The NetWork Project: Asynchronous Distributed Computing on Personal Workstations

Günther Sawitzki
Institut für Angewandte Mathematik
Im Neuenheimer Feld 294
D 6900 Heidelberg

Abstract

NetWork is an experiment in distributed computing. The idea is to make use of idle time on personal workstations while retaining their advantages of immediate and guarantied availability. NetWork wants to make use of otherwise idle resources only. The performance criterion of NetWork is the net work done per unit time - not computing time or other measures of resource utilization. The NetWork model provides corresponding programming primitives for distributed computing. An implementation of a distributed asynchronous neural net serves as test application.

Computing in an Asynchronous Distributed Environment....1
An Example: Neural Nets....2
Design Goals....4
Principles of Operation and NetWork Layers....5
NetWork Scheduling Strategy....7
Components of the NetWork Model....8

Idle Time Distribution and Economy of Recruitments....9
NetWork Communications: Economy and Flexibility....11
Implementation Environment and Experiences with NetWork....13
Looking Ahead....14
Literature....15
How to Access NetWork....16
Biographical Note & Additional Information....16

The recent development of workstations gives the user considerable computing power for immediate access. On the other hand, when the user does not access it, considerable computing power is left unused. There is a general desire to make use of this aggregate computing power, but one does not want to lose the advantages of a personal workstation, its immediate and guarantied availability. The NetWork project provides a general purpose model which tries to match both of the following aims: sharing computing resources and respecting the absolute priority of the resource owner.

The domain we address is that of *personal workstations*. We are not addressing those team installations where the installed computing power per working place exceeds what is required by the tasks to be achieved. In these installations, the UNIX *nice* mechanism may be sufficient support. We are addressing the domain of dedicated personal workstations. Using UNIX on a personal workstation eventually you grant access to other users, but the general experience is that sometimes the only solution to yet another nice process is to kill all of them. We want to keep the individual availability of each station.

The approach we take is to allow other users to borrow the computing power if a machine is idle, but to impose a strict rule: if the owner accesses the machine, the guest is given only minimal time to retreat. For example, if the owner touches the machine (if there is any owner action on the machine), the machine has to be completely available to the owner without any noticeable delay. This imposes a ‘time to leave’ of the order of 1/10th of a second - a time which might be too short for any proper notification or clean-up.

As a consequence, if you make use of an idle workstation in a network and still want to respect the absolute priority of the owner, you can hope for an advantage, but you cannot rely on receiving any results. And you cannot rely on receiving results when you expect them: computing will take place in a distributed asynchronous environment with random availability of remote resources. The NetWork project gives a minimal communication and management model to operate in this environment for experimenting with distributed computing.

Computing in an Asynchronous Distributed Environment

We want to use free computing power, while respecting the absolute priority of the owner. Hence we cannot assume a guarantied environment. This affects possible applications in various ways. There are tasks which always benefit from additional computing power, in particular those working on large data sets. Sorting with some appropriate merge/sort algorithm gives a class of examples: the global sort can take advantage if a subset is already sorted by another machine, but need not be affected if the result of the pre-sorting is not available. The same applies to searching. All major accounting tasks will give a class. Any statistical analysis based on exponential families (like normal/ gaussian distributions) gives another class of examples: in these analysis you can calculate global sufficient statistics from those of partial data sets, if available. We will call the problems of this type *completely splittable*.

The class of problems we are interested in are iterative and recursive problems which have a stronger internal structure. For these, it is not clear, a priori, that they can take advantage of additional computing power. Moreover, it is unclear how to take advantage if the completion of a task is not guarantied. To

have a formal example, take a mapping $F:\mathbf{R}^N \to \mathbf{R}^N$ to be iterated. By the restriction of F to a subset $S \subset \{1,...,N\}$ we mean the mapping $\tilde{F}:\mathbf{R}^N \to \mathbf{R}^N$ with

$$\tilde{F}(X)_i = \begin{cases} X_i & i \notin S \\ F(X)_i & i \in S \end{cases}$$

So $\tilde{F}$ has the full input F has, but only operates on the coordinates defined by S. The idea of distributed iterations is to have restrictions to different subsets $S_1,..., S_p$ allocated to different machines for a number of iterations. The (restricted) iterations are performed in parallel. The results are collected as they come in and new tasks are redistributed again repeatedly. In simple cases, an incoming result $\tilde{X}(t)$ at time t replaces the most recent state X(t–1) of the collecting system and the new task will be to calculate a number of (restricted) iterations based on $X=\tilde{X}(t)$. In more general cases, an updating function c will be used to define an updated state $X(t)=c(X(t–1), \tilde{X}(t))$ based on the achieved state X(t–1) and new information extracted from $\tilde{X}(t)$.

The behaviour of the distributed system is not clear even in a guarantied computing environment. The outcome of iterations in one part of the problem might critically depend on results from iterations in other parts. Moreover, in an asynchronous environment, the result of a previous iteration may or may not be available for the next round: even if the iteration of F converges nicely, $F^n(x) \to x_o$ for some x_o as $n \to \infty$, the limiting behaviour of the asynchronous distributed system is not clear a priori.

There are iterative problems which still can take advantage of a distributed environment, even if the environment has no guarantied performance. Baudet (1978) studies a special class of this kind, that of iterations of Lipschitz contractions. For a Lipschitz contraction, any asynchronous iterate will converge to a limit (in general the same as original) under asynchronous iteration. Many numerical methods can be formulated in a way which makes them fall into the class covered by iterates of Lipschitz contractions (see Bertsekas and Tsitsiklis 1989, part 2). Studying asynchronous iterations in a non-guarantied (random) environment was suggested by the work of Eddy and Schervish (1988).

An Example: Neural Nets

A lecture by W.F. Eddy on the work of Eddy and Schervish on asynchronous iterations was one starting point for the current project. Another root was provided by a joint work of Kühn and Sawitzki (1989) on neural nets. We use an example from this work, a neural net applied to picture reconstruction, to illustrate asynchronous iterations. The specific variant of neural nets we are using is a Hopfield net (see Kühn and Sawitzki (1989), or Arbib (1987) Ch.5).

For our simple demonstration example, the state X of this system corresponds to a picture which is being processed, $X \in \{-1,+1\}^N$, where N is the number of pixels in the picture (the number of neurons in our net). The dynamics of this model can be seen as iterations: In a classical environment, a transformation F is iterated, starting from an initial picture, until a stable state is reached. In a distributed environment,

we take a slice S, represented by a subset of the index set (1,...,N), and ask an idle workstation to perform a number of transformations on this. The restriction to S means that only pixels in S may be changed, although the full picture is available as initial information. While S is being processed on one station, we

are going to pass other slices as sub-tasks to other workstations. When we get a result, we will merge the processed slice with the rest of the picture; i.e. our updating function uses the processed slice to replace the corresponding part of our original picture. This may introduce an error because the processed slice may depend on the state in other slices which may have changed significantly in the meantime. We repeat the assignment of tasks until we reach a stable state. This example is not covered by the convergence result of Baudet (1978). However, under mild regularity conditions, convergence to the original limit still holds.

Neural nets are an interesting target for asynchronous distributed computing: if we accept that neural nets provide a useful model for cognitive functions, we still must admit that in real biological systems there is no indication of global synchronization except on a very large scale (e.g. daily rhythm). Information processing takes place in a distributed asynchronous environment. And we must admit that this is not a guarantied environment - some results may be late or may never be reached. This is true for the individual, and this will be even more important for collective, or social cognitive phenomena. So experiences with neural nets in our environment might shed a light on critical aspects of neural network modelling.

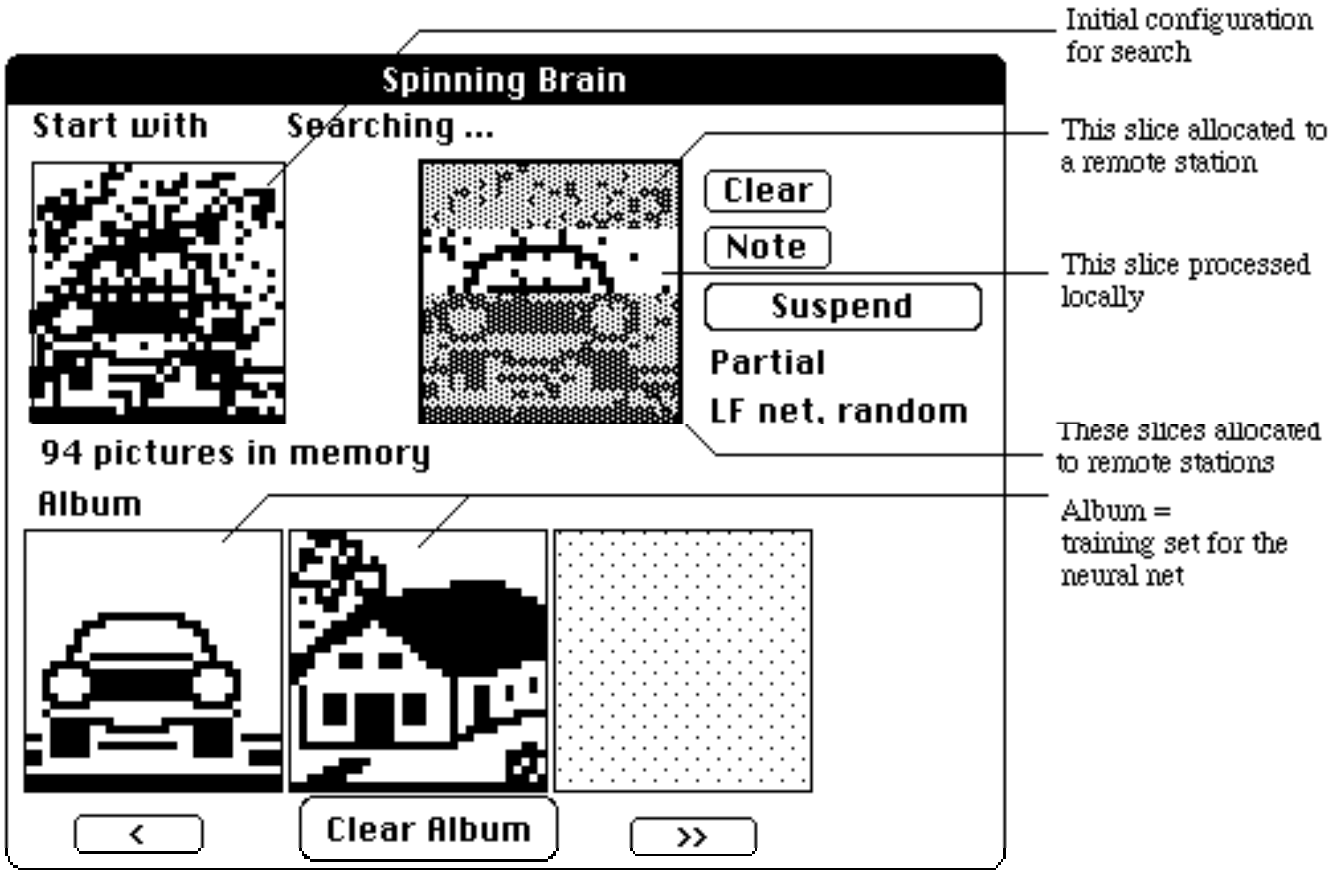

Figure 1: Screen dump from "Spinning Brain", a neural net used for picture reconstruction. The neural net was trained on a series of pictures, two of them visible in the bottom row. Starting from the initial picture (top left), the neural net reconstructs the original. The current state of the system is shown in the top right frame. Formally, the state space is $\{-1,+1\}^N$, N=32*32.The iteration on the local machine is restricted to one slice of the picture. The slices shaded in gray are allocated to other machines.

Design Goals

The goal of the NetWork project is to make use of the free resources of a network to provide a better net outcome. If the resources would be otherwise unused, or if the resources are free, measuring the resource consumption is a needless effort. What counts is the *net work* done, as measured in tasks per wall clock time. This is the performance criterion. The model implementation runs in an unobtrusive way, making use of *free* network resources, but interfering as little as possible with any user request.

The central idea of NetWork is that every **machine** has an **owner**. The owner is the source of events which have absolute priority on the corresponding machine. If the owner touches or accesses his/her machine, the machine has to be completely available without any noticeable delay.

An 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. Moreover, a NetWork machine in general will, but need not, correspond to a physical machine. For example, a cluster of CPUs may be considered a machine for the purposes of NetWork.

Even if there is no immediate owner access, a machine may be busy because an owner initiated process needs the resources of the machine. The absolute priority of the owner must extend to owner initiated processes as well. A machine is considered **idle**, or free for the purposes of NetWork, if there is no owner access and no owner initiated activity. NetWork is only allowed to take resources which are free in this sense.

The goal to run in an unobtrusive way, making use only of **free** network resources, also affects communication. The effect for any “owner” other than the one requesting network services should be barely noticeable, and care must be taken not to compete for network bandwidth. Unfortunately with current technology it is nearly impossible to avoid interfering with

other users. All that can be done reasonably is taking measures to minimize the number of network accesses and the additional network load.

To allow for open environments, independence of the underlying communication model and adaptability to heterogeneous hardware are additional design goals of NetWork.

- **immediate availability of any machine for its owner** (e.g. guarantied availability of any machine on any local request within 1/10th of a second)
- **minimal interference with "owner communication"** (i.e. "second class" communication where possible,…)
- **independence of communication model** (including network/file/bus based communication; network topology;…)
- **adaptability to heterogeneous hardware.**

Table 1: Network design goals

Finally, to invite experiments with our model, the implementation of an asynchronous iteration scheme should be as near to that of a (standard) iteration scheme as possible.

In the next section, we present an outline of the current model implementation for NetWork and its principles of operation. Special strategies are needed to cope with a non guarantied environment to cope for asynchronicity of results, and to random availability of partners. These strategies are discussed in the following sections. Then we will discuss the low level components and services necessary to meet the NetWork design goals. Measures to economize communication and to allow for flexibility of communication technology are discussed next. We conclude by a discussion of the current NetWork implementation environment and experiences with NetWork.

Principles of Operation and NetWork Layers

NetWork views the computing environment as a set of **machines** with **processes** running on these machines. Each machine has an **owner** who has absolute priority on this machine. Processes may be running on behalf of the (local) owner or they may satisfy a remote request. If a process is running on behalf of a remote request it should be terminated immediately when the owner accesses the machine.

A process handles **tasks** and eventually it may generate tasks for remote execution. A task may be delegated to another process, possibly on a different machine, and results may, or may not, be returned.

The NetWork programming model has three layers. The top layer, the application layer, contains the application specific code. Apart from initialization and clean-up sections this code should be able to define sub-tasks, and to handle results from sub-tasks if available. The specific details of this layer are - of course - application dependent.

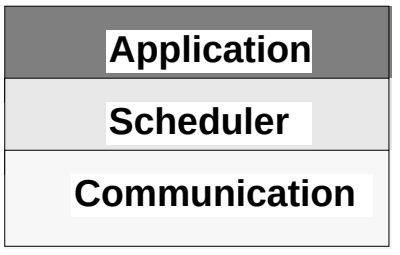

Figure 2: **NetWork layers.** The scheduler layer contains support for dynamic load balancing and adaptive scheduling. The communication layer has to provide transport shielding and communication in a non-reliable environment.

The scheduler layer provides support for asynchronous iterations. The NetWork scheduler monitors and stimulates the generation, assignment, and integration of sub-tasks. While the proper generation of sub-tasks is application dependent, the NetWork scheduler can monitor the overall system behaviour and try for dynamic load balancing. Task assignment is an interaction between scheduler and application.

The communication layer forms the basis of the NetWork design. It has to provide the basic communication services needed for the network system. In particular, it has to cope with a non-reliable environment. If necessary (for example to implement diagnostic or management tools) the services of the communication system may be accessed directly, avoiding the scheduler.

NetWork is implemented as a message passing system. A process may send task descriptions as messages and results are returned as messages. If a process is set up for task generation, the scheduler will ask the application periodically for the definition of a new task. If a new task definition is given, the scheduler will pass this information to the communication system for further transmission. If a process is set up for result handling, the scheduler will inform the application of any result received by the communication system.

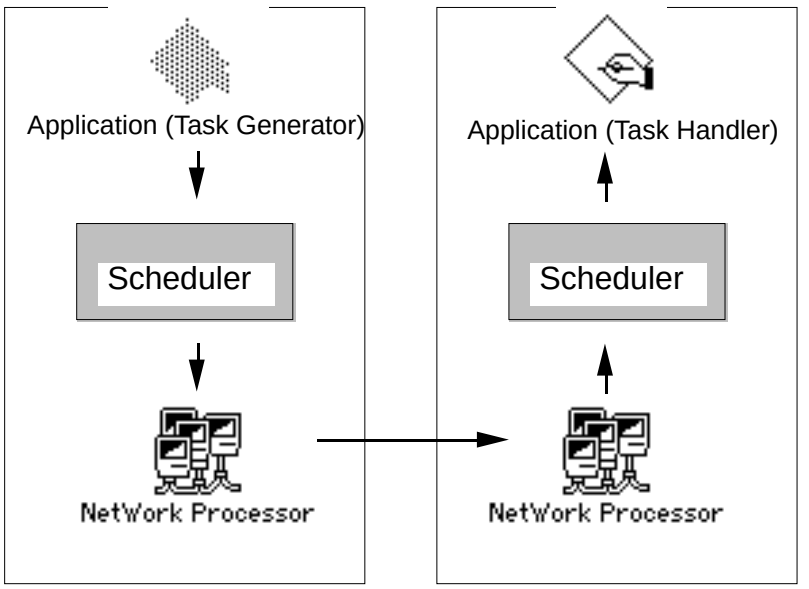

Figure 3: NetWork message flow: a simplified picture. The task generating application program defines a task message and hands it to the scheduler. The scheduler does the necessary housekeeping and passes the message to the NetWork Processor which communicates it to the receiving NetWork Processor. The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

Since NetWork is designed to work in a non guarantied environment, no assumptions about the life time of a communication partner should be made. Hence a process which is generating tasks does not have knowledge where to delegate a task to. The scheduler will make a proposal where to delegate the next task to when asking for a new task definition. The application is free to accept this proposal, or to select a different target using a look-up server or any other source of information.

Messages are addressed to processes, residing on machines. However, in a non guarantied environment, no assumption on the existence of a communication partner can be made. The address refers to a process class (defined as any instantiation of the underlying program) rather than to a particular process instant. On the recipient machine, NetWork checks whether the target is active, i.e. if there is a corresponding process. If so, the message is made available. If the machine is idle but no corresponding process is active, NetWork tries to locate the program and launch it first. If it fails, the message is discarded. There is no prolonged negotiation and no acknowledgement. The task message is an implicit launch command, and the completed result is the only acknowledgement, if any. If the state of a machine changes from idle to used, that is if the "owner" accesses the machine, NetWork will kill immediately any application it has launched.

NetWork Scheduling Strategy

A scheduler for NetWork may be integrated in applications and makes use of the services of the NetWork system. In the current NetWork implementation, a scheduler prototype is provided, together with a library which interfaces with the NetWork communication system. The scheduler will ask the proper application code regularly whether a new task should be defined, or informs about incoming messages. It also does a preliminary check for the usefulness of incoming messages, filtering out messages which can be identified as useless or outdated with respect to the application context.

To guarantee a fail-safe behaviour, tasks should be allocated redundantly. As a consequence, more than one result may be returned relating to a sub-task. This poses a problem to the scheduler. Assume we have some effective time scale (some measure of effective iterations done, for example). Assume we have two incoming partial results Y, Y', where Y is based on information available at effective time T, with K iterations done on Y, and Y' based on T' with K' iterations. Let Y arrive at time t, Y' at time t'>t. Should we replace the results of Y by those of Y' ? There are trivial cases: If $T' \leq T$ and $K' \leq K$, then Y' is clearly outdated . Else if $T' \geq T$ and $K' \geq K$ and not both equalities hold, then Y' is better than Y, so Y should be replaced. For the remaining cases, a decision must be taken.

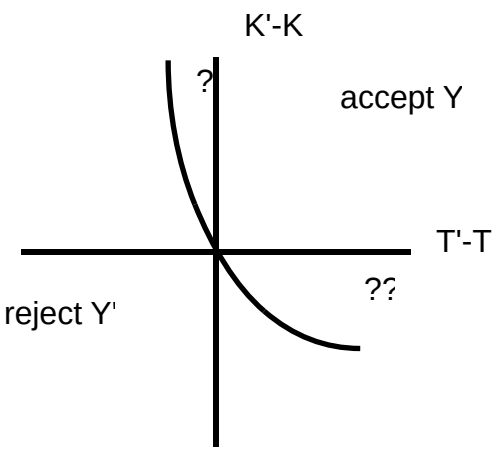

Figure 4: Critical decision: Limit of the acceptance region for conflicting results. Results based on better initial information (K'–K>0) and with better iteration count (T'–T>0) can be accepted a priori. Results based on poorer initial information (K'–K<0) and with fewer iteration counts (T'–T<0) can be rejected a priori.

Following a suggestion from W. Rheinboldt we adopted the strategy to only accept those packages which can be accepted a priori (see Figure 4). Instead of putting computational power into the evaluation of the optimal acceptance decision, we try to keep the probability of entering the critical region low by adapting our task allocation scheme. Since our criterion is the wall clock time to perform the task, and both acceptance decision and task allocation will be done by the same machine, there is a trade off between those two, and we can keep the expected loss due to a wrong decision small by keeping the probability of conflicts low.

The NetWork scheduler prototype uses an adaptive task assignment scheme to minimize the probability of these conflicts. An application can override or augment the generic strategy as provided by the scheduler with a more application specific strategy.

Components of the NetWork Model

To meet the design goals, NetWork needs certain services.

- **idle/busy state monitoring** to keep track of owner activity
- **process management** to launch a process to serve a remote request and to kill all processes launched by NetWork when the owner accesses the machine
- **communication** to pass message descriptions and results

NetWork needs an **idle monitor**. The only task of the idle monitor is to monitor whether the state of the machine is idle or whether the machine is active on behalf of its owner. Since this is machine specific information, each machine must be equipped with an idle monitor.

Second, NetWork needs a **process manager** which is capable of handling all process management on remote request. If the machine is idle, the process manager may launch processes to fulfil remote computing request, and it has the task to clean up all remote processes immediately if the state of the machine changes from idle to busy, that is if the owner accesses the machine. The process manager is informed of any idle/busy transition by the idle monitor. It is responsible for guarding the priority of the "owner". The process manager keeps track of active processes on the local machine.

Third, NetWork needs a **communciation system**. The communication system has to guarantee reliable services in a possibly unreliable environment. Moreover, it should take special precautions to minimize interference with “owner communication”, as required by the NetWork design goals.

Idle monitor, process manager and communication system form the core of the NetWork system. They must be present in any implementation of NetWork. This core provides convenient primitives for distributed

computing while shielding the transport system. In this respect it resembles other approaches (Gardner et al. 1986, Bernard et al. 1989). Going beyond these approaches, NetWork tries to provide a minimal model suited even for a non-guarantied environment.

A process requiring remote services will pass a task description to the (local) communication system. The communication system will pass the task description as a message to the communication system of the recipient machine. The recipient communication system will ask its process manager to find an appropriate process to handle the message. If it is found, the message is delivered. If the process is not found but the machine is idle, the recipient process manager will try to launch a corresponding process ("launch on task") and if the launch is successful the message is passed on.

If the owner accesses the machine, the idle monitor will give a signal to the process manager, and the process manager will kill any guest processes it has launched so far immediately.

If a (remote) process has completed a task it may return a result, or generate a subsequent task as appropriate.

NetWork does not assume any session maintenance. If the owner has absolute priority, session maintenance over the net is of little use. A process handling a remote task may be killed instantaneously at any time because the owner accesses the machine. Hence session maintenance would give little if any information about the chance of successful termination of a task. Moreover session maintenance is prone to produce additional communication load. Since it is not necessary for distributed computing, session maintenance is not required for NetWork.

However NetWork does not exclude session maintenance. The NetWork system can be extended to include session maintenance or acknowledgement schemes if required. A useful combination could be to use NetWork's message passing system to establish the first contact with a remote co-worker (launching the co-worker if necessary), with a session oriented protocol being used after that.

NetWork does not assume that a communication partner exists - in a non guarantied environment, no assumption on the existence of a communication partner should be made. NetWork must be capable of remote launching. Since a specific launch command would add to the communication load, NetWork provides a "launch on task" facility as described above.

There are situations where the "launch on task" feature might not be useful. For instance, if NetWork is used in a master-slave setting, a certain slave may be very late with its results. If the master has used a redundant task assignment, the whole job may already be completed and the master may have terminated. The messages support a "don't launch" flag, which is honoured by NetWork. These messages will only be delivered it the recipient does already exist. A recipient will not be launched automatically if this bit is set.

Look-up is not listed among the required services. With the lack of a look-up system, NetWork has only two initial possibilities: it can use fixed

addresses, or it can use random addresses. Both are useful, and the communication system has to provide at least one of these possibilities. A special case of fixed addresses it the use of broadcast addresses to ask for possible partners. All well known look-up strategies either imply the use of tables (hence fixed addresses) or use an implicit broadcast. So look-up is not restricted to broadcast mechanisms in NetWork.

NetWork provides look-up facilities. But there may be application specific information which would allow for better look-up strategies than could be provided by a generic system. To allow for more efficient strategies, look-up has been moved to the application level. In particular, this allows using of look-up servers which are implemented as separate programs and may be shared between several applications.

The services required by NetWork could be provided by the operating system. In general for the current state of art however NetWork has to augment the host operating system to provide these services.

Idle Time Distribution and Economy of Recruitments

We have to identify idle machines and must have a strategy how to allocate them for cooperation. The idle state is determined by the Idle Monitor, and idle machines can be registered as possible compute servers using a look-up server. Of course we would prefer using those machines which will be available for some time. We would like to avoid those machines which are free for the moment, but will be used shortly. To do this, we would need some method to tell promising machines from others.

Our first informal review of literature, and interviews with experts in that field, gave little hope. The general idea we met was that usable idle time would be controlled by a Poisson process. So the idle time would have an exponential distribution. But since an exponential distribution is memoryless, there would be no chance for optimizations based on waiting times. Disregarding any recommendations, we implemented an allocation scheme based on observed idle times, and then measured the availability. A sample plot is given in figure 5. If the idle time distribution in fact would be near to exponential, this plot should exhibit a line. Clearly this is not the case. Statistical analysis shows that the distribution is more adequately approximated by a Weibull distribution (figure 6). Whereas the exponential distribution is memoryless, the Weibull distribution within the parameter range indicated by our measurements has a decreasing hazard rate. This implies that the frequency of useless (short time) allocation of machines can be drastically reduced by waiting until a certain critical idle time has been exceeded. This is the approach we take in the NetWork implementation.

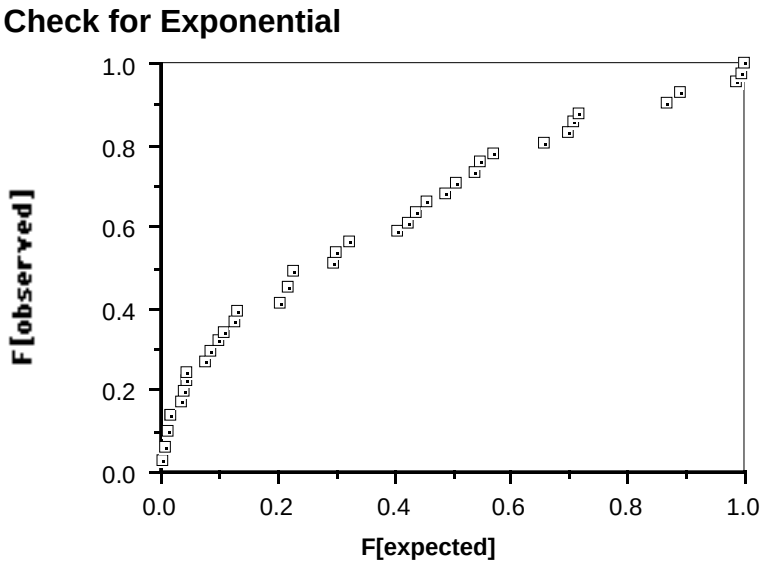

Figure 5: Diagnostic plot for exponential distribution of available idle time. Observed distribution function versus expected. If the time of availability would follow an exponential distribution, this plot would show approximately a straight line.

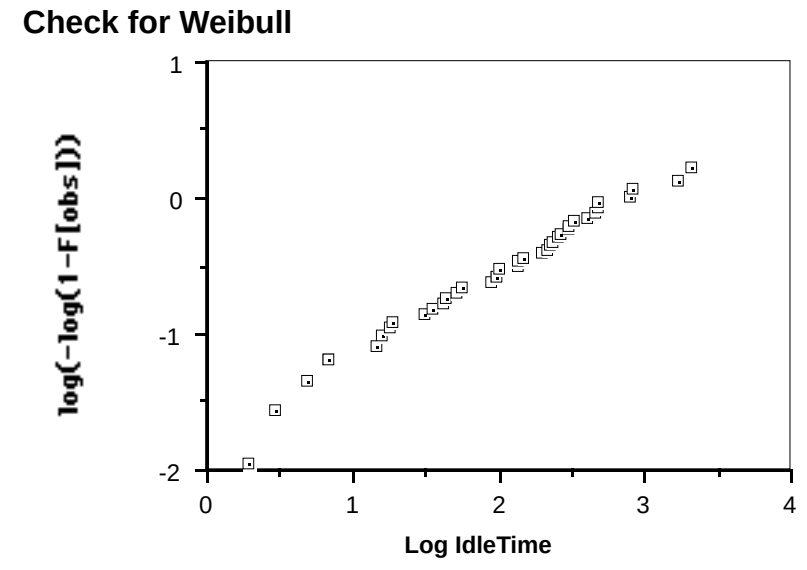

Figure 6: Diagnostic plot for Weibull distribution of available idle time. This plot shows an approximate linear behaviour, which is an indication for a Weibull distribution.

NetWork Communications: Economy and Flexibility

As stated above, the NetWork model has to minimize communication load to avoid competing with "real" users. We already mentioned that NetWork allows a process to be launched implicitly by sending a task addressed to it, and that NetWork avoids negotiations and explicit launch sequences. This is done to reduce additional communication load. Of course it is possible to use explicit authentication and authorization schemes and direct control over launching with NetWork, and in any environment where security is required this will be necessary. But it is in no way required for a minimal implementation of distributed computing, so it is not required in the NetWork model.

The decision not to enforce any session maintenance techniques, nor even any acknowledgement schemes, is another measure to minimize communication load. NetWork can operate in a connectionless mode, so session maintenance techniques or acknowledgement schemes are not required. Again, if needed, both can be applied of course.

Since NetWork is designed to work in a noisy environment where no guaranties for availability or performance are given, NetWork has to be

prepared for messages which are outdated or out of context. To minimize communication load in these cases, NetWork encourages a separation of descriptive information from bulk load. Conceptually, each NetWork message consists of a priority part, which should be small and contain just sufficient information to decide whether the message is usable in a given context, and the message core which should contain the bulk of information. When a message arrives, the priority part along with the usual administrative information is presented to the recipient for inspection. Only if the recipient accepts the message as usable, the bulk information needs to be transported. The separation in priority information and message core is only a conceptual one.

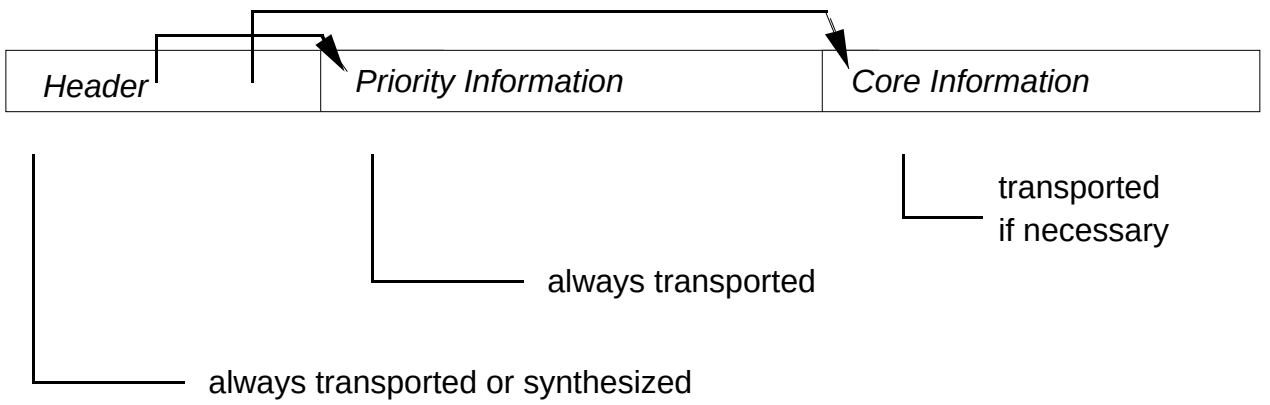

Figure 7: Separation into first/second class information. The core information need only be transported if requested.

The Communication Manager may optimize for a transport system and will do packing/unpacking and transport in a transparent way as seems optimal for the transport system. In particular for a packet oriented transport system, the Communication Manager will pack header and priority information into a first transport system package, and fill it up with as much core information as fits reasonably into this package. Subsequent packages with the remainder of the core information will only be sent if the recipient requires this information. Thus unnecessary information load can be avoided.

NetWork does not assume a master-slave situation. Freedom of topology is achieved by using a triplet of addresses in the message header. Each message has a source, indicating the process or program from which the message originated, a message target indicating the process to which the message is to be delivered, and a destination (like a reply-to address) to which a possibly resulting message is to be sent to. This allows easy implementation of hierarchical compute servers, forwarders, or genetic computation schemes (figures 8 and 9).

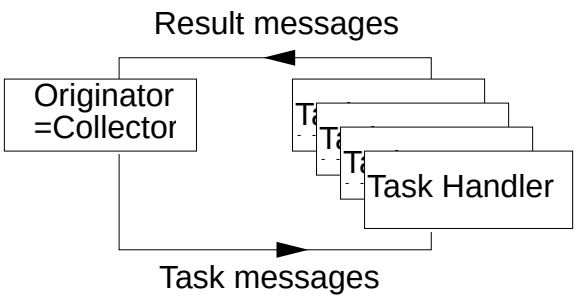

Figure 8: Arrangement for asynchronous iterations. The results are passed back to the originator to be used in the next round of iterations. Asynchronous iterations are a special case of the NetWork setting. The originator assigns sub-tasks to (anonymous) co-workers which handle them. The results are passed to a recipient=initial process and integrated there. If necessary, this cycle is iterated.

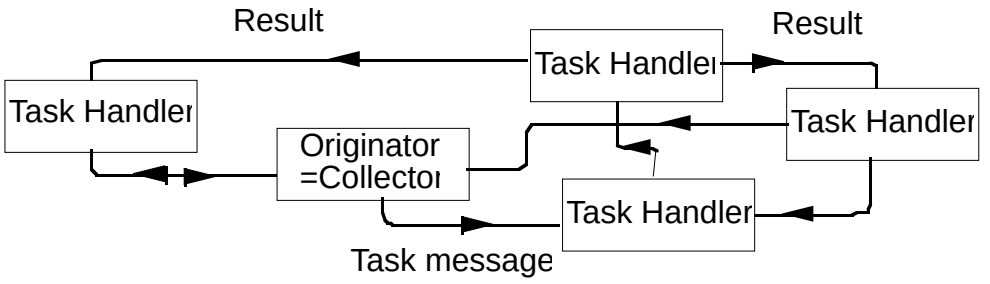

Figure 9: Arrangement example for genetic algorithms. The results are passed back to the originator to be used on a randomized selection basis in the next round of iterations. In this very special example, results may always be passed back to the generator (to guarantee convergence), but are also passed to parallel processes. Genetic optimization is another special case of the NetWork setting. In contrast to the asynchronous iterations, the results are not (or not only) returned to the originator, but are passed to a neighbour selected at random, which may or may not use parts of the information supplied ("random cross-over"). If necessary, this cycle is iterated.

Implementation Environment and Experiences with NetWork

The original implementation of NetWork uses the Macintosh as a target machine. The Macintosh Operating System is essentially a user driven system, with an event queue monitoring user action as the heart of the system. Hence there is a single, well defined point on the Macintosh where it is possible to monitor all user actions. Communication, in the form of the AppleTalk protocol, is another core part of the Macintosh OS. This makes the Macintosh a prime target for NetWork, although the NetWork implementation model is not restricted to the Macintosh. But defining and guaranteeing user based constraints is more complicated in a UNIX environment.

The Macintosh is designed as a single user machine. The usual memory and process protection schemes are not available on the Macintosh, and have to be substituted. On the other hand, the continuous unique address space under Macintosh OS allows for efficient communication between all processes on one machine. Memory and process protection of course are readily available on UNIX systems. A NetWork implementation for UNIX has to respect these mechanisms and UNIX' separated address spaces. This affects in particular the transport system for messages which stay on one machine.

A UNIX system tuned for rapid launch and optimized for inter-process communication would be a very interesting target for NetWork. The Mach kernel - as far as we know - has extraordinary capabilities in this direction. However we have not been able to work with a Mach kernel so far.

The current implementation of NetWork for the Macintosh consists of the NetWork Processor (a Control Panel extension), a library of low level routines, and a scheduler prototype. On the user's side, the NetWork Processor is copied into the directory containing the operating system, the System Folder. That is all what is required to install NetWork. As with all Macintosh Control Panel extensions, the NetWork Processor will be activated automatically on the next start up. Once the NetWork processor is installed, programs can make use of the NetWork system. For the programmer, the NetWork library will be used as is. All necessary calls to the library will be handled by the scheduler, unless direct access to the low level routines is requested. Two function of the scheduler have to be adapted: definition of new tasks, and handling of incoming messages. Typically this will be done by overriding the dummy actions provided with the prototype scheduler. To allow a typical Macintosh program to make use of NetWork, the scheduler will be activated at two points in the main event loop: in the case list handling new events, in case of NetWork events the scheduler method handling messages has to be called. In the default clause (the "idle" case), a call to the method which is defining new tasks has to be added.

The NetWork implementation model has been in use now since November '89. It is easy to give impressive figures showing the score as far as the main goal is concerned, net work throughput: Of course it is possible to design tasks that are limited in performance only by the minimal communication and scheduling overhead, and will show arbitrarily good performance. Without relying on this type of examples, the general experience was that about 70 % of the free computing power could be used effectively, with a reduction of free communication bandwith on an Ethernet of less than 5 %. These are results on small to medium sized local area networks (< 100 stations). NetWork is an experimental environment, and more systematic measurements are underway.

For larger networks, looking up idle machines will become a critical issue. In the present implementation, each station does its look-up. For larger

networks, a hierarchical scheme with specialized look-up processes will be preferable (Theimer and Lantz, 1989). This is a field of current research.

We try to give absolute priority to "real" users. Second class communication would be first choice. We have several communication models allowing for priority communication. Second class communication, communication only if there is free bandwidth, seems to be a neglected area.

When we started the NetWork project, we wanted to provide a minimal implementation. Asynchronous distributed computing, was our leading example. However when we finished the alpha phase of our project, we learned that there is more demand for distributed computing models than we expected. It seems that models like RPC are far from satisfying the needs for *guarantied* distributed computing. However we do not want to go into this area, because for us distributed computing in a non-guarantied environment is much more challenging.

There is a trade off between reliability and overhead. The model we are using is a minimal acknowledgement scheme. A task assignment for us is an implicit launch, and the only acknowledgement we are using is the completion message of a task. We got the impression that launch on task assignment would be a valuable feature in a more general context, and our acknowledgement scheme is sufficient for a large class of applications. However a basic implementation for a guarantied-completion implementation has been asked for repeatedly. Using our model it is easy to establish a session on first successful contact, satisfying the demand for guarantied completion.

Looking Ahead

The availability, or affordability, of computing power is subject to change. For many purposes getting the necesary computing power soon will be no issue, at least in the richer economies. But another aspect of NetWork will stay important. We are moving from installations with computers and workstations to a *computing environment*: a massive use of only partially coordinated or uncoordinated autonomous computing devices, with multiple threads of communication between them. These environments will not have a guarantied stability. They will change in time, and will have varying availability. We should be prepared for the possibilites and problems distributed computing environments bring with them. NetWork allows to study some of the effects, and proposes a design strategy for computing environments.

Literature

Arbib, M.A *Brains, Machines and Mathematics*. Springer, New York, Berlin, Heidelberg 1987. ISBN 0-387-96539-4

Baudet, G.M. Asynchronous Iterative Methods for Multiprocessors. *Journal of the Association of Computing Machinery 25*, 1978.2, 226-244

Bernard, G., Duda, A., Haddad, Y., and Harrus, G. Primitives for Distributed Computing in a Heterogeneous Local Area Network Environment. *IEEE Transactions on Software Engineering 15*, 1 (December 1989) 1567-1578

Bertsekas, D.P., and Tsitsiklis, J.N. *Parallel and Distributed Computation.* Prentice Hall, Englewood Cliffs, NJ., 1989. ISBN 0-13-648700-9

Eddy, W.F., and Schervish, M.J. Asynchronous Iteration. In *Computing Science and Statistics: Proceedings of the 20th Symposium on the Interface 1987*, 165-173. American Statistical Association, Alexandria, VA. 1988.

Gardner, T.J., Gerard, I.M., Mowers, C.R., Nemeth, E. and Schnabel, R.B. DPUP: A distributed Processing Utilities Package. ACM SIGNUM Newsletters 21, 1986.4, 5-19

Kühn, R., and Sawitzki, G. Spinning Brain: an Interactive Program for the Associative Recall of Visual Patterns. *Wheels for the Mind (Europe) 1/1989*.

Lindenberg, J. NetWork Communications. Universität Karlsruhe, Institut für Betriebs- und Dialogsysteme, 1990.

Sawitzki, G. NetWork Programmer's Guide. Universität Heidelberg, Institut für Angewandte Mathematik, 1990, 1991.

Theimer, M.T., and Lantz, K.A. Finding Idle Machines in a Workstation-Based Distributed System. *IEEE Transactions on Software Engineering 15*,1 (November 1989), 1444-1457

How to Access NetWork

NetWork is available upon request from the author. The NetWork distribution disk contains additional documentation. The NetWork Programmer' Guide (Sawitzki, 1990) is a first step if you want to implement a NetWork program. It is recommended to start modifying the examples which are provided on the NetWork distribution disk. The NetWork communication system is documented in a separate paper (Lindenberg, 1990) which is recommended for additional reading.

The NetWork software and documentation is available on electronic media: anonymous ftp from

statlab.uni-heidelberg.de <129.206.113.100>

provides the recent version of the NetWork software. Other sources of information are

sumex.aim-stanford.edu info-mac server
The Developer CD Series Vol. IV ff Path:
…:Programming&Utilities:…

A video demonstration of NetWork is available as "NetWork Developers Conference" 1990 from Apple Advanced Technology Group (ATG), Apple Cupertino 1990.

Biographical Note & Additional Information

Günther Sawitzki is at the Institute for Applied Mathematics, Heidelberg. He is working in computational statistics and data analysis if he is not busy with software engineering and development. The NetWork communication system was designed and implemented by J. Lindenberg, Karlsruhe. The other members of the NetWork project were R. Kühn, Heidelberg, and L. van Hemmen, Munich, both members of the Heidelberg Neural Network research group, and L. Taylor, Apple Computer R&D Europe, Paris.

The author thanks R. Beran, M. Hebgen, J. Lindenberg and L. Taylor for helpful comments.