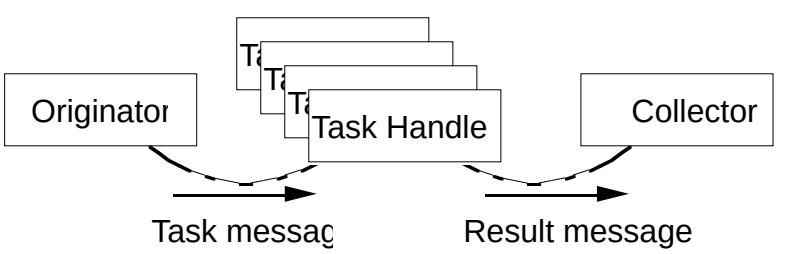
A **process** is any activity (process, task, application…) running on a machine and visible to NetWork. Processes reside on machines. A process has no identity for itself. Process classes are identified by their **signature**. The signature denotes any instance of a particular process class rather than a particular process. Processes are bound to a machine. We do not deal ith relocating processes. For communication, a process is identified by its **NetWork address.**The NetWork address of a process consists of its signature and the address of the NetWork node it is residing on.

Any process will eventually generate **tasks**. A task is a lump of work which someone has to do.

Distributed computing is viewed by NetWork as involving three elementary agents:

- an originator defining the compute task(s),
- a task handler , or task handler processes, performing the core of the computation,
- a collector handling the results.

The originator is a **task generator.** It generates a task message and passes it to the **task handler**. The task handler evaluates the message, performs the appropriate calculation, and eventually returns a result to the collector. The collector has the job of assembling the (partial) results to give a final solution (Figure 1). From an abstract point of view, the collector is just a special case of a task handler. It is specialized in integrating results.



Originator:	**Task Handler:**	**Collector:**
allocate co-workers assign sub-tasks	take sub-tasks evaluate sub-tasks reply result	take results evaluate them

Figure 1: Conceptual processes and messages.

The separation into originator, task handler and collector is a conceptual one. For instance, originator and collector may be implemented in the same program. This would give a master-slave scheme. Using a chain of task handlers with master and collector in in the same program would give a circular scheme. NetWork is flexible enough to implement any other topology as well.

Processes will eventually communicate. NetWork communication goes by **messages**. A message originates in one process and is transmitted to another process (messages are not shared). Typically a message either is a description of a task to do, or it contains the result of a completed task.

Message Destinations and Processes (Alive and Dead)[2]

NetWork provides a distributed computing environment giving absolute priorities to the owner of a machine. This is the basic design principle of NetWork: the priority of the owner must not be questioned. This implies that NetWork must be prepared for a random environment without guarantied availability. Moreover, NetWork has to economize on communication resources. Consequently no session is maintained between processes. All required information must be (either explicitly or inplicitely) passed in the messages. No computational state is maintained across tasks. A message may contain knowledge references, however, and the corresponding knowledge may be cached by processes.

A message comes from a source process and goes to a destination process. In a non guarantied environment, the destination need not exist: since process persistence is not assumed in NetWork, there is generally no guarantied process in NetWork. If the destination does not exist, NetWork may try to find the corresponding program file and launch it (“launch on task”). The original message is then forwarded to this new process.

Besides a destination, each message has a process to reply to.

- The destination is the process to perform the task.
- The reply-to process is the process to whom the result is delivered.

The reply-to process typically is the collector, or the next task handler in a chain of task handlers. It may be the same process as the source. Reply-to addresses are also found in SMTP headers. Compared to the SMTP header, or more specifically the RFC 821/822 recommendations used widely in electronic mail, the source process corresponds to the from-address, the destination corresponds to the to-address, and the reply-to process corresponds to the reply-to address.

[2]For detailed information, see the separate NetWork Technote "Message States".

Processes occuring as destination process may be launched by NetWork as a task arrives. Processes launched upon remote request are killed as soon as possible if the owner of the machine does anything. This leads to three classes of process which have different termination time rules.

- master processes, typically launched and terminated explicitly by the owner.
- local slave processes, typically launched to serve a (local) message and terminated using a message or if all master processes are terminated.
- remote slave processes, typically launched to serve a (remote) message and terminated using a message, or terminated by NetWork when the local owner accesses the machine.

> **(Macintosh only)** When the state of the system changes to busy, NetWork Processor terminates slave processes by faking command-Q during a call to WaitNextEvent, GetNextEvent or EventAvail. However it additionally tests if less than a tenth of a second passed since the systems state changed, or if the application does not respond to command-Q (calling MenuKey), or if a modal dialog window is front-most. If any of these condition holds, NetWork Processor calls ExitToShell instead of faking the event.

Although there is no guarantied lifetime of a process over the net, an active originator process should be guarantied until a complete task solution is guarantied. The originator process is only a source of tasks. The lifetime of the collector process should be guarantied from the arrival of the first partial result until the arrival of the last partial result to complete the task solution. The collector process is a “sink” of information. It will be a general process to reply to.

NetWork Architecture

The general NetWork architecture consists of three levels: communication, scheduler, and application. From the NetWork point of view, the application layer can handle two distinct things: it can generate sub-tasks, and it can handle tasks (which may be results, or new tasks).

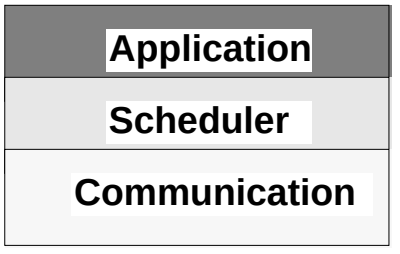


Figure 2: NetWork layers

The application layer contains application specific functions, written in a partitionable way. The proper definition of the tasks and the evaluation of the results are part of the application layer. The scheduler performs the administrative services. It is responsible for allocating co-workers, assigning tasks to them, and/or collecting results. The scheduler interacts with the communication layer by

passing messages. The communication layer handles the details of identifying (and if necessary creating) the communication partners, passing messages, and related tasks. The communication layer cooperates with the transport systems available to exchange these messages with other processes.

The NetWork design principles imply that neither a guarantied communication nor a persistence of processes can be supposed. Both the communication layer and the scheduler layer try to shield the application from a non guarantied environment. Both layers try to reduce communication load.

The core of the NetWork communication system, together with additional support and monitoring functions, is implemented as a separate process. This agent is the proper NetWork processor. The complete scheduler and access functions for the communication system are supplied as a library and linked to the application.

> (**Macintosh only)** The time critical communication code is implemented as a driver which will be installed by the INIT code attached to the NetWork Processor control panel extension. The scheduler interacts with the application layer by calling application-supplied task handlers.

Since NetWork is a message passing system, the general picture is best understood in terms of message flow. In a completely controlled system, there would be a simple message flow. The task generator defines a message (a task description) and sends it. A task handler receives the task, performs the appropriate calculations. Then the task handler would define a message (the result) and return it to the appropriate address. The recipient of this reply (which may, but need not be, the same as the first task generator) takes the result, and integrates it into its current state.

From an abstract point of view, we need not worry about results. They can be considered as a special task, i.e. integrate the result information in the current state. Everything which applies to a task holds for results as well. So we will ignore results for now.

As we have separated application specific code from general message control, and message control from transport, we have the simplified picture given in figure 3:

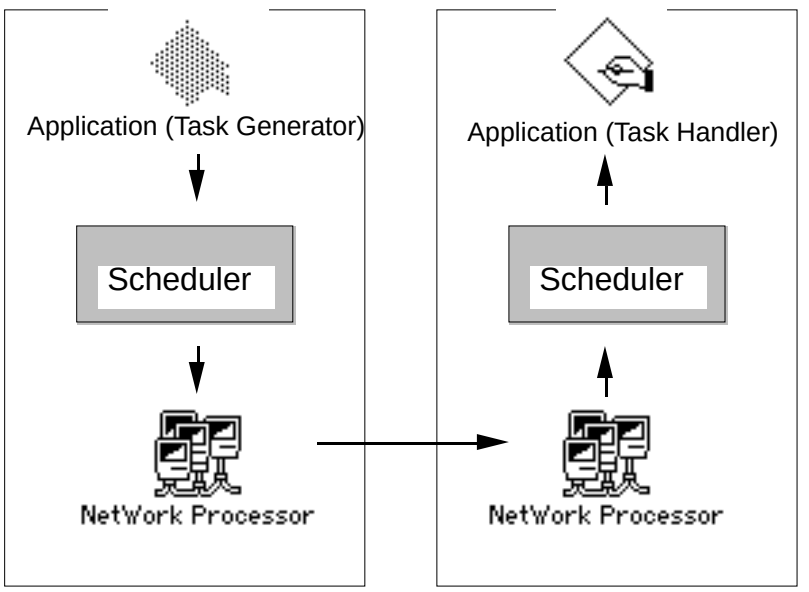


Figure 3: NetWork message flow: a simplified picture. The task generating application defines a task message and hands it to the scheduler. The scheduler does the necessary housekeeping and passes the message to the NetWork processor which communicates it to the receiving NetWork processor. The receiving NetWork processor launches the destination application (if necessary). The scheduler of the destination passes the message to the task handler of its application.

NetWork Messages[3]

Messages can only be created and destroyed by the communication system. All messages belong to the communication system. NetWork messages are referenced by message identifications. The identification is an internal identification controlled by the communication system. The communication is free to perform housekeeping action on its messages[4].

Messages are handled by the NetWork processor. An application sends a message via its NetWork Processor to the NetWork Processor of a remote machine. The destination node NetWork Processor makes best efforts to forward messages to processes.

For NetWork, a message consists of header information and **core information**. The core information is the real information load. The header information is a (in general short) description of the contents of the core information. In general, it will be accessed prior to using the core information. In communication, "header" is a standard term for a part of the message which consists of administrative information in a fixed form. Since "header" is a standard term in communication, we will reserve the name "header information" for the fixed (application independent) part and use the term "**priority information** "for the application dependent part of the header. From now on, we will even reserve the term "**header**" for the (application independent) part of the header. Header and priority information are used to identify the message and to decide whether it is usable in a given context. The core information will only be accessed when the message is usable.

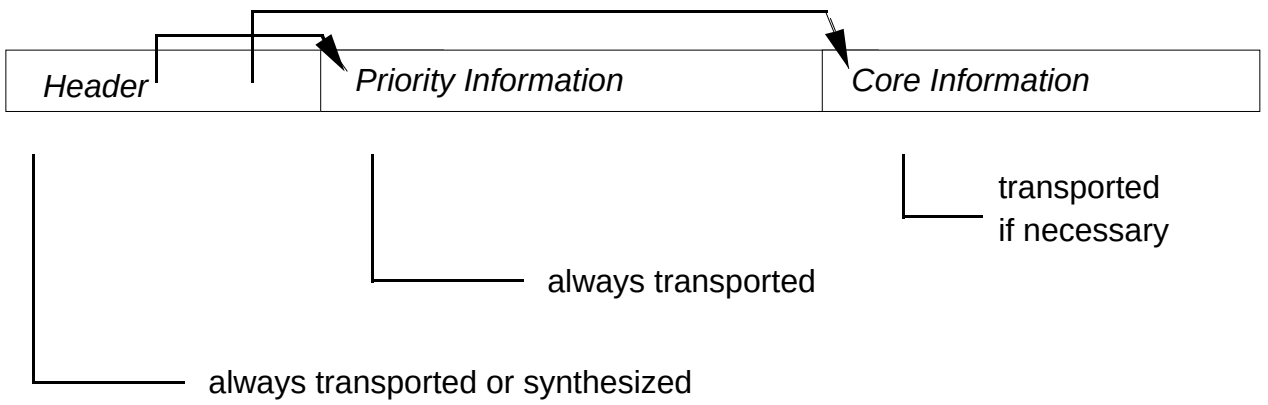


Figure: Separation into first/second class information. Header and priority information identify a message. The core information need only be transported if the message is usable.

In a random environment, no session maintenance is guarantied. Hence messages can be outdated or out of context. The recipient of a message, the task handler, has to check whether a message is usable. The header and priority information should contain all information necessary to do this check. NetWork needs only transport the core information of a message is accepted as usable. Details are implementation dependent and may depend on the communication system.

[3]An overview of the NetWork communciation model is given in the general introduction to the NetWork project. A detailed account is given in J. Lindenberg: NetWork Communications. Universität Karlsruhe 1990

[4]There is a current limit set to 2 minutes for messages which are processed internally after which a timeout condition is set. Messages must be removed explicitely using destroy.

The NetWork Processor does a first check of the message header and discards any incoming message which are not usable because no recipient can be identified or because the required capabilities are not given. If it does not discard the message, it will then try to find a matching process. If it succeeds, it will forward the message to this process. If not, it will try to find and launch an application with a signature matching the code identifier ("launch on task"). The original message is then forwarded to this new process.

The sending process defines a message and passes it to the communication system. If the message has been transmitted, the sending process may release its all buffers associated with the message. It then tells the communication system to destroy the message.

When a new message arrives, the receiving process first evaluates the message header. If the header indicates that the message is unusable, it tells the NetWork Processor to destroy the message. If the header is usable, it allocates buffers for the priority information and asks the NetWork Processor to get it. When the priority information is available, it can be checked. If after checking the priority information the message is not considered usable, the recipient tells the NetWork Processor to destroy the message. If the message is considered usable, the recipient allocates buffers for the core information and asks the NetWork Processor to accept it. When the core information is available, it is evaluated and the NetWork Processor is told to destroy the message.

Typical sequences of message/task handling are given in figure 5.

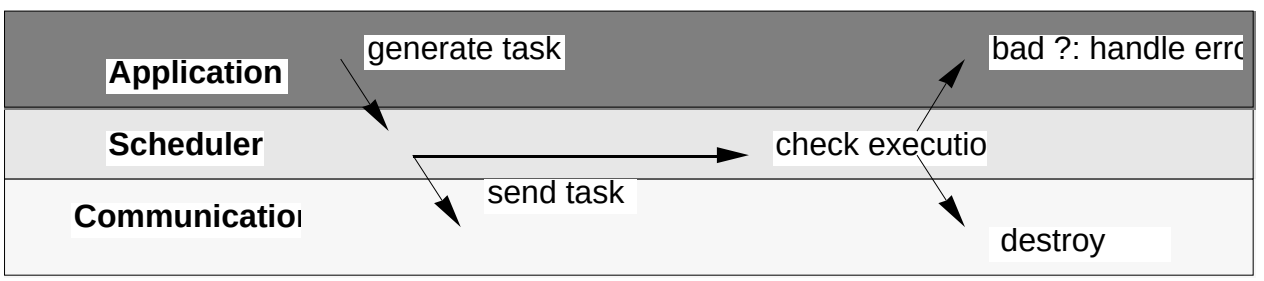


Figure 5a: Message sent: the message is passed to the communication system via the scheduler. On successful delivery, the scheduler will automatically destroy the administrative message information. On error, it will try handling the error condition, possibly by calling the application.

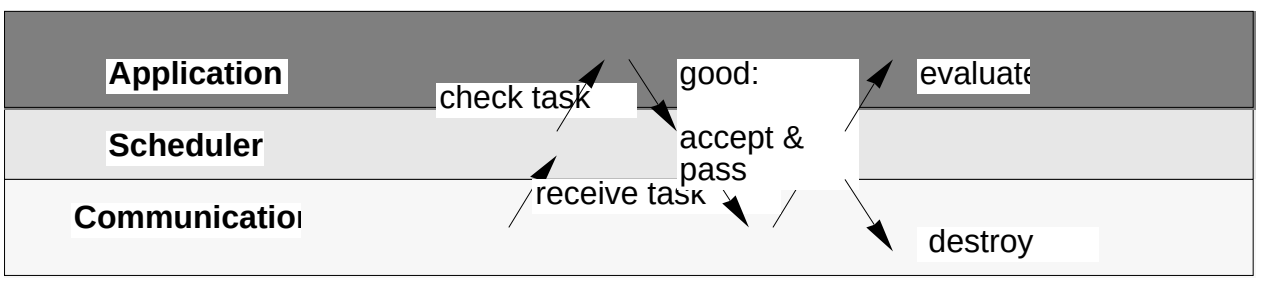


Figure 5b: Message received with success. A message is received via the scheduler layer. The scheduler will handle the (possibly asynchronous) receipt and then release the administrative information.

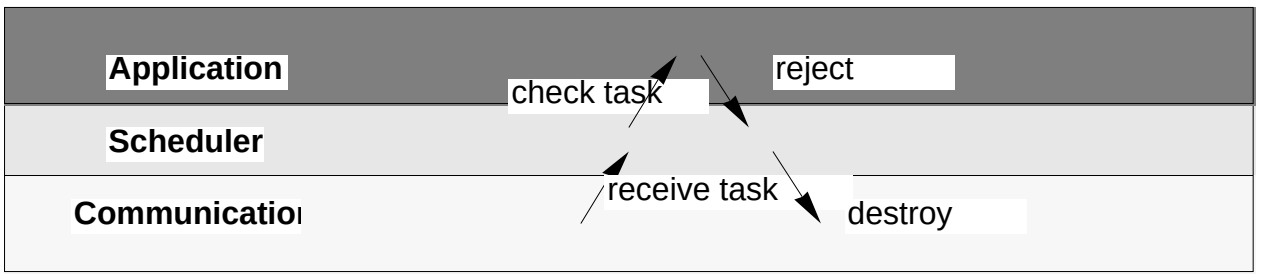


Figure 5c: Message received without success. The scheduler will try to prevent unnecessary transport and release the administrative information.

> **Important:** Depending on the communication system, any communication process may take considerable time. NetWork does **not** make temporary copies of the information. Use MsgStatus to make sure that there is no message transfer pending if you want to change the information associated with a message.

> **Important:** Messages are created by the NetWork driver. You must call DestroyMsg to free the space allocated internally in the NetWork system for a message. DestroyMsg does not affect the message information, or space you have allocated directly. Messages are freed if your process terminates.

> NetWork does not prevent forwarded messages circulating around. If you are writing a router or forwarder, it should take appropriate measures, for example implement a hop count or a message history table

Message Handling in a Random Environment

In the previous section we discussed the general message handling in NetWork. The NetWork Scheduler is an implementation model designed to handle messages in a random environment. To allow for flexible handling of messages, the scheduler associates a message handler to every message and invokes this message handler in the appropriate situations. Handling an outgoing message and handling an incoming message are not symetric[5]. As a consequence, message handlers come in two kinds: task generators, handling outgoing tasks, and task handlers, handling incoming tasks. The implementation of both is application specific. The time of invocation can best be understood by a picture:

[5]They could be designed in a symetric way. But this would require acknowledgements schemes, hence additional network load. This is against the design principles of NetWork.

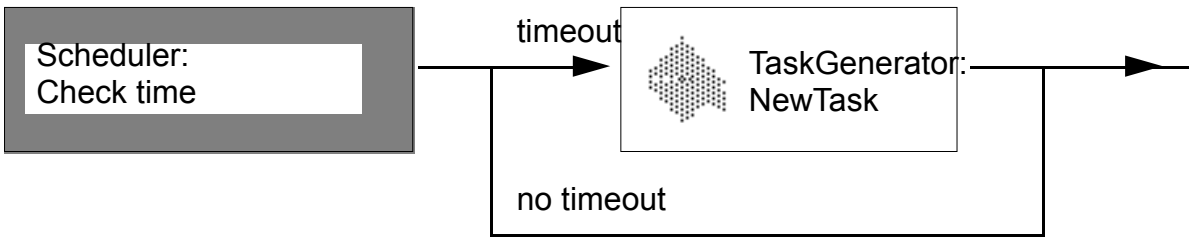


Figure 4a: NetWork control flow: a simplified picture. The periodic scheduler task does its housekeeping. Based on a timer, the TaskGenerator is asked for a new task.

The task generator is invoked periodically, based on a timer. Any application is free to invoke the task generator apart from the timing condition whenever it seems appropriate. To honour the NetWork design principles, an inhibition time is respected after a task has been generated and sent. This helps to avoid excessive network and computation load.

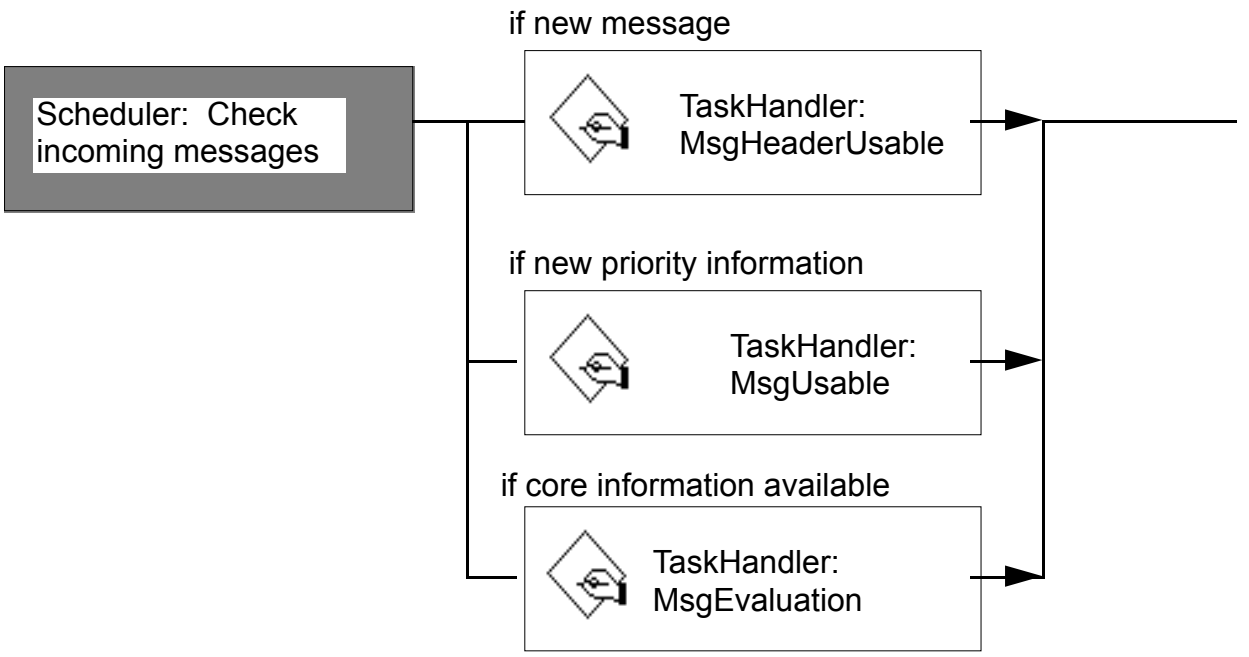


Figure 4b: NetWork control flow: a simplified picture. The periodic scheduler task does its housekeeping. If new messages are available, the TaskHandler is called to check or evaluate the new messages. Then, based on a timer, the TaskGenerator is asked for a new task.

The task handler invocation is based on events. Events are: a new message has arrived; the priority information is ready for inspection; the core is ready for inspection. Different actions can be applied in these three stages.

Neither picture shows an additional invocation: when a message is completed, it must be destroyed. Destroying a message implies telling the communication system that the message is not needed any more. For the task generator, a message is completed when it is sent successfully, or timed-out, or an error occurs. For the task handler, a message is typically completed immediately when the task evaluation is performed. However, time-out or error conditions may occur as well. The application should release all its buffers assiociated with a message after DestroyMsg has been called.

Inside NetWork [6]

This part of the documentation is only of interest to you if you want to implement a program using NetWork. It starts with an introduction to NetWork's data types. Next, the high level components of the NetWork software are presented: the NetWork Scheduler and NetWork message handlers. You have to define your own message handlers and you have to know how to interface them to the scheduler to write your own programs. NetWork utilities follow. You may make use of the additional NetWork services provided by these routines. You rarely have to go beyond this material in ordinary applications. If you want to modify basic components of the NetWork system however, you will find these components next: the look-up system, interface with the communication system, and transport system interface. The scheduler makes intensive use of these routines.

A cookbook and a collection of tricks and recipes are added.

Basic NetWork Data Types

Address Data Type

```
MsgAddr = record
    a : longint;
    p : longint;
end;
```

MsgAddr is used to hold NetWork addresses. The “a” component denotes the machine (or machine address), “p” denotes the process (or process class). NetWork supports multiple concurrent transport systems. Since a universal addressing scheme does not exist, the interpretation is specific to the transport system.

Fields and Methods Description

a	network address - interpretation depends on transport system
p	signature or program number - use longint , for example longint('NetE');

Message Data Type

```
MsgPtr = ^MsgRec;
MessagePtr = MsgPtr;
```

```
MsgRec = record
{ message information used by the communication system. This information is local }
    MsgLink    : MsgPtr;      { used by NetWork }
    Msg2Link   : Ptr;             { used by NetWork }
    MsgResult  : integer;         { >0 busy, =0 done, <0 error }
    MsgFlags   : SignedByte;      { reserved - lock & attn flags }
    MsgCmd     : SignedByte;      { command (phase)}
    MsgTicks   : longint;         { timeout (ticks) for this message}
```

[6]This documentation should be synchronized with your software version. Please consult the interface files in case of doubt.

```
    MsgUserRefCon   : longint;     {~used by NetWork Scheduler}

    MsgReserved1    : longint;     { reserved for NetWork processor }
    MsgReserved2    : longint;     { reserved for NetWork processor }
    MsgReserved3    : longint;     { reserved for NetWork processor }
    MsgTrpPtr  : TransportPtr;     { transport system used by message }
    MsgTrpRefCon    : longint;     { free for transport system use }

{   message header information This information is transported to the destination }
    MsgSource  : MsgAddr;          { may be overridden by NetWork processor }
    MsgDest    : MsgAddr;          { address the message is sent to }
    MsgReply   : MsgAddr;          { address to return replies/results to }
    MsgCapasVerb    : longint;     { capas : integer, verb : integer }
    MsgReference    : longint;     { transported to other nodes }

    MsgPrioSize     : longint;
    MsgCoreSize     : longint;


{   message header information This information is local }
    MsgPrioPtr : Ptr;
    MsgCorePtr : Ptr;

end;
```

MsgRec is used as message header, and MsgPtr points to a header. All messages belong to the communication system. MessagePtr is used as an alias for MsgPtr to point to application-controlled copies of message header which may be used as parameter blocks

Fields and Methods Description

MsgLink	reserved for NetWork
Msg2Link	reserved for NetWork
MsgResult	>0 busy, =0 done, <0 error
MsgCmd	command (phase). See command codes.
MsgTicks	time in ticks until time out of message
MsgUserRefCon	LOCAL(! !!!!) application use. Reserved for the scheduler.
MsgTrpPtr	transport system used by message
MsgSource	address of source process of the message
MsgDest	address of destination process of the message
MsgReply	address of the process to send results to
MsgCapasVerb	capas : integer, verb : integer
MsgReference	application specific message stamp
MsgPrioSize	size of priority information
MsgCoreSize	size of core information
MsgPrioPtr	pointer to buffer for priority information
MsgCorePtr	pointer to buffer for core information

Message Handlers

Message Handler Definition

```
tMessageHandler=object(tObject)
                  ContextStamp:longint;
                  NrPendingMessages:longint;
                  procedure init;
                  procedure restart;

                  procedure Stamp(Msg:MsgPtr);

                  function Destroy(var Msg:MsgPtr):OsErr;
                  function DisposMsg(var Msg:MsgPtr):OsErr;

                  {Buffer handling}

                  function NewPrioPtr(var PrioSize:longint):ptr;
                  function NewCorePtr(var CoreSize:longint):ptr;

                  procedure DisposPrioPtr(var PrioPtr:Ptr);
                  procedure DisposCorePtr(var CorePtr:Ptr);
                  end;
```

What Message Handlers Do for You

They handle messages. Usually, the message handlers will be called by the scheduler; you need not call the message handlers directly. Message handlers come in two kinds: task handlers, used to handle incoming messages, and task generators, used to define new outgoing messages.

What You Have to Do for Message Handlers

You need not modify any of the general message handler methods. You should override the buffer handling methods (NewXxxx, DisposXxxx) to implement a buffer strategy adapted to the problem at hand. Usually, you will not modify the basic class tMessageHandler, but derived classes. For the derived classes (task handlers and task generators) you have to define the application specific actions.

Where to Find Them

Message handler prototypes are provided in source form in file SchedulerUnit.p (interface) and SchedulerUnit.inc (implementation).

Message Handler Fields

ContextStamp — a user defined 32bit flag to verify context continuity. ContextStamp is compared with the MsgReference field of a message record. If both match, the message is assumed to fit in the current computation context. The ContextStamp is specific to a message handler. Multiple concurrent message handlers may have differing context stamps. A value of zero is recommended to denote an undefined (neutral) ContextStamp.

NrPendingMessages — number of pending messages for this message handler. Do not free the message handler unless NrPendingMessages is zero.

Message Handler Methods

```
procedure tMessageHandler.Init
```

Init is called once to initialize the message handler. Sets NrPendingMessages to zero and calls restart.

```
procedure tMessageHandler.Restart
```

Restores the message handler to a well defined operational state. Restart is used for a first start of the message handler within a context. For the sender, it sets ContextStamp to a new (locally unique) identifier. For the recipient, it sets ContextStamp to zero (=undefined/neutral).

```
function tMessageHandler.Destroy(var Msg:MsgPtr):OsErr;
```

Destroys the message record, and sets Msg to nil. Decreases NrPendingMessages by one, but does not dispose buffers.

```
function tMessageHandler.DisposMsg(var Msg:MsgPtr):OsErr;
```

Releases all buffers associated with Msg, and calls Destroy.

```
function NewPrioPtr(var PrioSize:longint):ptr;
```

Allocates a new buffer for priority data. Entry: PrioSize=Requested size; Exit: PrioSize=Allocated size. Defaults to NewPtr.

```
function NewCorePtr(var CoreSize:longint):ptr;
```

Allocates a new buffer for core data. Entry: CoreSize=Requested size; Exit: CoreSize=Allocated size. Defaults to NewPtr.

```
procedure DisposPrioPtr(var PrioPtr:UNIV Ptr);
```

Disposes buffer for priority data. Defaults to DisposPtr.

```
procedure DisposCorePtr(var CorePtr:UNIV Ptr);
```

Disposes buffer for core data. Defaults to DisposPtr.

Task Handlers

Task Handler Definition

```
tTaskHandler=object(tMessageHandler)
                  PrivilegedInterval:     longint;
                  PrivilegedTimeout:      longint;
                  PrivilegedAddr:         MsgAddr;
                  UsableCapas:            longint;

                  procedure Init; override;
                  procedure Restart; override;

                  function MsgHeaderUsable(var msg:MsgPtr):boolean;
                  function MsgUsable(var msg:MsgPtr):boolean;
                  procedure MsgEvaluation(var msg:MsgPtr);

                  end;
```

What Task Handlers Do for You

They handle incoming messages, typically containing task descriptions or results (=tasks to update your state). Usually, the task handlers will be called by the scheduler; you need not call the message handlers directly.

What You Have to Do for Task Handlers

You need to modify the MsgEvaluation method to perform application specific actions. You may want to modify the MsgUsable to check the priority information of a message and tell whether the core information is usable at all. You may want to modify the MsgHeaderUsable method to do a first check of the message, and delegate or mark it for discard it if appropriate. If you want to use your own buffer handling scheme, you have to override the buffer handling methods inherited from tMessageHandler.

Where to Find Them

A task handler prototype is provided in source form in file SchedulerUnit.p (interface) and SchedulerUnit.inc (implementation).

Fields and Methods Description

PrivilegedAddress	"Master" address. This is used to maintain a minimal session continuity. If PrivilegedTimeout is not zero, and master is not timed out, only messages from this origin should be accepted.
PrivilegedTimeout	next timeout point
PrivilegedInterval	timeout interval

PrivilegedInterval, PrivilegedTimeout and ContextStamp are dedicated fields to allow for a minimal session maintenance. The default implementation of MsgHeaderUsable uses these fields. If the address part of the DefaultMsg is not zero, and master is not timed out, only messages from this origin will be accepted. PrivilegedInterval is used to hold a timeout interval for the approximate lifetime of faithfulness to the privilegedOrigin: after a message from privilegedOrigin has arrived, only messages from privilegedOrigin are accepted for the duration of PrivilegedInterval. If PrivilegedInterval is exceeded, the process corresponding to privilegedOrigin is considered defunct and messages from other processes are accepted as well. PrivilegedTimeout is used to hold the next timeout (next timeout point =last encounter + PrivilegedInterval).

The context stamp is reserved for application dependent identification of a context. Intended use: Only messages matching this ContextStamp, or with context stamp zero may be usable.

```
function tTaskHandler.MsgHeaderUsable(var msg:MsgPtr):boolean;
```

MsgHeaderUsable will be called when the priority information of a message is available. Given the msg identification of an existing message, MsgHeaderUsable should investigate the message header to determine whether the message indicated is compatible with the current context, and if so return *true*, else it should return *false*. Whenever a task handler decides to accept a task for closer inspection, it should allocate appropriate buffers for the priority information, and stamp the corresponding message by calling the Stamp method.

The default method does a preliminary check, based on the priority fields of tMessageHandler.

```
function tTaskHandler.MsgUsable(var msg:MsgPtr):boolean;
                        {in general, this must be customized}
```

MsgUsable will be called after the header information of a message has been checked. Given the msg identification of an existing message, MsgUsable should investigate the priority information to determine whether the message indicated is useful. If it is, MsgUsable should verify that buffer space for the core information is available, adjust the CorePtr and CoreSize entries of the message header, and return *true*, else it should return *false*.

The default always returns *true*.

```
procedure tTaskHandler.MsgEvaluation(var msg:MsgPtr);
                        {in general, this must be customized}
```

MsgEvaluation will be called when the priority and the core information of a message are available. MsgEvaluation should initiate the execution of the task indicated by the message if necessary (e.g. by setting appropriate flags etc.).

```
procedure tTaskHandler.Init;
                        {in general, this must be customized}
```

Once only initialization. Init should set all fields of tTaskHandler.

Init is called by the scheduler on installation of a TaskHandler.

Task Generators

Task Generator Definition

```
tTaskGenerator=object(tMessageHandler)
                   TickleInterval:         longint;
                   WaitInterval:           longint;

                   DefaultCapasVerb:       longint;

                   procedure Init;override
                   procedure Restart;override;
                   procedure Stamp(msg:MsgPtr);override;
                   function NewTask(var msg:MsgPtr):boolean;

                   end;
```

What Task Generators Do for You

They handle outgoing messages, typically containing task descriptions or results (=tasks to update your state). Usually, the task handlers will be called by the scheduler; you need not call the task generators directly.

What You Have to Do for Task Generators

You need to modify the NewTask method to perform application specific actions. If you want to use your own buffer handling scheme, you have to override the buffer handling methods inherited from tMessageHandler.

Where to Find Them

A task generator prototype is provided in source form in file SchedulerUnit.p (interface) and SchedulerUnit.inc (implementation).

Fields and Methods Description

TickleInterval	approximate interval for trying for a new partner (in ticks).
WaitInterval:	approximate interval to wait before new task.
DefaultCapasVerb	default values to be used for MsgCapasVerb in NewTask

```
function tTaskGenerator.Stamp(msg:MsgPtr);
```

```
function tTaskGenerator.NewTask(var msg:MsgPtr):boolean;
```
{in general, this must be customized}

If a new task can be assigned, NewTask should compose the appropriate task information in the message header part of the message record and pass a pointer to the record. It should return nil, if no task can be defined. The default enters the task generator default values, but returns false.

Scheduler

NetWork Scheduler Definition

```
tScheduler=object(tObject)
                  MySelf:                 MsgAddr;
                  MyTransport:            TransportPtr;

                  receiving:              boolean;
                  sending:                boolean;

                  CoHandler:              tSchedulerCohandler;
                  TaskAddr:               MsgAddr;
                  TaskId:                 longint;
                  TaskIterations:         longint;

                  TaskHandler:            tTaskHandler;

                  TaskGenerator:tTaskGenerator;
                  NextTickle:             Longint;
                  NextWait:               Longint;
                  PrevDest:               MsgAddr;

                  Err:                    OsErr;
                  ErrQuiet:               Boolean;
                  ErrFrom:                tSchedulerPhase;

                  procedure handleError(from:tSchedulerPhase;which:OsErr);
                  procedure Init;
                  procedure Reset;
                  procedure Free; override;

                  Procedure SetSending(onOff:Boolean);
                  Procedure SetReceiving(onOff:Boolean);

                  Procedure InitTaskHandler(newTaskHandler:tTaskHandler);

                  Procedure InitTaskGenerator(newTaskGenerator:tTaskGenerator);

                  Procedure HandleMsg(Msg:MsgPtr);

                  procedure PeriodicTask;
                  function GetSleep:longint;

                  procedure KickOff(maxCount,maxticks:integer);

                  procedure DoNewTask(addr:MessageAddr;Transport:TransportPtr);
```

```
procedure sendMessage(msg:MessagePtr);
                  procedure replyMessage(msg:MesagePtr;flagToAdd:longint);

                  end;


var               NetWorkScheduler:tScheduler;
```

What Scheduler Does for You

The scheduler manages task handling and NetWork control flow. In general, there will be only one instance of the scheduler, the NetWorkScheduler. When a message is signaled by the NetWork communication system, the scheduler delegates it to the appropriate task handler. When a new task may be created, the scheduler calls the task generator, and submits a newly defined task if appropriate.

What You Have to Do for Scheduler

The scheduler is implemented as an object of class tScheduler. You have to generate an instance of this object using new(NetWorkScheduler). You initialize the new scheduler once by calling NetWorkScheduler.Init. PeriodicTask must be called frequently as indicated by NetWorkScheduler.GetSleep, and HandleMsg must be called whenever a NetWork event occurs. If your program is event driven, place a call to NetWorkScheduler.PeriodicTask and NetWorkScheduler.HandleMsg in your event loop. Use NetWorkScheduler.GetSleep in you calculation of the sleep value for the main event loop. If PeriodicTask is called regularly, the scheduler guaranties that the NetWork communication system is kept active. Call NetWorkScheduler.Free once when you are about to leave your program.

```
PROCEDURE EventLoop:
VAR
    cursorRgn: RgnHandle;
    gotEvent:       Boolean;
    event:          EventRecord;
BEGIN
    cursorRgn := NewRgn; {pass an empty region the first time thru}
    REPEAT
          gotEvent:=WaitNextEvent(everyevent, event, GetSleep, cursorRgn);
          AdjustCursor(event.where,cursorRgn);
          IF gotEvent(event) THEN
               DoEvent(event)
          ELSE
               BEGIN
                    NetWorkScheduler.PeriodicTask;
                    IF NLTask<>NoErr then ProgramBreak('Error in NlTask');
                    DoIdle;
               END;
    UNTIL FALSE;         {loop forever}
END;
```

A Simple Event Loop, modified to use NetWork.

```
PROCEDURE DoEvent(event: EventRecord);
BEGIN
    CASE event.what OF
          mouseDown:
               DoMouseDown(event);
          mouseUp:
               DoMouseUp(event);
          keyDown, autoKey:
               DoKeyDown(event);
          activateEvt:
               DoActivate(event);
          updateEvt:
               DoUpdate(event);
          osEvt:
               DoOSEvent(event);
          NetWorkEvent:
               NetWorkScheduler.HandleMsg(event.message);
          kHighLevelEvent:
               DoHighLevelEvent(event);
    END;
END; {DoEvent}
```

Processing Events, modified to use NetWork.

You should introduce two message handlers to the scheduler which will be used as defaults. You introduce a task generator by calling InitTaskGenerator. Whenever a message is to be generated, the default task generator will be asked. You introduce a task handler by calling InitTaskHandler. Whenever a message comes in which is not associated with a specific message handler, the default task handler will be asked.

You need not modify the scheduler.

Where to Find It

The scheduler is provided in source form in file SchedulerUnit.p (interface) and SchedulerUnit.inc (implementation).

Scheduler Message Handling

Messages come and go. The scheduler has no chance to know when a message has to go. You can determine the time to send a message on the fly, and call NetWorkScheduler.DoNewTask if a new message should be defined. Or you can call NetWorkScheduler.PeriodicTask periodically. PeriodicTask will check timing conditions, and eventually will call DoNewTask.

If a message might be prepared, the scheduler creates a new message record. It fills in all fields known to the scheduler, proposes a destination address, and calls the NewTask method of the default task generator. The task generator may refuse to define a new task by returning false, or it may allocate and set up appropriate buffers and fill out the appropriate fields defining a new task, or it may invoke another task generator to define the message.

Whenever a task generator decides to define a task for transmission, it should stamp the corresponding message by calling the Stamp method. If the scheduler recieves true as a result from a task generator, it passses the message to the NetWork Processor for submission. When the message is transmitted completely, or when an error occurs, the NetWork Processor will notify the scheduler. HandleMsg will pass the message to the appropriate task generator by calling the DisposMsg method.

The Scheduler does have a chance to know when a message comes in or when a message needs attention. With version 1.1 or higher, the NetWork Processor will use events to signal a message. Scheduler selects

the event type NetWorkEvent[7] in Scheduler.Init. You can select another type if necessary. Your main event loop should check for the appropriate message type, and call Scheduler.HandleMsg.

The Scheduler checks the MsgResult and MsgCmd fields of the message to find out the state of the message. It uses the MsgUserRefCon to associate a MessageHandler with the message - the MsgUserRefCon field is reserved if you are using the scheduler. The Scheduler uses MsgReference and MsgCapasVerb for context checks.

If a new incoming message is signaled, the Scheduler passes it to the default task handler by calling the MsgHeaderUsable method. The default task handler should inspect the message header. The task handler may discard the message by returning false, or it may accept it on first impression by returning true, or it may invoke another task handler to inspect the message header. If a task header is accepted, the MsgUsable method of the task handler now associated with this message will be invoked. If the message is accepted as usable, the Scheduler will get the bulk of the message, and call the task handler's MsgEvaluation method.

Whenever a message is to be discarded - for whatever reason - the Scheduler will call the appropriate methods of the message handler associated with this method to release buffers. If no message handler can be identified, the default handlers will be called.

At the obvious well defined moments, the Scheduler will inform the Cohandler to allow for adaptive variants.

Default actions

If a message provided by NewTask is not stamped, it will be considered an orphan. The default task generator will be asked to handle this task further on.

If a message accepted by MsgHeaderUsable is not stamped, it will be considered an orphan and the default task handler will be asked to handle this task further on.

If a message accepted by MsgHeaderUsable is not stamped, it will be considered an orphan. The default task handler will be asked to handle this task further on.

If a message accepted by MsgHeaderUsable does not have a priority buffer associated, but positive buffer size, the task handler will be asked to provide a buffer by calling NewPrioPtr. If NewPrioPtr does not return a pointer, an error condition exists.

If a message accepted by MsgUsable does not have a core buffer associated, but positive buffer size, the task handler will be asked to provide a buffer by calling NewStdPtr. If NewStdPtr does not return a pointer, an error condition exists.

[7]NetWorkEvent is defined in the MPW interfaces. The current interfaces define NetWorkEvent =10.

Error conditions

If a message provided by NewTask has a positive size for priority information or core information but no buffer can be allocated, an error condition exists. It will be signaled by the scheduler via the HandleError method.

Constants and Data Types

Masks defined to denote required/available capacities. The lower 16 bits are reserved for the NetWork scheduler. The upper 16 bits are reserved for the transport system.

```
cAnyCapas =    0;       {Scheduler reserved word. No known capabilities/no
                             special  capabilities required}
cMsgReply =    $8000;   {Scheduler flag. Message is a reply.}
cMsgNAttention=    $8000;    {Scheduler flag. Message needs special
                     attention/priority.}
UnknownFormat =    $00010000;{Message system flag. Reserved to mark future
                     extensions.}
cMustBeLaunched=   $80000000;{Message system flag. Recipient must be already
                              running. Don't launch.}


cNilError=MemFullErr;   {Scheduler error code:Nil error. Could not allocate
               storage.}


MessagePtr=MsgPtr;
```

This is an alias for the communication system pointer type. Communication system message pointers and record date structures are private to the communication system. We want to use the same data structure as a paramter block controlled by the application.

Fields and Methods Description

> The fields of the scheduler can be read, but should not be modified by objects other than the scheduler (or a scheduler cohandler, if installed).

MySelf	the address and type this scheduler is installed at.
receiving	receiving is supported if scheduler is running, and receiving is true. A TaskHandler must be installed.
sending	Task generation and sending is supported if scheduler is running and receiving is true. A TaskGenerator must be installed. **Note:** ReplyMessage and SendMessage do not require the scheduler sending to be active.
Cohandler	hook for adaptive extensions of the scheduler. The following fields are communication fields for the cohandler (if installed). The scheduler/cohandler may fill these entries with proposals. The task generator's NewTask is free to change them
TaskAddr	(proposed use:) address of the effective compute server for this task
TaskIterations	(proposed use:) the number of elementary actions to perform. Can be used for adaptive extensions of the scheduler.

TaskId	(proposed use:) a stamp to identify the task. The scheduler fills it with a time stamp upon InitTaskGenerator. Can be used for adaptive extensions of the scheduler.
TaskHandler	will be called whenever a message comes in.
TaskGenerator	will be called whenever scheduler wants to send a message.
NextTickle	used internally (next time to make a try call)
NextWait	used internally (do not call before this time)
PrevDest	previous destination of a task
Err	Latch error code here to allow for silent implementations.
ErrQuiet	Do not report error messages (used during init and exit)
ErrFrom	Latch error source here to allow for silent implementations.

```
procedure tScheduler.handleError(from:tSchedulerPhase; which:OsErr);
```

Handle errors during scheduler activity. The default just calls the debugger. **Should be customized.**

```
procedure tScheduler.Init;
```

The once-only initialization. Call this after initializing the toolbox and creating a scheduler with New(NetWorkScheduler).

```
procedure tScheduler.Reset;
```

Reset to no error, not sending, not receiving.

```
procedure tScheduler.Free; override;
```

Call this before leaving your program to dispose of the scheduler and NetWork services.

```
Procedure tScheduler.SetSending(onOff:Boolean);
```

Set the scheduler sending state.

```
Procedure tScheduler.SetReceiving(onOff:Boolean);
```

Set the scheduler receiving state.

```
Procedure tScheduler.InitTaskHandler(NewTaskHandler:tTaskHandler);
```

Installs NewTaskHandler in the scheduler, initializes the TaskHandler by calling NewTaskHandler.init, and resets the receiving flag. Call this after you have created a scheduler and a TaskHandler.

```
Procedure tScheduler.InitTaskGenerator(NewTaskGenerator:tTaskGenerator);
```

Installs NewTaskGenerator in the scheduler, initializes the task generator by calling NewTaskHandler.init, and resets the sending flag and TaskId. Call this after you have created a scheduler and a task handler.

```
procedure tScheduler.PeriodicTask;
```

The periodic task. **Should be called from the main event loop.**

```
function tScheduler.GetSleep:longint;
```

GetSleep returns the sleep value requested for the scheduler. The scheduler's PeriodicTask method would like to be called after GetSleep ticks.

```
procedure tScheduler.kickOff(maxcount,maxticks:integer);
```

Initiates a round of at most maxcount scheduler tasks for a total time of maxticks (whatever comes first). The usual send interval limitations do not apply during kick off. This can be used to have a quick start, for example to spray out an initial round of sub-tasks.

The following methods are for internal use of the scheduler only:

```
procedure DoNewTask(addr:MessageAddr;Transport:TransportPtr);
```

Call TaskGenerator.Newtask, with all support, for addr. Use this method to force the generation of a task.

```
procedure tScheduler.SendMessage(msg:MessagePtr);
```

Send the message indicated by msg. If msgowner is not nil, the msgowner's DisposMsg method will be called by the scheduler when the message is done. The message record indicated by Msg is free for application use immediately. Use this method if you have completely prepared a method and you want to transmit it immediately.

```
procedure tScheduler.ReplyMessage(msg:MessagePtr;flagsToAdd:longint);
```

Send the message indicated by msg, but to msg target (i.e. send a reply or result)If msgowner is not nil, the msgowner's DisposMsg method will be called by the scheduler when the message is done. The message record indicated by msg is free for application use immediately.

The Cohandler Object

The cohandler is provided to allow for flexible modifications of the scheduler, for example to implement adaptive schedulers. Cohandlers are optional extensions to the scheduler. If there is an adaptive scheduler installed, it should only take into account messages which are accepted by MsgHeaderUsable for its adaptive scheme, and just discard any message which is out of context.

For adaptive extensions, additional fields are provided. If you plan to make use of adaptive schedulers, TaskGenerator and TaskHandler should support these fields.

TaskAddr	address of the effective compute server for this task
TaskId	a unique identification of the sub-task corresponding to the current message

TaskIterations	indicator of the complexity of a sub-task. For asynchronous iterations, this may just be the requested/performed number of iterations.

The Cohandler should fill TaskId and TaskIterations with a proposal before NewTask is called. NewTask can make use of these fields, or use its own algorithm to determine the necessary parameters. If a TaskGenerator wants to make use of adaptive variants of the scheduler, it should notify the scheduler of any deviations from the scheduler's proposals by updating these fields of the scheduler with the values actually used.

For adaptive variants, the TaskHandler should notify the scheduler of any messages it has identified in the MsgUsable method (irrespective whether they are usable or not) by updating the scheduler's fields TaskId and TaskIterations, if appropriate. (The Destination field should contain the Destination to which the job resulting in the present message has been originally submitted, if any).

```
Type tSchedulerPhase=(pUndefined, pUsable, pUnUsable, pStartNewTask, pNewTaskDone,
                   pNoNewTask, pSendMessage, pHousekeepingDestroy, pInit, pFree,
                   pAcceptMsg);


tSchedulerCohandler=object(tObject)
                   procedure CoHandle(cmd:tSchedulerPhase;msg:MsgPtr);
                   procedure reset;
                   end;


procedure tSchedulerCohandler.CoHandle(cmd:tSchedulerPhase;msg:MsgPtr);
```

Inform the cohandler about the state of the scheduler

```
procedure tSchedulerCohandler.reset;
```

Reset the cohandler.

Utilities, Low Level Routines & Interfacing to the Message System

What the Utilities Do for You

They provide environment control and logging support.

What You Have to Do for the Utilities

Nothing, if you are using the scheduler.

If you are not using the scheduler, you have to call InitNetWork before using any utility routine.

Where to Find Them

The interface is part of NetWork.p. The object code is included in the file NetWorkLib.o – the implementation part of NetWork.p is included for your information only.

NetWork Routines

```
function InitNetWork:OsErr;function UseEventNo(eventcode : integer) : OsErr;
```

Initialization. Call InitNetWork once before use of any NetWork routine. Returns notOpenErr if no NetWork Processor is active. Sets event code for NetWork event. Then recommended value is eventcode=NetWorkEvent. If you use NullEvt, events are not signaled and you must use SignalMsg to find any messages which need attention.

Utility Routines

Logging & Debug Support

```
procedure AddrToString (Addr: MsgAddr; var s: Str255);
```

Convert an address to a standard string, for dumping etc.

```
procedure MsgToString (Msg: MsgPtr; var s: Str255);
```

Convert a message to a standard string, for dumping etc.

```
procedure LogString (s : str255);
```

Writes s to the NetWork log file if logging is switched on.

```
procedure LogStrTime (s : str255);
```

Writes s, followed by a time stamp, to the NetWork log file if logging is switched on.

```
procedure LogMsg (s : str255; Msg: MsgPtr);
```

Writes s , and the message, to the NetWork log file if logging is switched on. If logging is on and the next action is a call to destroy, it will not be logged.

```
procedure CheckError (s : str255; e : OSErr);
```

Writes s as an error message to the NetWork log file if e <> 0 and if logging is switched on.
Usage of CheckError may lead to inadequate overhead. It is recommeded to use

If e<>noErr then…

```
function TimeStamp :longint;
```

Returns a randomized time stamp. This can be used for example as a context stamp.

```
procedure ProgramBreak (s: str63);
```

Drop into debugger, if a debugger is installed.

Environment Control & Investigation

```
function Visible : boolean;
```

Returns true, if the process is supposed to display a user interface. Returns false for faceless background processes. Visible checks the “background only” bit in the SIZE resource, and the setting of the background entry of the NetWork control panel extension.

```
function Master : boolean;
```

Returns true if the process is considered a master process.

```
function Spare : boolean;
```

This returns the setting of the spare flag set with the NetWork processor control panel entry.

```
function GetProcessType(signature:longint;var ptyp:integer) : OsErr;
```

Given the signature of an active process, GetProcessType returns its process type.

```
function SetProcessType(signature:longint;ptyp:integer) : OsErr;
```

Sets the process type.

```
function GetIndProcess(var signature:longint;index:integer) : OsErr;
```

Get the signature of the process indicated by index relative to the internal tables of NetWork. A null signature indicates an unused slot.

Idle Monitor

What Idle Monitor Does for You

It monitors the idle/busy state of a machine

What You Have to Do for Idle Monitor

You have to call InitNetWork before using any Idle Monitor routine. Idle Monitor is an integrated part of the NetWork processor. Idle Monitor routines are only available after NetWork has been initialized.

Where to Find It

Idle Monitor is contained in the NetWork processor. The interface is contained in NetWork.p..

Idle Monitor Constants

```
imBusy   =    0
imIdle   =    1
imActive =    2
imLoaded =    3
```

Idle Monitor Routines

```
procedure PreventIdle;
```

Prevent NetWork from registering the machine as idle. This function may be used in applications which do not want to use "busy" cursors during lengthy operations. PreventIdle makes sure that an application is not considered idle prematurely. PreventIdle is honoured for master applications only. If needed, it should be called at least once every 15 seconds.

```
function Idle : boolean;
```

Returns true if the local machine is considered idle by the NetWork processor.

```
function IdleMonitorState : integer;
```

Returns the state of the idle monitor of the local machine.

```
function IdleTicks  : longint;
```

Returns number of ticks the machine has been idle, if idle is true. Else it returns -1.

Lookup Routines

What NetWork Lookup Does for You

It looks up possible partners. Upon request, it will return NetWork addresses of possible partners. By default, partners are identified by idle NetWork processors. Lookup can be used to register and look up other entities as well.

What You Have to Do for NetWork Lookup

Nothing, if you accept the Lookup prototype as distributed, and if you are using the scheduler. If you have a more efficient or economical look-up strategy, you can replace Lookup by a your own unit.

If you are not using the scheduler, you should call NLInit once to initialize it, call NLStart to start the look-up process and call NLTask frequently. You can use NLGetSleep to get the current sleep value (in ticks) for interval unitl the next call to NLTask.

Where to Find It

Lookup is provided in source form (interface and implementation) as file NetWorkLookup.p. The object code is included in the file NetWorkLib.o -- the implementation part of NetWorkLookup.p is included for your information only.

NetWork Lookup Routines

Constants Known by Addressing Routines

```
nlLocal             = 0    {can be used instead of local to denote this machine }
nlBroadcast         = -1;  {broadcast address, all of this cable except myself }
```

These constants are defined in NetWork.p.

Error Codes Returned by the Lookup Routines

```
const
    nlVersion       = -31100; { -- no appletalk version 48 or higher, could be
                          removed }

    nlTaskErr       = -31103;      { -- routines called in wrong order }
    nlNotFound = -31104;      { -- used internally }
    nlDupReg        = -31105;      { -- called NlRegister twice }
    nlNoReg         = -31106;      { -- called NlDeregister without NlRegister }

    nlAtkOffErr     = -31108; { -- appletalk off, cannot use function }
```

Lookup Routines

```
function NLinit;
```

Initializes the look-up system.

```
function NlTask : osErr;
```

Periodic name look-up task. Used to maintain the internal buffers of the look-up system. This function should be called frequently. The required maximum size is returned by NLGetSleep.

```
function NLGetSleep: longint;
```

Returns the preferred sleep value for the currently active look-up system. An optimal use of the look-up system would call NlTask approximately everytime NlGetSleep ticks have expired. More frequent calls may be redundant, less frequent calls may lead to outdated look-up tables.

```
function NlNode : longint;
```

Returns address of the local node. Returns zero if no remote transport system is active or if the driver is missing.

```
function NlStart : OSErr;
```

Start the look-up process.

```
function NlStop : OSErr;
```

Stop the look-up process. The settings of Lookup are preserved.

```
function NlCount : integer;
```

Returns the number of partners found (a positive number). Returns OSErr on error (a negative number).

```
function NLNext(after:longint) : longint;
```

Given the valid identification of a machine, NLNext returns the identification of the next machine based on the internal (circular) tables of NetWork. If the identification is not valid or zero, NLNext returns a random identification of a machine encountered. If there is no active machine encountered, NLNext returns zero.

```
function NLRandom:longint;
```

NLRandom returns a random machine based on the internal (circular) tables of NetWork. If there is no active machine encountered, NLRandom returns zero.

```
function NLActive(who:longint):boolean;
```

Given the valid identification of a machine, NLActive verifies whether this identification is still in the list of active machines. Whether the machine is indeed still active or reachable is not checked .

```
function NlSetSearch (NlName, NlType, NlZone : Str32) : OSErr;
```

Configure the look–up system. If no configuration is set, the default is to search for any name, type 'NetWork Processor', in the local zone (NlName '=', NlType 'NetWork Processor', NLZone '*').

```
function NlRegister (NlName, NlType : Str32) : OSErr;
```

Register current process as a compute server. To register under the name of the user as defined by the Chooser, pass an empty string as NlName.

```
function NlDeregister : OSErr;
```

De-register current process as a server.

NetWork Communication Interface

What It Does for You

The NetWork communication routines hide the underlying transport system from you. They take message descriptions from you and translate them to the appropriate low level communication actions. The NetWork library does not perform application specific buffer management for you - this rests with you.

What You Have to Do for It

Nothing, if you are using the scheduler.

If you are not using the scheduler, you should call InitNetWork once to initialize it, call the appropriate NetWork routines to use it, and call ExitNetWork when you are done with the NetWork services. If you have caled the communcation system to create a message record (using GetMsg or PostMsg), you have to destroy the message record if you are done with it. Destroying unused messages as soon as possible helps to keep the system efficient.

To create a message and mail it, you use PostMsg. To check whether a message is available - actually to get the number of available messages - call MsgAvailable. To get the priority information, call GetMsg. Eventually, you then will request the core information using AcceptMsg. You can check the current status of a message with MsgStatus. Call DestroyMsg if you have either completely received a message, or if you have decided that you will not use it, or if you have posted it and verified that it has been sent.

You can use the NetWork communication system based on events, if you have called UseEventNo. If you did not call UseEventNo, you have to poll for messages which need attention by calling SignalMsg.

> No assumptions about message sequence and transmission times should be made. The current status of a message processed by the NetWork message system can be queried using MsgStatus.

Where to Find It

The NetWork Communication library is provided in source form (interface and implementation) as file NetWork.p. The object code is included in the file NetWorkLib.o -- the implementation part of NetWork.p is included for your information only.

Addressing in NetWork

The internal communication of NetWork sends messages to processes on machines. In general, the process is identified by a system specific signature. The machines are identified by addresses.

NetWork deliberately tries to prevent addressing fraud. Whenever possible, NetWork will try to guarantee the proper originating address - to the best of its knowledge - as the source of a message.

> Signatures have to be registered with Apple. To experiment with NetWork, you can use the experimental signature 'NetE' (this spelling). This signature has been registered with Apple and is reserved for experimental use.

Message System

Message System Constants

```
const
{ general error messages }
    eQueEmpty      = -31000; {no more messages available -- out of memory }
    ePrio2Big      = -31001; {priority information to big }
    eNoSuchMsg = -31002; {invalid or NIL message reference, no message
                available (GET) }
    eNotLaunched   = -31003; {destination process does not exist - not used }
    eAbortMsg      = -31004; {message transfer aborted, e.g. timeout }
    eProcTableFull = -31005; {process table full (Init/Exit) }
    eNoSuchProcess = -31006; {specified process unknown }
    eNoMoreDynamics= -31007;      {maximum number of dynamic ids exceeded }
    eLaunchFailed  = -31008; {launch failed - not used }
    eInvalid       = -31009; {local message transfer aborted }
    eSizeLimit = -31010; {message larger than supported by transport }
    eVersion       = -31011; {version of library/driver/transport/system }
    eProtType      = -31012; {no transport, invalid network address -
                              Dispatcher}
    eLoopback      = -31013; {discard of broadcasted message }
    eTransportDown = -31014; {transport system not available }
    eCmdSequence   = -31015; {cmd sequencing error- bug of NetWork Processor
                              or Dispatcher}
    eProtIndex = -31016; {protocol index out of range }
    eProcessExists = -31017; {process with that creator already exists }
    eProcessIndex  = -31018; {invalid process index }
    eProcessType   = -31019; {process type illegal or does not match launch }
    eRestartListen = -31020; {a listener handled a request for more data,
                               and requires a restart therefore }
    eMsgTimeout    = -31021; {maximum message lifetime exceeded - AppleTalk }
    eNoSignature   = -31022; {couldn´t obtain signature of application file -
                              Library }
    eMsgLockFailed = -31023; { couldn´t lock message - NOT USED }
    eSigTableFull  = -31024; { signature table full }
    eSigRegistered = -31025; { signature already registered }
    eSigNotRegistered = -31026;   { signature not registered }
    eProcMgmtError = -31027;      { internal error }
    eSourceSig = -31028; { source signature wrong in Send/Post }
    eSourceAddr    = -31029; { source address wrong in Send/Post }
```

process types - see NetWork Communications

```
pUnknown   = 0;
pSlave     = 1;
pLocal     = 2;
pMaster    = 3;
```

layer prefs for launching

```
    LayerDefault   = 0;
    LayerFaceless  = 1;
    LayerBack      = 2;
    LayerFront = 3;
```

command codes

major command codes

```
    tGeneral   = $00;
    tListen    = $10;
    tGet       = $20;
    tAccept    = $30;
    tPost = $40;
```

minor command codes

```
    tStart     = $00;
    tTimeout   = $0C;
    tTimeout1  = $0D;
    tAbort     = $0E;
    tAbort1    = $0F;
```

```
var
    pDefault : integer;      {set this variable to pDynamic if you want a dynamic id}
    pFileSignature : longint;     { this is the signature of the application file-
                                  valid after InitNetWork }
    pProcessSignature : longint;  { this is the signature of the process. If the
                                    process type is pDynamic, it may be different from
                                    pFileSignature - valid after InitNetWork }
```

Except for MsgUserRefCon all of the components of the MsgRec structure are READ ONLY. If you are preparing messages to pass as a parameter block to the communication system, you should set all undefined components to zero.

Message Routines

```
function EqAddr (a, b : MsgAddr) : boolean;
```

```
function EqNode (a, b : MsgAddr) : boolean;
```

```
function IsLocal (networkaddress: Longint) : boolean;
```

True, if networkaddress is local.

```
function SetMsgAddr (a, p : longint) : MsgAddr;
```

```
function GetNetWorkAddr : MsgAddr;
```

Message Handling Functions

```
procedure DumpMessages;
```

Write a dump of all currently active messages to the log file.

```
function AvailableMsg : integer;
```

AvailableMsg returns the number of available messages.

```
function MsgStatus (Msg : MsgPtr) : OSErr;
```

Returns 0 if the Msg has been transferred completely, < 0 if there was an error, > 0 indicates that the Msg is still being transferred or waiting.

```
function SignalMsg (var Msg : MsgPtr) : OSErr;
```

Returns a pointer to the next message which needs handling. SignalMsg checks both old and new messages.

```
function GetMsg (Msg : MsgPtr;
                 PrioData : Ptr;
                 MaxPrioSize : longint) : OSErr;
```

Given a pointer to a message indicated as returned by SignalMsg, GetMsg reads the priority information to the area indicated by PrioData and MaxPrioSize.

```
function FlushMsg (DontFlushMask : longint) : OsErr;
```

Flushes pending messages. Only incoming messages corresponding to the current process signature will be flushed. Messages which are already accepted by the application (band(MsgCmd, tMajorMask) <> tListen) are not flushed. DontFlushMask can be set by the application. Messages will only be flushed if band(DontFlushMask,MsgCapasVerb) is zero.

```
function AcceptMsg (Msg : MsgPtr;
                 CoreData : Ptr; MaxCoreSize : longint) : OSErr;
```

Given a pointer to an existing message, AcceptMsg initiates the data transfer of the core data. CoreData is a pointer to a receive buffer of size at least MaxCoreSize. If the core information is longer than MaxCoreSize, the rest is discarded.

```
function PostMsg (var Msg : MsgPtr;
                 Trp :TransportPtr;
                 Capas : longint;
```

```
Stamp  longint;
                  Destination, ReplyAddr : MsgAddr;
                  PrioData : Ptr; PrioSize : longint;
                  CoreData : Ptr; CoreSize : longint) : OSErr;
```

Given pointers and sizes of existing information buffers, PostMsg generates a new message and initiates sending. It does not make a copy of the information that is referenced.

> The efficiency of the network access depends on the communication system used. NetWork does not impose restrictions on message sizes a priori. There are however constraints coming from transport systems.
>
> For AppleTalk, efficiency is best if the total size of both priority data and core data does not exceed 538 byte (May change in future implementations).
>
> For communication to MCP using A/ROSE, efficiency is best if the priority part does not exceed **12** byte if the A/ROSE user bytes should be used for the NetWork priority part.

```
function SendMsg (RefMsg : MsgPtr; var NewMsg : MsgPtr) : OSErr;
```

SendMsg uses the information in RefMsg to post a new message. All of the fields must be filled in. Undefined fields should be set to zero. The message posted is returned in NewMsg.

> You should set MsgTrpPtr:=nil unless you want to select a specific transport system, the availability of which you have verified. If you are replying to a result for a message which you have received via NetWork, and the ReplyTo address is the same address as the source, you can assume the transport system given in the message.

```
function ForwardMsg (Msg : MsgPtr; ForwardTo : longint) : OSErr;
```

ForwardMsg forwards a message to the same or another process on the same machine. Don´t call DestroyMsg for a message you forwarded except if you forward to yourself. ForwardMsg may be called after a GetMsg, but all buffer references will be removed

```
function DestroyMsg (Msg : MsgPtr):OsErr;
```

Given a pointer to an existing message, DestroyMsg signals that this message is not used by the application any more and the communication system is free to releases the data structures associated with the message. This possibly kills a transfer. DestroyMsg does not affect the information buffers associated with the message.

Transport System Interface

NetWork Processor allows additional transport systems to be added. NetWork has three built-in transport systems “Dispatcher”, “Local”, and “AppleTalk”. The “Dispatcher” is special: it is used to select one of the other transport systems to send a message on, depending upon the destination address.

Transport systems may be implemented as code resources of type 'NetT' (id range 0..3), which are copied into NetWork Processor and loaded at system start-up, or they can be implemented as applications, drivers, or any other piece of code and register themselves with the NetWork Processor.

For more information about expanding the NetWork transport system, see the separate documentation[8].

[8]J. Lindenberg (1990) NetWork Transport Systems

How to Implement a NetWork Program

Prerequisites

To use NetWork, you have to be able to

- define sub-tasks (“slices”) to be assigned to co-workers for asynchronous solution
- integrate information from solutions of sub-tasks as they are available.

A recommended implementation strategy is to start with a partitionable (single host) solution of the problem at hand. A first step is to separate task definition, task execution, and integration of results. A second step is to verify correct handling of memory (buffers, variables), as this will be critical in a distributed implementation. A final step is the distributed version, where the shared memory situation encountered in a single host is replaced by a message based system.

The definition of sub tasks will be implemented in a task generator. Handling of tasks (or results) will be implemented in task handlers.

How to Implement a NetWork Program: Cookbook

> Remember that if you want to use a new signature, you have to register it with Apple. To experiment with NetWork, you can use the experimental signature 'NetE' (this spelling). This signature has been registered with Apple by the NetWork project and is reserved for experimental use.

To start NetWork, you have to generate a scheduler by calling New(NetWorkScheduler) and to activate it by calling NetWorkScheduler.Init. You have to give the scheduler a chance regularly by calling NetWorkScheduler.PeriodicTask. The scheduler can be set back to inactive state by calling NetWorkScheduler.Reset. The scheduler is released by calling NetWorkScheduler.Free.

> If your program is event driven, you should place a call to NetWorkScheduler.PeriodicTask in your main event loop to be called on null events.

> If you have activated or used the scheduler, you should always call NetWorkScheduler.free before leaving your program.

If you are going to generate sub-tasks, you have to override the task generator. Take the prototype definition *tTaskGenerator* and adapt it to your needs. Create a task generator object and install it by calling NetWorkScheduler.initTaskGenerator. To customize a task generator, you have to write a function NewTask. NewTask should return nil if no sub-task can be defined, or a message pointer defining a new sub-task. The proper task definition is private to you. The scheduler's task sending activity can be controlled by NetWorkScheduler.SetSending.

Programming for NetWork in general will consist of writing a master process (the later compute client) and a compute server. The compute server has to be distributed to the co-workers[9]. To guarantee a fail-safe behaviour, all functions should be implemented on the original generating machine.

These functions must be implemented in the master program (compute client). The compute server must be able to accept sub-tasks, and handle them. Although it is possible to use the message handling system of NetWork directly, it is recommended to make use of the supplied scheduler model.

If you are going to accept sub-tasks, you have to customize the receive handler. Take the prototype definition *tTaskHandler* and adapt it to your needs. Create a task handler object and install it by calling NetWorkScheduler.InitTaskHandler. To customize a task handler, you have to write a function MsgUsable and a procedure MsgEvaluation. The scheduler will get the priority information of an incoming message to the PriorityBuffer indicated by MsgPrioPtr. MsgUsable should check any incoming task on the basis of the header information and the available priority information. If MsgUsable returns true, the scheduler will ask the message system to pass the bulk of the data describing the sub-task to the core buffer indicated by MsgCorePtr. You have to write a procedure MsgEvaluation which should take the data from the buffer and initiate the proper task execution. To return a result to the sender, you can make use of the ReplyMessage function.

With NetWork, programs can be launched automatically on remote request. Programs launched on remote request may be terminated by NetWork when the owner accesses the machine. Do not assume that it is safe to continue processing at that time if you receive a command-Q. You must clean up as soon as possible or you will not have another chance. Also note that you don´t have the time to report results, because all messages – including those that are about to be transferred – are killed when your application dies. Remember that NetWork' s priority is with the owner, not with the remotely launched application.

If it is necessary that you clean up, set your process type to "master" after program initialization, and call the function “Idle” regularly. However, users may become annoyed by having an alien application around, and your application is likely to be cancelled from the list of welcome visitors.

Chapters "NetWork Scheduler Routines" and ""NetWork Communication Routines" can be consulted for more information.

[9]To avoid virus proliferation, worms and other things, NetWork does not do any active transportation of code. The code to be launched has to reside on the destination machine and is under control of the destination home user.

Tips

Conceptually, NetWork distinguishes task definition, the proper handling of tasks, and collection of results. However, to allow for a fail-safe implementation it is necessary to provide all three functions on the machine of the initiating user, or to guarantee a minimal availability of the network.

The instances involved in NetWork, the processes, communicate by message passing. The absolute priority of owners implies a list of NoNos for NetWork communication:

- Do not assume that computing resources other than your home resources are available at all.
- Do not assume that a computing resource once accessed is available until completion of its task.
- Do not assume that a communication resource known is stable or available.

You could overcome these restrictions by verifying that the conditions you suppose are continuously met at the cost of putting additional load on the network. This however would obstruct other users. Hence additionally:

- Do not access the network unnecessarily if not forced to by a user.

You should not assume that every sub-task which is assigned is effectively carried out. If sub-tasks are being solved, you should not assume that the results are presented in the order the tasks are assigned. A positive way of thinking about this environment is: assume that good faith is presenting you with results to partial problems, free of charge, but at random. You should be able to make use of these gifts. On the other hand, you can ask for the solution of a subproblem. But this is only a request, not a guarantee of faith.

Tricks and Recipes, Hints and Warnings

Selection of Co-Workers

The NetWork communication system supports selection of co-workers by capabilities. See **capabilities.**

to come •

Authorization

The reply-to address can be used to implement a simple authorization scheme. Note that the reply-to address need not be a public address.

Caveats and Warnings

> On exit NetWork Lookup will try to re-install the traps it has patched. However, interference with other patchers (e.g. MacApp) cannot be excluded..

NetWork Processes and Multifinder, SIZE Resource

Each NetWork Process should be capable of operating in the background. Therefore the “can-background” bit in the SIZE resource should be set.

A slave application without the “can background” bit set may be launched in the background without ever receiving any time slice, therefore never doing anything useful. Still worse, it is possible that the process is not terminated automatically, and the user must call it to the front before it is exited.

Another reason to require background capability is: the NetWork Processor depends on SystemTask being called regularly, in order to keep message transfers alive, among other things. The only way to ensure that SystemTask is called regularly – assuming Multifinder – is to call WaitNextEvent or GetNextEvent periodically. But as a consequence it is always possible that your application is switched to the background, and if the application is not able to operate in the background, time-outs are very likely to occur and the application will lose messages.

A NetWork process can be written to operate both as a master and a slave. If you are separating your applications into two separate images for master and slave processes, the recommended implementation of a NetWork Process is a faceless background task with the “background-only” bit set in the SIZE resource.

Note that implementing a "slave only, faceless background task" maximizes the likelihood of taking advantage of available machines for of two reasons: It allows you to minimize memory requirements, and NetWork Processor will be able to launch your application, even if the user did not check “Background (invisible) processes” AND a modal dialog is frontmost. A faceless background task can be launched at almost any time.

If you are implementing master and slave modes within the same application, you can use the Master function to differentiate between the two modes. Master returns true if the user launched the application and false if NetWork Processor launched the application. You can tell whether you should display a user interface or windows showing progress by calling the function Visible. Visible is always true for a Master process and always false for “background-only” processes. For the remaining case, a slave process without the “background-only” bit set, it returns the setting of the "background only" check box in the control panel.

NetWork Processes and File Creators/Signatures

You should register the signatures you are going to use with Apple. The signature, or file creators, are used to associate files with icons. But beyond this, they are used to identify applications to be launched on an open/print request. NetWork uses the signature of an application to associate a message to a program file if there is no running application listening for messages corresponding to the given creator. If there is already an application running with a specific signature, NetWork Processor will never launch another application for that signature even if there are multiple applications with the same signature in the various folders used by NetWork Processor. An attempt of any application to register its signatures with NetWork Processor will fail if there is already an application registered for that signature.

Moreover, the user cannot register multiple copies of the same program using NetWork for the same signature, even if the program is launched manually. The attempt to register the signature another time will fail. If you want the user to be able to launch multiple copies of your program, then you´ll have to use a process type of **dynamic**, i.e. ask for a dynamically assigned creator.

> The current NetWork implementation does not allow you to implement a slave and a dynamic process identification using the same running process, except if you are sure that you are not communicating to local processes of that family.

> The library does not support multiple signatures for one process, but NetWork Processor does. If you want to be able to use multiple signatures within one application, you have to add calls to NetWork Processor yourself to register the additional signatures with NetWork Processor. Of course it is possible to use dynamic signatures as the additional signature. You´ll have to modify the library routines in order to allow you to specify which signatures to use when calling PostMsg.

NetWork Processor identifies processes internally using the signature. This may lead to inconsistent results if you are trying to register a dynamic signature from within a slave or local process. If you want to do this, be sure to change the process type associated with the dynamic signature to a slave immediately after launching.

NetWork Processes and Stack Requirements

NetWork Processor requires some stack space – especially if logging is turned on. The maximum that has been observed is about 2.5KB. Generally this does not present a problem to an application, except if you are using NetWork from within nested procedures using large arrays or within recursive procedures. If you are implementing a faceless background task however, be warned.

By default, Multifinder allocates a stack of 8 KB – or 24 KB if Color Quickdraw is present – to all applications launched. Faceless background tasks however are assigned a default stack of just 2 KB, which may be too small to call NetWork. Therefore you must include statements in your program to adjust the stack size prior to calling MaxApplZone. The same is of course true if you need more stack space for other reasons. In the case of a faceless background task using NetWork, however, you cannot get by without it.

The following procedure illustrates how to test if there is sufficient stack space and increase the size of the stack if necessary. This procedure should be called BEFORE any other statement in your program, in particular before calling MaxApplZone.

```
procedure InitStack;
const MinStackSize = 4096;
type LongPtr = ^ longint;
begin
      if LongPtr (CurStackBase)^ - LongInt (GetApplLimit) < MinStackSize then
            SetApplLimit (Ptr (LongPtr (CurStackBase)^ - MinStackSize ))
end;
```

Hints: NetWork Processes Which are not Applications

When NetWork Processor receives a message which does not correspond to a running process, it will search for an application in the “:NetWork Tools:” folder. Hence only applications are launched as slave processes. However it is possible to implement a master NetWork process as a desk accessory, MPW tool, or HyperCard XCMD, or something similar. It is also possible to use NetWork from an INIT or driver, but only if you register your creator before Multifinder is loaded.

NetWork Processor uses a very simple strategy when cleaning up. Whenever a file is closed,

NetWork Processor tests if its file reference number is that of the current application. If it is, Multifinder is asked for the current process id, and NetWork Processor de-registers all creators known for that process id. Hence if a DA Handler, MPW or HyperCard is exited, the corresponding NetWork process is no longer alive either.

Hints: Computing Contexts and Integrity

NetWork does not assume any sort of session maintenance. Hence the NetWork message system cannot suppose or guarantee any session or context integrity.

> If you need authorizations, you can use NetWork's addressing scheme. You can use the reply address to implement a simple authorization scheme if you use "private" addresses as targets. Only a process which has received a task message will know this address to reply to.

The minimum session maintenance, or context verification which in general will be required, is supported by the scheduler. The context may be different for the receiving or sending branch. The context is defined by:

- a ContextStamp.
- a privileged address
- a privileged timeout

For all, zero signifies a neutral or undefined state.

The ContextStamp is to be compared with the MsgRefernce of the message record. The scheduler routine DoNewTask fills in the TaskGenerator's ContextStamp before sending. If a context continuity is required, the MsgRefernce can be used to verify the context.

A message may signal that it is changing context, or else needs special attention by using
- a cMsgNAttention flag.

The TaskHandler's routine MsgHeaderUsable is responsible for verifying context continuity.

Hints: Expanding the Scheduler: Installing Cohandlers

Object Pascal does not allow overriding methods with varying parameters. As a kludge, various hooks are implemented in the scheduler to allow free experimentation without completely rewriting the scheduler. A user definable object, a cohandler, will be called at every critical step of the scheduler.

The formal definition is:

```
Type tSchedulerCohandler=object(tObject)
                   procedure CoHandle(cmd:tSchedulerPhase;msg:MsgPtr);
                   procedure reset;
                   end;
```

Cmd identifies the phase of the scheduler at which the cohandler is invoked. Msg contains a pointer to the current message (if any). The intended use of cohandlers is to provide adaptive allocation of tasks. A cohandler should feel free to modify the following fields:

```
TaskIterations
                    TaskId
                    Destination
```

when called at pStartNewTask. All other calls to the cohandler should be considered only informative to provide the basis for adaptation.

Examples

Ping: *Just Pings*

Ping documents the basic possibilities of the NetWork system: look-up of remote partners, message transport, and remote launching.

See the Ping source code for more information. Ping is the recommended first example if you want to develop your own software using NetWork.

RemoteJob: *Supports Distributed Use of MPW*

RemoteJob is a simple example which illustrates most possibilities of the NetWork system: going beyond the basic features it uses the scheduler for asynchronous distributed computing.

See the RemoteJob source code for more information. RemoteJob is the recommended starting point if you want to develop your own software using NetWork.

RemoteJob illustrates how to use the NetWork system with pre-NetWork software which at least supports start-up scripts, like MPW. RemotJob takes a task file Remote.Job, sends it to a target machine, stores it there as a file Remote•Job, and then launches the proper target (defaults to MPW). Instead of using a task file, commands can be sent on the fly using the New Task… item in the New… menu.

To use RemoteJob with MPW, the target's NetWork tools folder must contain RemoteJob along with a copy of the MPW shell, and the special start-up file. You would then use an MPW script to store the appropriate job files as Remote.Job, and keep track of successful execution of remote tasks. Note that RemoteJob does not clean up the job files - modify it if you want this.

To use RemoteJob for a distributed make, you can either modify RemoteJob to keep track of sub-tasks (thus introducing session oriented techniques), or you can run make repeatedly until it tells you that there are no more tasks to assign (thus following the ideas of asynchronous iterations).

see the "RemoteJob" example for details and implementation.

> **Warning:** Never put MPW or any shell into the NetWork tools folder unless you are in a completely trustworthy environment. If you allow a shell to be launched remotely, the owner has no control over the type of shell commands which may be invoked. In particular, if you allow remote launching of a shell, there is no safeguard against worms..

Spinning Brain: ***a Neural Net Using Asynchronous Iterations***

Spinning Brain is a full example of asynchronous iterations using the NetWork system. Spinning Brain uses the cohandler facilities of the scheduler for an adaptive version of the scheduler which adjusts itself to the effective power and reliability of possible partners.

see separate documentation and source code for “Spinning Brain”

ScreenSaver: ***Idle Time Launching***

If a folder “NetWork Idle Tools” is in the system folder, applications residing in “NetWork Idle Tools” will be launched automatically on transition to idle state. ScreenSaver is an example which uses this possibility. It **is** an ordinary application.

You can launch **any** application on idle time (for example MPW). Collect the things to do later in “NetWork Idle Tools”.

see the “ScreenSaver” sample

Hello, UDPTransport: ***Startup Launching***

If a folder “NetWork Startup Tools” is in the system folder, applications residing in “NetWork Startup Tools” will be launched automatically at start-up time. Hello is an example which might be used this way.

So far, it is a boring example. The start-up launching facility is necessary if you want to install additional transport systems (like TCP/IP).

see the “UDPTransport” sample

see the “Hello” sample

Mailer

to come •

Index