A Market Model for Controlled Resource Allocation in Distributed Operating Systems

Alan Messer

Submission for the Degree of Doctor of Philosophy

City University

Department of Computing

29th April, 1999

Contents

List of Figures

This is dedicated to Monika for her continued support and putting up with me whilst I finished.

Acknowledgements

This work was carried out in the Systems Architecture Research Centre in Department of Computer Science at City University, beginning in late 1993 and finishing in early 1997, during my time there as a postgraduate research student. This thesis was written in 1997 and early 1998, during my time as a Research and Development Engineer at SONY Architectural Labs, Belgium and later with SONY Digital Networked Solutions, Belgium. Many people have influenced my work over this time at these establishments whom I wish to thank.

Tim Wilkinson, *Java Virtual Machines to the gentry*, who provided the encouragement and technical discussions to bring this unusual work to light.

Peter Osmon, *supervisor*, who provided useful guidance throughout this time and helped give me the will to finish this thesis in the months of writing up.

Julie McCann, *the closet goth and illustrious leader that never was*, for her support, friendship and inspiration in those months of writing up.

Kim Harries, *a bounder and a cad*, flat-mate, friend, cultured-type person and philosopher.

Ludo Cuypers, *the Belgian contingent*, team leadership and encouragement while at SONY.

And finally, my parents, for their support and encouragement in getting this thesis finished.

Abstract

This thesis explores the potential for providing processes with control over their resource allocation in a general-purpose distributed system. Rather than present processes with blind explicit control or leave the decision to the operating system, a compromise, called *process-centric* resource allocation is proposed whereby processes have informed control of their resource allocation, while the operating system ensures fair consumption.

The motivations for this approach to resource allocation and its background are reviewed culminating in the description of a set of desired attributes for such a system. A three layered architecture called ERA is then proposed and presented in detail. The lowest layer, provides a unified framework for processes to choose resources, describe their priority and describes the range of available resources. A resource information mechanism, used to support choices of distributed resources then utilises this framework. Finally, experimental demonstrations of *process-centric* resource allocation are used to illustrate the third layer.

This design and its algorithms together provide a resource allocation system wherein distributed resources are shared fairly amongst competing processes which can choose their resources. The system allows processes to mimic traditional resource allocations and perform novel and beneficial resource optimisations. Experimental results are presented indicating that this can be achieved with low overhead and in a scalable fashion.

Chapter 1

Introduction

General purpose distributed systems consist of a network of computing resources, such as workstations, on which a variety of tasks may execute. Each task requires multiple resources which must be provided from the available resources through sharing when necessary. This thesis aims to answer the question: *In general purpose distributed systems, is it possible and useful to empower each task with informed control over its resource allocation while preserving fairness?*

Two approaches to resource allocation exist in such systems. In the first it is the responsibility of the operating system, or a third party, to share resources between tasks. By using its knowledge of other tasks, the OS is able to distribute and balance tasks. However, good task performance is difficult to ensure, since the importance and locality of each task's resources is not known by the operating system. In the second approach allocation is the task's responsibility: each can blindly use whichever resources it chooses. By choosing its own resources, a task can express the locality with which it intends to use resources, enabling good performance when there is no competition. Yet, since the selection is made blindly and with no expression of importance, this choice may be poor when competition is present. Also, since it has full resource control, little prevents a task using unreasonably large quantities of resources.

This thesis proposes that neither approach is optimal, since the operating system does not consider the task's function and the task does not consider competition from others. This thesis therefore advocates a new method of resource allocation in which tasks choose all the resources they must consume and express an importance for all their requests. This allows them to have influence over which resources to use and how important they are. To make informed decisions tasks use information about the resources surrounding them (their resource environment). This environment contains demand and utilisation information for each resource in its distributed locale. To preserve fairness, under competition for resources, the operating system controls the importance a task can express about the resources it desires.

Therefore, tasks have a free informed choice of resources. A task chooses resources appropriate for its functionality and expresses the importance of each resource in accordance with this. But tasks can only impose a limited effect with their choices. We call this approach 'process-centric' resource allocation, since all decisions are the process's.

1.1 Reasoning

Resource allocation is important for the execution of a task because it can greatly affect performance. This is especially true in distributed systems where there is a greater choice of resources. With choice comes increased complexity in determining the optimal resources for each task. This in turn becomes even more complex when there is a diverse set of applications executing across the system at the same time, as is common in general purpose distributed systems.

For example: scientific computation tasks require a variety of resources over a long period of time; while databases and graphical tasks require their resources concentrated towards memory or disk access where the performance is critical. In general purpose systems it is important to cater for a range of different types of resource requests.

This thesis proposes that a task should be able to select the right available resources for its purpose. The competition for resources requires that this choice be informed. In addition, to ensure fairness among tasks the operating system must retain control of the rate of consumption of each resource.

This thesis proposes a three tiered architecture to achieve these aims. The lowest tier provides a framework which gives processes full control over their resource requests, while allowing the operating system to maintain fair consumption of resources. Above this, the second tier provides a scalable distributed information system called a 'marketplace', to provide resource demand and utilisation information to processes. The third tier consists of resource aware processes which both lower tiers to make informed resource allocation decisions in order to improve their performance in dynamic resource environments.

1.2 Method

The work described in this thesis investigates a novel form of resource allocation which requires work on many areas of an operating system, including: operating system structure, resource allocation algorithms and information dissemination techniques. To demonstrate and analyse these changes example new algorithms to utilise the architecture also need to be developed. As a result, a great amount of potential work in this field is possible. The work undertaken for this thesis has tried to provide experimental evidence that the stated aims of the thesis are in fact feasible and useful.

As a result, a distributed operating system based on the proposed architecture has been developed and implemented on a simulation of distributed hardware. Using instrumentation on this operating system, the overhead and resource/monetary operation of the architecture can be examined. Experiments using this implementation are performed to analyse the fair distribution of resources to processes under competition from other processes. Additional experiments measure the overhead to support the architecture, its scalability and examples uses of the novel resource allocation framework.

1.3 Contributions

This thesis makes original contributions centred on the three tiers of the architecture: the framework for process-centric resource allocation, the scalable distributed marketplace and process-centric policies for process performance self-improvement.

1.3.1 Framework for process-centric resource allocation

The process-centric framework is based on the concepts of money and simple market operations and contributes:

Full process-centric resource allocation Processes have full control over the location and importance of all the resources they ever need to consume. Processes even have control over the resources in which they exist, such as the memory in which the process resides.

Fair control over resource usage This thesis introduces several methods for sharing resource usage fairly between competing processes.

Low overhead auction mechanism This thesis proposes a decoupled auctioned technique whereby processes are not guaranteed resource access but, instead, buy the ability to consume a resource. This aims to reduce the overhead of using resources compared to the direct auction systems used by previous money based systems.

1.3.2 Scalable Distributed Marketplace

The distributed marketplace provides a means for disseminating information and contributes:

Resource information feedback The ERA architecture provides a uniform and scalable means for providing resource information to processes so that they are aware of their resource environment and can respond to it.

Information distribution with dynamic locality The distributed marketplace provides a dynamic degree of locality for each node, dependent on the current system load. This is unlike previous systems which use fixed hop counts to limit the distance information is transferred.

1.3.3 Resource policies

These process-centric abilities require, a process to be written in such a way as to exploit them to be aware of its resource environment. To illustrate the benefits of the process-centric approach two new algorithms have been developed, contributing:

Self load balancing Using a novel non-deterministic algorithm, a set of processes are demonstrated load balancing themselves using only process-centric allocation and the feedback presented by the marketplace. Each process chooses cheaper nodes so as to increase the quantity of resources it receives for the money it pays. Results of initial experimentation in a simple scenario indicate that local optimisation of resource cost leads to a global cost optimisation and thus a load balance.

Dynamic parallelism This thesis advocates that a process can dynamically determine a near optimal level of software or hardware parallelism so as to consume its unused resources. This is made possible by a process using feedback from the marketplace to monitor its resource utilisation. This process controlled technique is new to the field of resource allocation. Initial experimental results are given that show a process determining a near optimal level of software parallelism available for a simple configuration.

1.4 Report structure

This chapter has introduced an overview of the content of this thesis. The next chapter introduces and analyses work related. Including both mainstream solutions and economic-based systems solving related problems. In Chapter 3, we consider the problems of using both these approaches for general purpose distributed systems. The chapter concludes with an introduction to the proposed resource allocation architecture along with an explanation of how it tackles these problems.

Chapter 4 describes the general framework for providing controlled process-centric resource allocation from both monetary and operating system points of view. Analysis of the effectiveness of resource control and overhead of the framework is given under a variety of configurations. Chapter 5 covers the design and implementation of a distributed marketplace and how prices are used in the system. This chapter also analyses the scalability and overhead of the marketplace. Chapter 6 describes example uses of the ERA architecture to allow a process to optimise its operation. Examples focusing on locality and utilisation are presented. Each is analysed with respect to its operation and potential performance improvement.

Clearly no solution is perfect, so a critique of the solution is presented in Chapter 7. The thesis ends with a summary of the conclusions drawn from the work and introduces future research opportunities pointed to by this work.

An appendix follows which describes the simulation used to inform the analysis contained in this thesis. This appendix describes how the simulation was designed and instrumented in order to obtain the results.

1.5 Summary

This thesis, therefore, asserts that current resource allocation techniques are inadequate for use in general purpose distributed systems. This is because processes are unaware of competition for resources and they do not constrain their resource consumption in proportion to their importance.

The thesis advocates a new method of resource allocation, called *process centric* resource allocation. In this method, tasks choose all the resources they must consume and express an importance for all the resources they need, allowing them to influence their consumption of resources. The role of the operating system is to preserve fairness under competition for resources by controlling resource consumption overall.

Ultimately, this thesis aims to answer the question: *In general-purpose distributed systems, is it possible and useful to empower a task with informed control over its resource allocation, while preserving fairness?*

Chapter 2

Development of resource allocation techniques

Resource allocation has become a complex and challenging problem. Multiple processes can make multiple requests for multiple resources in a given system. Each request can exhibit differing importance, locality and timing constraints. Having access to, and then considering this information presents many complexities for deriving even a single solution in a single configuration. In situations where the configuration changes over time, the problem becomes even more complex. It is infeasible to derive an optimum allocation in anything but simple scenarios.

In earlier systems this was not a problem since computers were only used in simple scenarios. For example, single user workstations need not consider complex priorities, locality or timing constraints, since each machine is dedicated to one user at a time. General-purpose servers run computationally intensive tasks which only need to be differentiated by abstract priorities. Dedicated systems, such as real-time, can be configured for well-defined situations with well-defined allocations.

The arrival of complex, performance sensitive applications that are distributed and demand more complex resource allocation. Progress in software technology allows more sophisticated applications to exist and coexist including multi-media, database and graphical applications. When resource allocation is extended to the distributed case (such as networks of workstations), the complexity increases again. Multiple instances of resources introduce both choice and competition, while the requirement for scalability of distributed systems introduces problems of access latency combined with out of date and incomplete information about remote resources.

This chapter covers the prevailing resource allocation techniques for use in general purpose systems. Techniques relating to resource allocation in single processor systems are discussed first. Following this, the techniques and problems of the distributed resource allocation case are discussed. Next, the chapter focuses on relevant previous systems which have used economic and market concepts to solve resource allocation problems. Finally, this chapter briefly covers some contemporary work in the area of this thesis.

2.1 Scheduling

2.1.1 Priority scheduling

Most conventional operating systems employ priorities for scheduling [19]. Each process is assigned an absolute priority which is represented by a numerical value. Higher priority values override lower values absolutely. A process with a higher priority always gets the resource in preference to the lower priority process.

The most common use for priority scheduling is for CPU and disk access. In this scenario, effectively multiple queues are used where resource requests from the highest priority queue are handled first (see Figure 2.1). For example, the ready queue of a CPU scheduler can be structured like this. Once a process completes its time-slice, or blocks, it is returned to the queue for its run priority. The main benefit of this technique is its low overhead (simply a priority queue insertion $O(n)$), where n is the number of priorities. It is also a simple way of providing a few distinct and exclusive levels of priority, and this is helpful for distinguishing between system level and user level work.

Unfortunately, static priorities can lead to resource *starvation*, since the priority levels

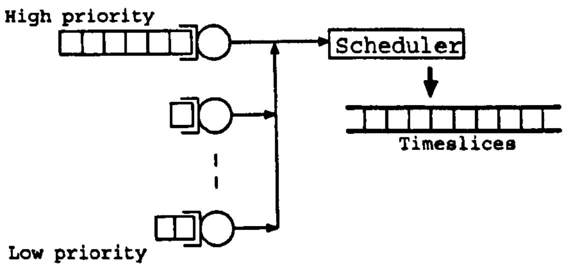


Figure 2.1: A priority scheduler of CPU time-slices. Higher priority processes have absolute priority over lower.

are exclusive. This is because, despite being runnable, a low priority process may have to wait indefinitely on a higher priority process. Several solutions exist for this problem based around the idea of *dynamic* priorities [2, 32]. In these solutions, the priority of processes is adjusted over time to prevent starvation.

Priority aging lowers the priorities of higher priority processes as resources are consumed. This enables lower priority processes to get scheduled, even in the presence of prolonged resource usage by a higher priority process since, eventually, the higher priority process will be demoted to a level lower than the lower priority one.

Decay-usage scheduling [2, 32] is similar. Averages are kept for the processor usage in the recent past, giving higher priority to those processes with low usage. Thus, as the highest priority process utilises the resource, its priority drops (decays) until it no longer has the highest priority. This approach is taken by some popular operating systems such as UNIX.

Dynamic priority scheduling has other problems. Firstly, the use of numerical priorities provides little in the way of meaningful control. Assigning an abstract priority to a process encapsulates little about the precise importance of a process over others. Because of this, it is not possible to ascertain the resource access each process will receive. Secondly, the functions used to calculate the dynamic priorities are ad-hoc. They are only controlled through a few crude scheduling parameters, which are poorly understood. These problems become more acute when many levels of priority are used. A third problem is that priority scheduling is not applied across all system resources. This can allow even low priority processes to consume more resources than their abstract priority dictates. For example, a low priority process can consume large quantities of memory and page fault bandwidth, with the consequence that higher priority processes may be suspended by page faults allowing the lower priority process to increase its CPU consumption. These problems arise because priority scheduling providing no control over the relative quantity of resources consumed by a process.

2.1.2 Bounded consumption

Many operating systems [32, 18, 20] incorporate bounds on the consumption of certain resources. These bounds represent the maximum allowed consumption of these resources by a process over time. In this way, any process can (if the bounds are checked) be prevented from consuming all the resources in the system. Once a bound has been hit the operating system prevents further consumption of that particular resource, which is usually terminal to the process. A count of the consumption of each resource can be kept for each process by the operating system and checked on each allocation to ensure the bounds are kept. This simple implementation imposes little overhead on system run-time.

When the run-time requirements for a process are well-known bounded consumption is very useful as a safe-guard against bugs. Bounds are also useful for limiting resources, such as: the number of processes per user, stack size and open files. This enables reasonable conservation of important resources such as internal operating system table entries.

The technique has two problems however. Firstly, exceeding the bounds is terminal to the process concerned. This can be overcome to some extent by using hard and soft limits to warn the process that it is nearing its bound. However, it is still possible for a process to hit the hard limit and be terminated. A solution is to block/suspend the process when the bounds are met, allowing the process to be restarted should it be needed. Consider, for example, a database server process. It needs to exist for an indefinite amount of time and can consume an undefined quantity resources depending on demand for queries. With bounded consumption, at any point in time the server may hit its resource bound and be terminated. This has two detrimental affects: the server is terminated preventing any further queries and any current transactions will not be completed potentially causing inconsistencies.

The second problem is that the bounds refer to absolute quantities of resources rather than the rate of consumption over time which is often more relevant. Consider CPU time-slices. A process may need to be run in the background performing a long calculation. In this situation the amount of the resource over time needs to be restricted to keep it in the background rather than restricting it to 5 minutes of computation.

2.1.3 Real-time scheduling

In real-time scheduling the timeliness of the schedule is important, if not vital. Real-time comes in two forms depending on how rigid the timing guidelines (deadlines) are. Hard real-time deals with scenarios where missing a deadline is mission critical. Applications which fall into this category are nuclear reactor controllers and flight controllers. Soft real-time, by comparison, has much weaker deadlines, where missing a few is not disastrous, but merely undesirable. Areas such as multi-media traffic management use soft real-time.

In order to provide the scheduler with some bounds for scheduling the various tasks, each task must have its resource needs – the amount of resources required and timeliness constraints (start time and deadline) – specified in detail. Thus all resource consumption can be known before the task is run. This provides a very rigorous framework, such that the execution time of all tasks must be safely predictable. This predictability also requires knowledge of the static interactions within the system and hence to know the overall timing constraints.

Given these constraints, the scheduler derives a schedule for which all the deadlines are met. In the case of hard real-time, deadlines *must* be met and the schedule is usually dry-run under all circumstances and guaranteed by providing safety margins on all resources used. This means that resources are typically very under utilised (around 50 percent).

In soft real-time it is not possible to provide such guarantees since the tasks to be scheduled are dynamic. One of the most popular techniques for scheduling such requests is *Rate Monotonic* scheduling. Each task is assigned a static priority which is based on the task's periodic execution rate. In this way, frequent tasks receive a higher priority than occasional tasks. This means, however, that the importance of the task is not absolutely reflected in the priority, since an infrequent task may be close to its deadline, yet still have a low priority. Thus, to provide a reasonable level of assurance about meeting the deadlines, resources are generally under-utilised.

To meet this need for priority as a function of timeliness, *Earliest Deadline First* scheduling can be used. At each reschedule the current set of tasks and their deadlines are considered in order to ensure the task with the earliest deadline is run. This approach provides a higher resource utilisation while also taking account of the importance of the deadlines, but at the price of a higher overhead, since at each reschedule the earliest deadline must be computed.

Real-time scheduling has been very successful in those application areas which fit into its framework. It is not, however, feasible in a general purpose system. Firstly, real-time scheduling requires detailed knowledge of the processes which is simply not available in a general-purpose operating system. Typical general-purpose tasks, such as compiling and editing, cannot fit into this structure, since it is impossible to predict the amount of resources and time each task will use. The complexity due to the number of possible interactions required by most applications. Also many of these interactions are through the operating system to system resources, such as disks, for which the time constraints of access are not known.

Secondly, since deadlines are absolute and the amount of resources each process obtains depends on the interaction between each process' requests, deriving the relative rates of computation of concurrent tasks, given the real-time requests of each is complex. Given this complexity, it is difficult for the user or process to provide any active adjustment of its requests in response to its surroundings.

2.2 Distributed resource allocation

When extended to the distributed case, resource allocation gains three extra complexities. Firstly, there is usually a choice of instances of each resource, which may be used interchangeably. Secondly, information about the state of the system is usually out of date due to network latency. Lastly, scalability of the software must match that of the hardware. In combination together these complexities make resource allocation a complex problem.

2.2.1 Process-based control

Process-based control is a simple form of distributed resource allocation. When a choice of resources exist, the process simply specifies which instance of the resource it wants to use. Each resource may be addressed by either an abstract resource identifier or the node's address. The choice of resource to use is up to the programmer and may be fixed at compile-time, as a parameter to the program, or determined at run-time given the input.

The advantage of this approach is that it requires little or no overhead and allows very good resource utilisation when used in custom distributed systems where the configuration is static and specialised for the purpose.

The approach does have its problems. Firstly, it does not take into account other processes in the system. In a multi-user system it is quite possible that the nodes a process uses are in heavy use by other programs. This can be seen in Figure 2.2. Secondly, the processes are unable to cater for changes in the size or structure of the distributed system. Should the distributed system change, the process must then be recompiled or suitably parameterised to use or tolerate the changes. In combination, these factors make process-based control suitable only for static, well-known workloads or single job scenarios.

Virtual processors

Virtual processors or nodes improve the situation by providing a level of abstraction between the placement of the task and the physical layout on the system. This allows the task to be written with a parallel structure which suits the software algorithm rather than

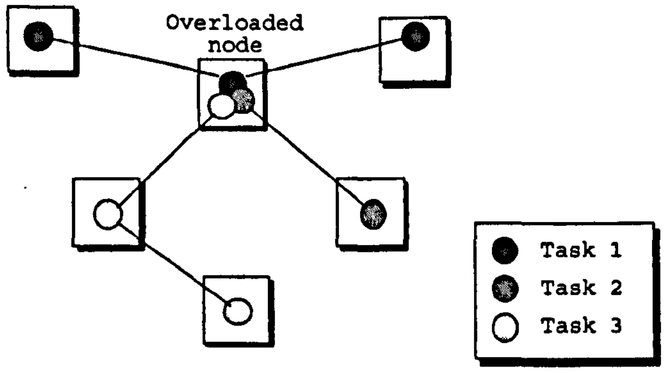


Figure 2.2: The centre node is overloaded by the node selection by three independent programs

the hardware. The task's virtual layout over the virtual processors can then be determined by the programmer or the operating system (as we shall see in the next section). Virtual processors are easy to implement and use, since the extra abstraction only requires the use and administration of abstract addresses/identifiers.

Popular systems such as PVM [4, 24, 25] use this approach. There are several advantages. Firstly, code does not need to be recompiled to respond to system changes, since all the addresses are abstract. Secondly, as a result it simplifies the task of the programmer, by reducing any need to consider the hardware structure when writing algorithms. Thirdly, the choice of address to placement mapping can allow multiple tasks to be configured at installation time so as to execute efficiently on the same hardware.

Problems remain with this approach when applied to a dynamic system. Since the workload is not known prior to dynamic changes, the virtual mappings must then be changed dynamically. This requires administration either by human intervention or operating system. Such administration is a compromise between process and operating system based control.

2.2.2 Operating system based control

A key problem with process-based control is that resource competition from other processes is unknown. The operating system, however, has this information since it is providing

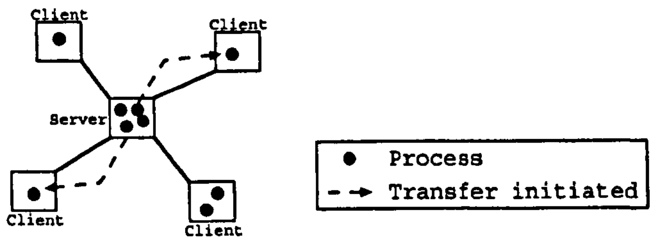


Figure 2.3: A server initiating load balancing

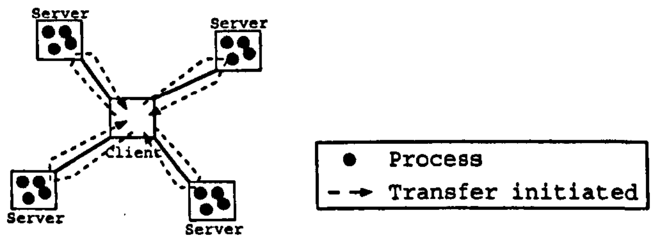


Figure 2.4: A client initiating load balancing

access to the resources. By using this information the operating system can perform *load-balancing or sharing* to equalise the load among nodes.

With operating system based control of the resource allocation, requests for resources are made generically without specifying any location. The operating system determines the location. When new jobs enter the system, a new allocation for the combined requests can be made. With location and access transparency, processes can run without knowledge of their initial placement or continuing migration. Operating system based control is ideal for multiple user situations, since it can cater for competition among processes.

With the operating system making decisions on which resources to use, there is an issue of where and when to make these decisions. Systems fall into three categories in this respect: initial placement, server-initiated and client-initiated systems. Initial placement systems only perform the decision making when new resources are requested, e.g. when a new process is created. Server-initiated systems load balance by initiating a distribution of the load when part of the system is overloaded. So, once a node becomes overloaded it can offload some of its work onto other nodes (see Figure 2.3). Client-initiated systems load balance by looking for additional work when a part of the system is under-loaded (see Figure 2.4).

Operating system based control does, however, have its problems. The biggest problem is that the operating system does not know much about the processes it is running. E.g. it does not know the importance or spatial locality of resource requests. It must therefore make general assumptions about the type of applications in order to make an allocation.

For the operating system to make its allocation it must store information about the distributed system's load. For good scalability this load information must be distributed too, which increases the overhead of the operating system and poses problems of scalability.

Heuristic systems

It is not feasible to derive an optimal load-balance in general scenarios because information about the rest of the system is incomplete. Heuristic load balancing systems have therefore been developed. Heuristic systems make assumptions about the process requests or the nature of the system load. Using these heuristics, good load-balances can be quickly calculated for situations when the assumptions apply. Since the assumptions are not usually too specific and are commonly true, such heuristics are the basis of many load balancing algorithms [29].

As an example, Bryant and Finkel [7] developed a simple cooperative heuristic load-balancing algorithm, the *Pairing* algorithm. The heuristic used here is that, rather than rely on spreading load to all nearest neighbours, it is only necessary to find a partner node to exchange work with, since it is common to have local hot spots together with local cold spots. Each node can send a pairing query to a neighbour, containing the processes it wants to offload and load estimation of each. On receiving a query a node can choose to reject it, since the list of processes would not benefit from running there, or form a pair so that processes migrate to the lighter loaded node. The pairing is breaks when a process in the pair decides it can no longer benefit. In this way the pairing heuristic balances hot spots.

Learning systems

Learning systems take the use of implicit knowledge one step further by trying to learn from past decisions. The heart of such systems is historical data about either process resource requests or alternative decision probabilities. Thus, by using historical information about each process (such as memory reference patterns) some functional information is available for future decisions and *probability vectors* allow the algorithm to benefit from previous inappropriate decisions, by reducing the probability of them being chosen in the future.

For example, Stankovic's Stochastic Learning Automata [38] use a probability vector to store choice information about how loaded nodes are. Each node periodically checks its own load and broadcasts its load value to all other nodes, which use this information to form their probability vectors. These vectors allow nodes to probabilistically choose where to balance an excess load. When a node receives a migrating process it can choose to reward or penalise the sending node. For example, increasing the probability of migrations to a sending node can penalise it for previously sending a high load.

Quality of service operating systems

Quality of service (QoS) operating systems [40, 33] retain the principle that the operating system is in control of resource allocation, but adds an extra parameter to the resource allocation requests: desired quality of service. This allows processes to specify more exactly the resources they need. For example, rather than simply allocating a network connection, a QoS OS allows the process to specify the bandwidth and possibly latency which it needs for that connection. QoS operating systems can be used in CPU scheduling and virtual memory allocation as well as networking. This kind of operating system is designed for resource intensive situations such as multimedia applications. In these situations, the resource requirements are well-defined, high and mostly constant.

For several reasons a QoS operating system is not well suited to the general purpose distributed systems considered in this thesis for several reasons. In such a system, it is usually not possible to define the quality of service required by an application. For example, what is the quality of service required for compilation? Unless quality of service can be quantified such systems simply act as traditional operating systems. In general purpose distributed systems resource requirements are highly dynamic. The quantity of processing required for a compilation varies with the complexity of the source file and stage of compilation. So any guarantees made to the compiler would have to be either continually re-requested or else re-evaluated. In a QoS OS, this frequently means that previous guarantees must be broken to ensure the new guarantees can be met.

Finally, in a QoS OS it is perfectly possible for a rogue or misprogrammed process to request an unneighbourly high quality of service from the system, in turn affecting other processes. But, in a general purpose distributed system as defined in this thesis, it is not acceptable for a process to be able to affect adversely the execution of others beyond their priority.

2.3 Economic techniques

Markets provide mechanisms for decision making and resource allocation in the real-world. *Producers* produce products which are consumed by *consumers.* To consume a product a consumer must first purchase it. Producers demand a certain price for the product, while consumers offer a certain price. This is the model known as *supply* (the producer's product price) and *demand* (the consumer's product price). The point at which the two meet is the price *equilibrium* and is the current market price. As supply increases the price drops, while as demand increases the price increases as the resource becomes more scarce (see Figure 2.5).

This simple view omits a lot of detail, but is a well studied model for the operation of market forces. In a distributed marketplace the picture is complicated by multiple producers and consumers. Since markets are not centralised, it is not possible for any consumer or producer to know the value placed on the product by all the others. In this way, the market price is derived from incomplete information and so is not constant across the marketplace, just between communicating participants. Each participant in the

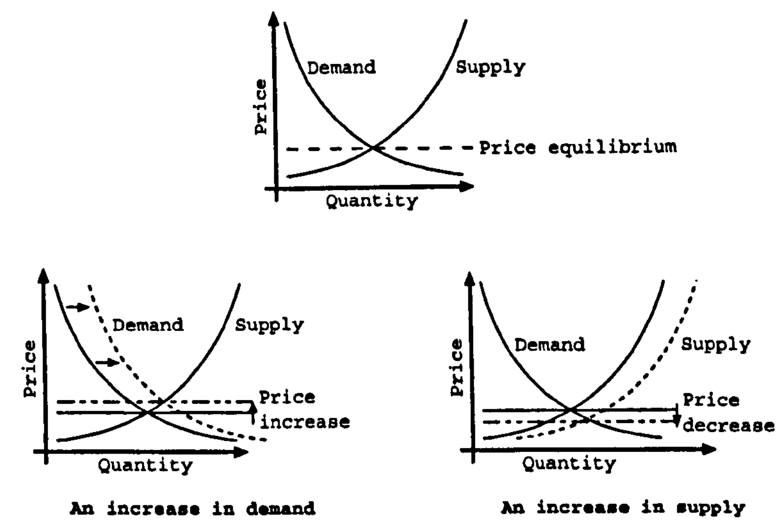


Figure 2.5: The resultant variations in equilibrium price given changes in supply or demand

market non-deterministically makes purchasing decisions which result in demand varying over time. The complexity of this market is increased further since the market can contain a supply substitutable products, i.e. products which are compatible but not identical.

What is interesting about markets is their ability to promote scalable distributed decision making about the allocation of resources in the presence of incomplete information. Real-world markets do, however, have some problems. Firstly, they can be unstable. This is especially true in fast markets such as stock-exchanges or other computer embodied markets, since the lack of deterministic behaviour can result in large price fluctuations. Secondly, the overhead for agreeing prices can be high, especially when competition/demand for a single product is high.

The world of economics and real-world markets is thus a complex one. Yet, the problem they solve is very similar to resource allocation in distributed systems, including: distributed decision making; use of incomplete information; and scalability. So, it is hardly surprising that several researchers have, in the past, tried to formulate versions of real-world markets for distributed resource allocation.

There are many similarities between the business world and a distributed system. A process running on the CPU can be viewed as doing work. Memory space can be likened to land. Network links compared to roads or public transport systems. Many other analogies can be pursued. Consider memory. Is it like rented land or purchased land or perhaps leasehold? What happens when you run out? Perhaps the memory could buy data to be stored in itself, rather than the process buying memory for data? The analogies are not perfect. For instance, the aim of processes is to complete and then die. This means they should die as soon as possible, a sentiment not usually shown in real life!

Some of these similarities have been noticed by other researchers. But given this multitude of options, analogies and conflicts the systems developed from them have been diverse.

2.3.1 Auctions

Many market systems are based on the idea of auctions to determine which among competing consumers gets access to the resource. Auctions can derive the market value for a single item/quantity of a product (resource). There are many ways to arrange auctions, but the general scheme is: each potential consumer is a *bidder* taking part in the centrally managed auction (by the *auctioneer*). A bid depends on the value the bidder places on the resource. There are two types of bidding: Open – where competing bidders can see one and others' bid; Sealed – where the bid is only seen by the auctioneer.

The auctioneer closes the auction, either once all the bids are in or after a certain amount of time has passed for competing bids, and decides the outcome. There are two ways of deciding the winner. The most obvious is *First price* where the auctioneer accepts the highest bid. The other is called *Second price* where the auctioneer accepts the highest bidder, who pays only the second highest price. This is supposed to discourage bidders from escalating prices deliberately to cause competitors to pay above the odds.

Sealed-bid auctions are a quicker means of agreeing a price, since the auctioneer only waits until a bid is in from all the potential bidders. In an open-bid system, the auction may last for some time while new bids are placed. The problem with sealed-bid auctions is, however, that the price can vary dramatically over time, since the competition is not seen. To prevent this, the *Dutch auction* provides an iterated sealed-bid auction. In this auction, the usual sealed-bid auction is executed several times after feeding back the chosen price to the bidders. This allows a more stable price level, and thus representation of demand, to be derived.

The problem with all these kinds of auction is that, for each item to be allocated, an auction must be held between all potential bidders. Since an auction must be held for every decision, there is quite a high CPU and network overhead (if distributed). The need to centralise the decision and wait for all bids to arrive also limits how many bidders can take part. There is also an implicit need for some synchronisation between bidders and the auctioneer as to when the bidding phase starts and stops. This imposes extra overhead, especially in the distributed case.

2.3.2 Money represents the importance of the process

When making analogies between markets and distributed resource allocation, one of the first concerns is money and the possession of money by processes rather than people. In real-world markets people possess money to purchase goods. It is people making decisions on the worth of goods which enables a market price to be found. In response, this market price affects the decisions of those buying goods, limiting how much they can purchase.

A process is very similar. It is the entity requiring resources, but it should be allowed only a finite amount. Decision making for a process using its money availability and current resource market values, should determine which resources it buys. It is not surprising that most economic distributed resource allocation systems have used the process as the unit of responsibility for money, whether or not the process itself has access to the money. This section covers the systems where money represents the resource acquisition power of the process.

SPAWN

With networks of workstations, programs can be executed remotely on neighbouring computers. With current work practices, it is very common for a large number of these machines to be idle at any one time. If these idle resources can be utilised then the network of workstations can be used as a multicomputer [17]. The problem of using idle workstation resources is a long standing one, several systems having been proposed [34, 48]. *Spawn* [55] applies market economics to the problem to investigate fairness in resource distribution and system stability.

In the idle workstation scenario, the wasted CPU is the commodity of principal value to other users in the system (although memory and network are also of value). The Spawn work considers a market for CPU. *Money* reflects resource rights while *price* reflects supply and demand, allowing processes to make use of remote idle CPU by buying cycles. In this market purchasers aim to maximise the CPU obtained for the amount paid. As a result, the load is balanced across the machine in accord with to the amount of money each process has. This is Adam Smith's classic *invisible hand* argument [50].

Suppose, for example, a user wishes to run a large computation over a period of time. Since the task is easily made parallel, the job can be split across the network of workstations and executed on the idle machines. Users not using their workstations will sell their idle CPU cycles so that the processing time for the large computation can be purchased across many machines, depending on the amount of money available.

Each seller executes an *auction process* to sell the workstation's idle cycles to buyers. The auction process continuously accepts bids from buyers for the next available time-slice on that particular workstation. The auction is a sealed-bid, second-price auction.

Each task is split into two subtasks: the worker and the manager. The worker is the process which actually performs the computation, while the manager is responsible for receiving money and performing resource usage decisions (see Figure 2.6). Thus, a manager process buys idle CPU from sellers for its worker process. Since the network of workstations is large it is not feasible for a manager to bid to all sellers so, instead, each manager has a list of "neighbouring" sellers which supply price and availability information.

Each manager process needs money in order to bid. Money is supplied via a tree of sponsors in which higher level sponsors sub-divide their sponsorship between their sub-managers for sub-processes. This is analogous to a system of pipes fed from the top with

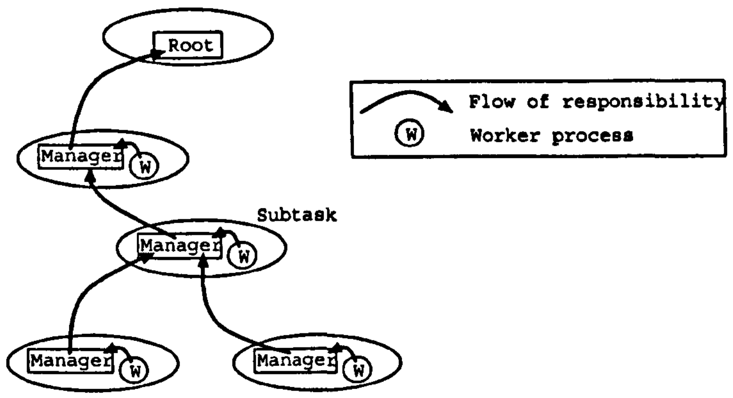


Figure 2.6: Under Spawn, each task is sub-divided into a manager subtask to handle the market side of processing and a worker process.

water. As the water descends it splits into smaller sub-pipes. At the ends of each pipe are the processes being funded to do work.

To buy idle CPU cycles from a seller, the manager process bids a price and the number of cycles required, and a brief description of the task. This allows the seller to consider each process' bid. The winning process is allocated the number of cycles and payment is made. The seller then starts accepting bids for the next available cycle.

Once the purchased period is up, the process executing has *first right of refusal* for continuing at the current market price, since under this implementation tasks cannot be migrated. As long as it can continue paying the current market rate the process continues to execute.

Simulations of Spawn have shown it to be effective for sharing resources fairly between processes. They have also shown that, when the Spawn market is under constant conditions, prices converge towards constant levels. Price levels also adapt as new tasks are added and subtracted from the system.

Spawn has a number of problems however. Firstly, the sealed-bid auctions give rise to significant variance in the market prices and thus shares received by processes. Secondly, the scalability of the marketplace is poor, since the information about local neighbours is only propagated in a 'nearest neighbour' fashion. This means spatial price fluctuations can be large, since processes are unable to exploit idle resources further afield. Thirdly,

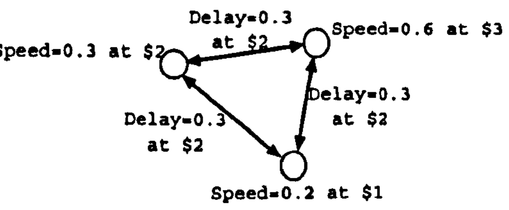


Figure 2.7: A priced distributed system waiting for a task to decide which resources to consume to minimise its run-time

the overhead of bidding to multiple sellers, in order to obtain resources, is high especially when the granularity of work is small.

Ferguson's work

Ferguson et.al. [21] take a more detailed approach to the problem than SPAWN [55]. They consider that in a distributed system there are several resources: CPU, Memory and Network which are scarce and competed for by tasks. Each task and each resource is represented as an agent which has maximising its own function (utility) as its goal. Every resource is parameterised. So a CPU has a speed, memory has a size and a network has a delay. Every task has parameters for its CPU requirements, memory usage and thus network transfer times. The agents representing the resources maximise their utility by varying their prices, while agents representing the task minimise their costs and run-time. It is therefore possible to view the distributed system as a set of resources with parameters (speed, etc.) and costs, which the task uses to minimise its run-time (its ultimate goal) while the resources try to maximise their utilisation through pricing. An example of this structure can be seen in Figure 2.7.

In this structure the process buys the resources it requires. This means that on entering the system, if it requires u_j seconds of processing it must purchase $\frac{u_j}{r_i}$ seconds on processor P_i where r_i is the speed of that processor. A task may migrate around the system, but it must purchase the network links it uses and must return to the node by which it entered the system.

On entering the system, each task is given an amount of money to use in order to complete its task. It then determines which resources to consume, based on its remaining money and the current prices. To do this, it performs three steps: computes its budget set; computes a preference for the budget set; and bids on the most preferred budget. Computing its budget set involves choosing the processors where the task can afford to run and get to and from over the links. Once determined, it chooses the cheapest budget combined with the shortest service time. For its bid, the task uses the calculated cost plus an element of its surplus cash. If it wins the auction the process reduces the amount of surplus cash paid, meaning that under little competition prices will fall. If, however it loses, the fraction of the surplus used is increased and the process repeats. In this way, while waiting, the level of the bid depends on the task's waiting time.

An auction process on each node receives the various bids. Ferguson implemented both sealed-bid and dutch auctions on each node and found little noticeable affect on the price level. The bids are converted into unit costs and the best price wins. This determines the market price, which is the maximum a process is prepared to pay. In order to allow the tasks to calculate their budget sets they need to know the prices of the local resources in the system, so each node also runs a *bulletin board* service which contains the prices from the nearest neighbour nodes. The bulletin board is updated whenever the local prices change and broadcast to all neighbours.

This scheme has a few problems. Firstly, the quantity of resources required must be known before any bids can be made. The assumption is that all processes are batch jobs with well defined bounds like real-time tasks. This assumption does not hold for interactive jobs or jobs where the computation time is non-deterministic. Secondly, as with SPAWN, if the granularity of the computation is small, the overhead of calculating the budget sets is unreasonably high. This is especially true when the Dutch auction, which is iterative, is used. Thirdly, the use of nearest-neighbour information dissemination provides poor scalability in large distributed systems.

WALRAS

Wellman et.al. use the WALRAS algorithm to allow resource allocation problems to be expressed in a market-oriented fashion. Walras (1874) proposed an algorithm called *tatonnement* [58] to find the equilibrium of supply and demand. In a tatonnement market, consumers and producers respond to price signals in order to maximise their self interest. The market interactions are controlled by a central auctioneer process which adjusts the price levels of goods towards a general balance in which all goods are cleared (allocated to a consumer from a producer). Each good is balanced in turn, but since demand for all goods are interrelated then the balance of other goods is affected as this happens. Once balanced, consumers receive a set of new prices and determine their level of demand for each good at the new prices. This 'groping' for a balance is iterated until a balance is found.

Strictly speaking tatonnement auctioning is not the way price setting works in the real world and finding an equilibrium is not guaranteed in this situation. This is because the decision-making processes are not monotonic and thus a *pareto optimal*[1] solution, not an equilibrium is found. In the situation of computer resource allocation, this does not really matter since a workable market can be derived. The WALRAS algorithm provides a distributed marketplace over goods, such that many markets exist, one for each good. Each market asynchronously receives demand functions from consumers. The demands are taken to derive a price which brings the demand closer to the supply (nearer equilibrium) and thus balances the market. The price is then returned asynchronously to the consumers for them to consider (see Figure 2.8). By using demand functions which are monotonic it is possible to ensure an equilibrium is found. The WALRAS algorithm differs from a strict tatonnement market in that demand functions are transferred by the consumer and each market only considers the supply of one good.

This system allows several resource allocation problems to be tackled [12, 6, 59, 41]. Wellman has applied the system to transportation problems (The Multi-commodity Flow

[1]A solution where an incremental change in any one variable will not lead to an improved solution.

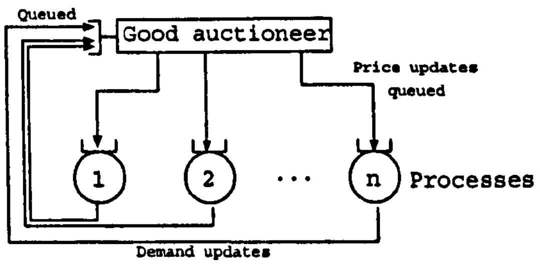


Figure 2.8: The asynchronous market for a good under the WALRAS algorithm

problem and Blue Skies economy) and simple CPU time-slice allocation. The Multicommodity Flow problem [59] is concerned with the allocation of good movements over a transport network. This is the problem encountered when trying to calculate the optimum transfer of many types of data across a computer network. CPU time-slice allocation [6] considers multiple processes consuming single CPUs time-slices.

WALRAS and tatonnement auctioning do present problems for overhead and scalability. The strictly iterative nature of the auctioneer increases the time to receive an allocation while the balance is being found. This can be overcome by allowing partial allocation (revocable) while a balance is found, but this itself introduces overhead in the allocation of the resources. As an example, a CPU time-slice allocation written in LISP [6] had an overhead greater than ninety percent. A key concern with distributed systems is spatial scalability. In a large distributed system it is not efficient to have consumers on each node considering goods in all markets, since coordination and communication are the limiting factors. Instead, incomplete information and locality must be used within the markets in order to ensure scalability.

Stoica's work

Stoica et. al. at Old Dominion University have taken auction based work and applied it to scheduling in parallel computers [27]. The parallel machines considered are MIMD with a fast network. Each processor can run a separate process or, more usually, groups of processors run tasks consisting of multiple processes cooperating to achieve a particular result.

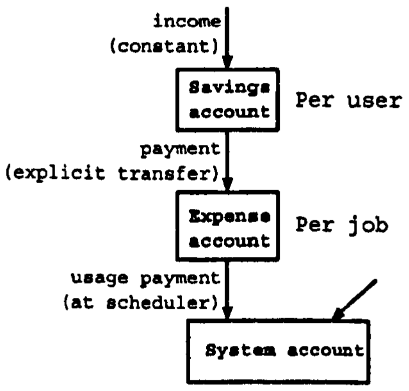


Figure 2.9: Monetary flow in Stoica's expense account work.

The problem being tackled is *fair* use of the machine by scheduling multiple tasks across the appropriate processors. The importance of a job is expressed by the amount spent by the *user* on the task. Each user is given a *savings account* into which a wage is paid. Each time a new job is created by the user, an *expense account* is created into which money for the task is deposited. The funds in the job's expense account are taken by the scheduler to pay for resources for the job to run (see Figure 2.9). Thus, the consumption of resources in the system is directly related to the wages paid into the users' savings accounts. To prevent any user disrupting the system each savings account has a maximum limit, above which wages cannot increase its size. Otherwise, had a user been "on holiday" for a period, he would afterwards be able to purchase prolonged exclusive access to the system's resources.

When a task and its expense account are created, they are submitted to a ready list along with an expected duration and the number of processors to be used. When a processor becomes idle the scheduler scans the list and selects the job with the best offer for the current number of idle processors and takes the payment.

There are two approaches to selecting the best offer from the list. The first is to recalculate the best offer on the list every time-slice and schedule the highest member. This approach has two problems: (a) a high overhead is incurred with the recalculation and in reloading all the processors; and (b) a task does not know how much it needs to pay in order to complete, since the price it must pay varies over its run-time. The alternative is to guarantee the price paid by allowing the process to execute until the end of its stated duration. At the end of the period the current executing task is given *first right of refusal* to continue, if it can continue to pay the current price (next selectable offer). The second approach has been chosen by Stoica et.al. The scheduler also takes account of the time processors have been idle, when a task has to wait for enough processors to become available, since a perfect allocation does not exist.

Stoica's system is suitable for batch parallel processing, but it does not suit general-purpose distributed platforms. The lack of distribution of multiple prices prevents scalability of the system, while the guaranteed minimum periods of each task make the system unsuitable for interactive tasks due to the poor response times that can be delivered.

2.3.3 Cooperative systems

There are two types of economic resource allocation; *price-directed* and *resource-directed.* Price-directed systems are like those previously described where resource prices exist across the system and the demand of consumers selfishly purchasing these resources drives the prices towards an equilibrium. Resource-directed systems are those where participants consider the marginal value to themselves of the resources they do not yet have. In this way, they are not concerned solely with prices but the increased advantage through obtaining the resources. Those participants with an increased utility from the resources receive the additional resources and vica-versa. Participants thus cooperate for resources through the consideration of their personal utility.

Kurose et. al. have applied this technique to two problems: distributed file allocation [31] and channel access policies in multi-access networks [30].

Distributed file allocation

The distributed file allocation problem involves finding the optimal layout of files and file access to a set of files stored across a distributed system. Access to file data x_i in a file of size K, on each node has a probability defined by a Poisson distribution parameter λ with average service time μ plus communication costs C_i over N nodes [31]. This gives a

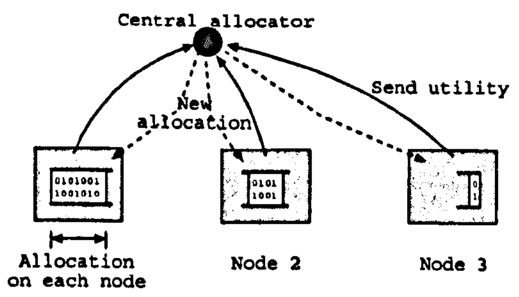


Figure 2.10: One iteration for refining the allocation in the distributed file allocation problem by utility functions

function U for having x_i of the file of each node i, this is the node's *utility*:

$$U = -\sum_{i \in N}(C_i + \frac{K}{\mu - \lambda x_i}).$$

Maximising this function produces the best allocation of the file across the distributed system given the model. By deriving the first or second derivative of this utility, each node can derive its *marginal utility* starting from an arbitrary feasible allocation. These derivatives are then sent to the central allocation agent which adjusts their resource (quantities of file) allocations appropriately (see Figure 2.10). This process is then iterated until a balance is found, when the increase in utility is determined to be insignificant. In this way, all nodes determine how effective their allocations are, then the central agent redistributes the file in order to improve these.

Channel access policies

In a similar way, the problem of arbitrating access in multiple access network channels can be solved. In order to make best overall use of the shared network links, there must be a trade-off between each node transmitting and staying silent, since staying silent allows other participants to use the link for performing useful work. Should a node transmit at its highest rate, the combined quantity of useful work might be reduced, since other nodes would not be able to gain sufficient access.

A set of the network links can be setup with each node considering its own performance, as before. This enables nodes to trade access (or silence), for the improved throughput of the system, in a cooperative fashion. Since the derivation of the access levels is iterative it would usually only be performed on initialisation and then at infrequent intervals.

Summary

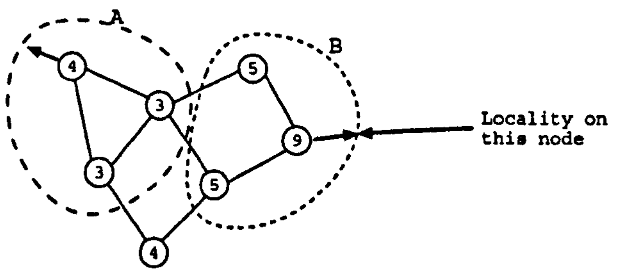
This technique provides a good model for the deviation of *pareto optimal* solutions to resource sharing problems. Unfortunately they suffer from both high overhead and poor scalability. The iterative process of deriving the solution has a very high overhead which can only be reduced by seeding the algorithms with rough solutions as the basis for refinement. The use of centralised allocator agents to determine new resource weightings prevents scalability, although systems such as channel access can provide allocation on a locality basis thereby increasing the scalability.

2.3.4 Money represents the power of the resource

In the previous systems the process has been the unit for monetary responsibility and thus money is spent by or on behalf of the process. An alternative view is taken by Clarke et. al. at Trinity College, Dublin [15]. They give the money to the resources and allow the resources to compete for and buy work. This is the inverse of the previous arrangements, where resources were in scarce supply and were competed for by processes.

The work centres around the *law of one price* [9], an economic idea that a product should have one price across the system. This differs from systems with nearest-neighbour information dissemination, that have the price of a product such as CPU varying across the system, since not every participant has access to all the information (see Figure 2.11). The use of one price is not realistic in real-world markets, since it does not reflect the variations in supply and demand due to location. Instead, the money available to each resource (consumer) merely affects how much demand they can exert.

In the computer situation this idea was used to represent the problem of resource allocation where the price of computation is constant across the system. With the price



Here prices in A are lower than in B, yet are offering the same commodity (CPU processing). The demand for the commodity determines price when supply is constant. when supply is variable both supply and demand determine local prices.

Figure 2.11: The price of the CPU commodity varies over the system, dependent on local demand

of work fixed, it is the quantity of money used by resources to buy work which determines the supply of the resource. This has the effect that the actual price of a unit of work is not important. Instead, the performance of each resource is denoted by the amount of money given to it. Thus the price of a unit of work can be fixed arbitrarily at one.

The problem tackled was balancing the load of a parallel make application. The product being sold is compilation of bytes of source code, with each node acting as a compilation resource buying source code to compile (see Figure 2.12). Each compilation node is allocated money to represent its speed. Since the system may be used for other purposes this can be considered to be a more general measure of each node's performance. The absolute amount of money given to each node is not really important, since it is their comparative levels which determine the load balance. However, since the product being consumed has a fixed price (one in this case) it is important that there is enough money in the system to enable all the source code to be brought. Likewise, if the level of money is far greater than the cost of all the source code, then the load may not balance, since a node could buy more than its share. So, while the absolute level is not important, the amount of money must be approximately equivalent to the amount of work to perform.

The source code is then shared among the compilation nodes according to their demand to perform work. Files come in integral units which means that the sharing among nodes has to be the best-fit.

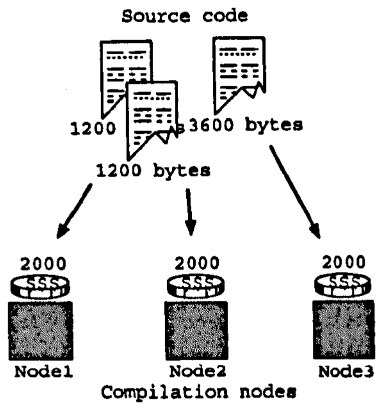


Figure 2.12: Multiple compilation nodes buy bytes of source code to compile.

The difficulty with this approach is that the market is a *closed economy*[2]. Prices are fixed to allow resources to compete for the goods dependent on their supply. This presents two problems. Firstly, the amount of money to distribute and its distribution is unknown. Secondly, when any new work enters, the system requires a centralised alteration in money supply to *all* resources and this is inherently unscalable. Also, in the distributed file allocation problem, scalability presents difficulties since all work comes in integral units which must be distributed using a 'best-fit' algorithm.

2.3.5 Summary

The economic and market-based systems discussed above address the task of resource allocation in various ways. Strictly economic systems [21, 12, 27] are interesting from many points of view, but for several reasons impose an excessive overhead if they are used in a real distributed system. Firstly, the mechanisms used for deriving pricing have high overheads, whether they are sealed-bid, second-price auction or a tatonnement system. Secondly, these systems impose a rapidly increasing overhead, when scaled by either the number of processes or the number of nodes, as contention in the auction increases.

Cooperative economic systems provide a simple technique for 'like' processes to find their maximum utility in an arbitrary system. Yet these systems fall short in a general-

[2]One where no money enters or leaves the system. All money is totally recycled.

purpose environment when the tasks are neither alike or guaranteed not to exploit co-operating processes. They also have overhead problems due to the time taken to find a balance.

2.4 Resource information

The need for knowledge about resource usage is common in distributed systems. Without this knowledge it is impossible to balance resource allocations. There are three aspects to resource information: how is it obtained; how it is measured; and how it is used. Measurement is an important part of the problem, since without accurate measurement of each resource any algorithms based on this information will be equally inaccurate. Most interesting, from a distributed resource allocation point of view, is how the information is obtained in terms of overhead, timeliness of information and scalability. How it is used is interesting given the process-centric proposal of this thesis.

2.4.1 Measurement

Knowledge of resource usage is common in distributed systems and is the heart of *load-balancing* systems. The knowledge is acquired and used on the behalf of the process by the operating system. It provides the basis for any current resource information to the process, though usually the process does not access the information itself.

Load statistics are the most common measurement of resource usage in operating systems. They use a benchmark which produces an abstract *load* of each node, which is used to judge its relative usage.

Run queues

Since most operating systems only control access and thus consumption of the CPU, it is not surprising that CPU information is the basis for most load statistics. The main metric used is CPU utilisation, which is reflected indirectly in the processor's 'run queue' length. Simple load statistics report the average run queue length over a particular period. Systems like UNIX record the average 'run queue' length over the last 1, 5 and 15 minutes to provide historical information.

Such 'run queue' load statistics are common in operating systems [18, 32] allowing a general view of the usage of the CPU resource. Many load-balancing systems [36, 47, 28, 16, 26] use this statistic in order to distribute work over a distributed system.

The problem with 'run queue' length is that it only refers to one resource and requires historical records to make predictions. Therefore it represents just a particular piece of information and not the whole picture.

Other statistics

Specific statistics can be collected for other resources such as network, disk and I/O bandwidth. Virtual memory systems [3] for example usually keep statistics for page faults (both hits and misses), active shares, page-ins and page-outs. This type of information is rather specific and low-level, but reflects the underlying usage.

These measures are problematic for two reasons. Firstly, the information is not consistent between implementations. While the information is available from each kernel, the system interface (be it system call, device or shared memory) is not consistently available between implementations and when available semantics differ. Secondly, deriving a useful statistic from the complex interactions is difficult. No common statistics have been agreed among vendors or researchers.

2.4.2 Information dissemination

Distributed systems can be both large and complex. They have to cope with communication delays, tolerate faults and be scalable. Communication delays make it impossible to obtain comprehensive information about the state of the rest of the system. Firstly, the communication delay itself means that any information obtained is out of date by the time it arrives.

Secondly, for large systems, it would take a relatively long time, obtain a global state of the system. This would require global synchronisation of all nodes in the system. For special-purpose systems with centralised code sections this can be done, since load information can be found at the same time as the algorithm is synchronising. But in a general-purpose scenario, where the coordination is required regularly, the load may be too high.

Thirdly, the additional network and CPU load caused by disseminating the information should reflect its importance. Thus, in a highly loaded system the information transfer should consume less resources than in a mostly idle system. This is especially true when information is not sent over the entire system. When lightly loaded, information should travel further, using more resources, than when the system is heavily loaded.

Given these problems, several techniques have been developed for transferring information, usually load information, around a distributed machine in a scalable manner.

Nearest neighbour

Distributed systems can often be considered as mesh networks, where each node has a set of neighbours to which it is connected and can transfer information. *Nearest neighbour* is a technique used in distributed systems similar to that used in network routing [5] where routers exchange routing vectors. The actual transfer of information can be either periodic or aperiodic dependent on how rapidly the information is expected to change. Periodic transfers impose a load on the system even when the information has not changed, by re-sending duplicate information. Aperiodic transfers require some notion of sufficient change in the information before updates are sent. Potentially this can result in sending more out of date information; or, alternatively, it can lead to large quantities of updates if information is changing quickly. The common solution is to use a hybrid system where updates are periodic, with aperiodic changes when the information changes sufficiently.

A simple example of this technique can be seen in Figure 2.13. In this iterated version, the information traverses one network link each time step. Thus, over time, the information

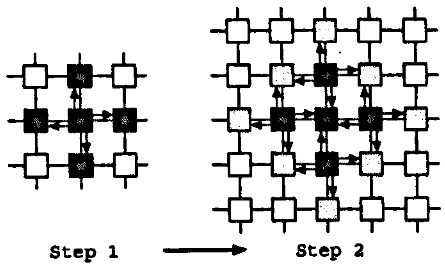


Figure 2.13: Simple iterated information transfer using nearest neighbour meshes

spreads from the locale of the originating point. The problem with this technique is that it can impose a high network overhead. Firstly, because transfers are not limited, the information must traverse the entire system. Secondly, multiple transfers can continue at the same time while the previous information transfer finishes. A variation of this simple version places an upper bound on the hop count of each piece of information. In this way the amount of information in each network traversal has an $O(n_{hops}^2)$ upper bound where n_{hops} is the maximum number of hops.

Nearest neighbour dissemination is often used in distributed systems of all types. Its uses range from: routing vector exchange [5], scientific algorithms [14, 37, 11, 51] and the load balancing work described in this chapter.

The problem with nearest neighbour as a method of disseminating information is that the distance information travels is not dependent on its importance or the load on the system. Distance travelled can be tied to importance by using the maximum hop count as a measure of importance, but this does not take into account the load on the system, however. So that the performance of the main computation can be affected by this information transfer.

Other approaches

Many network topologies are richer than a simple mesh, thereby allowing broadcast to locally connected nodes. A broadcast packet is one which is received by all nodes it passes by. In bus-based topologies, this means that all nodes on the same bus can receive the same information with a network load of one packet. This is a highly efficient technique for transferring information to physically connected nodes.

The problem with this technique is that bus topologies are inherently unscalable, due to electrical timing and power considerations. One solution is to use *multi-cast* over non-broadcast networks. Multi-cast packets allow a packet to be addressed to multiple recipients, rather than broadcast's *all* recipients, thereby reducing bandwidth requirements. The problem still exists that under either mechanism, the distance information is transferred is not dependent on its importance, or the system load. In particular the broadcast still involves sending the information to *all* nodes in the system.

A common solution is to perform a *subscribe-announce* policy using multi-cast. In this way, clients (*subscribers*) subscribe for information which is announced in a multi-cast. This solution has a low overhead as long as subscribers are close to announcers and the number of subscribers is relatively small.

More controlled locality can be achieved by using a *hierarchical* approach. On each level information is distributed by the parent to its children and may be forwarded to its peers via its parent. At each level a process determines when the information is distributed, and whether it should be distributed further afield by forwarding it to its parent. This has the effect of minimising the responsibility for sending the message, imposing structured locality and hence controlling the spread of information in the system.

The problem with this approach is that the structure of the hierarchy must be explicitly configured by somebody. Thus, the level of locality does not depend on the load on the system, but instead on an explicit arrangement, be it physical or virtual.

2.5 Contemporary work

Towards the end of the time spent working on this PhD some similar work has been done elsewhere, particularly in the field of proportion-sharing. This section briefly discusses this contemporary work and how it relates to the work reported in this thesis.

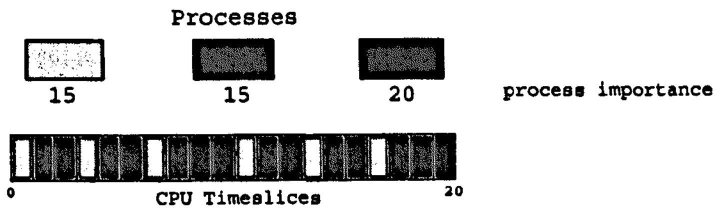


Figure 2.14: Proportion-sharing of 20 time-slices between 3 processes

2.5.1 Proportion-share scheduling

Proportion-sharing is the means used by resource servers in the ERA architecture to allocate resources. The ERA architecture does not impose any particular type of proportion-sharing on these servers, but advises the semantics which should be used for sharing resources. Proportion-share scheduling allocates a process access to a resource in direct proportion to it importance in the system. Each process has an explicit importance which represents quantitatively the process' right to resources in the system. The magnitude of the rights does not imply an absolute amount of resources, as in real-time scheduling, but instead a proportional share of the resources. An example of a proportion-share allocation can be seen in Figure 2.14.

Proportional-sharing of resources was developed for example servers in this thesis, while the contemporary techniques discussed below were being developed. This contemporary work is largely based on techniques developed by Carl Waldspurger at MIT [57, 56]. He introduces two algorithms for proportion-sharing, primarily of CPU, on a uniprocessor. *Lottery scheduling* uses the concept of random lotteries, while *Stride scheduling* deterministically selects which process has access to each time-slice.

Much of the contemporary work focuses on the derivation of algorithms for allocating fair shares of a singular resource (usually uniprocessor CPU), whereas this thesis is focused on the ERA architecture, marketplace and providing process-centric resource allocation. The proportion-sharing work proposed in this thesis differs from this contemporary work, since it is focused on the derivation of shares with low overhead rather than accuracy and with the interrelation of scheduling with the allocation and derivation of price.

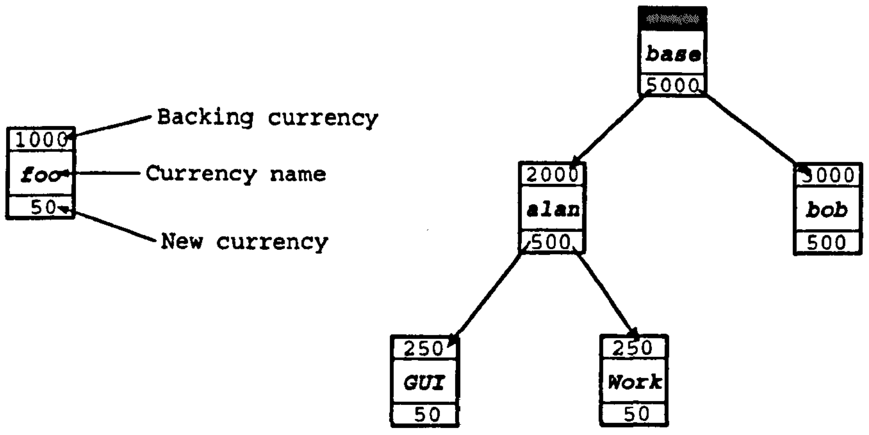


Figure 2.15: Lottery scheduling: currencies being subdivided into new currencies

Lottery scheduling

Lottery scheduling is based on the idea of a *lottery* where each process holds a number of lottery *tickets.* From the number of lottery tickets available, one winner is picked at random. Unlike commercial lotteries, tickets are not used up once access is granted, they may be reused, but they may only be used for one resource at a time. Thus a process with more lottery tickets, in the long run, wins more lotteries. Over time, each process will win lotteries according to the number of tickets it has.

The number of tickets each process has, in the entire system, is controlled by transferring tickets and ticket currencies. Tickets are first-class objects and they can be passed between objects as part of a resource policy, or used to transfer priority to another process performing work on a process' behalf. Ticket currencies allow construction of groupings of types of tickets. A currency provides a new ticket at a rate compared to tickets in its base currency. Thus 1000 base type tickets may be reflected in a new ticket currency 'foo' as 10 tickets, each worth 100 base type tickets. This allows the proportion-sharing between clients using 'foo' currency tickets to be altered without affecting others. If the exchange rate used for currency 'foo' is altered, all clients using foo receive a smaller share compared to others, while increasing the total number of base tickets reduces all the proportions of those currently holding tickets. These currencies can be formed into a tree structure all derived from the base type. An example currency tree is shown in Figure 2.15.

To deal with partial allocation of resources, such as fractional or variable size quanta, *compensation tickets* are issued temporarily to clients in proportion to their usage. Under-usage results in positive compensation tickets, while over-usage results in negative tickets. These then affect following lotteries to compensate for the earlier unfairness.

With the added complexity of proportion-share scheduling, overhead is an obvious consideration. Lotteries require random number generations per allocation, plus a lookup of the winner. Even with tree structured state and quick random number generators the overhead for reasonable numbers of processes can become significant. Waldspurger claims an overhead of approximately 5 percent, on top of traditional scheduling overhead, when five processes perform simple database lookups.

Lottery scheduling also suffers from large fluctuations in allocation. Since the access selection is random, for small numbers of allocations the variance from the ideal allocation can be large. The variance for lottery scheduling is $O(\sqrt{n})$ where n is the number of allocations performed.

Stride scheduling

Stride scheduling reduces the variation in allocation and the overhead of lottery allocation by deterministically allocating access to ticket holders. Taking *rate-based flow control* as its basis, stride scheduling aims to calculate the time interval (*stride*) which a process must wait between successive allocations.

For each process, the scheduler keeps: *the ticket allocation, the stride and the pass.* The stride is derived as inversely-proportional to ticket allocation, the pass indicates the virtual time which the process is currently at. Allocation is simply a matter of selecting deterministically the process with the lowest virtual time. Once allocated, the pass is incremented by the stride. Thus, a process with twice the ticket allocation will receive half the stride of another process, which will lead to it receiving twice as much access to the resource being scheduled (see Figure 2.16).

As with lottery scheduling, dynamic operations such as new processes and ticket changes

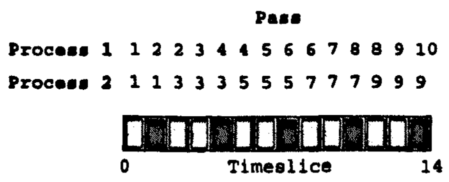

Pass
Process 1 1 2 2 3 4 4 5 6 6 7 8 8 9 10
Process 2 1 1 3 3 3 5 5 5 7 7 7 9 9 9
0
Timeslice
14

Figure 2.16: Stride scheduling: using strides to allocate twice as many quanta to a process with twice the tickets

also need to be taken into consideration, but these are not dependent on the scheduling parameters. Since extra state must be kept by the scheduler there is a corresponding increase in work to be performed when dynamic changes are required, such as storing the remaining proportion of the process' stride. Non-uniform quanta can also be handled this way.

Due to its deterministic nature stride scheduling has a much reduced allocation variance. Ideally the variance is at most 1, but with a skewed ticket distribution the variance can be $O(n_c)$ where n_c is the number of clients.

Charge-Based Proportional Scheduling

Maheshwari [35] at MIT created *Charge-based proportional scheduling* based on Waldspurger's work on Lottery scheduling. As with other ideas in this thesis, this approach improves on the work of Lottery and Stride scheduling by adopting similarities with Bresenham's [8] line drawing algorithm. They use integer arithmetic to approximate a continuous gradient line in the field of discrete pixels on a screen.

This method can be applied in n dimensions to the problem of proportion-sharing. This work falls between the proportion-sharing work of Waldspurger and my own work [1]. The result is a simple and deterministic means of achieving proportion-sharing without the large variances which can occur with lottery scheduling.

Stoica's work

Besides his work on parallel scheduling on MIMD machines, Stoica and Abdel-Wahab have undertaken work on proportional-share allocation. Two different approaches have been proposed: Deterministic Selection [52] and Earliest Eligible Virtual Deadline First (EEVDF) scheduling [53].

The Deterministic Selection method approach takes a simplistic approach to the problem. Clients are selected in proportion to the amount paid as per Lottery or Stride scheduling. Instead of probabilistic or stride methods for calculating which client to select next, this work proposes using a cyclic count and reversing the bit pattern in order to generate a pseudo random sequence. This random sequence is then used to select the next client to run to produce an allocation in proportion to the money held by each. The method is deterministic and has lower overhead and less variance than Lottery scheduling.

This method was improved on with EEVDF. It uses the concept of virtual time, like Stride scheduling, but improves on the technique. Each process has a weight and a requested duration and this combination is used to calculate both the next eligible virtual time when each process should run and also the current increment of virtual time. The system keeps a list of (Time, TotalWork) pairs for each process. These hold the next eligible time for the process and the relative amount of work it has to have received by that time. The current virtual time increment is calculated as $1/sum(w_n)$ where w_n is each process' weighting. This means that processes with a higher weighting receive fewer time slices.

Virtual time is incremented by the current virtual time increment and the process at the head of the list of pairs (ordered on virtual time (deadline)) is selected. Once run, its virtual eligible time and work are updated to find the next eligible deadline. In this way, time slices are accurately distributed amongst processes.

The Moo Scheduling algorithm

These other approaches all have an overhead of $O(log\ n)$ for most operations, since an ordered list insertion is required. My work at this time at SONY concerned the need for an even lower overhead approach for use with embedded multimedia devices and also the consideration of latency with which tasks are scheduled. Multimedia tasks are sensitive to both bandwidth (proportional-share) and latency (time to schedule). The aim of this work is to follow the simplicity motivation of Cost proportion scheduling, combined with a need to consider latency of access combined with good proportion-share accuracy. This work tries to minimise this to $O(1)$ for most cases, while including latency in the scheduling.

This approach is based around a simple cyclic buffer which represents future time slices after the current time. Each element of the buffer is a list of processes which will require that time slice. Processes are run by taking the front of the list at the current buffer position, and incrementing the buffer position cyclically when the list is empty. Each time a process suspends, or its quanta expires, it is placed in the buffer at $time + priority\ MOD\ bufferSize$ where $time$ is the current time and $priority$ is the proportion-share parameter of the process. This means processes with a low share parameter receive a large share and vice-versa.

Besides this share parameter, each process has a latency parameter. When a process resumes after waiting more than *priority* time slices, it enters a *latency queue* which is ordered by the process' latency parameter. Processes in this latency queue are serviced in preference to the cyclic buffer. If the process resumes before *priority* time slices then it next executes when it would have been scheduled had it not been suspended. This means that processes with low share parameters receive high shares and that processes that have blocked for more than very short periods are resumed ordered by latency and therefore quickly.

The simplicity of the buffer approach means that reschedule, suspend and resume operations are all O(1). By using clever buffer management means that other operations are O(1). This reduced complexity makes this well-suited for embedded multimedia systems where processing resources are scarce.

Summary

Proportion-scheduling is useful in scheduling access to resources in general purpose systems. By adjusting the proportions given to each process, the rates of resource consumption and thus computation can be varied. Unlike earlier systems, the control is logical, controllable, fine-grained and easy to reason about. Such semantics make it useful in concurrent shared uniprocessors where providing fair access to a number of different priority level tasks is important.

However, these algorithms do not easily extend to distributed resource allocation. In distributed resource allocation scalability is important. With the use of currencies, lottery and stride scheduling must consider the centralised knowledge of exchange rates between currencies. This is true even if currencies are stored in *base* currency, since any interaction with the currency (process oriented or exchange rate control) requires this information. Likewise, if tickets are to be held by the client and reused, then the overhead of using remote servers becomes prohibitive, since remote access would be required to check ticket values.

More generally, while tickets specify the rate of resource consumption and thus computation, clients appear to have no informed control over the tickets they have or present to each resource server. In order to maximise use, in a dynamic resource environment of a distributed system, it is desirable that a process should be able to adjust such controls in response to its surroundings. The problems of obtaining, analysing and acting on this information become even more of a concern when applied to the distributed case where there is a choice of resources.

2.5.2 Continued exploration of economic computation

Wellman's latest work

Wellman et. al. at the University of Michigan are continuing their work on solving distributed scheduling problems using microeconomic principles [61, 60]. Previous work using *Tatonnement* style market equilibrium has been followed up by work on traditional auctions, such as Dutch and Vickery auctions. These enable equilibrium prices on goods to be found more easily than by the tatonnement approach, especially in a multiple good market.

The work has also been applied to the field of Agents and Artificial Intelligence, where solving distributed problems in "human" ways is popular, while the overhead of achieving the result is less of a consideration.

This work is closer to the work of this thesis than Wellman's earlier work. Processes are allowed to be resource aware, but the system does not address the other problems present in such a system, such as the overhead of deriving auction prices, or the need to control the very resources which the process is using in order to execute, or the scalability of such a market, or the need for working with incomplete information.

Matt Welsh's interest

Matt Welsh, of Linux fame[3], now works for the Nemesis project at Cambridge University. His work there is about supporting networking and distribution of the Nemesis operating system. As a sideline, however, he is interested in the use of *Resource-aware* systems for Real-time and Interactive Parallel systems. His Nemesis work aims to provide:

- Resource management on a single node. Simple resource aware management on a single node.

- System-wide resource management. Extending resource aware management to the system wide case in a scalable fashion.

- Application support for resource allocation. Providing applications with information and how they can use it.

These are the areas which the ERA system has already explored. Novel new single node and multiple node resource policies are demonstrated along with the low overhead and scalability of the system. The ERA framework, marketplace and process-centric applications describe and demonstrate the provision and support for application resource allocation.

[3]Matt Welsh was a figure in the Linux movement and wrote the book 'Using Linux'

The existence of this work nicely verifies that the goals of ERA are common among emerging operating systems and applications. Its also nice to know that others believe these goals are worthwhile.

2.5.3 Beneficial use of resource information

Resource information, once obtained, must be processed in order to decide about the best allocation. Traditionally, this processing is performed either by a human operator or by the operating system/third-party. As described in the Sections 2.2.2 and 2.3 many different approaches have been taken on how best to process the information. This thesis advocates that the process itself can use the resource information to make informed decisions about which resources to use.

Processes, when allowed to process the information themselves, have the potential to act on the resource information combined with their operating needs so as to use the resources to achieve their goals rather than simply balance the load. Some contemporary work on application specific resource allocation has been reported [46].

Ranganathan et. al. [46] propose that network-based programs in a message-passing environment can make use of network latencies in order to improve their locality and hence reduce response times. The programs themselves access the monitored latencies between nodes and use this information in order to migrate to a position where latencies are minimised. In their experiments, an interactive talk program is simulated which tries to place itself dynamically on a node which minimises the latencies for current participants. Thus, when a new user joins, the locality may be affected (see Figure 2.17.). This locality is impossible to determine by simple load balancing since the relevant statistic for this application is network latency rather than CPU load. This example shows how improved performance can be gained by something more sensitive than blind load-balancing resource allocation. This contemporary work hints at the flexibility that a process-centric approach can bring through the use of both resource information and application specific resource knowledge.

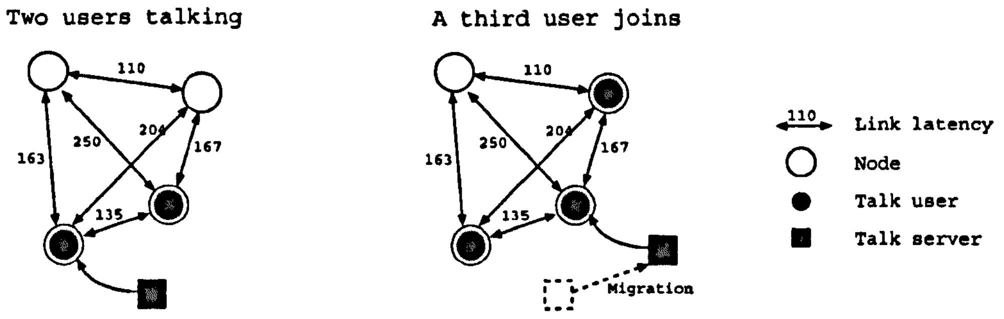


Figure 2.17: Latency migration as a new user joins in the session

2.6 Summary

Single node computer techniques such as Priority Scheduling and Bounded Consumption provide crude control over the execution of processes both in terms of consumption over time and total consumption. A multi-user scenario with a diverse set of jobs, can lead to unequal resource access and consumption with equal priority users. However, these technique's simplicity does produce low-overhead systems which can scale linearly.

Extending resource allocation to the distributed case introduces three problems: scalability, out of date information and incomplete information about the load on the system. The simplest solution, such as PVM, ignores these problems and allows processes to make blind placements of resource usage on the system. This solution does not perform well, when several tasks execute on the distributed system, since each is unaware of the other as competition or the possible availability of alternative idle resources. Virtual processors help alleviate this blind placement of programs on resources by adding a layer of abstraction at load time.

Operating system controlled resource allocation overcomes these problems by optimising the local cases (or wider cases with the use of heuristics). By using nearest-neighbour information, scalability can be maintained with an additional overhead limiter, such as a hop count. This has three side-effects: information is by definition a fixed subset of what is potentially available; there is no problem with out of date information, since the information is obtained directly; the load imposed by information dissemination is not related to the system load and can affect the speed of overall computation. Various algorithms, mostly in the field of load balancing, have been devised for using this load information to decide on how best to allocate the load. Operating systems based systems know the competition and the quantity of the load, but nothing else, and can thus only balance loads according to generalisations. But, the application's specific needs vary dynamically and concern different complex parameters per resource, such as access patterns, access latency, etc. It is difficult to express, this information dynamically to the operating system.

Economic and market-based systems solve the problem in various ways. Strictly economic systems, while interesting from many points of view, impose an excessive overhead in a real distributed system. There are several reasons for this. Firstly, the mechanisms used for deriving pricing have high overheads, whether they use sealed-bid, second-price auction or a tatonnement system. Secondly, these systems impose a vast overhead, when scaled either by number of processes or number of nodes, as contention in the auction is increased. Cooperative economic systems provide a simple technique for 'like' processes to find their optimal compromise for an arbitrary system. Yet these systems fall short in a general-purpose environment when the tasks are neither alike nor guaranteed not to exploit cooperating processes. They also have overhead problems due to the time taken to find a balance.

Contemporary techniques supporting *Proportion-sharing* have shown it is possible to fairly share single resources. However these techniques have some problems: they are not scalable to more than one node due to the centralised nature of expressing resource rights; they place control of resource rights outside the reach of the process, leaving a passive system; and they concentrate on accuracy rather than overhead.

Chapter 3

Overview

3.1 Introduction

Hardware and software are changing their relationship as computer technology advances. Hardware is a plentiful resource compared with a few years ago, with large quantities of workstations now commonplace in most computing environments. With the wider adoption of networks, networked workstations collectively have a similar computing power to an average supercomputer. Each workstation usually has its own user, yet most of the resources available in the system remain idle most of the time due to the dynamic demand for resources from each user.

At the same time, software is becoming increasingly complex, placing increased load on the resources of the user's workstation. This increase in software load comes from two directions. Firstly, the increase in size and complexity of application programs combined with the increase in the size of the datasets being processed. Secondly, new types of applications are becoming widely used. These new applications include: database processing, data mining, multi-media and network software itself. From the increase in software and dataset size, combined with the increased availability of hardware, comes the idea of using the machines collectively to get the job done. This is the field of distributed systems of networked workstations.

Typically, the users of such a network of workstations present a complete spectrum of potential uses and thus the distributed system needs to act as a general purpose computing platform. i.e. be capable of executing a multitude of types of application without specialisation or reconfiguration. A user at each workstation should be able to execute programs as if they were being run on the local workstation. The size of the distributed system used should not affect the performance adversely as it is scaled, while increases in resources in the system (network bandwidth, number of nodes or node speed) should be utilised efficiently as and when they are available.

Since the resources are shared, each user's personal resources can be affected by the processes of other users in the system. To limit this, it should be impossible for a process to consume all the resources in the distributed system or on a particular node, unless there is *no* competition for those resources. Processes executed should make best use of all resources, dependent on their functionality, and their resource usage should alter dynamically as system conditions and configuration change.

Such distributed systems are only now becoming a realistic technology. With the techniques outlined in the previous chapter, the resources of a network of workstations can be harnessed. Both computation and data can be distributed over the system and the resultant computations can be coordinated to achieve the desired result. In environments where the resource demands of each process can be treated as generic, load balancing systems can share a dynamic workload across the system with varying degrees of optimality depending on the assumptions made. Systems such as Parallel Virtual Machine (PVM) [4, 24], using virtual processors, allow the structure of the distributed computation to be mapped to specific hardware configurations.

Problems still exist, however, before networks of workstations can be accepted and used as general purpose computing platforms. In this chapter the requirements for distributed resource allocation in such systems are outlined. The problems that remain are then considered. Finally, the proposed new resource allocation architecture to solve these problems is introduced.

3.2 Resource allocation needs

For a network of workstations to be accepted as a general purpose computing resource it must satisfy certain requirements, from both performance and programming perspectives. From a programming perspective it is important that the resource allocation system provides three properties: Fairness in access to resources; Liveness (freedom from starvation, deadlock and live-lock); and Uniformity of access to resource allocation. Also since application performance is frequently an important issue, resource allocation and access must be: Appropriate to the application using the resources; Responsive to changes in the system's loading or configuration; Scale with the number of processes, users, services, nodes in the system; and these must be provided at an acceptably low overhead in resource consumption.

3.2.1 Fairness

In any multi-process system whether it be multi-user or simply multi-process, there is competition for resources between executing processes. In cooperative systems, this is not a problem since each process can show consideration for the resource needs of others. This is not true in the competitive systems considered here, where a process can make as many requests for resources as it wishes and it is entirely possible that the total number of resource allocation requests exceeds that which is available. This poses a question of fairness as to how the resources are allocated among competing processes. Resource fairness comes in two forms: *consumption fairness* and *spatial allocation fairness.*

Consumption fairness

Consider the allocation of CPU time-slices. Allocation here is by consumption alone, since there is only a temporal consideration. From a consumption point of view, the scheduler has two considerations: which processes are runnable and the importance of those processes. Two equally important and runnable processes should, by resource *consumption fairness*, consume equal shares of the CPU over time (see Figure 3.1).

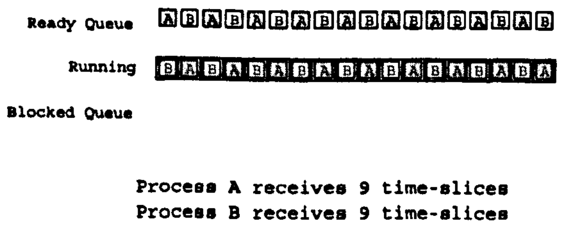

Ready Queue
Running
Blocked Queue
Process A receives 9 time-slices
Process B receives 9 time-slices

Figure 3.1: Two equal priority processes receiving equal shares of a time-sliced CPU

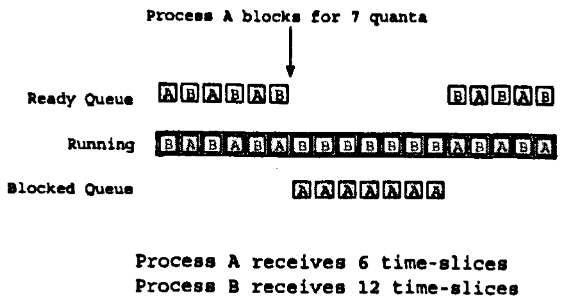
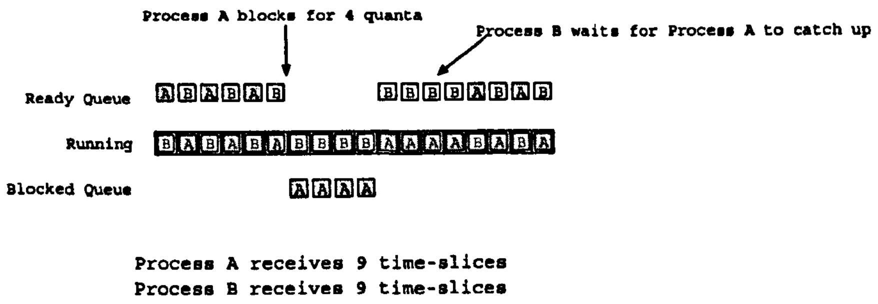

Process A blocks for 7 quanta
Ready Queue
Running
Blocked Queue
Process A receives 6 time-slices
Process B receives 12 time-slices

Figure 3.2: Two equal priority processes under instantaneous fairness. The non-blocking process consumes more in the long-term

The situation is more complicated when demand for the resource varies over time among processes. When a process is suspended, its demand for CPU becomes zero. In this scenario, there are two ways fairness can be interpreted. The first is instantaneous fairness, where fairness is only applied between currently competing process, with no consideration for long-term fairness. If we again consider the scenario of the two equally important processes competing, then with instantaneous fairness, when one blocks the other gets full access to the CPU until the other resumes. Once resumed they compete on an equal basis (see Figure 3.2). This of course allows the non-blocking process to receive a higher share than the blocked one, which implies longer-term unfairness.

Alternatively, long-term fairness can be considered, by allowing the resuming process an increased importance to allow it to catch up on its consumption while blocked (see Figure 3.3). While this maintains fairness in the long term, it imposes a large variance on the bandwidth and response time received by processes over time, as others block.

What is required is a short-term fairness correction, such that processes receive an effective short-term improvement to permit good response-times for low latency tasks such



Process A blocks for 4 quanta
Process B waits for Process A to catch up
Ready Queue
Running
Blocked Queue
Process A receives 9 time-slices
Process B receives 9 time-slices

Figure 3.3: Two equal priority processes under long-term fairness. Both processes receive equal shares, but suffer higher variance in short-term fairness

as interactive jobs, while in the longer-term fairness is maintained. In this way variance in latency to access is minimised and fairness maintained.

CPU Fairness is usually the only form of fairness considered by a modern operating system since, without it, all resources can be accessed excessively. With modern applications, which impose a high consumption on a variety of resources, fairness involves the use of other resources in the system, such as memory and network.

Spatial Allocation Fairness

Memory poses an interesting problem since it has potential for both quantity and consumption fairness, since at any time there is a finite amount of backing storage (virtual memory) with a finite amount of real memory to access it. The real memory consumption problem is similar to the consumption of CPU problem. The problem of *spatial allocation fairness* is more interesting.

Consider a simple two-level memory hierarchy of real memory and disk. In this arrangement, real memory acts as a paged cache for the larger disk backing store. There are two potential sources of unfairness here. First, consider a program which allocates all the available memory to itself as either a denial of service attack or accidentally through a bug. All other processes are immediately denied any new allocations, even though they may be more important in the system than the attacking process. Here the potential for memory allocation on backing store is not associated with the process' importance in the system.

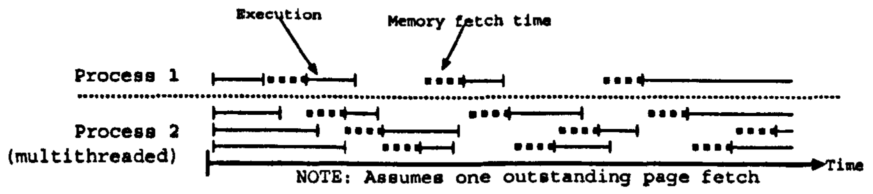


Figure 3.4: Process 2 has a low importance and exhibits poor locality, yet receives three times as many page fetches, evicting pages of Process 1 and thus slowing it down

Secondly consider a process allocated a reasonable quantity of memory in backing store and which consumes most of the paging bandwidth (again either deliberately or accidentally) by having poor locality. As the process accesses memory it evicts data from real-memory used by other processes. While the other processes can evict the pages of the process with poor locality, they can have their performance hit by the increased page times due to the reduced temporal locality times caused as a result. This is especially true if code is evicted too. The poor locality process could naively suffer low CPU consumption and as a result leave an increased share of CPU to the other processes. But, alternatively, if the process is multi-threaded it may be able to increase its CPU consumption too by allowing multiple outstanding memory fetches to be pipelined (see Figure 3.4). In either case the process is able to consume more of the memory bandwidth than its importance in the system warrants, thereby affecting the performance of other processes.

Bounded Consumption and Priority Scheduling [19] techniques used to provide control over access to resources (for example CPU) do *not* support fairness to any extent. Instead they provide an at best crude and ad-hoc control of the consumption of resources. Proportion schedulers such as lottery [57] and stride [56] provide the necessary fine grained support for fair consumption of resources. The proportion of shared resources consumed can be dynamically controlled to a good degree of accuracy. Unfortunately, due to the centralised nature of converting tickets to their base currency, the lottery and stride scheduling techniques would not scale well as the system size is increased. Controls such as those supplied by the microeconomic systems, described in the previous chapter, only provide this bounded consumption at a large overhead due to the use of sealed-bid, second-price auctions for price setting.

3.2.2 Liveness

An important criteria for resource allocation is freedom from starvation and deadlock. Deitel [19] states that deadlock in resource allocation can be effectively prevented by breaking at least one of the following four rules of resource allocation:

- Mutual Exclusion. Only one process can use the resource at one time.
- Hold and Wait. A process holding one resource waits for other resources to be freed by other processes.
- No Preemption. A resource can only be released voluntarily.
- Circular waiting. The head of a chain of processes can be waiting for the tail to release a resource.

Freedom from starvation is also very important since, while a process may potentially be able to obtain the resources or access to resources it requires, it should not have to wait indefinitely. Priority Scheduling [19], for example, can result in such starvation.

3.2.3 Uniform access

If all resources are to be controlled then it is desirable to provide a uniform application interface for the allocation and acquisition of resources. Traditionally, allocation and acquisition of resources may be performed in various ways. For example, consider UNIX [3], where there are three different API forms for the allocation of CPU, Memory and Network access. CPU allocation is performed with calls to threads and fork(); while memory allocation uses malloc() and free(); leaving network with calls such as bind() and socket(). If these resource interfaces were to be extended so as to include expression of the importance of the resources required, then *each* interface would need to be extended separately.

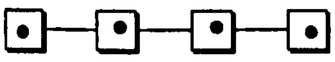


Figure 3.5: A balanced workload of 4 sub-tasks across 4 nodes

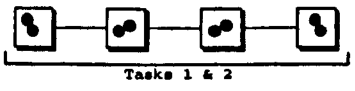


Figure 3.6: Two tasks spread across the available resources

This approach clearly presents increased programming effort as the numbers and types of resources are increased and is therefore unsuitable for providing a uniform extensible interface.

This approach clearly presents increased programming effort as the numbers and types of resources are increased. To improve on this, *Plan 9* [44, 45] takes the UNIX approach further by including access to all resources through the file system interface. This includes all network access, memory and process access. It does this by binding special namespaces into the file system.

While these systems provide a uniform interface, it can be strained for unusual devices, such as CPU. A more general and frequently used approach is to use an object-oriented model which allows standard interfaces to be hierarchically composed through inheritance. Many modern operating systems [42, 43, 10] provide their system resource interfaces in this way. It has the advantage of modularity, through inheritance, to provide standard and extensible interfaces. Problems of protection of these interfaces can be solved either through memory protection [43] or capabilities [10, 54].

3.2.4 Responsiveness and application appropriateness

A general-purpose distributed system is a dynamic entity. The system's configuration and workload change over time as resources and tasks come and go. Therefore, the resource allocation needs to change over time in order to tolerate and take advantage of the changing situation.

A simple scenario for these changes is the absence or presence of extra nodes or tasks.

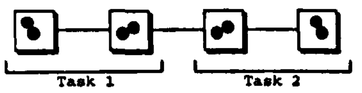


Figure 3.7: Two tasks allocated across two nodes each from the available resources

Consider the code of a task composed of four sub-tasks representing a balanced load across four nodes, as seen Figure 3.5. When another similar task arrives, the resources can be used in two reasonable ways. The first (Figure 3.6) places the resources equally across the system. The second (Figure 3.7) places the sub-tasks of each task on the same nodes. Neither placement is the definitive ideal allocation for all situations.

These examples introduce the types of problems which traditional load-balancers are poorly suited to solve, since they do not know the needs of the particular applications. If the tasks are computationally heavy then it is best to place the tasks together on as few nodes as is feasible thereby reducing the network communication required to gather the results. Likewise, if the tasks require a lot of synchronisation and coordination, then reducing network communication will improve the task's performance. If, however, the sub-tasks are memory intensive, such as a database, and are well partitioned on the data they access, then placing the work across the four nodes increases the memory bandwidth which may be exploited.

This simple example brings out two requirements. Firstly, the resource allocation needs to respond to changes in the nature of the workload. Secondly, and more fundamentally, the optimal resource allocation does not simply depend on the even placement of the sub-tasks, but depends on the operation of the sub-tasks in question, their 'functionality'.

Also, this application responsiveness does not only affect placement of resources, but other factors such as parallelism. In this example, if the best placement is one task per two nodes, then it may be better to change the number of executing sub-tasks or threads to improve performance. For example, if the process is computationally intensive it is probably better to replace the two processes with a single process. Likewise, if the processes are memory intensive then moving to a single process with many threads to perform computation would be advantageous, since some application threads can continue computation while others are blocked awaiting memory page requests.

Overall it is important that the allocation is appropriate to the application and that it is capable of changing dynamically as competing workloads change this allocation or new nodes become available.

As we saw in Chapter 2 traditional resource allocation mechanisms [4, 24, 25, 36, 47, 28, 16, 26] cannot support such responsiveness since they either make the allocation blindly [4, 24, 25] or based on operating system information alone [36, 47, 28, 16, 26].

3.2.5 Scalability and Overhead

With any system the scalability of the algorithms used is of primary importance, since it is usually impossible to predict the magnitude of the system parameters. For resource allocation these parameters include the number of processes (schedulable entities) and users. As these increase, the performance of the algorithms should degrade as gracefully as possible. At the same time, the overhead imposed by the resource allocation system should be low. This is orthogonal to scalability of the system, although most scalable algorithms are also low overhead.

Algorithmic scalability on a single node has to a large extent been tackled for resource allocation fairness problems such as CPU scheduling, resulting in $O(n)$ and $O(n \; log \; n)$ algorithms [57, 56]. Scalability of fair allocation algorithms used in distributed situations still remains to be solved.

The acquisition and dissemination of remote resource information for making resource allocation decisions requires an algorithm which is low overhead and scales as new nodes are added to the system as well as when new client processes (those using the information) and server processes (those producing the information about the resources) are added.

Traditionally, information is disseminated using a node's *nearest neighbours* [14, 37, 11, 51]. Unfortunately, this approach imposes a constant overhead, no matter how loaded the system is. The usual solution makes this overhead small by transferring little information infrequently. This means the information available at each node is small and potentially quite out of date.

Besides these problems, the fixed overhead of information dissemination when the system is heavily loaded seems inappropriate. It may be better to have the system disseminate more information when less loaded.

3.3 The ERA solution

No current resource allocation system meets all or even a large subset of these needs. Fundamental requirements, such as freedom from deadlock, can be provided by following certain design rules. A few techniques exist in some form that meet some of the other needs. Unless these needs are more closely satisfied, distributed systems will provide a poor platform for general-purpose use (multi-user, multiple application domains).

A new resource allocation architecture, called the Economic Resource Allocator (ERA), is proposed in this thesis as a solution to these problems. This architecture provides 'process-centric' resource allocation. Processes themselves are responsible for their resources and their allocation. Process-centric resource allocation differs from conventional process controlled resource allocation, such as PVM, in two fundamental ways. Firstly, a process is responsible for *all* resources it has to use, including the memory to hold its code, data and stack segments. Secondly, it is enabled to make *informed* decisions about the resources it requires during its entire run-time.

Fairness between competing process-centric processes is provided by controlling the importance a process is allowed to express towards a resource it wants. Resources are then allocated with respect to this importance so that each process can have access in proportion to the importance placed on it.

These two principles are combined in the ERA framework. On top of the framework a dynamic locality information dissemination system is constructed and several novel resource allocation policies provide improved application-specific resource allocations. The

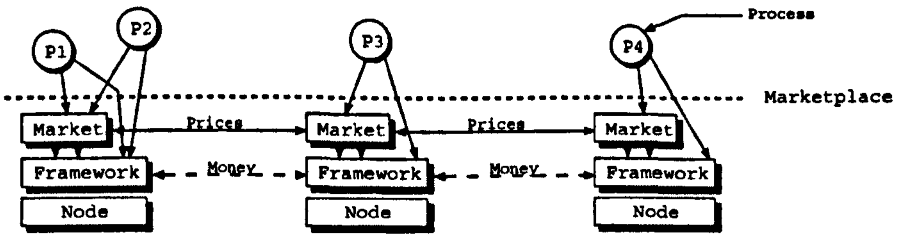


Figure 3.8: The interrelation of the three parts of the ERA architecture

design of these systems is such that they should impose little overhead and are fully scalable.

3.3.1 Overview

The fundamental idea behind the ERA architecture is a market in which goods are bought and sold. Market concepts and marketplaces have many similarities with distributed resource allocation. Past researchers have used economic ideas, including supply and demand to solve other resource allocation problems. Markets have three desirable features which we wish to exploit. Firstly, they provide fair resource allocation under competition. Secondly, each consumer can have at most a limited effect on the market because money is controlled. Lastly, they are scalable and low overhead but, as a result, it is difficult to know the price of all the goods at once. These correspond to some of the key requirements for distributed resource allocation as we have outlined them.

The ERA architecture is formed from three distinct parts. Firstly, there is the *framework* which provides a fair and yet controlled environment for allowing 'process-centric' resource allocation using market ideas. Secondly, there is the *marketplace* which provides a scalable means of disseminating information with dynamic locality about the resource environment. Lastly, there are 'process-centric' algorithms for making informed decisions using the marketplace to improve a process' performance in novel ways. The three parts form a hierarchy. Thus, the marketplace requires the framework and the 'process-centric' algorithms need both the framework and the marketplace. This can be seen in Figure 3.8.

The framework provides the monetary support for the entire ERA architecture and provides fairness and control between competing processes. Consumers (client) processes have money in the form of wages to spend on all the goods (resources) they want to access. Producer (server) processes allow access to the resources they serve in proportion to the amount paid by each consumer.

Unlike a traditional auction system, consumers and producers in ERA are monetarily decoupled. Processes spend money with a server, but this does *not* guarantee them a certain amount of access at a particular price. Instead, the process pays up front to gain access to the resource and whenever it uses the resource this pre-paid money is consumed at the current price. As demand for a resource changes dynamically, so too does the price, which is derived from the current demand and utilisation of the resource. The framework can encompass any resource, but those considered here are: CPU time slices, virtual memory access and networking through a Distributed Shared Memory (DSM) system.

By analogy, imagine a transportation system that sells travel permits of any value. To use the transportation system the holder uses the permit in a ticket machine which deducts the current price of travel. Attempts to travel when the price exceeds the holder's balance are rejected and the holder must purchase a new permit. Rides are cheaper when the system is quiet, allowing for more travel for the same amount of money and vice-versa.

Each resource's price over time indicates the demand for it. For processes to make informed purchasing decisions they need access to these prices not only locally but also remotely. A distributed marketplace on top of the ERA framework provides this access. A resource server pays to advertise its price in the marketplace. Agents take this money and traverse the nodes of the distributed system to spread the price information. Since these agents use the advertising money to exist and run in the system, how far they travel depends on the current system prices. So, when the system is unloaded and the prices are low the price information travels further than when heavily loaded and prices are high. This provides a distributed information dissemination system with locality depending dynamically on system load.

The framework and marketplace combine to form an environment where a process can be in total control of its resource allocation, while the operating system maintains control over fairness. With this control over their allocation, processes can request resource allocations which are 'application appropriate' to improve their performance. Algorithms can be devised for all sorts of performance improvements. An example is optimising the location of resources purchased so as to increase resources obtained for the same expenditure. This results in self load balancing, where each process, by optimising its own costs, contributes to a global load balance due to the pricing interactions. Many more such algorithms are possible and easily programmed, since they are simply part of the process' code and can be written by the application programmer.

3.3.2 Meeting the needs

The resource allocation requirements for a general purpose distributed operating system were detailed earlier in this chapter. How the ERA solution meets these needs and thus overcomes the problems of previous resource allocation architectures will now be outlined.

Fairness

Access to resources is achieved by spending some of the process' regular wage with a resource server process. The location of the chosen resource server is the outcome of the process' ability to choose resources in the presence of competition. In order to gain access to a resource an amount of money must be spent with the server to indicate the importance of the resource to the process. Unlike previous economic systems, this money does not go into an auction to derive a price, instead it goes directly to the server to allow access.

A process receives access in proportion to the amount spent with the server over time. Therefore, at any point in time, each process has, at most, the ability to consume access in proportion to its current reserves (the money deposited with the server). Thus the ability to access the resource is proportional to the amount spent on the resource and fairness is assured.

Liveness

The liveness of the resource allocation architecture depends principally on freedom from starvation and deadlock. Deadlock implies that all the given rules of resource deadlock apply: mutual exclusion, hold and wait, no preemption and circular waiting. Mutual exclusion holds, but since most resources cannot be used simultaneously, the other three do not. Resources are not held by one process while waiting for another. All resources are preemptable, in that they are forceably released since access is only allowed by purchasing finite units. Circular waiting cannot occur since the resource allocation algorithms do not allow waiting for resources from other processes.

Starvation is more interesting. Access to resources relies on the ability to pay. This in turn depends on: having some money to pay with; and the ability to spend the money on the resource. Since a process receives a wage at regular intervals it cannot run out of money forever since it will eventually be paid more wages.

The real problem lies with the ability to spend the money on the resource. If a process has no resources and it has money, but the operating system has not allocated any resources on its behalf, then how does it run in order to buy the resources it needs. A 'chicken and egg' situation exists. Which comes first, the money or the resource? Using a technique called 'standing orders', a process can ensure the operating system buys vital resources on its behalf, without being able to subvert the operating system into performing any other extra work on its behalf. In this way a process cannot be starved of resources indefinitely.

Uniform access

The monetary system in the framework provides a uniform interface for access to all resources. Using any resource involves purchasing access to the resource from a server. This is achieved through a single kernel API call. By using object-oriented techniques, standard client, server and resource interfaces can be extended to suit resource specific parameters. Allocating physical resources, such as a new thread or a new memory block, is achieved orthogonally to obtaining access to use the resource, with standardised specialisable object interfaces.

Responsiveness and application appropriateness

Responsiveness and application appropriate resource allocation is a new and significant type of resource allocation. The 'process-centric' resource allocation architecture aims to provide this directly. Process-centric resource allocation combines the proposed system, where resources are solely in the control of the process, with the knowledge of competition found in operating system controlled resource allocation.

Control over the resource choice is provided by the process being able to choose its resources at particular locations, as with process controlled resource allocation. Also the process is able to express the importance of the resource to it by adjusting the amount it spends on the resource. This classification of the importance of the resource allows the process controlled resource allocation to be used in situations where competition exists. Rather than allowing unlimited access to a resource, simply because a process has chosen to use it, the process instead receives a share of the resource in proportion to the importance it expresses. To maintain fairness this expressed importance is controlled through the money supply to each process as described in the next chapter.

The process must be informed in order to make its choices, otherwise its decisions will be blind, like traditional process-controlled allocation. The distributed marketplace provides this functionality. Built on top of the framework itself, the marketplace provides price and timeliness information about resource servers in the current locale, which varies dynamically with system load. Price information derived from servers depends on two parameters, the utilisation of the resource and the demand (competition) for the resource. A process can use this price information, in order to choose which resources it should spend its money with.

Scalability and Overhead

It is important to preserve the scalability of the algorithms and keep the overhead low, while providing the additional functionality provided by this architecture. Several types of algorithms exist in the architecture: those to manage the money supply; those to allow access to the resource proportionately; those to disseminate the price information around the distributed system; and those to decide on which resources to use. The scalability of the ERA architecture is directly affected by the first and third of these, since they form part of the framework (operating system). But the proportionate access and resource decision algorithms lie in the process' domain due to the 'process-centric' nature of the architecture. Example access and decision algorithms presented in this thesis show good scalability and low overhead; as do the algorithms to manage the money and disseminate the price information. Money management imposes very low overhead and scales linearly. While, by its very nature, the price information dissemination overhead is directly related to the importance placed on it by the advertising server and the cost of disseminating the information across the system. It also scales linearly.

3.4 Summary

The computer world is changing quickly with greater proliferation of hardware and more complex software. With these changes, future systems will need better resource allocation to meet software demands. This thesis considers the requirements of such a future system for general-purpose distributed computing and proposes an architecture to satisfy them.

The requirements are: fair access to resources with expression of their importance; safe acquisition of resources with freedom from deadlock and starvation; uniform access to resources to minimise programming effort; responsiveness of resource allocation to changes in the system or workload; resource allocation which is not blind but instead is appropriate to the application's operation; scalability of the important resource allocation algorithms and overall a low overhead.

Traditional resource allocation techniques [2, 32] do not meet these desired requirements. Single node fairness can be provided by contemporary proportion-sharing techniques [57], but do not scale well or impose a particularly low overhead. Uniform access to resources can be provided by either the file system or object interfaces, though none of the existing interfaces express resource access importance. Depending on whether resource allocation is under the control of the process or the operating system, responsiveness or application appropriateness are sacrificed. Process control, such as PVM, is application appropriate, yet unresponsive. Operating system control is responsive, yet inappropriate for anything more than general resource requirements. Dissemination of information across distributed systems either provides low overhead and good scalability with little information, or high overhead and poor scalability with large information dissemination.

To overcome these problems this thesis proposes the ERA architecture. Its three constituent parts have been outlined: the ERA monetary framework for fairness and control in a process-centric fashion; the ERA distributed marketplace for dynamic locality information dissemination; and so have the algorithms for resource choice by the processes to improve their performance in dynamic scenarios. ERA aims to meet the identified need for a more open and flexible resource allocation system.

Chapter 4

Providing control

"Power is nothing without control"

(Unknown)

4.1 Introduction

The key component in the ERA architecture is the ERA framework. The framework provides two parts of the architecture. The first is the 'process-centric' aspect. This is the ability to be responsible for all resources required and be able to access these resources and determine their importance. The second part constrains the freedom of the first part to provide fairness under competition through a monetary system, so that resources can be shared while preserving proportional fairness. It is the ERA framework which provides the monetary abstraction used at the heart of the ERA architecture to satisfy these competing goals.

The principle of the ERA framework is that a process controls its own resource allocation choices. The framework specifies as little as possible, so as to provide a light-weight system and maximum flexibility. For this reason, the framework does not specify how a process should spend money, or how prices should be set for each resource. Instead, the framework provides the interaction between processes, the mechanisms for using money and monetary management. All other work required by the framework is achieved by higher levels of the architecture or the process itself.

This framework allows for the distributed case, because the mechanisms used in its design are scalable with increased resources, processes and system nodes. Besides this, the mechanisms are designed to have an acceptably low overhead. Previous economic systems [56, 27] have ignored aspects of scalability and overhead when considering similar problems.

This chapter covers the abstract design of the framework, its structure and mechanisms. In doing so, this chapter discusses the scalability and overhead of the framework. Experimental results are presented indicating that the mechanisms scale linearly with the number of processes and system size. Results show that additional overhead of the system, as the number of processes and wage periodicity are increased is less than five percent.

4.2 The framework

Process-centric resource allocation is the cornerstone of the ERA framework. This means that a process operates in an environment of resources which it consumes and to consume these resources efficiently, it must be able to discover which resources are available. With this awareness, the process must be able to choose which resources to use and do so uniformly (w.r.t. API). Since many process-centric processes can exist, fairness must be enforced among processes competing for resources. Thus, the priority or importance of each resource to a process, must be quantified to allow requests to be arbitrated. For a process to control all its resources, it is important that the operating system does not perform any resource allocation on the process' behalf, since this would take control away from the process.

The ERA framework fulfils these requirements by providing the mechanisms to maintain fairness among competing priorities and to allow a process to: uniformly choose which resource to use; quantify the importance of its resources; allow processes to control the use of vital resources such as CPU and memory; and be able to discover the resources in its locale.

The fundamental feature of the framework is the use of 'money' as a means for quantifying the importance of resources to the process. This differs from previous systems in several ways. Firstly, the processes are in absolute control over the use of their money and decide how to spend it as part of their normal execution, since the process' code controls money spent on resources. Secondly, the connection between buying and selling resources (previously auctions were used) has been decoupled. This requires the rest of the framework to work in this decoupled way too. Lastly, the framework is scalably distributed and only provides the mechanisms and does not enforce any unnecessary policies.

A process is given the ability to choose the importance of *accessing* a resource over time. This access importance determines the relative importance of obtaining a time-slice or page fault, for example. A process expresses this relative importance by spending an appropriate percentage of its wage.

The ERA framework provides control over access to resources. The framework does not attempt to perform spatial fairness, such as limiting the total virtual memory consumption of a task. Work on spatial consumption has been done by Cheriton and Harty [13]. But as discussed in Section 3.2.1, this does not necessarily lead to resource access fairness.

4.2.1 Money, money, everywhere

Money forms the connection between the two goals of the framework: process-centric allocation and resource consumption control. For simplicity the unit of money is called a 'credit'. It represents the fundamental unit of resource access under ERA. Without money, access (page fault, time-slice, etc.) to a resource is not possible. Credits are, however, *not* related to real-world money. The amount available is simply a system configuration parameter. Credits are allocated on a per-user basis and held on a per-process basis. When 'logging in', the user is allocated money and all the processes run by that user have some of that money. Threads (or other smaller units of execution) share the money of their containing process. Processes are the unit of responsibility for resource allocation and access. This overall monetary structure can be seen in Figure 4.1.

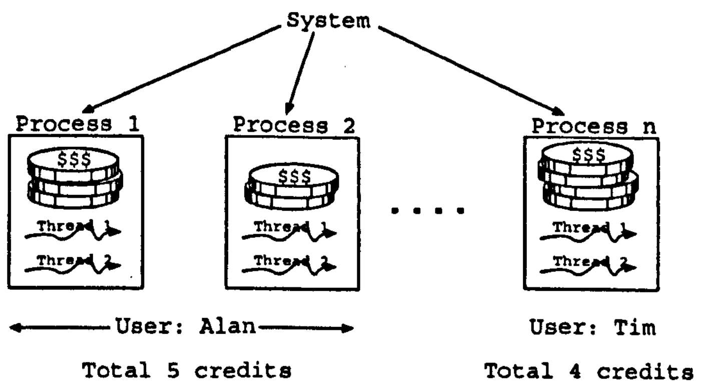


Figure 4.1: The subdivision of 'money' between users and thus processes, but not threads

By locating responsibility for money at the process level, a process can execute with multiple threads. At least one thread is used for its normal functionality and one thread is used to execute decisions on monetary spending without explicitly interrupting the main execution. Most of the time this extra thread does not need to change its resource decision and it blocks while monetary work is not required, leaving the main thread(s) to execute as normal.

Money is spent by client processes (resource consumers) with server processes (resource producers) to obtain access to their resources. Each resource has an advertised price which indicates the utilisation and demand for the resource being served. A process receives access in proportion to the amount spent with that server and the current price of the resource from that server. Thus, by spending money with cheaper servers the process (consumer) can receive more resource access.

A normal user-level process is a client process. It must spend its money to access all of the resources it needs to perform its execution. Since several resources will be required (at least CPU and memory) the process must divide its money accordingly. The division represents the *relative* importance of access to resources. Thus if a process divided its money $\frac{2}{3}$ to $\frac{1}{3}$ for CPU and memory access respectively, then the process expects accessing the CPU (running) to be twice as important as accessing memory (page faults).

A server process receives payments for access over time, each representing a client's estimate of importance for accessing the resource. Resources which can be served in this way must allow finite/preemptable access to preserve access fairness. This thesis considers: CPU (time-slices) and memory (page faults). The server process uses these payments to determine what proportion of access to grant each client. This can be seen as the probability of access P_i for each client i of n clients each paying C_i to that server:

$$P_i = C_i : \sum_{i<n}^{i=0} C_i$$

Since it is resource access which is considered under the framework, spatial allocation must be achieved orthogonally beforehand, through specialised object interfaces (for example allocating space in memory). Accessing a particular resource therefore is a two phase process. Firstly, a call the server to allocate some of the resource as usual. Then, secondly, money is spent with that resource server so as to obtain access to that allocated resource. An example of this two phase process for memory can be seen in Figure 4.2. (1) Memory is allocated in a similar way to normal. (2) The memory is accessed, but since it has not been paid for, it is refused. (3) Ability to access the resource is requested. (4) Access is granted since it has been paid for. (5) Cost of the access is deducted [1].

4.2.2 Providing and controlling money

Such a monetary based control system has undesirable properties as well as desirable ones. The most obvious example is *starvation*. Since the possession of money is the only means for accessing (consuming) a resource then the simple question arises: *What if a process runs out of money?*

Starvation in the real-world occurs when a consumer has no food and no money in time to buy more. One-off payments do not work in situations where you can not guarantee that consumers can afford all the resources they need. In distributed resource allocation this can occur under unpredictable loads because, since the load is unpredictable so is the price. Under the ERA framework this starvation problem cannot occur since 'money' is

[1] Assuming an implementation based on Cost proportion-sharing

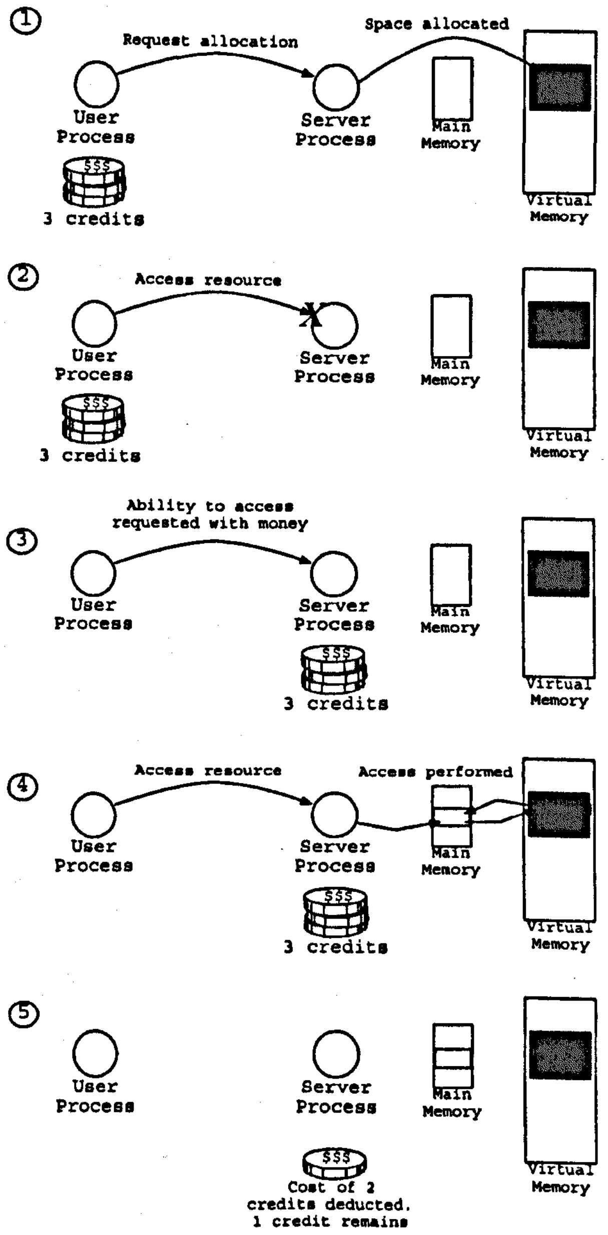


Figure 4.2: The two stage process of using memory under ERA

paid regularly as a wage. This also copes with situations where the run-time of a process is not predictable (such as compilers and system processes) and for which the necessary sum of money cannot be calculated in advance.

Each process has a wage which it receives in payments at regular intervals. In between payments, a process can choose to spend its money on resources and should it run out of money, or decide to change its spending, then it is assured of receiving more money within a finite time.

Since access to a resource is based on the proportion paid then, in the long term, each process will eventually receive *some* access no matter how little money it has. Together these two mechanisms ensure that a process is never starved of money or access to a resource because of price.

Wages do, however, pose a few problems. A wage implies a continuous and guaranteed supply of money to a process. It is important, in order to preserve fairness, that this source of money cannot be subverted. Potential problems exist when new processes are spawned. When a process is spawned it must be given a wage level in order to obtain any money. If, for example, all processes are given the same wage, then a malicious process could spawn sub-processes to gain extra money. In this scenario the child processes simply exist to send their income to their parent and in this way a process can essentially mint its own money. Since the possession of money is directly related to access to resources, this would lead to the ability to monopolise access to a resource. The framework must prevent this.

So, wages always conform to a conservation of money supply, such that: "the wage of a process is shared by the parent process and any child processes after a 'spawn' operation". This means that at any spawn operation the parent must give up some of its wage to its child/children as their wage. This conservation is better seen as a 'wage tree', see Figure 4.3. The root of the tree represents the operating system, which according to system defined parameters increases the money supply by fabricating money when new users 'log-in'. At each lower branch, the money each process was given by its parent is now the supply of its current wage level and the wage level of all its children. Thus, at each branch, wages are conserved. To ensure this conservation, the reverse of this process

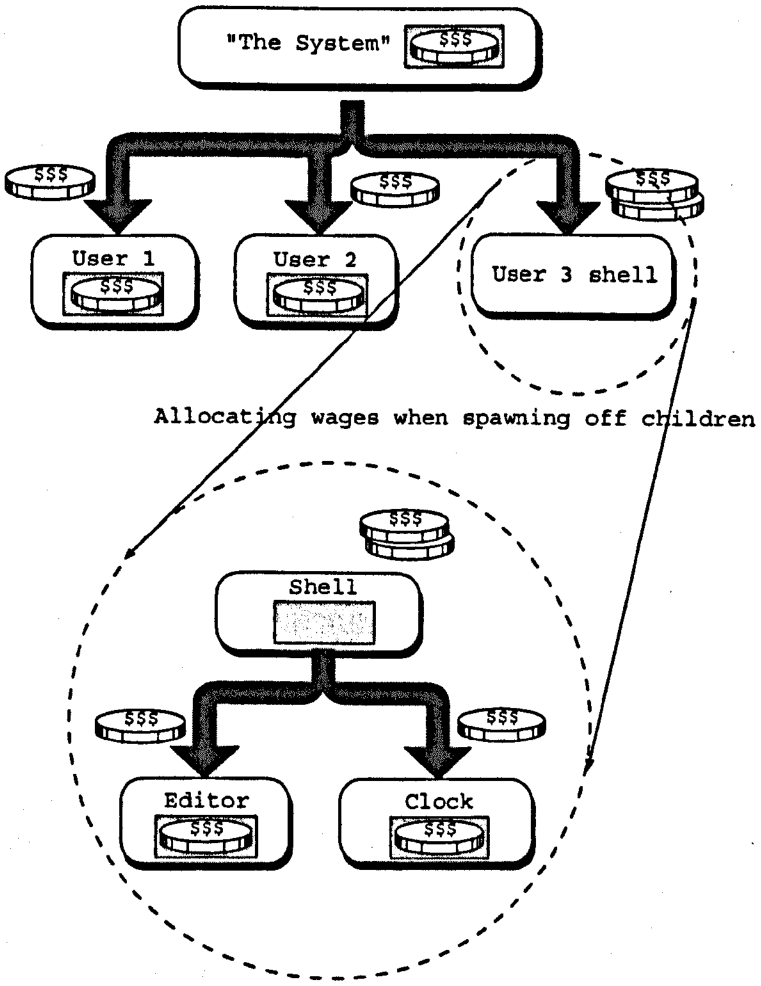


Figure 4.3: Wages are conserved when a process spawns children. This ensures money cannot be fabricated by processes

takes place when a process is killed by repaying the wage to its parent and re-parenting its children. Seemingly, paying wages to a large tree of processes will be very expensive, since money will have to be passed down the tree, allocating portions to parents and children as it descends. This will be especially costly if wage rounds occur frequently.

However, ERA takes a mostly static view of wage values in the tree. Since, values in the tree, especially higher up, are unlikely to change frequently (e.g. not on every pay round), the process' wages can be calculated at creation time. Then, if a parent has its wages changed, its children's wages can be altered accordingly. This update will be expensive

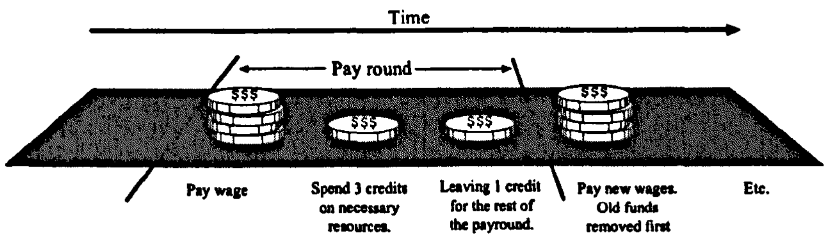

Time
Pay round
$$$
$$$
$$$
$$$
Pay wage
Spend 3 credits on necessary resources.
Leaving 1 credit for the rest of the payround.
Pay new wages. Old funds removed first
Etc.

Figure 4.4: Basic money flow in a pay round

since the entire subtree must be traversed, but no more expensive than the processing that would be require to calculate the entire subtree every pay-round. Once each process has a wage level, then each process' wage can be paid separately and in parallel, without overall system coordination. This simplification removes wage payment as a potential system wide bottleneck. This in turn improves scalability of the system across wide-area distributed systems.

Simply conserving a process' wage is not enough to prevent monopoly. Since a wage is a guaranteed income there is nothing so far stopping a process accumulating money and not spending it. A process which simply went to sleep for a day, woke up and tried to spend all its money performing a denial of service attack on a particular resource would be exhibiting undesirable behaviour.

To prevent this, the ERA framework controls money one step further, by preventing its long-term accumulation. It does this by removing any unspent money before the next wages are paid. Therefore, money is paid to a process for its use in that particular pay round (time period between wage payments). This step is unique to the ERA framework. The flow of money during a typical pay round can be seen in Figure 4.4. The effect of this level of control is that a process can, at most, spend the money it earns as a wage for its own purposes. This in turn means the effect on consumption of resources is at most the number of resources it can purchase with this finite amount of money.

This is a high degree of control, but implies that a process cannot express more importance than the money available in a single pay round. This can be a problem for alarm style processes, which may need more money, but infrequently. So, the ERA framework

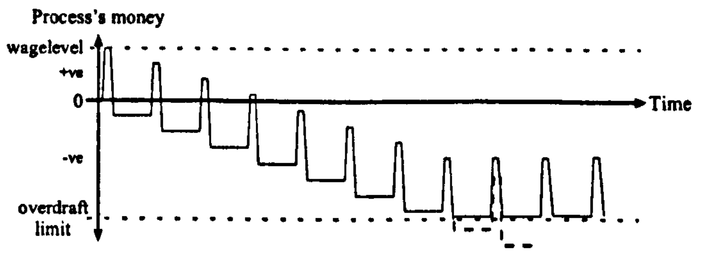


Figure 4.5: Graph of wage level over time, of a process trying to continually spend more than it has

provides an automatic overdraft facility for each process. The overdraft is simply the ability to spend more money than a process has, up to a fixed (system parameter) multiple of its wage level. For example, if the overdraft multiple is 3, a process with a wage of 10000 credits can at any time have between -30000 and 10000 credits. Any over-spend is then paid back immediately when new wages are paid. Thus, the overdraft will be paid off in a few pay rounds, if no further money is spent. This means that in the long-term a process cannot over-spend or subvert extra money, but can temporarily express special priorities should it wish to do so (see Figure 4.5).

Distributing the wage framework

When wages are paid over a distributed system by multiple ERA kernels there is no need for the wage payment to be synchronised. Processes on a particular node will receive their wage to spend on their resources at a regular interval and this money may or may not be paid at the same moment on another node. The period of the wage payment must, however, be constant across the system in order that the value of money is constant across the system.

This means that a global clock is not required so long as the clocks run within close tolerances, thus ensuring fairly constant wage payment times. Otherwise, a distributed clock synchronisation mechanism should be used to ensure that the relative clock rates do not vary greatly.

With distributed wage payment comes the possibility of distributed resource consumption. Since wages may be paid out of synchronisation then, as a result, a shared resource can be purchased at different times. However, this does not imply that there is an unfairness towards those that are paid first, since a process purchasing the resource earlier will finish earlier than a process which was paid later. For example, assuming that each process uses the resource just for one pay round, the earlier process will receive a large share until the second buys into the resource. While both have paid for the resource both receive a share according to their payments. When the first process finishes using the resource the second process receives an increased share of resource. As a result each process receives on average a share in proportion to the total paid. Ultimately this means that processes compete for a resource only at the time they wish to use it and do not receive any unfairness.

4.2.3 Obtaining resources

Once processes have a means for expressing, in a controlled fashion, the importance of resources to themselves they must be able to access their chosen resources uniformly. The mechanism adopted in the ERA framework is contracts. Contracts are a standardised means to request access to a resource. A contract specifies: the destination server, the type of resource to which access is requested, the amount of money which expresses the importance of the resource to the client[2].

Contracts are the only means of requesting the ability to access a particular resource, but they do not arrange or specify an absolute reservation of access to that resource. Instead, passing a contract to a server asks the server to allow it to access to the resource which it serves for the next pay round time period. The type specifies the server resource which the client wishes to access. The money sent with the contract indicates the process' budget for access to that resource. It does *not* have to bear any relation to the current price of that resource. Instead, it represents the forward payment to the client's allocated budget, for this pay round, for access to the resource. In return for this money, the server

[2]These contracts are represented as objects and can be sub-classed to add additional parameters between client and server, but these few parameters represent enough to obtain access to any resource to be expressed.

is expected to allow the client access to the requested resource in proportion to this client's budget, when compared to the total sum of budget payments made to the server for this pay round.

Contract requests are sent to servers be the framework. This is achieved by the framework which handles the job of transferring the contract along with the money which has been pledged. The framework first checks that the calling process has enough money to fulfil this contract request. Assuming it does, the money and contract are transferred to the server.

However, there is a problem with process-centric resource allocation. Since processes have control over all their resource allocation, they must buy the resources within which they exist. In practice means that they must buy the CPU and memory access required to run. By the ERA framework mechanisms just outlined, this must be done by the process constructing and sending contracts to servers for both these vital resources. Unfortunately, in order to achieve this, the process must already be running and must therefore already be consuming these vital resources.

A simple solution is to relax the constraint that processes must buy the resources they need to execute. This brings two difficulties. Firstly, it defeats the objective of process-centric resource allocation, since the process cannot control the importance or placement of these resources. Secondly it is then possible for the process to subvert the resources without any control over their access and this defeats the fairness goal of the ERA architecture.

The solution in ERA is a mechanism called 'Standing Orders'. For each process, the operating system stores two contracts for it to use for vital resources. A standing order contract is exactly the same as a normal contract except the kernel stores it and provides process access through an API call. The most usual use for these standing orders is to specify the contracts for purchasing CPU and memory. The kernel consults these contracts when a process' wages are paid. On payment, if the contract is correct (i.e. it can be afforded), a copy is made and it is sent to the server specified in the contract on the process' behalf. This allows the process to buy these vital resources without actually running. When, a process is initially created, or it has run out of money, it can buy its vital resources, to continue its task.

Standing orders are initially created when the process is created. This can either be achieved either by the parent specifying them when creating the process or, if they are not specified, they are a copy of the parent's adjusted to match the same proportions with a lower wage.

The benefit of the standing orders approach is that the process still has process-centric access to these resources, since it can modify the contract parameters once running. This means the process can change the location of these resources (causing migration) or their importance so as to affect the amount of access received. This is done without the process subverting any processing or resources on its behalf from the operating system. Also, the only overhead incurred is the transfer of two contracts per process to local (same machine) servers, which is a low overhead operation.

4.2.4 Money transfer

Preserving fairness means that the architecture must preserve monetary control, such that a process cannot appear to have more money than it really has. An area where this is important is the transfer of money between clients and servers.

This protection problem is important since, as part of the resource acquisition process, the client transfers payment to the server up-front for all resource access in that pay round. Thus money is not gradually transferred in small amounts, as the resource access is consumed, but as a lump sum in advance. This makes protection of the transfer even more important.

This question of protection can be considered from two angles. Firstly, a server should be protected from clients fabricating money when transferring it to the server. For example a process could claim to send more money than it has, or request to send money while it has some, then spend it and then try and send the request. These problems, however, do not arise under ERA, since money can only be transferred by calling into the kernel, where money is stored. This means that money transfer is mediated by the operating system thereby allowing atomicity and authentication on any transfer request. While the kernel mediates transfers they can occur in parallel between nodes since each runs its own separate kernel.

From the other viewpoint, the money transfer problem is about protection of the client from the server. This is an issue, since the client expects to receive resource access in return for the money spent with the server. Unlike the client to server problem, there is no operating system mechanism that can be applied to ensure the server performs the action it claims to perform. This is because it can simply take the money and return no resources or little resources to the client. In this scenario the problem of protection becomes one of trust.

This trust is formed in two ways. Firstly, by only using server addresses from sources which are trusted. These addresses can be acquired from the system in the form of default system servers, or addresses received from other trusted software. Secondly, by using a server and only reusing that server if the previous service received was satisfactory, although this may be difficult to measure.

In fact, this form of trust is common in most distributed software, since information must be used which is returned as a result of an invocation of a remote service which is outside the control of the client.

4.2.5 Message queues and money transfer

Performing money and contract transfer is a three stage process. Firstly, the money is deducted from the client, according to the contract passed to the kernel through the sending API. Secondly, the money is added to the server so it has the money it needs to serve the client. Thirdly, the server must be informed that a contract has been sent to it and the contract must be obtainable by the server.

The money deduction and addition tasks are fairly simple and can be achieved by the kernel modifying each process' state accordingly. The transfer and reception of the contract is achieved with message queues. The ERA framework supports a message queue in the form of a mailbox [49] per process. This message box stores messages which contain a type

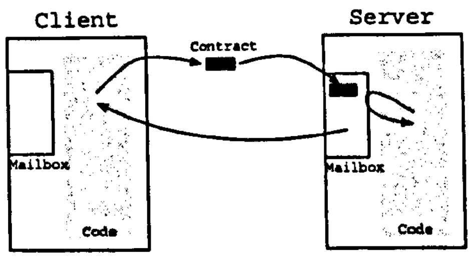


Figure 4.6: The use of a server's message box to provide the transfer of contracts

of message identifier and a pointer to the actual data of the message. The queue acts as a mailbox in that a thread retrieving a message from its mailbox is suspended when the mailbox is empty until a message exists. And likewise, when the mailbox is full, a sending process is suspended, see Figure 4.6.

Access to these message queues is provided by simple kernel API calls which take a message type identifier and a pointer to the actual information. These provide a simple interface for sending point to point contracts to other processes. Client processes use these calls to send contracts to servers. A server uses these calls to block until a contract is received. When the contract is received payment is ready at the server and the contract is processed.

The ERA framework supports two forms of contract transfer. 'One-shot' transfers provide for one way transfers to servers, such that the client does not want to or cannot know if the request has failed. This type of transfer is used by the standing-order mechanism to dispatch contracts for standing orders. This mechanism is used here, since there is no possibility that the client can do anything useful should the contract be rejected for some reason.

The other form of contract transfer uses 'Request-Reply' style semantics. The contract is sent with the money to the server which can either accept or reject it. While the server is processing the contract the client is blocked waiting for its reply. Since a server requires little processing in order to service the contract and contracts are rarely rejected, the client does not need to wait for long. Should the server reject the contract, such as when the resource cannot be further divided due to finite units of allocation, then the framework transfers the money back to the client.

Obviously this poses a problem. It is possible for a server to reject a contract having spent the money it had received from the client. This is possible, but it is the same problem as the trust problem for general money transfers between client and server. The server *must* be trusted to use the money given to it correctly.

4.2.6 Monetary protection

Money itself must be protected against duplication and fabrication. The ERA framework stores the monetary worth associated with each process outside its reach. In this way money is not directly manipulated as an integer would be, but instead via kernel API calls. This level of protection can be created in two ways. Firstly the monetary values themselves can be stored in kernel-space. The process can then access this money indirectly through the contract sending API calls. Secondly, the information can be provided in user-space and protection provided through access control lists (ACLs). The capability to access the user-space memory can then be used by the process itself or by the OS on its behalf. This latter approach allows greater flexibility to the process, but it must be trusted to manage its money correctly.

Another important aspect of protecting money is ensuring that it cannot be kept between pay rounds. The ERA framework employs kernel-space storage of money. This ensures that at all times all the money exists in kernel-space under its control. This in turn allows the framework to revoke the money easily between pay rounds.

One further potential problem exists with the monetary system, when considered in the distributed case. Since wages are paid in parallel over the distributed system and processes can migrate, it is possible that the wage updates on each node are not synchronised. It is thus possible for a process to migrate during the pay round process. This can lead to one of two undesirable properties. Firstly, the process may not receive any wages at all, since it migrated from a node before its pay round to a node which had already had its pay round. Or secondly, a process may move after its pay round to another node which has not had its pay round yet and thus receive two pay rounds.

Both conditions are only temporary, since at the next pay round things will be correct again, but it is possible that the correct execution of the processes may be disrupted. To prevent this, each process has a logical clock associated with its wage and each time its wage is paid, the logical clock is incremented. Each node also has a logical clock to say which pay round it is currently on. These clocks can then be checked on wage payments and migration to ensure the process receives the pay rounds correctly. Since these are logical clocks there is no need for synchronisation across the system to ensure correctness in the wage payment process, which would severally limit the scalability and thus speed of the process.

4.2.7 Framework summary

To summarise, a process is given a regular wage at each pay-round which it spends to receive access to all the resources it requires across a distributed system. This money is available to the process only during that pay round, any unspent money being removed before the next wage is paid. Processes spend their money on obtaining resource access and in so doing, they specify the chosen resource, its location and the importance of the resource through the amount of money offered. Each process requesting a resource receives a share of access to that resource in proportion to the amount spent on it in that pay round.

Each process has an automatic overdraft, as a fixed multiple of its wage, to allow it temporarily to spend more than it earns. Overdrafts are repaid on wage payment. New money is only created by the system when new users 'log-in'. When new sub-processes are spawned they must be given part of the parent process' wage. So as to prevent the malicious manipulation of money, by processes seeking to subvert the priority through money they can express, access to money is only provided to processes through a kernel API

Resources are bought by sending contracts through the kernel to resource servers. A contract specifies what type of resource the client wants to buy and the amount the client wishes to spend on it. The kernel takes the money and the contract which it sends to the server.

Each process has two standing orders which specify the contracts for its vital resources. These are sent by the kernel on the process' behalf, but the process still has access to modify them.

Once a resource has been bought it is consumed through usage. The process has access to the resource for as long as the paid money allows. Logical clocks at each node and process ensure that wages are correctly paid even while processes are migrating between nodes.

4.3 Ensuring the framework works

In resource allocation is it important for the techniques used to exhibit freedom from deadlock and starvation. In the ERA framework, deadlock can be viewed from two points of view; deadlock of resource access by clients to server resources; deadlock in the resource allocation mechanisms. From the point of view of client access deadlock cannot occur, since by definition all resources are preemptable to allow them to be proportion-shared. Thus 'No Preemption' does not exist (see Section 3.2.2), because preemption does and thus proportion-share access is deadlock free.

From the point of view of the resource allocation mechanisms, absence of deadlock is harder to prove. Consider the allocation of two closely related resources, CPU and memory. Both are closely controlled by the framework. To ensure deadlock does not occur, this interaction must be free from one of the four conditions for deadlock. Initially, when resources are requested, a contract is constructed and placed in the server's message box. Here there is no potential for deadlock, since transfer of the contract is performed by the operating system which must claim mutual exclusion on the server's message box. In this scenario, the first three conditions specified in Section 3.2.2 hold, but the fourth

(Circular waiting) does not, since each process is claiming access to the entire message box and thus only one resource.

Once in the message box, the contract is accessed by the server. Again, access to the message box is performed at the granularity of the entire box, so one resource is waited for and thus 'Circular Waiting' does not hold. Besides, since the server serves a vital resource, we must ensure deadlock does not occur with the server waiting on a vital resource which it serves itself. Due to the vital nature of both CPU and memory, neither runs in user-space. This means that the code and threads used by the servers do not use ERA controlled CPU or memory, thereby preventing deadlock.

Starvation is the other important consideration under the proportion-share resource allocation. Starvation could occur under ERA for two reasons: a process is unable to spend its money to obtain resources; indefinitely it has no money to spend.

Failure to obtain resources could occur either because a client is unable to spend its money or because a server refuses access to the resource. Standing orders ensure the first condition cannot occur. Standing orders are used by processes to obtain the resources required, such as CPU and memory. Standing orders, by being dispatched by the kernel rather than the process, allow a process to place contracts in server message boxes without using any resources. In this way, a process is able to spend any money it has, so long as it has set up its standing orders.

Allocation of resource access to a client is a server's responsibility. Since the implementation of servers is not defined by the framework, they must be trusted to allow access to the resource in proportion to payment. Failure to do so will cause starvation.

4.4 Server mechanisms

A server process is an ERA process which has, as part of its responsibility, the allocation of one or more resources to client processes which, in return will pay money for access. Server processes present the supply side of traditional supply and demand microeconomics. How the server performs its duty is not defined by the ERA framework, beyond:

- Access must be shared fairly between clients, in accordance to the amounts of money paid by each process. This prohibits badly behaving server processes which accept money and give no or little access in return.

- A meaningful price, which reflects historical demand, must be offered to any enquiring process. The price should have an equivalent meaning across all resources with the same identifier. For example, all "CPU" resources must have the same price semantics, but "CPU" and "SparcCPU" resources need not.

- A server should set its prices so as to maximise its utilisation (supply) while discouraging over-demand. Thus the price is higher when the demand is high than at times of low demand.

This means that a server is presented with two goals: to share a resource fairly between competing, paying clients; and present a meaningful price so as to control the number of clients it has to ensure good utilisation. In order to achieve these goals a server can use what ever information it can acquire or store (typically historical information).

These two server goals need not be directly related to each other. This means that the act of proportionately sharing the resource between clients need not directly effect the price being advertised. A server may advertise a price of 80 credits, but this does not mean that a client paying less than 80 credits will receive nothing. Likewise, a client paying 160 credits does not mean that that client will receive 2 units of the resource.

The price advertised by the server is an indication of the resource usage to allow clients to make an informed choice. The received share of the resource should depend solely on the competition for the resource at that time and the money paid by the client is used to indicate to the server the importance/priority for the resource to the client.

One can also derive a *'cost'* of using the resource which is the amount paid for the amount received. It is possible to make *cost* = *price* and this possibility is discussed in

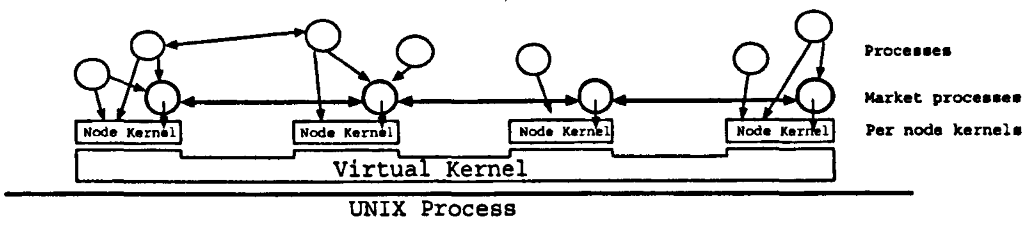


Figure 4.7: The structure of the ERA simulation

more detail in Chapter 6. Cost is a server's internal valuation of the resource and over short periods of time will fluctuate depending on demand. For stability reasons, the price advertised should represent a longer term indication of demand and utilisation.

It should be noted at this point that the length of a pay round and the time over which a price is taken and advertised are simply configuration parameters of the system at installation time.

4.5 Simulation

In order to study the ERA architecture, a detailed simulation has been constructed of a distributed system running an ERA single-address space operating system (SASOS). The simulation emulates a configurable number of distributed nodes connected by point-to-point network links. Each node runs an ERA kernel under emulation inside a UNIX process using POSIX threads and the UNIX memory subsystem. The code for CPU and memory servers runs on each kernel along with example workload applications. This simulation has been instrumented to record the number of instructions spent in each code section and the various operations performed by the operating system.

The simulator was constructed in two parts: the emulation environment and the code to provide the functionally of a real SASOS based on the ERA framework. The general structure can be seen in Figure 4.7. The simulation was constructed under FreeBSD UNIX [3] as a single process. The emulation environment provides a virtual kernel to convert this UNIX process into an emulated distributed system. In doing so, the virtual kernel: multiplexes ERA node kernels in turn; provides management of the ERA processes onto the underlying UNIX process context; and provide the emulation and protection of a shared memory address space between executing ERA processes.

Above this, the ERA per node kernels provide all operating system functionality required on an individual node. These kernels are the ERA based operating system and exist outside the ERA process-centric system. Above these, using the ERA process-centric framework, both the application and marketplace processes exist. The principle of this separation is that the code above the virtual kernel is as close as possible to a real ERA-based OS, with the actual code being executed by the emulated environment.

The simulation provides control of CPU, real memory and the marketplace as process-centric resources using the ERA framework. Both CPU and memory are shared to allow preemptive proportion-sharing. For the purposes of the simulations in this thesis CPU is shared by Cost proportion-sharing (see Chapter 6) and memory by page fault demand costing (see Chapter 6). These represent the simplest forms of resource sharing discussed in this thesis although other algorithms such as real memory page rental have also been implemented. Lastly, the marketplace is implemented according to the description presented in Chapter 5.

In order to provide a faithful simulation of the ERA environment it is important that the execution of each node's kernel and its processes are emulated[3] realistically. To this end the ERA simulation supports the emulation of process contexts using a POSIX threading system and the UNIX memory system to emulation virtual memory. This allows the execution of code, above the emulated systems, which is as close as is feasible to real code.

The code running above the emulation systems was then instrumented to count the number of instructions in total and in certain sections of code. All sections of code which belonged to functions which would normally be required by a distributed operating system, or functions which were only required for the ERA architecture, where marked both in the operating system and in the processes using the architecture. From these instruction counts

[3]In this thesis we distinguish between emulation and simulation. Simulation as an embodiment of a theoretical model. While, emulation as a mapping of one environment to a different underlying system. Note, sometimes emulations require simulations in order to provide part of the emulated environment.

the overhead can be estimated from the relative execution times of parts of the system, against the total time.

The data from this simulation is used in this and following chapters in order to analyse the operation of the ERA architecture. Unless stated, the configuration of the simulation is not changed between experiments. For more information about the simulation see Appendix A.

4.6 Analysis

In this section the operation of the ERA framework is examined in simple scenarios. Firstly, the ERA simulation is used to analyse that the system does indeed preserve proportion fairness under different experimental workloads. Next, the scalability and relative overhead of the algorithms used in an ERA system executing on one node in a simple ERA implementation are shown.

Four programs are used in this section to demonstrate the effectiveness of the ERA framework. These programs cover the range of process types common on general purpose systems today. The first is the `dhrystone` benchmark[4] which provides an example of a computation bound process. The process continuously runs a C version of the dhrystone program over 500000 loops. Secondly, we use the matrix multiplication problem to demonstrate a memory intensive task common in scientific computing. Two randomly generated floating point square matrices are continuously multiplied. Thirdly, the algorithmic section of the X11 MPEG decoder `mpeg_play`[5] program exhibits the computation and memory bound nature of many multimedia class programs. (The decoder is run continuously on a 100k MPEG file.) Lastly, a simple genetic program[6] provides an example of a CPU intensive task whose data set changes over time. (The genetic program evolves a solution

[4]Reinhold P. Weicker, CACM Vol 27, No 10, 10/84 pg. 1013. Translated from ADA by Rick Richardson
[5]Written by Lawrence A. Rowe, Ketan Patel, and Brian Smith at the Computer Science Division-EECS, University of California at Berkeley
[6]Kindly donated by Kim Harries at the Computer Science Department, City University, England - kim@soi.city.ac.uk

to the calculation of parity over 5-bits of data. In doing so, it builds large numbers of tree structures of the evolved programs, which it scans to evaluate their fitness.)

4.6.1 Proportion-sharing

Fairness is achieved under ERA when the share of resources each entity receives is directly related to its money, since money represents its resource rights. To validate the framework the first experiment measures the received quantities of resources. These are compared to the ideal proportion-share fairness of their wages.

For brevity, two types of program mixes and money allocation are used, providing four scenarios. An application-homogeneous mix is used to exert multiple similar resource request patterns, in this case those of the genetic program. An application-heterogeneous mix uses each of the test programs to provide a contrasting resource request pattern. Two wage level configurations are used: equal money allocation (1:1:1:1) representing equal priority to clients; and a 4:3:2:1 ratio of wages representing direct distinct relative levels of priority among processes.

The simulation was configured to have a time slice duration of 20ms and a pay round duration of 50 time slices. A single memory server, which serves 1Megabyte of 4k paged memory to processes, was run. The single ERA node ran 3 system processes in addition to workload: a CPU server, a memory server and a market server (see chapter 5). The node was configured to have a total of 13000 credits split up as 1000 credits per system process and 2500 per workload process (assuming a 1:1:1:1 distribution). Each process spent its money in an equal 50/50 allocation between CPU and memory.

Figures 4.8 and 4.9 show the equal wage resulted for both application mixes. Both experiments resulted in equal quantities of CPU time being allocated per pay round. In the homogeneous case (Figure 4.8) the mean levels obtained are: 1148, 1191, 1189 and 1207 milliseconds. These deviated from the ideal level of 1184 milliseconds (based on the 1:1:1:1 wages ratio) by -3, 0.1, 0 and 2 percent. In the heterogeneous case (Figure 4.9) the quantities of CPU time obtained initially varied strongly and then stabilised. This

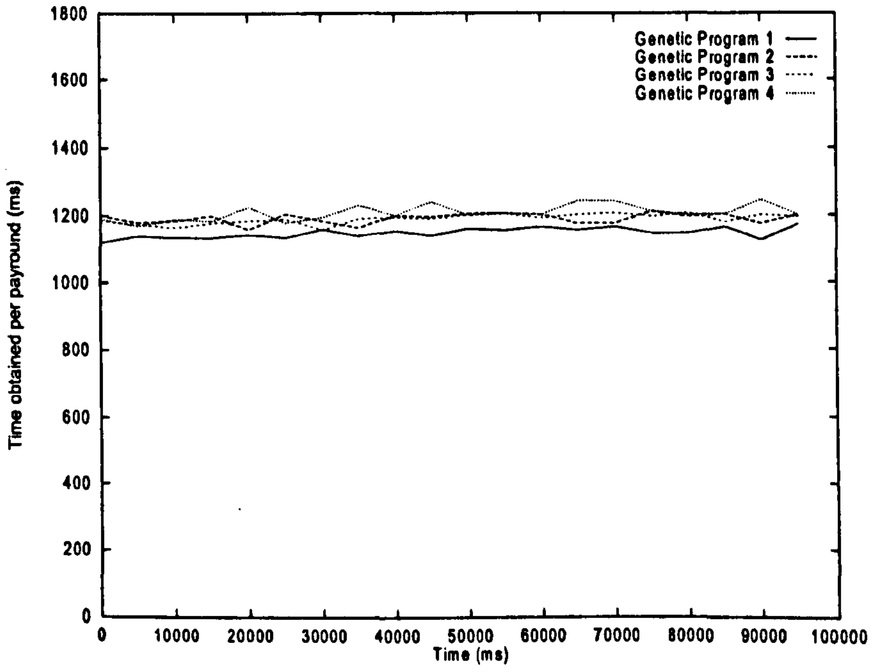


Figure 4.8: The average time obtained over five pay rounds against time of an application-homogeneous program mix using an equal wage allocation

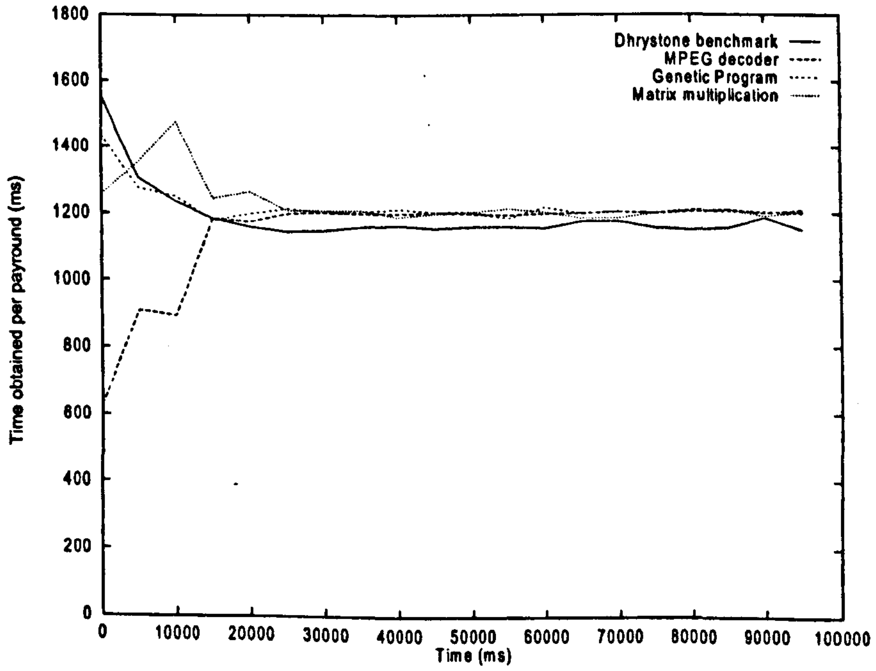


Figure 4.9: The average time obtained over five pay rounds against time of an application-heterogeneous program mix using an equal wage allocation.

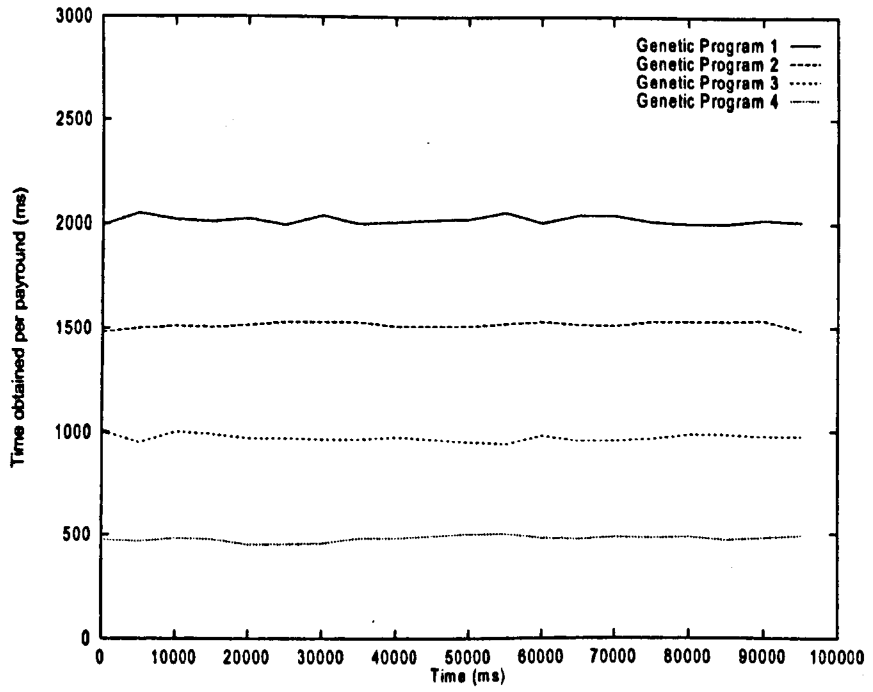


Figure 4.10: The average time obtained over five pay rounds against time of an application-homogeneous mix using a 4:3:2:1 (starting at the top) wage allocation

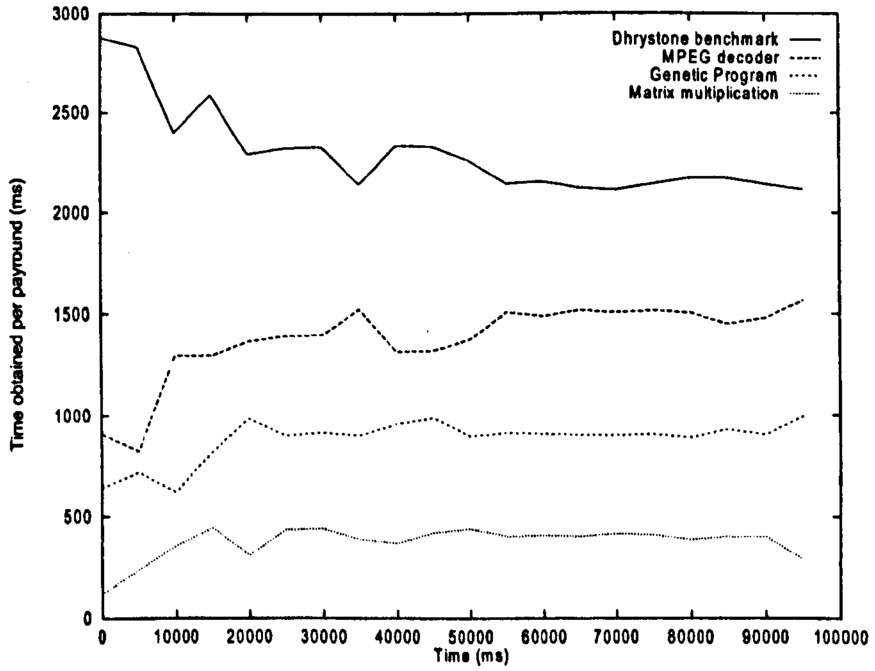


Figure 4.11: The average time obtained over five pay rounds against time of an application-heterogeneous mix using a 4:3:2:1 (starting at the top) wage allocation

is because memory intensive applications require a large number of page faults in order to acquire their 'working set' of pages. While they are blocking waiting for page faults to complete they cannot obtain their full share of the CPU resource. Once unblocked, however, they have an effectively higher priority with the CPU server due to their unspent money. Once stabilised, the mean levels obtained were: 1180, 1201, 1205 and 1203 milliseconds. These deviate from the ideal level of 1197 milliseconds by -1.5, 0, 0.2, 0.5 percent. Thus, proportion-share fairness was preserved under equal wage ratios. It should be noted that the two experiments showed a slightly different average level due to random factors such as application randomness and page fault times.

Figures 4.10 and 4.11 present the 4:3:2:1 wage results for both application mixes. In the homogeneous case (Figure 4.10) the quantities of resources received show close approximation to four distinct levels. The mean levels were: 2019, 1516, 971 and 477 milliseconds. Taking the total of these levels and the wage ratio, the ideal levels would be: 1993, 1495, 997 and 498 milliseconds. This gives us a deviation of: 1, 1, -2.5 and -4.2 percent for each level. While a result accuracy equivalent to the equal wage case is not achieved, the maximum variation of 4 percent is reasonable under a simulation environment, since accuracy in timing measurement is not good when running inside a networking UNIX system. Likewise, the heterogeneous case (Figure 4.11) presents a similar situation to the heterogeneous equal wage case. This time the page fault disruptions lasted longer, due to the lower priority of some of the memory intensive processes, which slowed the page fault rate over time. Once this traffic subsided the levels returned to similar levels to the homogeneous case. Thus proportion-share fairness was apparently preserved for distinct wage levels, but because of the relationship between CPU and memory page traffic reduced the potential share obtainable.

The examples just given showed fairness under stable situations. The next experiment examined the change in fairness both as new processes were spawned as children and as new processes arrived at the node. The experiment begin with a matrix multiplication process started at time 0, to compute matrices of size 150x150 with wages of 7500 credits (here there was a total of 10500 credits in circulation). At time 1, it spawned off a child

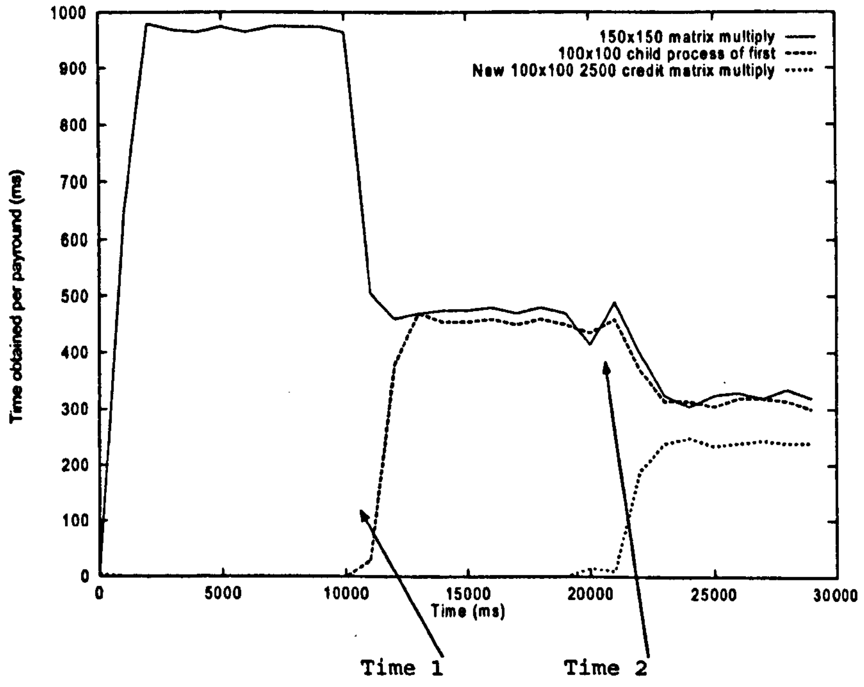


Figure 4.12: Fairness, as the number of competing matrix multiplications was altered

matrix multiplication to compute matrices of size 100x100 with half of the parent's wages. Then at time 2, a new process arrived on the node to compute 100x100 matrices, bringing 2500 new credits to the node (bringing the total circulation back to 13000).

Figure 4.12 shows the quantity of CPU obtained by the processes in each one second period. Initially, the first process received a 100 percent share of the resources, since it had no major competition. Then, after T1, the quantities received halved as the wage was halved between the parent and child. Here the new levels averaged: 485 and 480 milliseconds which approximate very closely to equal shares of the previous 970 millisecond 100 percent share. At time T2, the arriving process brought 2500 new credits to the node, causing it to reduce the quantity of resources claimed by the previous processes, by claiming some for itself. At this point, the new levels of resources obtained by each process dropped to: 325, 327 and 246 milliseconds which closely approximate the new wages ratio of: 3250:3250:2500 credits. Thus, even during disturbances to the number of processes and quantity of money spent on the resource, the levels of the CPU resource obtained

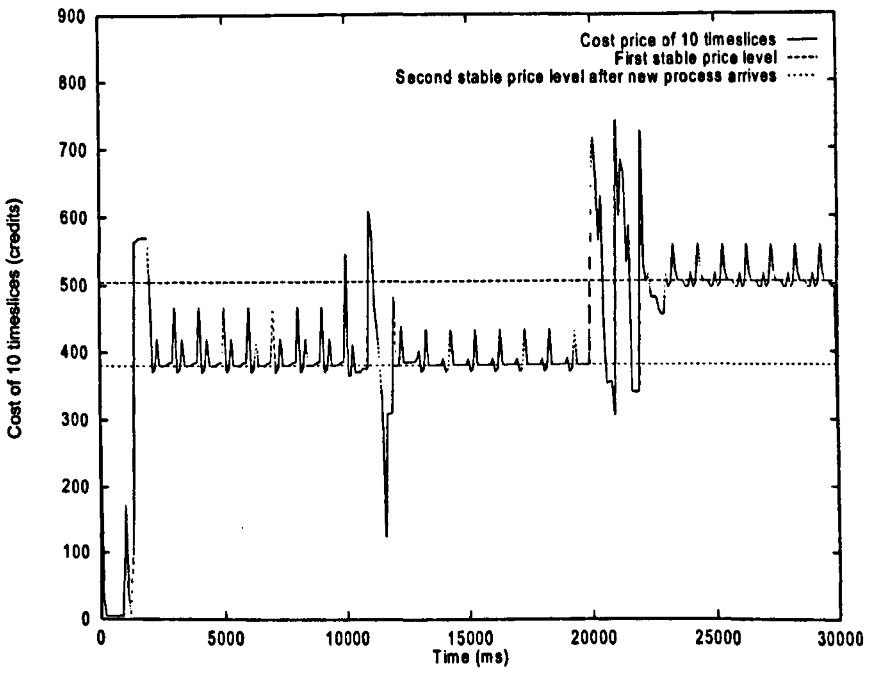


Figure 4.13: Cost, as the number of competing matrix multiplications was altered

closely approximated the wage levels of the process. Thus proportion-share fairness was apparently maintained. The initial latency in obtaining each new level was due to the new processes acquiring their memory 'working set'.

It is also interesting to view this experiment from the point of view of the current CPU cost as in Figure 4.13, since this reflects the share obtained. This cost is the current cost of receiving a time slice given the current competition and is the internal value calculated each time slice in the Cost proportion-sharing algorithm. Initially, the cost was set by the first process since it had no competition[7]. After T1, the cost remained the same even though two processes were now using the CPU, since it viewed them as each having half the importance (money) of the initial process. Thus they shared the CPU equally with the same cost. When the new process arrived at the server at time T2, it spent the money it brought thereby increasing the total money spent with the server, and hence the cost per time-slice. The increase in time-slice cost shared the CPU fairly between the client

[7]The only competitors were the threads used for the ERA system itself, such as wage payment and advert agents, these correspond with the small peaks on Figure 4.13.

processes. Again, cost fluctuations occurred while each process acquired its 'working set'.

These simple experiments provide evidence that, both under stable situations and when there are system disturbances, processes receive a share of system resources approximated by the amount of money spent and thus proportion-share fairness is achieved. Page traffic and RPCs reduced the share obtained by each process, due to blocking, but as a result in the short term once unblocked processes received increased priority.

4.6.2 Scalability of the ERA system

The ERA framework is designed to be scalable both algorithmically and with the size of a distributed system. This section presents experiments to demonstrate the scalability of the ERA framework.

The algorithmic scalability of the monetary mechanisms can be affected by the number of client-server relationships and the duration of the pay round. Firstly, the number of client-server relationships in the system directly affect the amount of work which must be performed by the kernel, the server and clients. Wages and standing orders must be processed for each process, while each server must hold state information for each client. Secondly, the duration of the pay round determines the overhead for wage payment and standing-order dispatch, assuming processes only interact once per pay round[8]. Besides these direct overheads, page faults are also incurred by ERA processing.

Figure 4.14 presents simulation results for the percentage overhead incurred by the ERA system on top of traditional operating functionality. Processes interact only at the beginning of each pay round. The process mix used is the application-homogeneous set of genetic programs. These processes produce a constant number of page faults over time as they evolve new programs.

The dominant factor in Figure 4.14 was the increase in overhead as the number of processes per node was increased. This was due to a dependence on the number of processes in most of the algorithms in the ERA framework, for example: client-server relationships;

[8]Under the general framework, processes may in fact interact more often

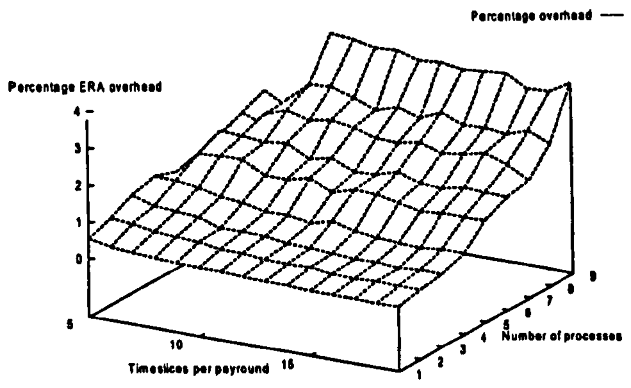


Figure 4.14: Percentage overhead per second produced by the ERA system when compared to a traditional operating system, as both the length of pay rounds (in time-slices = 100ms) and the number of active processes are varied

money interactions; page fault traffic. Two other features are observable in the data. Firstly, there is a linear relationship as the duration of the pay round was decreased. The magnitude of the gradient of this relationship was slight because the overhead for updating the wage figure and dispatching standing orders was small compared to that for page faults. Secondly, the furthest point of the graph drops to a lower level. This was the point at which processes could not receive at least one time-slice each in the pay round, since the number of processes exceeded the number of time-slices in a pay round. This in turn reduced the page faults produced and RPC calls and dropped the resultant overhead.

In comparison with previously published figures for proportion-share systems, the ERA framework performs well in these experiments. For SPAWN [55] an estimate of approximately a 10 percent overhead under a simulation while running Monte-Carlo simulations is given in [55]. Waldspurger's Lottery scheduler [57] is a similar proportion-share system implemented on the Mach operating system to provide single node sharing. Figures on the overhead of this system are not well documented, but [57] cites a slow down of 2.7 percent for 3 Dhrystone benchmarks executing concurrently. ERA imposed a measured overhead of 3.7 percent when running 9 genetic processes with a short pay-round. Under a 5 process scenario ERA imposed an overhead of 1.6 percent. Given that the ERA experimental results were unoptimised simulation results, the ERA framework seems to impose less overhead than its competitor and is more general.

4.7 Summary

The ERA framework is based on the idea of a monetary framework to combine the competing attractions of process control and operating system controlled resource fairness. In the monetary scenario, each process buys the resources it needs from resource servers. In the act of purchasing, each process spends an amount of money with the server which quantifies the important of that resource to it. In return, the process receives access to the server's resource in proportion to the amount paid.

Processes compete for resources, and so, to control the amount of access each process receives, the operating system controls the amount of money which the process has. This ensures fairness, prevention of deadlock and prevention of starvation, while allowing a process as much control as possible. Several mechanisms have been designed to support this monetary architecture including: operating system support, process support.

Experiments with simple test cases of application-homogeneous and heterogeneous mixes have demonstrated that each process appears to receive its fair share of resources within a variation of 5 percent. At the same time, processes which do not consume their total allocation of resources allow their competitors to receive more resources. The fairness of the framework was also examined over time, under conditions of spawning new processes and new processes arriving at a node. In both experiments the results indicated that proportion-share fairness was preserved.

Further experiments examined the overhead and scalability of the ERA framework running on a single node. Important parameters such as pay round period and the number of processes were varied under a typical load. In these experiments the result showed that, under these conditions the overhead of the ERA framework remained below 5 percent, and the trends for each of the parameters showed good scalability.

To summarise, these results indicate that it is possible, using a monetary framework, to provide processes with control over all their resource allocations, while maintaining fairness during competition from other processes. The experiments indicated this fairness control can be achieved at fine-granularity and with good accuracy, while imposing little extra overhead and scalability constraints on the system.

Chapter 5

Providing the marketplace

In the last chapter, the ERA framework was introduced. This allows processes to perform process-centric allocation in an environment of dynamically shared resources. Process-centric resource allocation allows a process to be in control of its own resource allocation and thus choose resources which are appropriate for its function.

A simple example is a server process serving some functionality across a distributed system to many clients. In this scenario, the location where the server should run is dependent more on the server's function than a general balance, when performance is an issue. For example, the position of the server will affect the latency of access to the server by its clients. At the same time, its location may provide it with fast access to an important resource for its function. Consider a file system server connected to a Redundant Array of Inexpensive Disks (RAID). The server could be placed at the machine with the RAID, since doing so improves access by the server to the disk. But, the server can position itself on a caching server near clients thereby reducing the latency of servicing requests.

For this choice to be meaningful the choice must be informed. The client needs to make decisions based on a knowledge of its surroundings rather than blindly. In such a scenario, the more information available, the better the decision can be.

A process using the ERA framework only has access to resource servers it knows about. Access to these servers includes the ability to buy their resources, but also the ability to enquire their price. The resource price indicates the demand for and utilisation of the resource. In order to make a decision, a process needs this information about both the resources it uses and those which it could potentially use. Using the ERA framework alone, a process only has the ability to directly query each resource server in turn.

This chapter introduces the ERA marketplace which provides a service to satisfy the information service needs of processes. The marketplace is constructed using the ERA framework itself and using the monetary concepts embedded within it. It is a scalable information resource for the discovery of new resources and their price information.

Firstly, this chapter introduces the issues of a marketplace through an overview of the desired properties. Following this, an algorithm is introduced to provide them. Using this algorithm, the actual design and operation of the marketplace is discussed. Finally, in this chapter, results from simulating the marketplace are given, demonstrating its scalability with system size and the operation of dynamic locality.

5.1 Motivation

When extended to the distributed case, the ERA framework consists of many individual nodes, which share nothing but their communication network links. Each process can obtain resources by using its default servers. This is sufficient for simple processes, like UNIX's `'ls'` this is sufficient. The process can purchase the necessary resources, from the servers given to it by its parent, in the form of standing-orders to CPU and memory. At creation, the standing orders are dispatched to these servers and the process can execute for as long as it is allowed to. Simple processes only need to interact with the ERA framework to buy any resources blindly, if they do not care about the quantity of access to resources they will receive.

Some processes will want to control which particular resources they receive. By controlling which resources a process receives, it can improve its performance in several ways. Typically a process may choose to place itself on the cheapest node (this leads to a load balance) or else might consume other under-utilised resources to increase performance. In these scenarios, since the process is solely responsible for its resource allocation it must choose which resources to buy in order to satisfy its aim. Since the ERA framework provides a shared-nothing resource information configuration, a process executing on a particular node has access to resource servers on its local node and access to any other servers it knows about. For each of these servers, the process can enquire about current prices to make its decision. Potentially, this can cause two problems: a process may have little information on which to base its decision; repeated direct querying of remote resource servers causes poor scalability. What is required is a means of disseminating the resource information in a scalable way, so that a process can find the optimal resources for its use.

The simple solution is a directly shared resource, such as a shared memory segment or a centralised server. While suitable for small systems, such solutions impose scalability problems. A shared memory segment requires any writing process to gain write access to those shared pages which, due to false-sharing and invalidations, leads to poor scalability. A centralised server scales poorly due to the finite power of the hosting machine, network latency and bandwidth.

The usual solution is to use 'nearest-neighbour' algorithms as discussed in Chapter 2. These algorithms provide information about the resources of nearby nodes in the network configuration. The key problem with nearest-neighbour techniques is that the limited information available at each node can limit the quality of the decisions made. Even when Time To Live (TTL) counts are used, the bounds of the propagation are finite, through imposing a fixed load on the system, which reduces the available resources to the applications themselves.

To overcome these problems, the ERA architecture uses some of the monetary ideas of the ERA framework to provide an ERA marketplace. In this context, the term 'market' means the abstract idea of a place where goods are bought and sold, although is often easier thought of as a bulletin board. In a market, goods are bought and sold at an advertised market price. Each market may contain several similar goods, each with different prices. Also, the prices in one market may differ from a remote market. Real-world markets present information, about resources (goods), which is potentially incomplete and out-of-date, across a distributed environment (the real-world) in a scalable fashion. This is similar to the information present in distributed computer systems where, to ensure scalability, each node can only contain potentially incomplete and out-of-date information about the rest of the system.

The idea of a marketplace has some other appealing features. Firstly, the overlap in information in each part of the market leads to global price equilibrium, since active participants in the market tend to choose the cheapest resources to achieve their goals. This is Adam Smith's classic Invisible Hand argument [50]. This equilibrium leads to a balanced use of resources in ideal market conditions[1]. Secondly, advertising a good in the market costs money. If a lot of money is spent advertising a product, then its market propagates further afield than if little money is spent. This is a difference between the advertising ability of multi-nationals and small local producers.

The ERA marketplace uses these features of real-world markets to provide a more flexible solution than 'nearest-neighbour'. The ERA marketplace provides a *shared* information resource which describes resources available at and near each node. This environment describes the price information of various resources, which can be used by the processes to make their resource selection decisions. Firstly, information held at each node only describes its locale[2]. This is similar to 'nearest-neighbour'. This information is thus potentially incomplete and out-of-date. Secondly, the amount of information at each node varies, depending on the cost to advertise, and thus system load. When the system is heavily loaded advertising costs more and so the market contains more local information. However, when the system is more lightly loaded advertising is cheaper and will contain information from further-afield. This enables the system resources used by information dissemination to vary with load so not to interfere with the executing processes.

[1]In the real-world things are not so simple. Unequal wages, structural unemployment, locality, etc. cause market features such the North-South divide.
[2]The area surrounding each node over its network links.

5.2 The Marketplace

The marketplace is a set of interconnected markets, each of which contains information on resources in its locale. The size of this locale at each node varies according to load. A process accesses its local market and receives information which is currently applicable to it. To provide this functionality a new algorithm for information dissemination called the 'Dying Sandwich-board Men' algorithm (DSBM) is needed. This algorithm will be described in outline, before the ERA marketplace is described.

5.2.1 Dying Sandwich-board men

A sandwich-board man[3] is a person who advertises a product by walking up and down the street carrying an advertisement on a "sandwich-board". Each sandwich-board man is paid to perform this tedious task, with the amount paid indicating the importance, duration or distance the sandwich-board man will walk. This algorithm takes its operation and name from these fellows.

The basic idea behind the algorithm is that it costs money to advertise your service to others. Sandwich-board men are paid money to wander about advertising a product. The more paid, the further or longer they'll perform this task. Imagine that the money paid to a sandwich-board man must be spent on obtaining all the resources needed to live, in this case food and drink. Since the sandwich-board man is only given a finite amount of money he can only advertise for a finite distance before keeling over and dying.

With this simple approach the time it takes to propagate the information to distant places can be high, since the one man must walk the entire network. A sandwich-board man may, however, sub-contract other sandwich-board men to advertise the same information, for a share of his money, with each man heading off in a different direction. If sub-contracting happens at every junction, then information can be distributed in parallel.

This propagation of information continues, with sandwich-board men heading down

[3]May be Sandwich-board person would be more politically correct.

different streets with the same advert, until eventually every sandwich-board man runs out of money and keels over and dies. At this point the information being advertised will have propagated a finite distance from the originating point. The distance travelled in each direction will be affected by how expensive food and drink are at different points in the system. Where these are expensive then the information travels less far than in cheap areas. This in turn affects the amount of information available at each point (locale) the sandwich-board men have visited.

5.2.2 Resource information

Advertising mechanisms in the ERA marketplace must be adequate for processes at remote nodes to make informed decisions. From a local perspective, each resource server calculates a current price for its service, but there are potential problems in the distributed case.

Firstly, there are problems of price information and heterogeneity. A price encapsulates the demand and utilisation of a resource. A resource in high demand is priced higher than a low demand resource. In a heterogeneous environment, comparing prices for different resources presents many problems. For example, prices may be set such that equal prices occur when an equal share of the number of time slices of CPU is available on each piece of hardware. Yet, one machine may be faster, have a larger processor cache, etc. all of which make it difficult to quantify the performance a particular application will receive from the resource. Similar problems exist in heterogeneous memory and network environments. For these reasons, heterogeneous environments are left outside the scope of this thesis. However, this is not a inherent limitation of the architecture, since such problems could be overcome by creating processes which monitor their resource usage and determine the relative usefulness of the resources they consumed.

Secondly, resources can generally be used locally or remotely. i.e. a process can either migrate to the location of the resource and use it or, it can use the resource remotely (see Figure 5.1). This means that potentially there are three potential kinds of cost associated with using a resource: choosing a resource; migrating to be able to use that resource; and

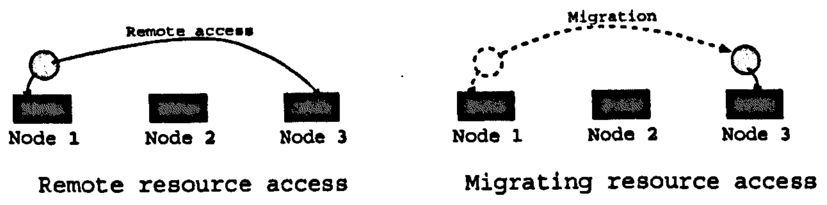


Figure 5.1: The two different ways to access a resource: remote access and migration access

accessing the resource itself. To simplify the simulation work in this thesis, a shared memory model is assumed although any model could in principle be used. In this environment, the costs of accessing and migrating to a resource are hidden by the memory system. So, in this thesis, it is assumed that only the resource access costs need be advertised.

Thirdly, while price information represents the demand and utilisation of a resource, the information's timeliness must be considered. Out-of-date information about distant resources is likely to be less useful than up-to-date information about nearer resources.

In the ERA marketplace to overcome these problems resource information is presented as the price of the resource combined with the time at which that price was set.

5.2.3 Provision of information

Information provided in the marketplace must be paid for by users of the advertising server. This means that there is a finite of amount of advertising each resource server can obtain, since it has finite funds. This means that a resource server must choose when to advertise resource information and with what importance. For example, a server with 100 credits to spend per pay round could advertise its resource price information once per pay round with the full 100 credits (synchronously). Or, alternatively, intermittently when the price changes sufficiently with 30 credits while money lasts (the asynchronous approach). Either approach is fine, but the quantity of advertising a server can exhibit is finite and is that server's responsibility.

To provide access to the marketplace with good locality, the operating system on each node holds the address of its nearest point of access into the marketplace. Each process uses this to access the marketplace. Advertising is obtained by resource servers in the

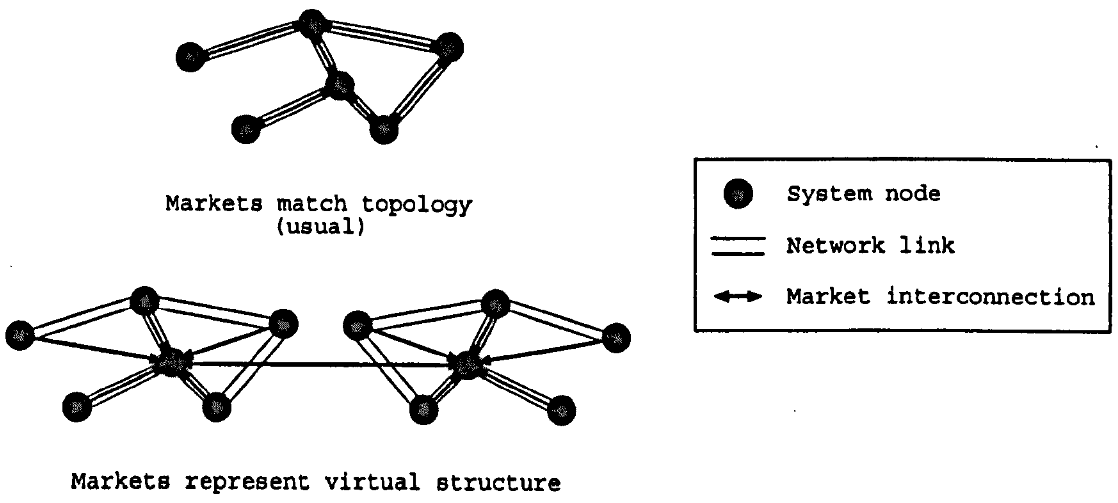

Markets match topology (usual)
System node
Network link
Market interconnection
Markets represent virtual structure

Figure 5.2: The market 'nearest-neighbour' interconnections can either match the physical links or represent a virtual arrangement

same way that processes buy resources such as CPU and memory. They send a contract to the market server along with up-front payment. When a process wishes to advertise a resource, it calls its market with the resource price, which is then time-stamped. The information is then distributed across the marketplace using the DSBM algorithm.

5.2.4 Distributing of information

In ERA, the marketplace is formed from a set of interconnected overlapping markets. Each market is a ERA process which cooperate to form the marketplace. Each market represents a point of presence in the overall marketplace. Each market is a storage point for resource information (adverts). Each market has a list of the addresses of its neighbouring markets in the marketplace, these are its nearest-neighbours. The usual configuration of this marketplace is such that a market exists for each node and the neighbour addresses reflect those of the underlying network, but more abstract configurations are possible. This structure can be seen in Figure 5.2.

When a resource server decides that information needs to be advertised, it calls its local market with the price to be advertised. At this point the DSBM algorithm is used. Each market acts as a store of information and potential splitting point for more sandwich-board men. Initially, a process called an *agent* is created representing a sandwich-board man. Each agent's sole purpose is to take the information it is given to a particular destination market and these advertise the information there. When the market is called to start advertising a particular price it creates an agent, giving it the resource information and a destination. Each agent is given a once-off payment when created (not a wage). This money must be used by the agent to achieve its goal, i.e. running to advertise the information remotely. Each agent performs the algorithm:

1 Travel to the node which contains the destination market. (If necessary this involves migrating the agent's context to the remote node.) In doing so, some of the agent's money is consumed.

2 Call the market (through a special API) to advertise your information with the market.

3 The market takes the information and stores it locally. It then creates duplicate agent processes to propagate the information to its neighbouring markets. Each new agent is given an equal share of the money the calling agent has left over from executing to this point. Likewise, each agent is given a differing neighbouring market as its destination.

4 Repeat to step *1* while sufficient money remains.

5 Eventually the agent and its children run out of money and there is no further spread of the information.

In this way, the information is stored at each market and enough additional agents are created to propagate to neighbouring markets. The important property of this method is that the distance the information travels is determined by the cost of the path where the cost of the path depends directly on the demand (competition and utilisation) for resources on that path. The importance of disseminating the information is indicated by the amount spent on it and so important information travels further.

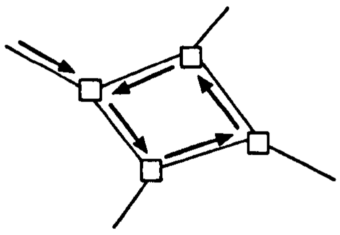


Figure 5.3: A cycle in the information flow due to the cyclic nature of the underlying 'nearest-neighbour' interconnections

Note that most of the functionality of the algorithm is embedded in the market, not in the agent itself. The agent is simply created with the information and a destination and repeatedly loops calling the next market from the current market, migrating as it goes. By separating this functionality prevents the marketplace being subverted by imposter agents. The market code itself runs under the agent's context, stores the information and spawns additional agents to do its advertising work.

Caveats

There are two problems with the algorithm just described. Firstly, since the graph of network links can be cyclic, it is possible that the advert information can be sent in cycles. Secondly, since money is consumed continuously by its execution it is possible for money to run out and the agent to be terminated before vital updates are performed, potentially causing inconsistencies.

Cycles

Cycles in the information flow can occur because the directed graph of network links is cyclic (see Figure 5.3). At any particular market, a new agent may be created which, through the network links, will eventually return to the current market and so computational effort and network bandwidth are wasted with these cycles continuing until money runs out for each agent.

Simple cycles such as returning the same information down the link which was just used can be overcome by checking the new destination against the source of the agent. This information is available since each agent keeps a record of the last market it has visited and passes this to the market when called.

A market can easily overcome more elaborate cycling by using time-stamps. Each agent is time-stamped when originated, so that the advert it carries is unique across the system. When a new agent arrives at a market the time-stamp of the advert is checked with that of the stored information. If the time-stamp is newer, the information is updated and new agents spawned, etc. If the time-stamp is the same or older, then the information is discarded and no further work performed. In this way, information can travel around the cycle at most once since, when an agent reaches a node which has already been visited, the spread of information is stopped at that point. Since money is preserved, the agent's money is returned to its parent.

In-place structure updates

Updating in-place structures is a problem in any situation where execution may accidentally be terminated while the update is being performed.

In the ERA marketplace the agent processes update resource-information list structures. The marketplace itself need not run with any special privileges and thus runs in user-mode. Even with the use of mutual exclusion on the list structures it is impossible to ensure that the user-domain code does not terminate while updating these shared structures. An obvious solution is to run part of the market in kernel-mode with all interrupts disabled. This would prevent termination due to a lack of money, but does not prevent problems with buggy code. It also has the side-effect of making the market a special case. Instead, what is required is a general way of performing the list updates atomically by a user-domain process.

To provide this functionality, a generic atomic pointer update function was added to the kernel. This function allows a pointer value to be updated atomically with a new value, even if money has run out. If sufficient funds exist the memory pointed to by the old pointer is freed, otherwise the data pointed to by the new data is freed.

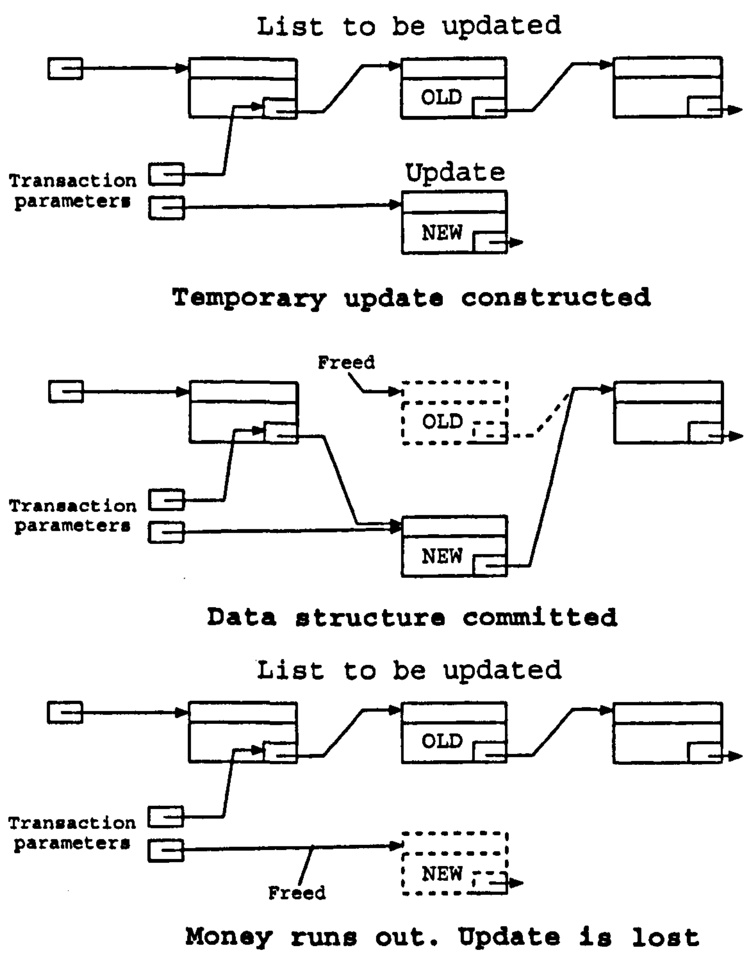


Figure 5.4: The operation of the atomic kernel pointer update primitive when used to update lists

The function takes two arguments, the address of a pointer and a new value. If money is available, the data which the first address points to is set to the new value by the kernel, without using the ERA framework and with interrupts disabled. This ensures atomicity. If money has just run out (rare) then the old pointer is not affected and when the process dies the data pointed to by the new value is freed. To achieve this effect, when memory is added to a shared data structure it must removed from ownership by the process. This prevents other parts of the list being deleted by later terminating agents.

The operation of this primitive for list updates is shown in Figure 5.4. The agent creates the new information in a temporary list element and makes the structure point into the list at the correct place. When the temporary element for adding into the list is complete, the address of the old list pointer and the new block are passed to the primitive for swapping. On success the element is part of the list; on failure the list is unaltered and the temporary element is deleted.

5.2.5 Consumption of information

Processes can use information in the marketplace in order to improve their process-centric decisions. The market allows any process to query it about a type of resource for which price information is desired. It returns a list of address, price and timeliness triplets of all the information which it has concerning that type of resource.

When and how a process uses this information is up to it. Typically, when a process wants to make a decision (periodically), it will obtain the information from its market. It can then use this information and any historical information it may have kept, in order to decide the type of resources to purchase. Example techniques are examined in Chapter 6.

5.3 Analysis

The emulated system used to obtain the results presented in this section is based on a ring of nodes connected by point-to-point links presenting a one dimensional view of the scalability and information locality of the algorithm. This network configuration is used to emulate distributed shared memory combining each node's emulated storage. For the purposes of the emulated distributed memory system, remote page faults are modelled as random events with a normal distribution with mean 10ms and variance 1.5ms. This means that memory latency, rather than bandwidth is modelled.

Information dissemination

The first experiment measures the information dissemination as the load on the system is altered. A distributed system of 10 nodes was emulated. The load on the system

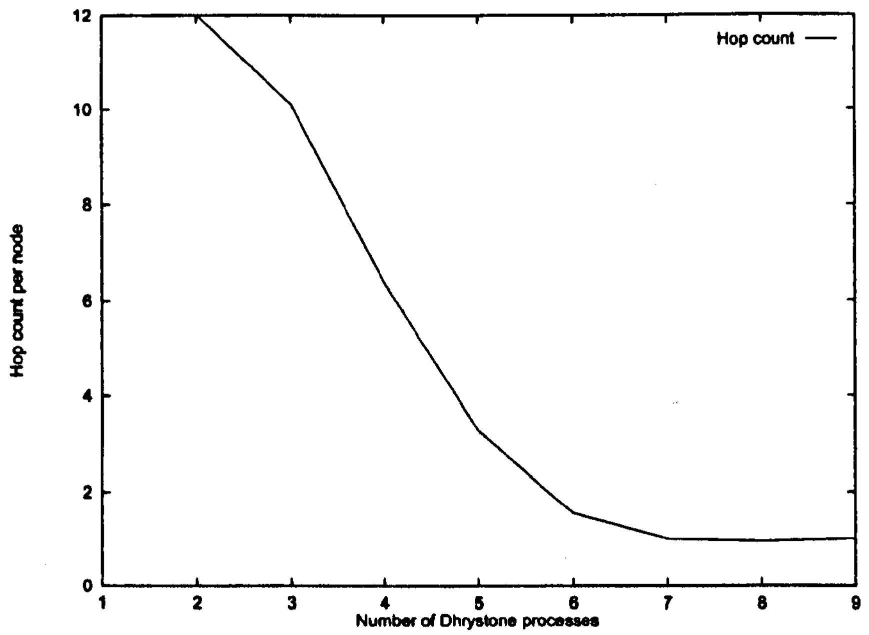


Figure 5.5: The average number of hops per node for each loading of benchmark processes

was uniformly increased by running successively more processes repeatedly computing the Dhrystone benchmark (although any load has the same affect). Several runs of the system were performed as the load was increased from 1 benchmark per node up to 10. All nodes ran benchmark processes with 1000 credits each (spent equally on CPU and Memory) and CPU was advertised with only 1000 credits.

Figure 5.5 shows the average number of hops per node for each loading. Figure 5.6 shows the corresponding average prices for CPU at each loading. Initially, when the price of the resources was low the number of possible hops was found to be 12. In this region the information propagated around the entire ring (10 hops), plus initialisation and cycle detection (2 hops). As Figure 5.6 shows, as the load increased the price of CPU increased causing a decrease in the number of possible hops the information could travel. As the load increased further the increase in price fell off as the ability to afford the CPU reduced. This continued until eventually the load was so high that the prices were too high to make remote advertising possible leaving one initialisation hop.

The second experiment measured the information dissemination again, but this time

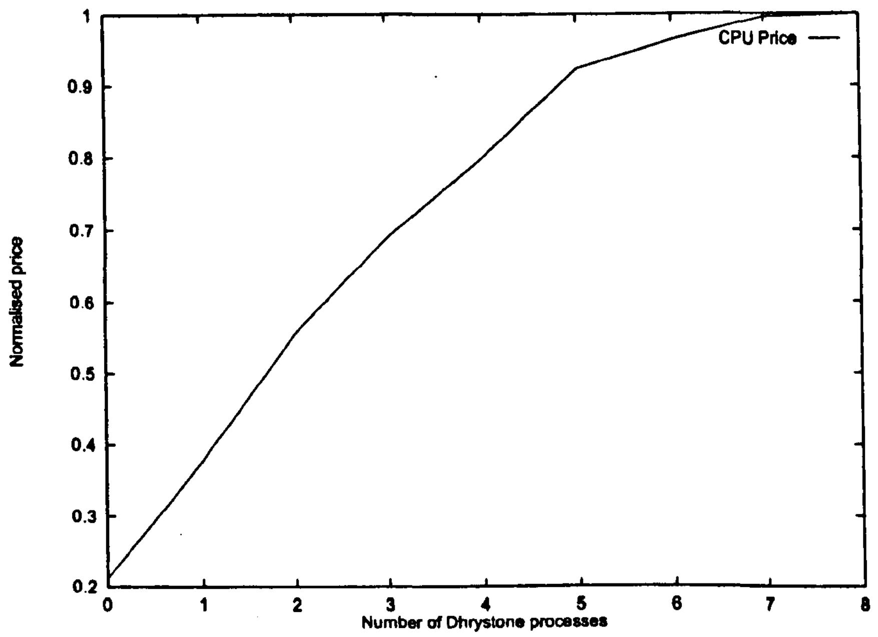


Figure 5.6: The average prices for CPU at each loading of benchmark processes

the load on only one node was increased to show the effect of a point load on the distance travelled by information from neighbouring nodes.

Figure 5.7 shows the average number of hops for each node with six additional benchmark processes running on node 5, each with 1000 credits. As before, all other nodes ran a single benchmark process with 1000 credits and CPU was advertised with only 1000 credits. The proximity of the nearby busier node affected the distance information can travelled from that node, since travel through the busy node was expensive. This resulted in a 'V'-shaped curve of information locality as agents got closer to the busy node. In this experiment, node 5 had the increased load and thus had the smallest locality, since all information disseminated in either direction incurred the same high costs. As the distance from the loaded node increased, the average locality increased, since the need to use the busy node reduced. This demonstrated that, in this simple scenario, how far information travelled was affected by non-uniform workloads. This illustrated that the distance travelled by agents can be affected by the load of the system.

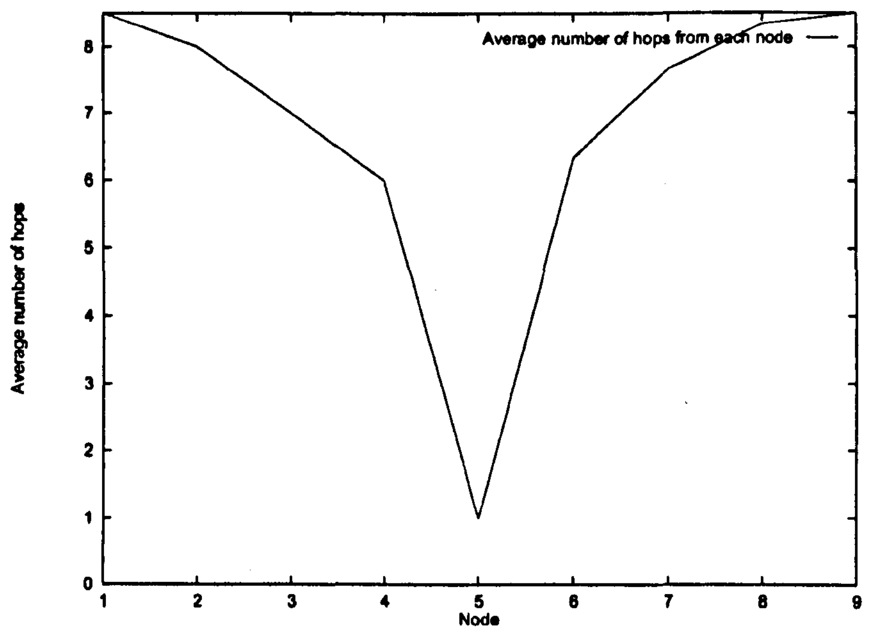


Figure 5.7: The average number of hops for each node with six additional benchmark processes running on node 5

Scalability

To demonstrate that the ERA marketplace, and thus the DSBM algorithm, are scalable, the overhead of an emulated system was measured as the system size was varied. In this experiment, the scale of the system was varied from 1 to 25 nodes. Each node ran processes for the market, CPU and memory servers as well as an application process repeatedly computing the Dhrystone benchmark. Each process was given a wage of 1000 credits. The CPU servers advertised their prices regularly with their entire income.

The result of this experiment is presented in Figure 5.8 and shows the total normalised average overhead of maintaining the marketplace using the advertising agents. Initially, the overhead increased non-linearly to a scale of 7-8. In this region, the scale of the system was smaller than the possible distance that could be afforded and so was bounded by the system's scale. From this point onwards the overhead continued linearly. In this region, the economics of the system bound the information dissemination. These simple experiments indicated that the increase in overhead with system scale is linear and thus the DSBM algorithm scales linearly.

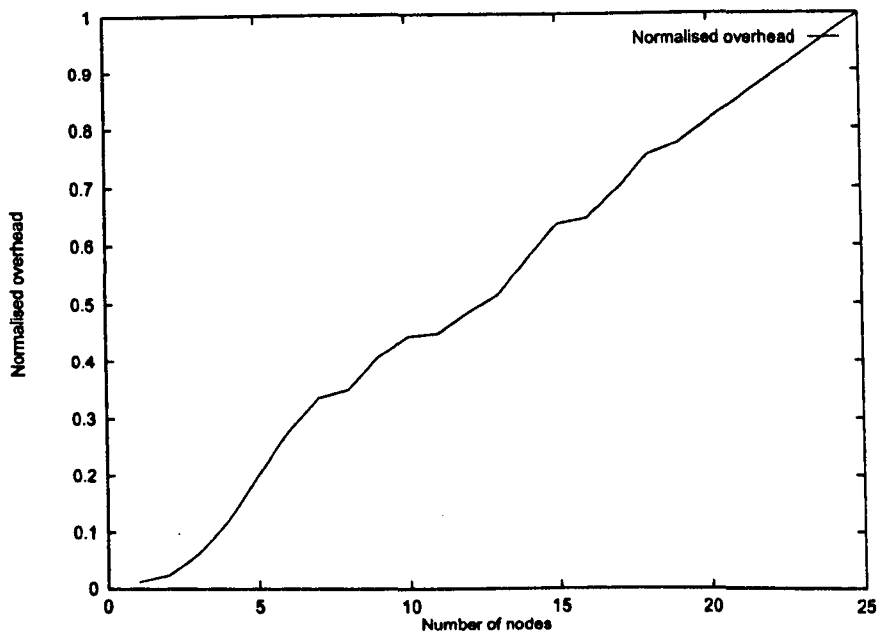


Figure 5.8: The total normalised overhead of maintaining the ERA marketplace as the size of the system is increased

It should be noted that this linear finding holds only for when the quantity of advertising (i.e. the number of advertising requests made) is constant. Clearly, if advertising is increased (for example by adding a new service to the system) then the overall cost will increase, however, this cost should scale linearly. This is also true if the frequency at which adverts are placed is increased. In these experiments servers only advertised once per pay round and at the beginning of that pay round. If a server placed many adverts then this also would increase the overall overhead.

5.4 Summary

The ERA marketplace is designed to provide a more effective solution to the problem of disseminating information in a scalable fashion than the traditional 'nearest-neighbour' techniques. The marketplace is constructed on top of the ERA framework so as to provide an additional resource "advertising" which allows process-centric processes to purchase information dissemination. The motivation for this arrangement is that spreading information around a distributed system is not free. It requires resources and so should be paid for. As a side-effect of this approach, the locality of dissemination of information is designed to vary with system load thereby reducing the distance information is disseminated when parts of the system are heavily loaded.

To provide the marketplace "the Dying Sandwich-board Men" algorithm is proposed. This uses the monetary framework to provide the dissemination semantics. The algorithm is fairly simple, but presents a couple of problems: updating in-place structures under termination conditions and cycles in the dissemination of information. Simple solutions to these problems are proposed.

The effectiveness of the ERA marketplace was investigated in a simple situation using the ERA simulator. Firstly, the attenuation of information under varying load was examined. The experimental results indicated that as load increases, so does the price, reducing as a result the distance over which information is disseminated. The dissemination of information was then examined for non-uniform load conditions. The results indicated that advertisements near high load situations have their distance curtailed appropriately. These experiments indicated that information can travel large distances up to a hot-spot. Finally, the scalability of the marketplace was examined as the system's size was increased. Results from a set of experiments indicated that, even with the extra functionality, an ERA marketplace scales linearly up to 25 nodes.

Chapter 6

Process-centric resource allocation

"Money makes the world go round"

(Unknown)

An important purpose of the work described in this thesis is to provide resource allocation which is under the control of a process while in an environment of multiple processes. With this control, a process should then be able to make resource decisions which are advantageous to its own execution and which should be superior to generic decisions made by the operating system. This approach is termed *'process-centric'* resource allocation.

The previous two chapters have described techniques and algorithms to provide the support for this approach. Chapter 4 describes the ERA framework this provides support for allowing a process to express the importance of resources to its execution as a means of controlling its resource allocation. The framework presents this control in such a way that control is relative to each process allowing the effect of a process to be controlled by controlling its relative priority. This allows the combination of process controlled resource allocation, while the operating system controls the overall affect which each process can have.

To provide this support, monetary ideas are adopted as a means of expressing the importance of each process and the importance of each resource to each process. The framework does not describe how prices are determined by the buying or selling processes. Neither does it describe how price information is obtained or distributed.

In Chapter 5 distribution and access to price information was discussed. This is the ERA marketplace, which disseminates information within a locality which varies with system load. The ERA marketplace was built upon the ERA framework. In combination, the techniques introduced in these chapters provide all the operating system support needed for process-centric resource allocation.

This chapter discusses process-centric resource allocation from two process viewpoints. Firstly, the provision of a resource server in this process-centric environment and pricing the resources it serves. Secondly, that of using price information by processes in order to perform resource allocation decisions based on their functional needs. Also in this chapter examples are presented of how these prices can be set and used, as a proof-of-concept. In this way, this chapter aims to answer the question posed at the beginning of this thesis:

Is it possible and **useful** *to empower a process with more control over its resource allocation while preserving fairness for general-purpose distributed systems?*

6.1 Server pricing mechanisms

A server process is an ERA process which has, as part of its responsibility, the allocation of one or more resources to client processes which will pay money in return for access to those resources. Server processes present the supply side of traditional supply and demand microeconomics. Thus the service's duty is to maximise its utilisation (supply) and avoid over-demand for resources in limited supply. Thus the price of a highly utilised resource should be higher than an under utilised resource and a resource for which the demand is higher than supply should be even more expensive. How the server performs its duty is not defined by the ERA architecture, with the exception that all servers must follow the following constraints:

- Share access fairly between clients, in accord with the amount of money paid by each process. This constraint prohibits poorly behaving server processes which accept money and give no or little access in return.
- Provide a meaningful price, to any enquiring process, which reflects the historical demand for the resource. The price should have an equivalent meaning across all resources with the same identifier. For example, all resources "CPU" must have the same price semantics, but resources "CPU" and "SparcCPU" need not.

In order to achieve this aim under these constraints a server has access the three sets of data: the potential client demand, the actual current client demand, and historical demand/utilisation data. The potential client demand is indicated by the total money paid to this server at the start of this pay round. The actual current client demand is the short-term demand which the server has seen, for example, since the beginning of the pay-round. Historical demand/utilisation data can be recorded by the resource, along with other resource specific information about past demand, utilisation and pricing. It is important to note that sending a contract to a server does *not* determine the quantity of the resource that a client will receive. This means that the server cannot calculate the actual demand it will receive from its clients.

The server must therefore, balance the actual demands of the client against the supply. It does this by altering the advertised price of the resource according to both the payments made and the demand presented over time by the clients.

6.1.1 Cost proportion-sharing

Sharing a resource fairly between clients is about determining an efficient proportion-share technique for the particular resource and using the money paid by each client as a basis for its proportion-share. This thesis uses a method, for proportion-sharing, based on the monetary principles already in the ERA architecture, called "Cost proportion-sharing". Cost proportion-sharing performs a proportion-share by combining the calculation of a useful price for the resource with the action of sharing it. The principle behind this technique is that if the cost is correct then each client can only afford to buy a fair share of the resource. We term this the "cost-price", since the cost of each access is calculated and is used directly in the advertised price.

Consider a simple scenario. At the beginning of a pay round a server has two clients; one (A) pays 80, while the other (B) pays 60. Before the next pay round the server has 10 units of resource allocations which it can share between the clients. By cost proportion-sharing the price is determined to be:

$$CostPrice = TotalPaidToServer/NumberOfAllocations$$

This would give in a cost price of 140/10 or 14 per allocation. A client is able to receive a share of the resource if it has more money than or equal to this amount. Initially, both clients have enough money and receive a share of the resource in turn. At each allocation the cost price is deducted from their current credits at the server.

This leads to the following credit sequence:

$$(80,60)(66,60)(52,60)(52,46)(38,46)(38,32)(24,32)(24,18)(10,18)(10,4)$$

This results in an allocation sequence: *AAABABABAB*, resulting in a 60/40 (3/2) share of the resource. This is an integer approximation of the correct 80/60 (4/3) share. The variation is due to the division of the resource into integral time slice units. The correct allocation would require allocations of 5.71 and 4.28. 6 and 4 was a close rounding.

One problem with this simple scenario is that, when a large difference exists between the prices paid by each client, the richest client monopolises the resource, until the clients have approximately equal value whereafter they receive equal shares. This results in an allocation sequence like: *AAA...AAAABABABABAB*. This monopolisation can be overcome by serving resource requests in round-robin fashion (while enough money exists for all clients), by using the decayed prices as the basis for probabilistic selection of each client, or by using an algorithm such as Bresenham's line drawing algorithm to distribute the differences more effectively.

This research took an approach based on Bresenham's line drawing algorithm [8], but several contemporary proportion-share approaches have appeared during this Ph.D. work. So consideration of algorithms for proportion-sharing was no longer pursued, instead work concentrated on the framework, marketplace and process-centric policies. For more information on this contemporary work, see Chapter 2.

One important advantage of cost proportion-sharing is that it is possible for a client to quickly determine whether or not it is completely utilising its *potential* share of a particular resource. The amount of money left at each server, for each process, varies in *direct* proportion to the utilisation of the resource. This means that should a process have to block due to, for example, a page fault the client can see this lack of utilisation. In response and to better utilise the resource, a client can then chose to use the rest of this resource for other purposes. This applies to all resources in the system.

6.1.2 Serving CPU

Allocating of time on the CPU is the most common resource allocation problem for operating systems. The resource is multiplexed over time in units, known as time slices. At any time, units of execution (here assumed to be a thread) can be in one of two states: *ready* to run, *waiting* for another resource. Timeslices are a simple resource to allocate. The ready queue processes (those that are ready) is the current demand for the resource. Those in the wait queue are not currently contending for the resource and so may be ignored.

To allocate the CPU between threads, the server must keep the traditional queues of process IDs plus a list of the current credits remaining for each client. This list is reset at each pay round. While a thread has money, on this list, and it is on the ready queue, then it is able to be run. This means of course that a thread has two potentially separate pieces of state stored in the CPU server. Firstly, it has its Process Control Block (PCB) stored on the ready queue. (This indicates the thread wants access to the CPU resource.) Secondly, it is on the monetary list (indicating that this thread has credit to use the server.) If only one of these pieces of state exists on an instance of this CPU server, then the thread is unrunnable. Standing orders provide the means by which a thread is always able to place an entry in the monetary list, without running thereby ensuring that its state is kept together and that a thread's runnability is only determined by whether it is *ready* or *waiting* and whether it has any share remaining from the proportion-share allocation.

The price advertised by the server, using cost proportion-sharing, could simply be the current price calculated by the algorithm using the ready queue as data. This, however, has the affect of presenting a volatile price over time, as the cost can vary quickly when processes block. Instead, in this thesis, servers advertise an average of the price over the last pay-round.

6.1.3 Serving memory

Serving memory presents more problems than serving CPU time slices. In the case of the CPU server, the allocation model provides the server with some useful information about its clients. Firstly, due to the ready queue it knows the current demand for the resource so it is able to determine an effective future allocation for these requests. Secondly, because time slices are integral units the CPU server can easily determine the number of remaining allocation units it has available. It therefore knows when to select the next client and accordingly revokes the CPU as and when necessary. Thirdly, the resource only has one variable, time, which must be multiplexed between clients.

Memory, however, has several additional details which complicate proportionate resource allocation. Firstly, the entire memory hierarchy is *not* preemptable. Typically a memory system is divided into three layers: cache (this may be two levels), real memory and virtual memory. Virtual memory is not preemptable, while the others are. This means it is not feasible to revoke a virtual memory page, since the important data stored within it will be lost. [1]

Secondly, the resource has an additional parameter for allocation besides time, namely

[1] Some systems provide this functionality, but require that the process be informed about the potential revocation and be allowed to attempt to keep the page [13].

space. A memory system can maintain multiple clients simultaneously, by having different areas of memory allocated to different clients. This means that the resource has potential to be managed in space over time (where preemptable).

Thirdly, the resource does not consist of a well-defined finite supply which must be shared between clients with high demands. This is because the time to access of pieces of memory is potentially variable. For example, depending on where the disk heads are when a virtual memory page-in request is made. This makes it difficult to quantify a maximum supply.

Fourthly, the actual demand is not known until the memory request is made and for significant periods of time the resource may remain idle, only to receive a large demand moments later as some action occurs. With the CPU resource, demand is less volatile allowing easier control.

Another possibility is to share the space of the memory resource and disregard page faults. Since the entire memory resource is not preemptable, then the memory server can only share the space of real memory. In this way, even if a page gets evicted from real-memory due to an insufficient share, the information is still present in virtual memory. Under this scheme the number of real pages which each client is allowed to consume is controlled according to its share. This means that a client with poor locality is unable to evict (unfairly) the pages of another client. This scenario has problems too, since a client may exhibit poor locality and consume more than its fair share of the disk bandwidth (for example), thereby slowing the progress of the other clients generating only modest page faults.

Neither alternative is obviously superior. Therefore, both approaches were implemented and evaluated. To distinguish them, the two approaches are named: Demand sharing and Rental sharing respectively.

Demand sharing

Demand sharing works by limiting the share of page faults which a client can have serviced in a particular period. For the purposes of this thesis, the period of the pay round was taken to be the period. As with the CPU server, the "demand sharing" server keeps a list of how much each process has paid in the current pay round. Also a theoretical maximum throughput of the virtual memory paging system is assumed by the server. The server uses these two values to determine the maximum number of page faults which each client can potentially use in a second.

When a page fault occurs the memory server is called in order to fulfil the request to fetch a particular page. If the client has sufficient money remaining the request is allowed and the page request sent. Otherwise, the process is suspended, until more money is paid to the server on behalf of this client. This is performed by the standing order mechanism in the same way as starvation was prevented for the CPU server.

A meaningful cost price for the resource is a little harder to produce, since initially (immediately after the pay round starts) there is no actual demand on the server, because there is no ready queue of outstanding requests, in contrast with the CPU server case. This implies that the resource is idle and must be cheap. But in the next few milliseconds, when the various threads are running, the page fault traffic may increase drastically. In this scenario the price would not give a reasonable estimate of the demand for the resource. To overcome this problem an average is taken. The price advertised, to indicate the demand and utilisation of the resource, is based on the average cost of the page faults generated in the *last* pay round. This figure is easy to calculate, since the server has the total amount paid in the last pay round and the total number of page faults in the last pay round. These can be divided to give:

$$pageFaultCost = \frac{\sum_{i \epsilon N}^{i=0} moneyPaid_i}{numberOfPageFaults}$$

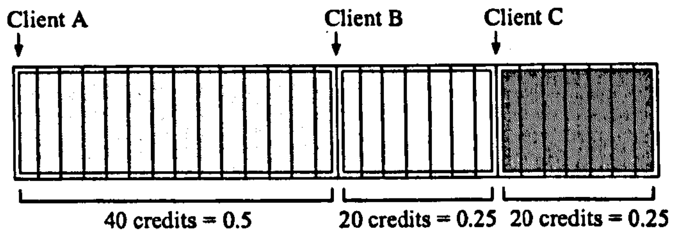


Figure 6.1: The sharing of real memory between clients paying 40, 20 and 20 respectively. (Within each section a traditional page replacement policy is used.)

Using this technique, if the number of page faults remains constant then the price reflects the average demand. Fluctuations in demand take one pay round to filter through the delay and allow the new share to be catered for.

Rental sharing

Rental sharing works in a similar fashion. Again, a list is kept of the payments made by each of the clients. Instead of using a share of page faults to determine fairness, real-memory is shared between potential clients. This means that the page allocation policy is altered to allocate pages only inside certain ranges of real memory, dependent on which client is currently active (see Figure 6.1). This means that a page fault for a process can only remove a page from the process itself, rather than unfairly evicting another process' page.

When the shares change the sections of real memory allocated for page replacement need to be changed. Two methods are possible. Firstly, the boundaries can be shifted appropriately. This may leave a resident and utilised page by another client outside its new range. Or, instead, a sparse map or list of used pages per client can be kept and can increasing a share can utilise infrequently used page from a client which is being shrunk. This method has a higher overhead in the maintainance of the share information but reduces false page faults. For the simulations described in this thesis, the simpler first method was used.

6.2 Process-centric client policies

With process-centric resource allocation, it is up to the processes to exploit their increased control over their resource allocations. The environment presented to each process is more complex than is traditional. Each process has more parameters to control how it allocate/uses its resources. But a process also has more information available to it. Firstly, it has information present in the local market. This includes information on the current price for all resources in its locale. The price information indicates the demand and utilisation of each resource in a uniform manner. Each piece of information also has its timeliness recorded with it, so that its usefulness can be assessed. Secondly, a process can obtain directly the current price of a particular resource from a server. Thirdly, a process can ask for each server it is using how much money was unused in the last pay round. This information indicates how much the process was under utilising the resource for its past expenditure. Lastly, it has its own functional requirements and potential access patterns to guide its future resource requests.

This thesis proposes that if a process can combine this information and control its resource allocation effectively then it can improve its performance. Such improvements can be viewed from two perspectives. Firstly, the throughput of the process is increased. This results from either resource allocation matching the resource demands of the process more closely, or from the process being able to exploit additional resources which it could not previously use. This can result, for example, from being able to react to dynamic changes in demand, so that the process is able to consume extra idle resources. Secondly, the timeliness of the process improves. By being able to vary the parameters of resource allocations over time it is possible to vary them to match the varying demands of the process. For example, a process can pay more for CPU when it knows it has a lot of processing to perform.

In the next section two simple example policies for making such improvements are proposed and examined. Each policy only relies on the information presented to it by the

ERA architecture and only uses the parameters available through the architecture. In this way, the resource allocation policy is implemented entirely in the user process domain.

6.3 Performing location optimisation

The marketplace presents each process with the possibility of choosing between many interchangeable instances of a resource. There are two likely motivations for choosing among resources. Either the process has some special knowledge about a resource, such as its physical location. i.e. using a printer. Or, it may hope to improve its performance by using resources which are in less demand. This reduced demand, is indicated in the marketplace through the uniform price mechanism.

The most effective mechanism for improving the performance of a process in distributed resource allocation is migration. Traditionally, migration is performed by the operating system which can use client or server initiated migration techniques to balance the load. By performing the location optimisation, for its vital resources, occurring to price, an ERA process can perform its own load-balancing: *Self Load Balancing.*

Another method is for a process to try and expand its resource consumption by including other locations. In this way, a process can react to new resources (such as a workstation becoming idle at the end of the day) and use them in addition to its current resources. In this way, a process can improve its hardware parallelism. In fact this is similar to the problem of load-balancing and utilisation optimisation (presented next), except that new instances are created/deleted on other nodes rather than migrated. Therefore, to avoid repetition this thesis will focus on the possibility of load-balancing and utilisation optimisation.

The advantage of self load balancing is that there is no requirement on the operating system to implement a policy on behalf of the processes. The initiation of a load balance of a particular process is not dependent on the operating system deciding it is best to load-balance now, but instead is dependent on the process's own needs. This benefit is only possible because the operating system is based on 'process-centric' resource allocation.

The benefits are clear. A process about to terminate shortly is unlikely to benefit from being migrated. An operating system based policy would be unable to predict this outcome and so could choose to move it at this inopportune time. This can occur surprising often, since the run-time of most processes is short and does not benefit from distributed resource allocation.

Another case might include a process heavily using a local resource. In this situation the state must be rebuilt on a new remote resource after migration. An operating controlled system may be unaware of the accesses generated or state that is kept for the process. For example: cache (disk and memory) state, or a database server's state. In these situations, it is often not worth moving the process.

The system also has the usual benefits from being able to respond to changes in demand for resources, since the environment dictates that other users may start other parallel tasks. In this situation, the cost of resources on the nodes affected by the new load goes up and, as a result, some tasks may move away to use lighter loaded nodes.

6.3.1 An algorithm for self load balancing

To perform this type of balance under the ERA architecture a process attempts to try to find the cheapest server to execute on. Since price reflects utilisation and demand, balanced prices directly means balanced utilisation and demand. When the prices only vary slightly on other nodes, they no longer appeal greatly to the process and so it doesn't wish to migrate. Therefore, each process tries to optimise its own individual performance through balancing of prices.

While this argument suits the behaviour of real-world markets, it does not suit computer systems. In a computer system, running a simple algorithm for processes to buy resources from the cheapest resource server does not result in a price balance. Instead a *herding* behaviour results, since all the processes leap to the particular node, which appears least loaded, since the algorithm expects the future state to reflect the current state. This kind of event occurs in the real world when there are large attractions. The difference is that people expect and realise that the important resource will be in high demand, since others will have seen attraction of the resource too.

The situation is made worse since a process may make its decision based on out of date information about remote resources which may have been lightly loaded, but which have now found sufficient load. A simple deterministic algorithm does not bring about movement towards an equilibrium condition. What is required is an algorithm which tends towards a long-term load balance, but may tolerate a load imbalance in the short-term.

This thesis proposes introducing non-determinism into the selection of whether to migrate and where to. By making the non-determinism in the algorithm to be dependent on a particular process' imbalance, the resulting migration decisions will tend towards a local balance, and by the 'Invisible Hand' [50] argument, to a global balance.

In order for a process to decide whether it is worth migrating anywhere else in the system, an algorithm calculates the probability of a migration being beneficial. To calculate this, the process obtains, from the market, the price of the resources available. The CPU resource is assumed here for simplicity. From this, it then determines an average price and thus any excess it may be paying compared to its current resource. If there is any excess, then the process selects to migrate based on the ratio: excessPaid:currentCost. Thus the more excess a process has the more likely it is to decide to migrate.

Once the decision to migrate has been made, there will most likely be many destinations to which the process could migrate. While the cheapest node would seem like the only choice, its future utilisation of the resource must be considered if the process is to improve its performance.

The algorithm then probabilistically determines which of the cheaper nodes to migrate to. This is achieved by calculating the excess paid over each cheaper resources. The probability of each client being chosen is: $excess_1 : excess_2 : ... : excess_n : totalExcesses$. The result is that the greater the excess that is being paid with respect to a particular client then the more likely it is to be chosen. But that if the excess is equal between clients, no matter how large, the clients have an equal probability of being migrated to.

This results in an algorithm as follows:

```
• Collect most recent prices from local market.
• Decide if it is worth migrating:
    – Calculate current average price of resources used.
    – With probability (excess paid:currently paid) migrate.
• If migrating:
    – Find location to migrate to:
        * Calculate excess compared to cheaper servers.
        * With probability 1st excess:2nd excess:  ...  :  last excess.
        * Choose destination node.
    – Migrate.
```

In situations where a perfect balance does not exist (9 equal processes on 8 nodes), then a process can choose not to migrate to any node until the reduction in price from moving to the new node is noticeably smaller than it is on the current node. In this way, a process does not see any extra advantage in the new node, since it's effect will be the same.

6.3.2 Analysis

To examine the effect of this algorithm, an experiment was designed involving eight processes running the algorithm on 4 nodes. The type of the process is unimportant, since the same migrations will be made, since the algorithm does not refer to the functional requirements of a process. A four node system was simulated repeatedly with the processes initially distributed across nodes in the ratio (1:1:3:3). The results of this experiment are presented in Figure 6.2, which shows the cost prices on each node (and thus their load) as the processes determine the location of the cheapest resource.

Initially, prices shift rapidly as local price imbalances are corrected and then quickly tend towards an average value, due to the algorithm's probabilistic nature. Since prices

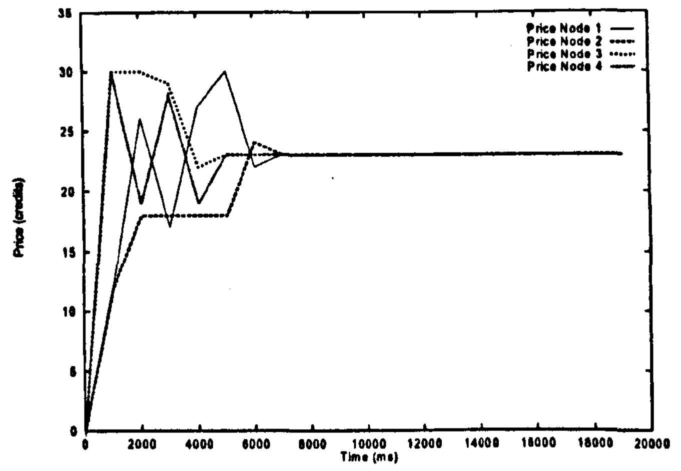


Figure 6.2: The advertised price on each node (and thus their load) sampled once a second as the processes determine a load balance from their local market information

reflect load (demand and utilisation) on each server, when the prices are balanced so is the load.

This experiment provides evidence that it is possible to perform load balancing using a relatively simple algorithm, when the decisions are taken by independent processes. The effect of shared indirect pricing information seems to be sufficient for a long-term decision to be made to improve the performance of the process.

6.3.3 Conclusion

This section described and demonstrated a technique for balancing the load of processes across a distributed system. Unlike previous techniques, the technique described here places the decision on whether or not to migrate in the user process' domain rather than in the operating system.

This ability of a process to load-balance itself is made possible by the process-centric properties of the ERA architecture. By providing each process with additional information and allowing it to decide where its resource rights are placed, a process can itself make decisions on resource location in the presence of competition for resources from others.

Using a non-deterministic algorithm to act as a predictor for future load, experiments have indicated processes using this technique can load balance themselves. This indicates that processes can mimic the behaviour of an operating system managed system. Besides this mimickery, a process should be able to choose *not* to migrate, for whatever reason. Such behaviour would be difficult, if not impossible, to provide through an operating system.

These experiments thus suggest the possibility of providing load-balancing using a process-centric approach which, in a short time, tends towards a global equilibrium under constant demand.

6.4 Performing utilisation optimisation

Optimisation of location and priority are the two most obvious forms of control by processes. They follow directly from the parameters which a process can vary. Traditional resource allocation techniques usually control location and leave the priority of processes to be set by the user. However, process-centric resource allocation can afford extra resource policies in order to gain an improvement in the performance of the process.

An example of these extended resource allocation policies can be seen in the optimisation of resource utilisation. Due to bottlenecks in other system resources, the utilisation of system resources, especially CPU, is rarely 100 percent. The most common example of this is the time a process must wait for memory page traffic when paging is necessary. When memory is accessed in a virtual memory system it happens that the data is not currently in real memory and instead resides on the backing store (disk). While the data is being fetched from disk the calling thread or process is blocked until the data is available. The probability of this increases as the size of the current working set of all processes increases. Such large working sets exist in scientific computations, multimedia processing and genetic algorithms. In these situations, a process may spend a noticeable percentage of its time blocked.

Traditionally, optimisation of this particular problem can be performed statically by the programmer (algorithmic structure, annotations and data placement) and by the compiler (loop reordering) to minimise the performance bottleneck due to page traffic. However, in a dynamic environment, the utilisation due to page faults there is over time as the competition for real memory changes. Increased competition means that less of each process' working set is maintained in memory. This in turn increases the number of page faults reducing the utilisation of the CPU.

Traditional techniques aim to avoid the problem by trying to minimise the number of page faults issued by minimising process working sets over time. A complementary alternative is to perform work on another thread in the background, while blocked on another thread. An example is the use of memory access look-aheads. In this situation, in order to amortise the cost of the page fetch, other pages are fetched at the same time. Another example is performing simultaneous processing in a multi-threaded environment, in the hope that not all threads will be blocked at the same time. This last alternative is especially appealing in unpredictable situations where large quantities of information are processed, such as real-time video or database processing. The problem with this approach is that increasing the parallelism (multi-threading) of a process may hinder, rather than improve, the performance of the process, since the extra threads may reference information which is not shared with other threads thereby increasing the number of page faults and slowing the overall throughput.

To demonstrate this phenomenon, an experiment was performed to observe the time spent waiting and thus throughput as the number of threads were increased, in a simple matrix multiply computation on a single node. A process was created which continuously multiplies two 250x250 matrices on a simulated node with 200k of real memory. In such a situation, a complete matrix multiplication requires 800k of virtual memory (3 matrices at 4 bytes per float) and so requires page traffic in order to compute. The code was compiled with GCC 2.7.0, with no special programmer annotations or loop unrolling. In order to analyse the effect of performing work while waiting for page faults, the degree of parallelism used to compute the matrix multiple was increased every 3 seconds, by spawning a new thread to compute rows in parallel.

Figure 6.3 shows the CPU price, and thus utilisation, in this experiment. The utilisation reaches a maximum at eight threads and levels out as additional threads are created (indeed

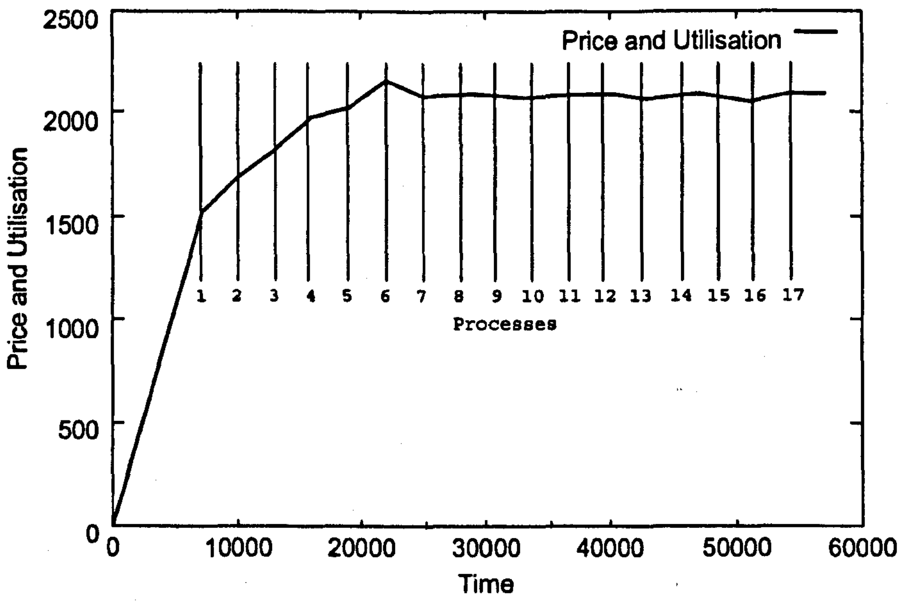


Figure 6.3: The price and thus utilisation, of a single CPU performing a matrix multiply process as the degree of parallelism is increased.

in certain circumstances the level can drop again). Two effects occur, when the number of threads does not closely match the utilisation which a process can exploit. Firstly, other additional threads cause additional context switches and page faults which slow the computation. This is especially true if additional threads cause existing useful data to be paged out. Secondly, the latency to compute a particular result can become longer as other parallel computations try to find useful work. For example, a WWW browser fetching pictures in parallel will result in many pictures being downloaded slowly rather than the necessary ones quickly. This shows that, performance is affected even in a simple example, if the resource consumption (more specifically the degree of parallelism) does not match that available.

If a process could know that it was fully utilising its share of the CPU then it would know not to increase the number of threads, and in this way the optimal level of parallelism to hide the effects of the paging latency would be found. This thesis proposes that, by using process-centric resource allocation, a process can obtain the information to do this, with a low overhead.

The marketplace contains price information about each resource. The price of a resource indicates the past active load on the resource and thus measures its utilisation, combined with the demand for it. When the price is low, either the utilisation is low or the demand is low or both and vice-versa. For example, when the price of CPU increases, this may be the result of an increase in demand for the resource while the utilisation of the CPU for each client may remain the same or decline.

A process needs to be able to obtain only the utilisation information. This is held accessibly at the server of the resource, since it keeps information on the quantity of unspent money. If money is unspent at the client by the server, then it has not fully utilised its share. Thus, unspent money at the server indicates past under-utilisation of that resource. Unfortunately accessing the information at the server may be expensive in the general case since the server may be remote. Instead a process can use the price information as an indication of changes in demand and utilisation. When it sees a change of a sufficient magnitude it can then query the server to determine whether it was a change due to utilisation.

Analysis

To demonstrate a process monitoring utilisation, an experiment was carried out in which the level of parallelism of a process was measured as the process determined whether it was under-utilising a resource. The matrix multiply process used previously was modified to actively participate in the marketplace and observe the pricing information of the CPU it is using. Since, there are no competing processes, any change in price is due to changes in parallelism by the matrix multiplication process causing changes in utilisation. The process analyses the pricing information over three seconds, and averages it to damp variations from page traffic access times. If the price it receives is higher than the previous one it knows that the utilisation has been improved and so spawns another thread, to try to improve it further. This continues until the prices no longer increase, indicating utilisation has been maximised and thereafter no more threads are spawned.

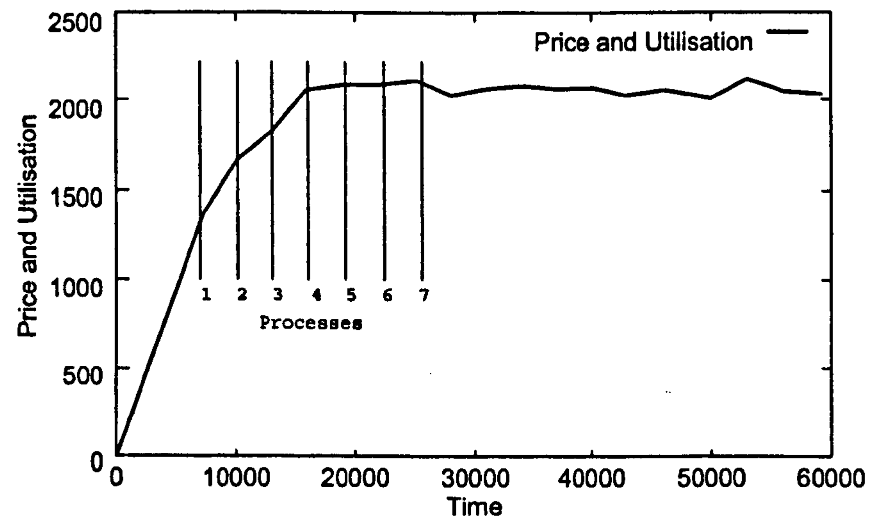


Figure 6.4: The price and thus utilisation of a single CPU as a matrix multiply process actively monitors the marketplace to maximise its parallelism.

Figure 6.4 shows the price and utilisation as before. Once the level of parallelism reached seven threads, the price no longer increased and so no new processes were spawned. Subsequently, the utilisation remained maximised along with the price. Thus, by using the marketplace a process appears to be able to maximise its performance by maximising its utilisation of the CPU by pipelining multiple page requests.

Previous techniques described in this chapter have mimicked optimisations which an operating system might attempt to perform. In this section optimisation of utilisation was demonstrated. This kind of optimisation is simply impossible for an operating system, since it has no control over the number of threads or function of each process. They are treated as 'black boxes'. By having access to information about the utilisation of a resource, a simple experiment showed a process changing its function in order to hide the latency of page fetches from disk.

6.5 Summary

ERA gives the programmer the responsibility for devising suitable implementations for serving and obtaining resources. Earlier results have shown that proportion-share techniques can be used to provide fair access to resources such as CPU and memory. We have now demonstrated two techniques to allow an application process to optimise its own resource consumption.

The first optimisation mimics the action of load-balancing, traditionally is the responsibility of operating systems. A simple non-deterministic algorithm is used to allow a process to migrate towards the best node for its execution. In a simple experimental scenario, applications could balance their load to a general equilibrium. This indicated that with process-centric resource allocation, indeed it is possible to mimic operating system controlled load-balancing. In this experiment the balance was obtained quickly and with little overhead.

Secondly, more complex utilisation optimisation is possible, whereby a process optimises its utilisation of a resource by consuming unused, but paid for, resources acquired for other purposes. The example presented in this chapter was a process performing a matrix multiplication. To optimise its utilisation, the process increased the number of parallel threads computing the multiplication, until the benefit in throughput was balanced by the increase in page traffic due to memory constraints.

These experimental results demonstrated that it is possible to provide and use process-centric resource allocation policies with a low overhead and processes can dynamically improve their own performance by choosing resource optimisations which best suit their operation.

Chapter 7

Critique

This thesis has introduced many new concepts, techniques and algorithms. Instead of fixed resource allocations performed in the process or else leaving the choice of resource to the operating system, this thesis proposed *process-centric* resource allocation. In process-centric resource allocation, algorithms determining the location, importance and selection of resources all execute in the process itself. Using information in the form of prices, a process decides how to control its allocation. This is made possible by the ERA framework and marketplace. These enable the resource allocation decisions to be made in the process itself and to provide them with useful resource information.

This chapter provides a critique of this new approach as compared with traditional techniques. Each section of the ERA architecture: framework, marketplace, process-centric policies, is critically considered in turn together with design decisions and other techniques proposed for use in the ERA architecture.

7.1 ERA framework

The placement and responsibility for resource importance (money) is one of the key concepts of the framework. The basic concept of money came from market motivations, but how this money is kept and used can affect the way in which the framework is able to provide process-centric resource allocation. It is imperative that while control of its money is given to each process, a process must not able to subvert this control and increase its expression of resource importance (money).

To support this goal, several design decisions were made. Firstly, that money is kept by the operating system and only manipulated through kernel APIs. This increases the overhead of using and manipulating money, since a kernel-user mode switch must take place, or at least a protection domain must be switched. Alternatives such as capabilities still depend on operating system primitives for manipulation. Capabilities would have the benefit of allowing money to be more readily passed around the system, but other design choices lead to minimisation of the number of monetary transfers.

Secondly, processes must not be allowed to accumulate money, since this would allow a process to express a high importance, temporarily, and thus affect the quality of service available to other processes. To impose this restraint, the concepts of pay rounds and the removal of money were introduced. These effectively limit the amount of money a process has and thus control the expression of its importance, but at the same time impose rigid constraints on the use of money.

The simplest use of wages and pay rounds is for a process to buy resources at the beginning of the pay-round and then consume them during the rest. This leads to a very bursty behaviour. When a pay-round is started typically several threads on a node are resumed. These threads make many resource requests to the resource servers at the same time. A server must then allocate its resources and respond. The client threads subsequently suspend themselves, waiting for more money at the next pay-round.

Thirdly, money must be provided up front to servers. A process does not allow a server to have access to its money, since this would allow the server to consume it all maliciously. Instead quantities of money are transferred to the server.

Moving to a system where money is not transferred would minimise the information transfer when a process sends a contract to a server, but there is a downside to this approach.

Every time a resource access is performed the consumer of the resource would have to be found, its money checked and the money updated if applicable. This would increase the overhead of using a resource. Also, there are scalability issues since these updates *must* be performed remotely in a distributed system. Monetary caching would be an obvious solution to this problem, but issues of cache coherence are then raised and these increase the complexity of the system beyond.

Rigid monetary constraints make *money* a less mobile entity than is desirable for future plans (see Section 7.5). The system is not able to respond quickly to changes in the configuration, demand or process requirements. For example, a change in demand within a pay-round is only noticed by others in the longer term should the situation persist. This is fine for medium grain tasks which are not expected to produce changes or react to changes in short periods of time. Such tasks are common in the field of multimedia, but with agent processing (see Section 7.5) this becomes a restriction which needs to be overcome.

7.2 Information dissemination

Resource allocation decisions are made by the processes themselves executing in the framework. In order that the decisions made by the process are useful, the process needs as much information as possible. Several different pieces of information are implicitly available at each server: price and unspent money. However, global access to this information does not scale well in the distributed case, so some scalable way to disseminate the information has to be found.

The ERA marketplace approach taken in this thesis, is as unconventional as the framework it uses. Rather than simply providing a fixed level of information dissemination (locality), the framework was used to enable the degree of locality to vary inversely with system load.

The result is a system, based both in the ERA framework and agent-like techniques, to pass the information around in an 'active' fashion. This achieves the desired functionality at a cost. The most obvious cost is the overhead of maintaining the agents and migrating them around the system. This is an order of magnitude of increased complexity compared with a simple nearest neighbour system. Migration of ERA processes is not very heavy-weight, but the overhead is significantly greater than the cost of sending a single message. This overhead could be reduced if a lighter-weight monetary system were used. It is an advantage that when the system is heavily loaded, very little agent activity occurs and there is then less overhead than with a fixed locality nearest neighbour system. Also, when the system is lightly loaded, information can travel further than with a fixed approach thereby increasing the information available to clients.

Besides presenting price information, the marketplace also presents timeliness information. This is important, since prices which were generated some time ago may have taken some time to reach the local market and this information is less useful than new information from servers. But the timeliness information also presents some problems. For example it increases the complexity of the client's code to process price information. A simple way to handle this complexity is to ignore information older than a certain time limit (for example a few pay rounds). This has a similar effect to using a nearest-neighbour approach which receives information conditionally from a fixed locality, although the benefits of dynamic locality would still remain. While simplifying, this approach negates some of the effect of the dynamic locality, since the timeliness cut-off may lose useful information which has travelled from further afield.

The monetary system can affect the ability of the marketplace to respond to small quick updates, rather than wide area updates. This is because the use of wage updates has a tendency to link the use of the market to the periods in which a server's wage is updated. It should be noted that the described architecture does not mandate this, it is just simpler and adequate to use this approach here. It does lead to periodic activity across the marketplace.

A server can remove some of the periodicity by spreading its advertising money usage over time. It is difficult for a server to determine how much to spent on advertising at any particular moment, since it will wish to keep money to spend later in the pay round too. This difficulty stems from two things. Firstly, it is not possible currently for a server to know how effectively its money is being spent, since all the money given for advertising is distributed when agents are created. Therefore, it is not currently possible for a server to adjust its expenditure as client processes do with other resources.

Secondly, it is not possible to spend money on advertising and have it used as and when necessary for the latest advertising. Consider a server advertising its price 5 times a second (1 pay round). In a simple scenario it could spend all its money with the market, since it only needs to advertise. Then, every time a new price was sent to the market, the remaining funds would be used to distribute it. Intuitively it can be seen that a process advertising 5 times as often as another, but spending the same money, might distribute its information less far but more often. Such an approach is prevented by the current system for two reasons. Firstly, money must be paid up front for all resources. Secondly, the market has no idea whether its client (the resource server) wishes to advertise information frequently or over a wide area.

The use of up-front payment highlights another feature of the marketplace. Money is paid up-front to each agent at each market. The remaining money of the incoming agent is divided equally among the agents carrying the information further. This is making the assumption that each agent needs to perform the same quantity of work. In reality, at least one of the agents will find itself completing a cycle. At which point it dies taking all its unspent money with it (out of the marketplace). In this way, a large sum of money may be spent with the server, of which a quantity (depending on topology) may be put to poor use.

A system could be envisaged where money is not paid up-front, but instead used from a *source*. In this situation all the agents would be able to use the same source. An agent dying, due to the discovery of a cycle would, instead of losing its money, reduce the demand on the source, thereby enabling others to travel further. This arrangement could also help solve the problem of advertising frequency and expenditure described previously. If a client had control of a 'source' it could choose to revoke the source and continue with any remaining money on a new set of adverts. Alternatively, where a process wanted information to travel as far as possible, it would never revoke the source.

This approach seems to solve several problems, but it would come at a cost. *Sourced* money supplies do not scale well across a distributed system, since they lend themselves to centralised control. An agent process consuming resources would require frequent access back to the source. The act of revoking a source would either require distributed revocation or a centralised solution for switching of monetary rights. Even in a centralised solution it would never be possible to know when a money 'source' could be removed and garbage collected, without a costly distributed garbage collection. This approach requires much further work before it can be claimed to be feasible.

An orthogonal issue is the impossibility for an advertiser to specify the speed with which the information should be advertised. A speed parameter might cause an agent to purchase quick memory and CPU to improve the speed of advertising rather than distribute itself as far and wide as possible.

The area where the ERA marketplace is really useful is where either the work to be performed at each node is complex and/or bandwidth is scarce. Consider a system where information in the form of databases is distributed over a wide-area. The size of the system makes it infeasible, in peak periods, for all the databases to be queried. Besides, the processing at each node can be sizeable and thus require large resource consumption. In such a scenario, the marketplace and in particular the Dying Sandwich Board Men approach, is very appealing. Computation agents can be created with the DSBM approach to perform a database query. More or less databases are queried depending on the importance of the user and the load of the system (network, CPU, memory). This example is not so remote, the Internet is such a system.

The marketplace itself is a good example of a process-centric system, since the agents which form the marketplace use the framework and make useful resource decisions with it. The resource consumption protection provided by the framework cannot be easily provided by a traditional operating system. The marketplace is a powerful new technique for dissemination, although still in the experimental stage.

7.2.1 Algorithmic considerations

The algorithm used in the marketplace relies on dynamic locality in order to maximise the information available for decision making. One of the benefits of this approach is the potential to speed up the time within which load balance occurs. When a point load exists (compared to the rest of the system) information travels over the network up to the point load. Near the point load information travels less far, since resources are more expensive. In general information travels further over lightly loaded regions than heavily loaded ones.

From the viewpoint of the point load, this result seems unhelpful, since as the information approaches the point load, it is deterred from travelling in that direction any further, since the costs are high, and so less likely to reach the centre of the point load. This means there is less diversity of information at the point load than in the lightly loaded region. This seems counter to the goal of removing the point load quickly.

However the viewpoint from the general load is different. Information travels from further afield while it is viable to do so. Near a point node, the information penetrates the heavily loaded nodes from the outside. This enables the outer nodes to dissipate the load to the further out lighter loaded nodes. Work on the central more heavily loaded nodes can then be dissipated to these now relieved nodes. In this way, the information gradually reaches the centre of the problem and enables the point load to be dissipated from the outside, rather than centrally. This has the benefit of preventing a load being dissipated towards other heavily loaded regions, since information from those regions is even less likely to reach that point load from the heavily loaded direction.

7.3 Process-centric resource allocation

In Chapter 6, two examples of process-centric techniques for optimisation of resource allocation were given. These techniques fall into two categories: those which mimic resource allocation procedures used in traditional operating system resource allocation; and those which perform novel resource allocation optimisations.

The load-balancing example is in the first category. Load-balancing has long been a reason for the use of operating system based resource allocation techniques. In simple scenarios the process-centric technique gives a good load-balance which can respond to changes in demand. The key motivation for process-centric resource allocation it that a process does what it wants with its resources. In an application-heterogeneous case, it may be that a particular resource allocation policy used by a set of clients results in a worse load-balance. This is not a flaw in the architecture, simply a flaw or preference in a client's code. Some processes may choose not to migrate, or all may migrate to a single node. By doing this they affect their own execution. Other processes with more reasonable policies will simply not use such nodes and will balance their load across the other nodes.

The second example illustrates a radically new resource allocation policy. A process is demonstrated modifying the work it performs in order to take advantage of extra available resources. In this example, the resources are those wasted due to page faults. The obvious criticism of these experiments are that the scenario is simple. The reason is the complexity of the task being performed. It is not that the client code is especially complex, but the algorithms are new and require understanding in depth. Unfortunately, time constraints imposed on the duration of Ph.D research restricts what can be undertaken and so further investigation of the policies had to be curtailed.

The policies and experiments presented in Chapter 6 are evidence of the behaviour of process-centric resource allocation. Techniques traditionally reserved for operating system based resource allocation can be mimicked by client-side process-centric resource allocation policies. And, fairly simple policies can provide improved resource allocation compared with current operating systems of any kind.

7.4 Modelling

It is clear that a system like the one described in this thesis is complex, from both from operating system and economic viewpoints. Construction of models – simplified versions – is the standard scientific method for studying complex systems. While there has been pioneering work to establish a formal basis for operation systems, current operating systems however have very pragmatic designs. By contrast in economics especially including econometrics modelling is commonplace. In fact, one might say that economics is the science of monetary modelling.

In the science of economics, there are several means and levels of description for economic systems. Economic models aim to describe economic systems using mathematical formalisiums to model many things from the flow of money over time, to supply and demand. These models of real world economic systems have taken many years to develop. The economics of the ERA system are artificial and fabricated for a particular purpose. As a result they do not conform to existing real world economic models. For example, in ERA money is revoked at the end of a pay-round.

Considering artificial economics and operating systems is a modelling challenge. Modelling techniques exist in economics and models for programs and processes exist in computer science, but no modelling technique reflects both. For example, a model of the ERA system could be developed in Z, UML, state diagrams or one of many other computer modelling languages, but this would not reflect the dynamics of money in the system.

During the work on this thesis the possibility of modelling was investigated in a preliminary way with the help of a member of staff versed in modelling techniques. While it was clear that no one modelling technique could encapsulate the system, it is possible to examine the system from a number of perspectives, for example: wage money flow, pricing and supply/demand; and traditional process interaction for deadlocking and feature interaction. It was clear that it is a very interesting problem, however, it is necessarily outside the scope of an already large PhD.

7.5 Future work

This thesis presents several novel techniques and algorithms, which are used in combination to form a radical approach to allocating resources across a distributed system. This inevitably promotes more questions than answers This section discusses some of the work which might follow on from questions raised.

The real advantage of process-centric resource allocation is the number of new resource allocation techniques which can be provided using this basis. The example policies described in this thesis represent the tip of the iceberg for this form of resource allocation.

Much of the work in this thesis could be studied in far greater detail if time allowed. Only simple algorithms have been presented here (although these are frequently the most effective). There is obvious scope for close examination of other types of resources, such a networks and disks and how best to proportionately share them. From a modelling point of view, there is much work to be done on the derivation of prices for resources. In the current implementation, prices are determined by utilisation and demand formulas. Simple formulas such as these may be the best, there is however, scope for evaluating different pricing mechanisms and how they affect the quality of the resource decisions made by processes.

Each of the process-centric policies could be studied in far greater detail. In the load balancing example there is scope for a lot more research. The speed and accuracy of the load balance will be affected by the quantity of information at each node, the timeliness of that information and the route by which information reached the node. Also it would be interesting to investigate how large loads get distributed in point load situations.

Similarly, the process-centric utilisation case which was demonstrated for a single node has many other possibilities. For example, the same utilisation optimisation could be provided for the network resources and this could allow a process to utilise a reserved bandwidth channel fully. A WWW browser could optimise the number of simultaneous connections used, dependent on both the user's importance and the utilisation of the network link to *that particular* site. Also, the system could be studied for multiple processes and new processes entering and exiting the system, while utilisation is being optimised.

Many other process-centric scenarios can be envisaged and implemented under the ERA framework. For example, processes can optimise the balance of resources they consume in order to maximise their performance in return for the money spent by the process; or they could perform other forms of prefetching in order to maximise their throughput, or parallel applications could load balance themselves, such that the processes are migrated together; or perhaps processes can make use of extra or idle distributed resources that become available dynamically. The options seam endless and it is a pity there is not enough time to investigate them further.

Currently the use of monetary revocation is a central principal of the market. It was introduced to prevent large accumulations of money, either accidentally or deliberately. By preventing large accumulations of money it effectively stops any process having more of an effect on the system, in a pay round, than its current wage dictates. This gives good control over the proportion-shares obtained and thus allows the system to be used with general-purpose programming (where programs may not always be correct). A future research direction could consider the system as described, but with this feature relaxed. This would affect several other features of the architecture. For example, loans could be removed, since processes could save. At the same time, the possible effect a process could assert would be less limited. However, since monetary values are simply relative it would not be possible (theoretically) for a process to perform a complete denial of service (the ultimate effect of a process spending a large amount of money), since other processes with receive a share no matter how small. Is this important for general purpose systems? Can the constraints be relaxed to allow a fixed maximum total of accumulated money? Would it then be possible for a process to hide money to bypass this limit? How is this affected by the type of monetary system used (capabilities for money, kernel access to money, etc.)?

7.5.1 Agent ERA

One of the principle motivations for the resource allocation work of this thesis was application to the area of *distributed agents.* This field is large and poorly defined, so a definition of distributed agents is probably appropriate. A distributed agent is a lightweight task (execution) which performs work for its creator across a distributed system. The principle use of agents is to allow the processing of information across a network and on the server rather than the client. Thus, agents are tasks which are created at the client and which migrate across the network to perform remote processing, returning the result to their creator. In this way, a large amount of information can be processed remotely and then a smaller amount (the result) is returned. This field has many similarities with traditional distributed systems, where similar client/server relationships exist. The principal difference is that with agents the server software does not do the processing on the client's behalf. Instead, a client program uses the server by migrating to the server to perform processing. The server is no longer active, per se, instead it acts as a resource for an active query. This is similar to a distributed process using the functionality of a remote server, by creating a thread which migrates and executes on the remote node.

There are been several, mostly unsuccessful, commercial attempts to provide this desirable form of processing. Notable examples include General Magic's Telescript language [62], Sun's SunScript and server-side Java [22] and Javascript [23]. Of these, Telescript is the earliest work and, while technically superior to the others, it failed to take hold in real systems. SunScript and server-side Java are still viable options although they have yet to be exploited to any real extent. In all cases, however, the use of server-side processing is limited and the consumption of resources by aggressive processes is not considered.

In such a system, tasks (threads of execution) must execute on a remote and potentially foreign host (not created as part of or for the particular client). This type of application is not a problem in purpose built systems, since the interfaces and resource consumption of both the client and server are known. But in a general-purpose system this information is not known and the server interface must accept tasks which will execute on the remote environment. Currently it has no control over the amount of resources consumed by this client, or over the number of clients. Consider, for example, server-side Java programs. These programs execute as a result of the client (the browser) on the server (HTTP server). A standard Java environment is provided in which potentially any program could execute. When a Java program is downloaded into the server environment the load on the server is affected. Since the number of programs and their resource consumption is not known, the performance of the server can be severely affected. This is especially true in a denial-of-service attack, which reduces the ability of the server to provide the WWW pages, which is its main purpose.

Clearly, for such a paradigm to become widely accepted, the execution of such downloaded tasks must be controlled so that their affect on the server can be defined by the server. The agents know their purpose, execution, and resource consumption (where to execute, for example). The ERA architecture is designed to separate the act of resource allocation from the control of allocation of resources so in principle it is appropriate. However, the ERA architecture so far lacks the right functionality and emphasis to work well in this field. But firstly, as mentioned in Section 7.1, the monetary structure used by the ERA architecture could be made more appropriate to lightweight agents which migrate quickly over the network, perform some remote task and deliver the result back quickly again. The act of migration is a comparatively high overhead task. Secondly, as discussed in Section 7.2, the definition of the money system makes it difficult to allocate money in small, short-lived amounts.

The ERA architecture introduced in this thesis can, however, meet the needs of such a system effectively and efficiently. Processes can be created which have the sole control of their own resources and themselves choose whether to migrate and how they wish to execute. This allows agents to be created which can adapt their resource consumption according to the loading of the system. For example, under certain loading constraints it is possible that remote execution would be preferable to local execution, since the network and remote execution are under utilised. At the same time, the use of a monetary abstraction allows the operating system (or user-mode operating system independent servers) to be in control over how much resources agents consume over time.

A version of ERA for such agent-based applications has been considered and called *Agent ERA*. Initial ideas for this system are discussed here. The basis for the approach is a move away from kernel controlled money to a monetary system based on capabilities. These monetary capabilities, called *tokens*, represent the ability of a process to give another process access to part of its resource importance. The ability to use the money can be revoked or created readily by the process, through the use of the operating system.

The operating system acts as a creator and validator for these tokens. These tokens exist in user-space thereby allowing processes to send them as with any other data (just like capabilities). Each token represents the right to use a proportion (percentage) of a 'source' of money. Each agent has at least one money-source associated with its execution from which it makes tokens. Agents may share the same sources in order to share expression of resource important. These tokens are then given to resource servers to allow access to a proportion of that source. A server can then use this money by requesting the operating system to make more smaller tokens from one of these passed tokens. This is like a bank converting a large denomination bank-note into smaller denominations. No money is created by the act, simply a different set of "handles" represent the same purchasing ability.

A source of money is a fixed quantity which is not consumed or depleted as wages are in the current ERA architecture. So there is no need to revoke or update tokens and sources at particular intervals. Sources can be created from tokens or as a proportion of other sources.

All resources are allocated, as with proportion-share resource allocation, according to the relative sizes of the tokens which a server has received from its clients. This allocation is performed by the servers as part of the execution environment for the agents and ensures that the operating system controls the quantity of resources which each process can obtain, since tokens are validated and examined in the execution environment. This means that even if a token is presented frequently, the probability of receiving a share is a result of the size of the token rather than how often it is presented.

The key to making this approach work is ensuring that the money-sources can be accessed efficiently even in the distributed case. The simplest technique is a centralised money-source, which is split down and given to the distributed agents. Then, whenever a token is used, the central resource is consulted (this must happen because tokens are relative to their source). The technique is quick for small systems, but scales poorly to large systems. Instead, it is proposed that a money cache be based on a source. Whenever a token is used the base source is obtained. The base source is either fetched from the location which created the source, or fetched from the current holder of the source (by indirection). The cache works in a similar way to a traditional cache. Client may snoop tokens out of other nodes' caches and each client must have exclusive access, to convert the token into a real value, although relative division can proceed without such exclusive access, since the value is not important.

The technique should scale better than a centralised one. But like similar caching systems (such as distributed shared memory - DSM), it might not scale to really large systems. Here, however, it is proposed that a solution exists. It is proposed that sources may be made from other sources, this means that tokens need not be resolved to a root source, but instead to its parent source. This can be considered like a hierarchical directory structure being more efficient to lookup pathnames than a flat one. The creation of a source requires exclusive access to the parent source, but from then on may be used in its own right potentially improving scalability.

This separation of the source of a money-source from its use does, however, present new problems for the revocation of access rights. Tokens can be revoked easily from a particular resource, since the use of a token relies on the source itself. Deleting a source, however, requires tracing of all the sources created from the source being modified. To maintain this state, the kernel could keep a list of the sources which it authorised (not those based on sources it modified). When a source is changed it follows this source tree to allow the effect to be propagated to other tokens derived from the source.

This technique should allow the same features of the ERA architecture to be supported, but provides fine grain usage of money along with lightweight migration of agents. Scalability would be slightly compromised, by comparison with the 'shared-nothing' approach of the ERA architecture described in this thesis.

7.6 Summary

The ERA framework provides the support necessary for processes to be control their own resource allocation in respect of its location, priority, and utilisation. The framework is not perfect, however, since control of the monetary system by the process can be slightly coarse grained. Small computations using the framework can really only benefit from simple allocations. Further work in this area, towards a lighter weight approach, has been proposed in this chapter. This approach, called 'Agent ERA', moves towards a money-source approach where the right to access resources is sent around the system as 'tokens'.

The marketplace disseminates information to give processes as much information as possible in order to improve their decision making. It does so in a novel and scalable fashion. The provision of information in a way that is inverse to system load, means that information travels up to, rather than into, heavily loaded regions. The consequences for the effectiveness of the decisions made by processes should be investigated further.

Giving information and control to clients creates a need for client-side resource allocation policies. This in turn introduces the possibility of many different resource allocation techniques on a process basis and two such techniques have been demonstrated. Each shows, at an experimental level, the additional power of the process-centric approach. Each investigation spawns further interesting questions for research.

Chapter 8

Conclusion

Distributed resource allocation presents many complex and challenging problems. Each task needs multiple resources for which demand will vary over time. Each resource will receive resource requests from multiple tasks, each with different access and usage requirements. The complexity of these resource needs makes the derivation of a perfect solution infeasible. The problem of resource allocation becomes even more challenging when extended to the distributed case. Besides the requirement to allocate resources in a fair and deadlock free fashion, allocation algorithms must take account of the availability of a choice of resources across the system. As a result, computer scientists have proposed many techniques for performing resource allocation which result in a feasible solutions for particular situations.

These resource allocation algorithms fall into two categories: those where the client requests the use of a resource, which is allocated by the operating system (*operating system controlled*); and those where the processes requests particular resources themselves (*process controlled*). Operating system controlled resource allocation provides a means for coordinating the choice of resources in a distributed system. The operating system has access to all the information about which processes are requesting what resources. This makes operating system controlled resource allocation appropriate for systems where load-balancing is important. By contrast, process controlled resource allocation gives the choice of resources to the process. A process can request which particular resources it wants. This makes process-controlled systems very suitable for fixed, high performance systems, since the precise resources can be chosen and little operating system overhead is needed.

An important new application field in distributed/networked computing is networked and multimedia-style tasks. In these systems local area networks of workstations and servers provide resources to one another. Each workstation runs networked multimedia style applications with high resource consumption. Typically, each node does not solely perform computationally or I/O intensive tasks (as is traditionally the role of servers) or small tasks and graphically intensive tasks (as is traditionally the role of workstations). Instead, each workstation is required to perform a combination of computation, I/O, small tasks, real-time and graphical tasks for a variety of users.

It is proposed in this thesis that process and operating system controlled resource allocation have their place but in such networked workstation systems, an alternative resource allocation approach which combines the best features of each is required. Process controlled resource allocation cannot respond to varying resource usage, when competing for resources with other processes, since processes are unaware of this competition. Operating system controlled systems take account of this competition in multi-tasking systems, but cannot consider the needs of individual processes because processes are treated uniformly.

This thesis has set out to try to answer the question: *Is it possible and useful to empower a process with more control over its resource allocation while preserving fairness for general-purpose distributed systems?* In doing so, a complete architecture for resource allocation using a process-centric approach has been proposed and analysed.

The basic premise of this thesis is that the two traditional approaches to control can be combined so that a process has control over its own resource allocation, while the operating system maintains control over fairness between competing processes. This allows a process to select resources which are the most appropriate for its application, since it is has this knowledge. This provides a process with the ability to choose how important a resource is and which (of a choice of resources) to use (for reasons of locality). The operating system controls the quantity of resources that each process can consume in relation to the resource requests from other processes to maintain fairness, which is vital when common resources are shared.

8.1 Process resource choice

The provision of process-centric resource choice stems from the the provision of resource allocation based on a monetary system where money represents the ability to consume resources by a process. The ERA framework gives control of money to the process rather than embed the monetary system into an operating system controlled resource allocation technique. By using several techniques (such as loans, standing orders and pay round money revocation), a process can buy the resources it needs, even those vital resources, such as CPU and memory, without which it could not run.

This strategy gives the process much more flexibility and control, than is usual. Not only does it have several more variables for resource allocation, such as priority and utilisation, than with a traditional process-controlled technique, but money also provides a uniform means for presenting information about the demand for resources. This enables a process not only to describe its resource needs in more detail, but to do so in an informed fashion. A process can choose not to use a resource because it is heavily loaded, rather than blindly choosing.

Results from Chapter 4 provide evidence that its *possible* to provide this functionality using money and resource decision making within the process itself. Processes in this chapter run a variety of simple tasks, in they which are able to survive, execute fairly and proceed without starvation or deadlock.

8.2 Operating system control

This degree of resource choice by processes brings the need for control. The framework relies on two techniques to provide this control. Firstly, provision of control over the money which each process has where techniques such as the use of revocation of money are used to control the money a process has, to ensure that it cannot subvert the resource importance which it can express. Secondly, the use of proportion-share techniques for resource allocation. Proportion-share resource allocation techniques were used in this work at the same time as contemporary proportion-share work was developed elsewhere. These techniques use a relative resource requirement approach to sharing resources rather than the absolute requirements in approaches such as real-time. This allows the effect which each process can impose on the system to be relative to competition from others. So that it is *impossible* for a process to consume all the resource bandwidth of a particular resource, if it is being used by another process at the same time.

In Chapter 4, these techniques were demonstrated, in two simple scenarios, preventing computationally intensive or memory intensive processes obtaining more of the resources in the system than their resource importance (money) dictated. Monetary revocation prevented saving money from previous pay rounds to provide additional importance in future pay rounds. The use of proportion-share techniques in the process-centric based resource servers prevented any particular processes from using the resources more than their relative importance dictated and so gave controllable long-term fair usage of resources.

These techniques indicate that it is *possible* to provide effective control over a process which has been empowered with the ability to select its own resources. The degree of control which can be afforded prevents a process from subverting the amount of money that it has and also allows each process to express the importance to be used.

8.3 Empowerment

Processes have to be empowered to use this control over their resource allocation. This means a process must be able to make resource decisions which are meaningful, such as choosing to use a different CPU resource. The operating system must respond, to this change in resource allocation decision on the part of the process, by transferring the necessary state from one node to another. Also, so that its decisions are not solely based on function, but on demand and utilisation information too, the process must be able to obtain the information it needs to make resource allocation decisions.

The transfer of state is not a complex problem. Servers can transfer state between similar instances (the only case considered here). With some carefully managed interaction between CPU and memory servers on different nodes, the state kept by each server can be maintained. In addition, the use of logical clocks per pay round prevents processes being starved, or paid twice per pay round, when vital resources are changed.

The provision of information is a more complex issue. Information is provided by the framework, for each resource – such as price and remaining money. This allows a process to derive the demand and utilisation per resource. Importantly, this information covers both overall information about the resource and information about the resource from the process' perspective. Demand represents the overall competition for the resource and thus acts as an indication of the size of the share that the process will receive, as a result of proportion-share allocation. But resource utilisation information only refers to the utilisation of the resource by the process itself and thus indicates the usage of its potential share. This difference gives processes information, which is not available with techniques such as the UNIX load average and allows a process to make more informed decisions, such as performing 'self load-balancing' based on resource demand. In addition, a process can also consider its own resource usage and expenditure together to implement novel resource allocation policies based on its *own* resource allocation needs, rather than the needs of a general load balance.

The use of the marketplace extends into a scalable information resource the ability for processes to have information. The marketplace provides dissemination of information with dynamic locality, so that the provision of information to resources does not interfere with the running of application processes. As a result, the information resource can provide better than a basic nearest neighbour approach to information dissemination (which would result either in a fixed overhead and/or a fixed locality of information). Experimental results in Chapter 5 demonstrated this ability to provide dynamic locality of information by using the framework as a user-domain process. The result indicate that the marketplace appears to maintain linear scalability even with the additional functionality provided by the algorithm and the underlying framework. These techniques considered together indicate it can empower a process with the information it requires to make effective resource decisions in a distributed environment.

8.4 Usefulness

One final question remains to be asked *"Is process-centric resource allocation useful?"*. A process may be able to control its resource allocation through more parameters than previously. The operating system may still be able to control the quantity of resources allocated under competition. A process may be able to obtain information on which to base resource allocation decisions. But can they in combination improve the performance of processes?

The marketplace provides the evidence that process-centric resource allocation is *useful*. Without process-centric resource allocation it would be much more difficult to provide resource information in which the locality varies dependent on system load.

In Chapter 6, two example process-centric resource allocation policies were considered. Firstly, optimisation of the location of resources provided experimental evidence that is possible for processes to perform their own load-balancing based solely on resource prices collected from the marketplace. These experiments showed that the information available to a process is sufficient to mimic resource allocation policies usually imposed by operating systems. The algorithm used is simple and has low overhead but, under constant conditions, quickly determines a load balance. also, since the decision to load-balance is based in the process, a process can to decide not to load-balance and instead consume resources based on other criteria. As a result, processes run on the most lightly loaded node or on the most appropriate node, improving their performance.

Secondly, a more ambitious resource allocation policy was proposed and demonstrated. A process was shown reacting to its surroundings and determining a suitable level of resource usage as a result of the dynamic availability of resources such as typically occur in general purpose systems. Specifically, a process was demonstrated optimising the level of parallelism in a matrix multiplication so as to make use of spare memory bandwidth on a single node. This experiment attempts to demonstrate the key advantage of process-centric resource allocation: the ability of a process to react in accordance to its surroundings and its own resource requirements.

8.5 Concluding remarks

This thesis has proposed a novel architecture for distributed resource allocation, called ERA. It aims to provide uniform access to resources and fair resource allocation in general purpose distributed systems. ERA is based on a new model for resource allocation, called process-centric resource allocation. This model aims to give processes control over their resources while allowing the operating system to maintain fairness. Existing architectures do not provide this combination. Ultimately this thesis tries to say whether it is "possible and useful" to provide this kind of control for applications.

This thesis has presented strong evidence that such an architecture is both possible and useful. A set of experiments has provided evidence that the architecture can provide fair access to resources in a uniform manner with low overhead and good scalability. Other experiments have provided evidence that process-centric allocation can be a scalable and flexible means for implementing a distributed marketplace of resources. Together these sets of experiments provide strong evidence that such an architecture is feasible for real applications

Further experiments have indicated that the architecture can be useful. Provision of a distributed marketplace which responds to the load a distributed system, shows the utility of the architecture. Using this marketplace, applications can be easily programmed across a distributed system to avoid resources in high demand and discover alternative resources.

Besides this, load balancing experiments demonstrated that the functionality of existing operating system controlled resource allocation can be emulated. Also, a process seems to be able to decide whether to load balance itself. Utilisation optimisation extends this by demonstrating use of utilisation and price information by the process to allow it to perform some parallel computation (essentially prefetching) so as to improve its performance.

These results taken together are strong evidence that it is indeed possible and useful to give processes control over their choice of resources in a resource aware environment.

References

[1] Ives Aerts, Ludo Cuypers, Alan Messer, Philip Rademakers, Kristof Vanbecelaere, and Erik Wybouw. The MOO scheduling algorithm. In *Sony Research Forum, Tokyo, Japan*, 1997.

[2] M.J. Bach. *The Design of the UNIX Operating System.* Prentice Hall, Inc, 1986.

[3] B. Bashrim. *4.4 BSD System Manager's Manual (SMM).* O'Reilly, 1994.

[4] A. Beguelin, J. Dongarra, A.Geist, R.Manchek, and V.Sunderam. A user's guide to PVM: Parallel virtual machine. Technical report, Oak Ridge National Laboratory, 1991.

[5] P. Bell and K. Jabbour. Review of point-to-point network routing algorithms. *IEEE Communications Magazine*, 24, 1:34–38, 1986.

[6] N. R. Bogan. Economic allocation of computation time with computational markets. Master's thesis, Department of Electrical Engineering and Computer Science, Massachusetts Institute of Technology, 1994.

[7] Raymond M. Bryant and Raphael A. Finkel. A stable distributed scheduling algorithm. In *Proc. 2nd Intl. Conf. on Distributed Computing*, pages 314–323. IEEE, 1981.

[8] Peter Burger and Duncan Gillies. *Interactive Computer Graphics.* Addison-Wesley, 1989.

[9] M. Chacholiades. *Microeconomics.* Collier Macmillan, 1986.

[10] Jeffrey S. Chase, Valérie Issarny, and Henry M. Levy. Distribution in a single address space operating system. *ACM Operating Systems Review*, 27(2):61–65, April 1993.

[11] H. H. Chen and R. K. Joseph. Exchange interaction model of ferromagnetism. *Journal of Mathematical Physics*, 13(5):725–739, May 1972.

[12] John Q. Cheng and Michael P. Wellman. The WALRAS algorithm: A convergent distributed implementation of general equilibrium outcomes. Technical report, University of Michigan, 1994.

[13] David R. Cheriton and Kieran Harty. A market approach to operating system memory allocation. Technical report, Computer Science Department, Stanford University, 1993.

[14] E. S. H. Cheung and A. G. Constantinides. Fast nearest neighbour search algorithms for self-organising map and vector quantisation. In A. Singh, editor, *Conference Record of The Twenty-Seventh Asilomar Conference on Signals, Systems and Computers (Cat. No.93CH3312-6)*, volume 2, pages 946–50, Los Alamitos, CA, USA, 1993. IEEE Comput. Soc. Press.

[15] D. Clarke and B. Tangey. Microeconomic theory applied to distributed systems. Technical report, Department of Computer Science, Trinity College Dublin., 1994.

[16] A. I. Concepcion and W. M. Eleazar. A testbed for comparative studies of adaptive load balancing algorithms. In B. Unger and D. Jefferson, editors, *Distributed Simulation 1988, Proc. SCS Multiconference on Distributed Simulation*, pages 131–135. SCS International, February 1988.

[17] David E. Culler, Andrea Arpaci-Dusseau, Remzi Arpaci-Dusseau, Brent Chun, Steven Lumetta, Alan Mainwaring, Richard Martin, Chad Yoshikawa, and Frederick Wong. Parallel computing on the Berkeley NOW. In *9th Joint Symposium on Parallel Processing*, 1997.

[18] H. Custer. *Inside Windows-NT.* Microsoft Press, 1992.

[19] H. M. Deitel. *Operating Systems.* Addison-Wesley, 1990.

[20] Digital. *Vax/VMS System Software Handbook.* Bedford, 1985.

[21] Donald Ferguson, Yechiam Yemini, and Christos Nikolaou. Microeconomic algorithms for load balancing in distributed computer systems. In *8th International Conference on Distributed Computing Systems*, pages 491–499, June 1988.

[22] David Flanagan. *Java in a Nutshell: A Desktop Quick Reference for Java Programmers.* Nutshell handbook. O'Reilly & Associates, Inc., 103a Morris Street, Sebastopol, CA 95472, USA, Tel: +1 707 829 0515, and 90 Sherman Street, Cambridge, MA 02140, USA, Tel: +1 617 354 5800, February 1996.

[23] David Flanagan. *JavaScript: The Definitive Guide.* O'Reilly & Associates, Inc., 981 Chestnut Street, Newton, MA 02164, USA, second edition, February 1997.

[24] G.A.Geist and V.S.Sunderam. Experiences with network based concurrent computing on the PVM system. Technical report, Oak Ridge National Laboratory, 1991.

[25] Bill Gropp and Ewing Lusk. A test implementation of the MPI draft message-passing standard. Technical report, Mathematics and Computer Science Division, Argonne National Laboratory, 1992.

[26] P. Gupka and R. Gopinath. A hierarchical approach to load balancing in distributed systems. In *Proceedings of the 5th Distributed Memory Computing Conference*, pages 1000–1005, 1990.

[27] A. Pothen I. Stoica, H. Abdel-Wahab. A microeconomic scheduler for parallel computers. Technical report, Department of Computer Science, Old Dominion University, 1994.

[28] Mourad Kara. A global plan policy for coherent cooperation in distributed dynamic load balancing algorithms. Technical report, University of Leeds, 1994.

[29] T. Kunz. The influence of different workload descriptions on a heuristic load balancing scheme. *IEEE Transactions on Software Engineering*, 17(7):725–730, July 1991.

[30] J. F. Kurose, M. Schwartz, and Y. Yemini. A microeconomic approach to decentralised optimisation of channel access policies in multicast networks. In *IEEE*, pages 70–77, 1985.

[31] J. F. Kurose and R. Simha. A microeconomic approach to optimal resource allocation in distributed computer systems. In *IEEE Transactions on Computers*, volume 38, pages 705–716. IEEE, 1989.

[32] S. Leffler, M. McKusick, M. Karels, and J. Quarterman. *The Design and Implementation of the 4.3BSD UNIX Operating System.* Addison-Wesley, 1989.

[33] Ian Leslie, Derek McAuley, Richard Black, Timothy Roscoe, Paul Barham, David Evers, Robin Fairbairns, and Eoin Hyden. The design and implementation of an operating system to support distributed multimedia applications. Technical report, University of Cambridge, 1997.

[34] M. J. Litzkow, Miron Livny, and Matt W. Mutka. Condor - a hunter of idle workstations. In *International Conference on Distributed Computing Systems*, pages 104–111, June 1988.

[35] Umesh Maheshwari. Charge-based proportional scheduling. Technical report, Laboratory for Computer Science, M.I.T., 1995.

[36] E. P. Markatos and T. J. LeBlanc. Load-balancing vs. locality management in shared-memory multiprocessors. Technical report, Computer Science Department, University of Rochester, 1994.

[37] P. L. Mills. The systolic pixel: A visible surface algorithm for VLSI. *Computer Graphics Forum*, 3(1):47–59, March 1984.

[38] R. Mirchandaney and J. A. Stankovic. Using stochastic learning automata for job scheduling in distributed processing systems. *Journal of Parallel and Distributed Computing*, 3(4):527–552, December 1986.

[39] A. Mohindra and U. Ramachandran. A survey of distributed shared memory in loosely-coupled systems. Technical report, College of Computing, Georgia Institute of Technology, January 1991.

[40] David Mosberger. *Scout: A Path-based Operating System.* PhD thesis, Department of Computer Science, University of Arizona, 1997.

[41] Tracy Mullen and M. P. Wellman. A simple computation market for network information services. In *First International Conference on Multiagent Systems*, June 1995.

[42] S. Mullender, G. van Rossum, A. Tanenbaum, R. van Renesse, and H. van Staveren. Amoeba: A distributed operating system for the 1990's. *IEEE Computer*, pages 44–53, June 1990.

[43] K. Murray, A. Saulsbury, T. Stiemerling, T. Wilkinson, P. Kelly, and P. Osmon. Design and implementation of an object-orientated 64-bit single address space microkernel. In *2nd USENIX Symposium on Microkernels and other Kernel Architectures*, September 1993.

[44] R. Pike, D. Presotto, K. Thompson, and H. Trickey. Plan 9 from bell labs. In *Summer UKUUG Conference, London*, pages 1–9, July 1989.

[45] R. Pike, D. Presotto, K. Thompson, and H. Trickey. Plan 9, a distributed system. In *Spring EurOpen Conference, Troms.*, pages 43–40, May 1991.

[46] M. Ranganathan, A. Acharya, and J. Saltz. Distributed resource monitors for mobile objects. In *International Workshop on Operating System Support for Object Oriented Systems*, October 1996.

[47] Thierry Le Sergent and Bernard Bertomieu. Balancing load under large and fast load changes in distributed computing systems - a case study. Technical report, University of Edinburgh and LAAS/CNRS, 1993.

[48] John F. Shoch and Jon A. Hupp. Computing practices: The 'worm' programs — early experience with a distributed computation. *Communications of the ACM*, 25(3):172–180, March 1982.

[49] M. Sloman and J. Kramer. *Distributed Systems and Computer Networks.* Prentice Hall, 1988.

[50] Adam Smith. *An Inquiry into the Nature and Causes of the Wealth of Nations.* University of Chicago Press, Chicago, 1976.

[51] Mark Stewart and Peter Willett. Nearest neighbour searching in binary tree search trees: simulation of a multiprocessor system. *Journal of Documentation*, 43(2):93–111, June 1987.

[52] Ion Stoica and Hussein Abdel-Wahab. A new approach to implement proportional share resource allocation. Technical report, Department of Computer Science, Old Dominion University, 1995.

[53] Ion Stoica, Hussein Abdel-Wahab, and Kevin Jeffay. On the duality between resource reservation and proportional share resource allocation. Technical report, Department of Computer Science, Old Dominion University, 1996.

[54] J. Vochteloo, S. Russell, and G. Heiser. Capability-based protection in a persistent global virtual memory system. Technical Report 9303, School of Computer Science and Engineering, The University of New South Wales, March 1993.

[55] C. Waldspurger, T. Hogg, A. Huberman, J. Kephart, and W. Stornetta. Spawn: A Distributed Computational Economy. *IEEE Trasactions on Software Engineering*, 18(2), February 1992.

[56] C. A. Waldspurger. *Lottery and Stride Scheduling: Flexible Proportional-Share Resource Management.* PhD thesis, Department of Electrical Engineering and Computer Science, Massachusetts Institute of Technology, 1995.

[57] C. A. Waldspurger and W. E. Weihl. Lottery scheduling: Flexible proportional-share resource management. In *First Symposium on Operating Systems Design and Implementation*, November 1994.

[58] L. Walras. *Elements of Pure Economics.* Allen and Unwin, 1954.

[59] M. P. Wellman. A market-oriented programming environment and its application to distributed multicommodity flow problems. In *Journal of Artificial Intelligence Research*, volume 1, pages 1–22, 1994.

[60] M. P. Wellman, Willian Walsh, Peter R. Wurham, and Jefery K. MacKie-Mason. Some economics of market-based distributed scheduling. In *Eighteenth International Conference on Distributed Computing Systems*, 1998.

[61] M. P. Wellman and Peter R. Wurman. Market-aware agents for a multiagent world. In *Modelling Autonomous Agents in a Multi-Agent World*, 1997.

[62] J. E. White. Telescript technology: The foundation for the electronic marketplace. White paper, General Magic, Inc., 2465 Latham Street, Mountain View, CA 94040, 1994.

Appendix A

Simulating ERA

This appendix describes the ERA simulation. It is important for two reasons to present the issues and construction of the simulation. Firstly, to provide a particular context for the mechanisms and algorithms of the general framework. Secondly, to provide more detailed description of its function to validate the simulation's results presented here.

This appendix starts by discussing the evolution of the simulation to the current version as ideas in this thesis were worked out. This leads to a description of the kind of system which has been chosen for simulation. Following these, the various parts of the simulation are then detailed, such as threading and the memory sub-system. How the architecture is reflected in the simulation is discussed in these sections, along with the problems discovered in using the ERA framework. To conclude, this appendix describes the instrumentation of the simulation so that the information presented in this thesis could be recorded.

A.1 Introduction

The ERA architecture has been through many versions, as ideas have been examined, before settling on the version described in this thesis. For the start, due to the complexity of interactions all the ideas had to be analysed in terms of their effect on the rest of the system.

In ERA Version I, processes bought guaranteed resource access. Money was distributed and the amount of money held was strictly limited using the techniques described in this thesis. A system of loans provided temporary priority for processes. A simple event-driven simulation was constructed for this system, in an object-oriented environment. Objects representing processes interacted through an event-queue which allowed contracts to be sent and monetary transfers to be performed and monitored. The simulation assumed a single CPU and other resources, such as memory and network, were ignored.

These ideas represented lead to problems. Guaranteed allocation of resources presented fairness problems between competing processes. Since, because of the guaranteed nature of the resources, race-conditions existed with respect to prices and timing, when an allocating process tried to ensure it obtained the resources it required.

Version II of the architecture and the simulator were derived to overcome these problems, and to investigate the issues of multiple resources on multiple nodes. This simulation included CPU, Memory and Network over a configurable number of nodes. The operating system style used was based on the message-passing paradigm. In this way, the cost of network links were explicit to process-centric processes, and so were the memory systems on each node. Large events, such as sending contracts, were broken down into the events of sending, transferring and receiving the contracts. Loans were dropped in favour of the more efficient overdraft system. Preemptive proportion-sharing was adopted, although resources were deterministically allocated between competing processes. This simulation was also event-driven, although the event structure was far more complex than before due to a finer-granularity. Also, this simulation did not provide data on the overhead or scalability issues of the architecture. At this stage, the ERA architecture only consisted of the framework.

Version III of the architecture and simulation were then designed so as to provide an emulation environment, rather than a simulation, and allow the execution of real code for both the ERA framework and user-domain processes. After the construction and testing of the ERA framework in this simulation, the ERA marketplace was conceived along with examples of process-centric resource allocation policies. This simulation is necessarily more detailed and complex, but the execution of real code allowed easier testing of new ideas; and the measurement of real overhead and scalability.

This is the simulation described in this appendix. Details of how the emulation environment is provided together with other issues uncovered in the implementation are also discussed.

A.2 The simulator

The simulation models a set of networked workstations. Each node has its own processor and memory and network links to other workstations. These links are assumed to be point-to-point network links with neighbouring nodes. Point-to-point links are the common denominator of all networks and so can, potentially, be used to examine most types of networks and topologies. Each workstation is assumed to have a user who wishes to run programs in the distributed environment which runs across the entire system.

The distributed operating system implemented is a Single Address Space Operating System (SASOS) [43]. A SASOS provides a single shared address space across all nodes of a distributed system, using a Distributed Shared Memory (DSM) [39] system. The system assumed by this simulation is a DSM rather than a DVSM. Thus, it is assumed that the other nodes of the distributed system act as the backing store for each node. Thus, a page fault generates network requests to obtain the page which must exist on another node.

A SASOS was used for simulation purposes for several reasons[1]. Firstly, work at that time in the Systems Architecture Research Centre (SARC) involved the ANGEL SASOS [43]. This provided an introduction and learning ground for SASOS issues. Secondly a SASOS simplifies issues of migration and location independence through its use of a single shared address space. Simply knowing the address of an object in the system is sufficient to access it. This simplifies the namespace for processes in the ERA architecture by using global object addressing. Thirdly, the single address nature of the emulated operating system simplifies the simulation. A single address space can be provided easily along with paged-based memory protection.

[1]It should be noted that the ERA architecture is as applicable to the message-passing paradigm as the shared-memory paradigm.

The simulator is constructed in two parts: the emulation environment and the code to provide the functionally of a real SASOS based on the ERA framework. The simulation is constructed under FreeBSD UNIX [3] as a single process. The emulation environment provides a virtual kernel to convert this UNIX process into an emulated distributed system. The responsibilities of the virtual kernel are as follows: Multiplex ERA node kernels in turn, map the ERA processes onto the underlying UNIX process context, provide emulation and protection of a shared memory address space between executing ERA processes.

Above the virtual kernel, the ERA per node kernels provide all operating system functionality required on an individual node. These kernels are the ERA based operating system and exist outside the ERA process-centric system. Both the application and marketplace processes exist above these, using the ERA process-centric framework. The principle of this separation is that the code above the virtual kernel is as close as possible to a real ERA-based OS, with the process code being executed by the emulated environment.

The simulation provides control of CPU, real memory and the marketplace as process-centric resources using the ERA framework. Both CPU and memory are shared to allow preemptive proportion-sharing. The two proposed memory sharing techniques discussed in Chapter 6 are implemented: real memory page rental and page fault demand costing. Lastly, the marketplace is implemented according to the description presented in Chapter 5.

A.2.1 Execution support

In order to provide accurate simulation of the ERA environment it is important that the execution of each node's kernel and its processes are emulated as realistically as possible. To this end, the ERA simulation supports the execution and preemption of real code for each part of the emulated operating system. In this way, each kernel has a separate context to provide execution of kernel code while each of its processes is allowed to have multiple contexts to allow for interrupt/message box processing along with its normal execution.

The simulation uses threads to provide this support. The virtual kernel provides support for ERA processes, each of which can have an arbitrary number of threads. Each node's kernel, by appearing as a process, can also have multiple contexts.

POSIX threads are one of the most common implementations of a thread package available and are included under the ANGEL operating system. POSIX threads provide cooperative threading and mutual exclusion support by switching the UNIX process' context. Unfortunately, the ERA architecture requires preemptive threading, so that the CPU resource can be revoked when a timeslice finishes.

To overcome this shortcoming, the POSIX thread kernel class was subclassed to allow the thread package's scheduler to be overridden (becoming the "virtual kernel"). Whenever a thread is suspended or resumed, the virtual kernel is called to allow it to pass on the handling of these actions appropriately. In addition, a time-slice clock interrupts the virtual kernel to inform it when the current thread must be preempted.

The virtual kernel uses these stimuli to arbitrate the running threads onto each node's kernel. For example, when a thread is suspended, the CPU server for that thread must be informed. Likewise, when a time-slice expires, not only must the virtual kernel change the executing contexts, but must also inform the old CPU server and the new CPU server of the need to change their state.

Figure A.1 shows a simplified version of the virtual kernel's execution support. This figure is greatly simplified with factors such as the management of the idle thread and the simulator's own thread removed.[2]

With this basic support, the POSIX thread contexts must be converted to be ERA threads which belong to a particular ERA process. ERA processes are then 1st class entities acting as the unit of responsibility for resource allocation.

To achieve this, two classes are provided. Firstly, an ERA thread class subclasses the POSIX class. This class provides a means of finding the ERA process which a thread belongs to, see Figure A.3. The thread contains information on whether the thread is running in Kernel or User mode and what state the thread is in. This is necessary for some threads, principally those for each node's kernel, to run outside the ERA system and have access to privileged information. Since this is an emulation it is impossible to run in a

[2]Since the thread package provides its own idle thread, this thread must be treated as a special case (which belongs to no particular node, CPU Server or ERA process context).

```
void VKernel :: timeslice(void)
{
  block_interrupts();
  {
    if (saveContext(currentThread)==0) {
      currentNode→cpuServer→suspendThread(currentThread);
      currentNode=nextNode();
      newThread=currentNode→cpuServer→nextThread();
      restoreContext(newThread);
    }
  }
  unblock_interrupts();
}

thread *VKernel :: resume(void)
{
  newThread=currentNode→cpuServer→nextThread();
  return newThread;
}

void VKernel :: suspend(thread *tid)
{
  currentNode→cpuServer→suspendThread(tid);
}
```

Figure A.1: C++ like pseudo-code for the virtual kernel's execution support

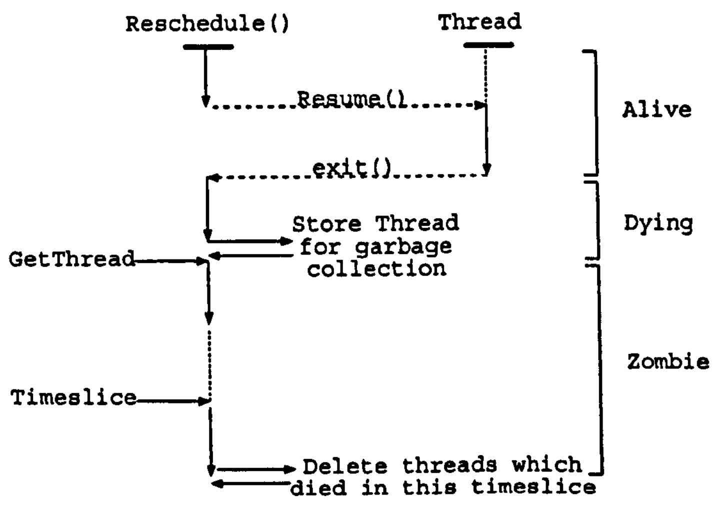

Reschedule()
Thread
Resume()
exit()
Store Thread for garbage collection
GetThread
Timeslice
Delete threads which died in this timeslice
Alive
Dying
Zombie

Figure A.2: The sequence of state changes when a thread is deleted

privileged processor mode (Kernel Mode, etc.) so, instead, the state of this information is stored in each thread. Storing this information in the thread enables a single ERA process to be able to run partly in kernel mode and partly in user mode by using different threads.

The variable *DeathState* is used to monitor the state of each thread as it dies. A thread can be deleted at any point in its execution, either by a thread itself or by the ERA system. In either case a thread may be destroyed with references still existing in the node kernels and virtual kernel. These kernels need to tidy their state. For example, when a thread is deleted the rest of the context switch to the next thread must still be executed. In addition, its kernel must remove this state from the run queue rather than saving it for further execution. This variable enables a thread to be flagged as 'Dying' to allow the thread to be terminated gracefully. Once the thread has finished the simulation code and returned to the thread package, the thread is marked as a 'Zombie' on the current CPU server. Zombie threads are those which are alive, in that they can run, but their state has been removed from the ERA simulation. At the end of the timeslice any zombie threads are garbage collected and deleted using their destructor method. This interaction can be seen in Figure A.2.

A process class provides the context for each process, including the ERA monetary information, see Figure A.4. The 'process' class is stored by each node's kernel as its

```
class Thread : public thread
{
  /* ... access methods ... */

private:
  Mode mode; // KernelMode or UserMode
  Boolean ecoThread; // Account for the resource usage by this thread?
  DeathState deathState; // Alive, Dying or Zombie
};
```

Figure A.3: C++ like pseudo-code for the state for an ERA thread

Process Control Block (PCB). The state of the 'process' class falls into two categories: simulation support and ERA support. Attributes, such as name and code, represent the simulation support category. *Name* is a name assigned to the process for debugging and logging purposes.

The *code* attribute indicates the code for this process. This means that a *process* only represents the PCB of a process and not the code itself (text segment). A simple object-oriented solution to extend the process so as to include the different code for each process type, would be to subclass the 'process' class for each process. For the first two versions of the simulation this is sufficient, but not for the emulation. There are several reasons for this. Firstly, separating the code from the PCB more closely represents the structure of a real operating system with the kernel/user space separation. This is reflected in the emulation, by allowing the code for the process to exist in the ERA managed user space, while the process PCB exists in a node's kernel space. Secondly, there is a need for different types of process in the system, for example system servers and user servers. The process' code can be subclassed appropriately from the 'code' class according to process type without alteration to the PCB represented by the 'process' class.

ERA support requires further state. To support the use of servers for vital system resources, each process has the address of its current server for CPU, Memory and Market kept for it. When access to each resource is required these values are consulted. A record of the node the process is currently located on is also kept. This is useful for migration and system call purposes.

For each vital resource, the standing orders of the process are kept separately from this information. This enables a process to change its standing orders for the next pay round, while using its current servers, which are kept in the *currentXXXServer* variables.

For each process, its monetary state is also kept here. A *money* attribute indicates the current amount of money the process has. While *wageLevel* indicates the level of the process' wage. These are used by the kernel when updating the process' wage each pay round. *maxLoan* represents the maximum overdraft which the process can have. To provide management of these under migration, a logical clock is kept to determine which pay round the process is currently on. This is initialised, when the process is created, with the current logical clock of the node's kernel and allows the wage payment to be ensured when migrating between nodes where pay rounds may not be synchronised.

The *upcalls* variable is the process' upcall message box. This structure allows messages to be sent to a process, for example sending a contract to a server. The process' thread may block on the upcall message box until messages are available. Likewise, a sending process may block on this message box until space is available or the message has been processed (Request-Reply semantics). For more information see Chapter 4.

This all results in the system structure shown in Figure A.5. Each node's kernel is the unit of multiplexing from the viewpoint of the virtual kernel. Thus, the virtual kernel multiplexes kernels, each of which has one currently executing thread of execution. In this way, the kernels map their timeslices onto the simulation of that node. Thread calls are passed onto the appropriate CPU server for that node, as are memory requests generated during that period.

Each node's kernel provides all kernel support for the ERA monetary system. Since the ERA system requires little extra kernel-wide support, the node's kernel need only deal with the pay round process. Each kernel executes a thread to pay the wages of the processes executing on its node. This thread waits for the pay round period using the timer facilities of the virtual kernel and, when the timer interrupt occurs, is woken into performing the payment of wages. For each process, this thread executes the code in Figure A.6 with interrupts masked. The messages EndPayround and StartPayround allow the process to

```
class Process {
   /* ... Access functions ... */

private:
  char              name[80];
  // Parent process for wage tree
  Process          *parent;
  // The CPU server running the process
  Process          *currentCPUServer;
  // The local Memory server receiving memory requests
  Process          *currentMemServer;
  // The Pricing server being used by default
  Process          *currentMarketServer;
  // The Kernel of the current node
  NodeKernel       *currentNodeKernel;

  // Amount of money the process has
  Money             money;
  // How much the process will get paid
  Money             wageLevel;
  // The limit of the overdraft it can obtain. (-ve number)
  Money             maxLoan;

  Upcalls           upcalls;               // Message box
  // Pointer to Code object of the process
  Code             *code;

  // Contracts for each standing order
  StandingOrder    *CPUStandingOrder;
  StandingOrder    *MemStandingOrder;
  StandingOrder    *MarketStandingOrder;

  // The payround for which the entity has been serviced. -1 if not.
  int               payround;
  // How many threads are using this e ntity. Used for deletion.
  Card              refCount;
};
```

Figure A.4: C++ like pseudo-code for the state of an ERA process

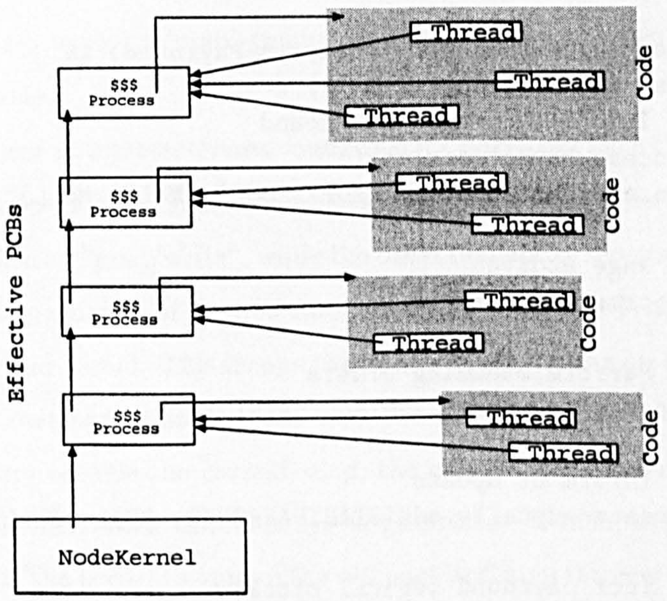


Figure A.5: The per node structure for the simulation's execution support

know that more money is available for it to spend. Most of the work is simply updating the Process structure. Standing order dispatch is simply a matter of sending a message. This message is sent asynchronously, so as not to block the kernel.

A.2.2 Memory sub-system

The memory sub-system supports the Distributed Shared Memory (DSM) system of the ERA memory servers which, in turn, support the SASOS single address space. Each node uses its real memory as a cache of currently held pages from the shared single address space. Since this is an emulation, this structure must be provided for the memory servers inside a single UNIX process. Issues, such as memory protection between processes, which for purposes of simulation are unimportant, are ignored by the virtual kernel.

Each memory server, independent of the algorithm used, needs to keep a virtual address to real page table. It is important to support this table so that each server is then able to access a page of the shared address space when a page fetch is required. Since the emulation exists in a single UNIX process, a single address space for the UNIX process already exists. This simplifies the simulation of this page fetch, since implicitly every thread can access

```
if ((process->getPayround()  < currentPayround) &&
    (process->getWageLevel() > 0)) {
        // Inform of end of pay-round
        process->upcalls.clear();
        process->upcalls.add(ENDPAYROUND, NULL, NULL);

        // Wage update
        process->updateWage();

        // Perform standing orders
        process->performStandingOrders();

        // Inform of update
        process->upcalls.add(STARTPAYROUND, NULL, NULL);

        // Incr payround logical clock..
        process->incPayround(currentPayround);
}
```

Figure A.6: The per process code executed by each node's kernel in order to process wage updates

every page in the simulation's process. Each memory server therefore only needs to be able to request, from the virtual kernel, which *page(s)* it wants to currently have as its real memory. Then the virtual kernel can allow access to those pages which are considered real memory and protect access to those which are not.

It is important to know which pages to fetch and when. When a program executes it accesses text and data segments, not all of which may currently reside in the emulated real memory of the current node. There are two ways of determining which pages the code is accessing so that this can be detected. Firstly, the compiler can be modified to cause a well-defined function to be called every time memory is accessed. This not only requires modification of the compiler but it also imposes a high overhead, since each access (for example, each instruction) requires a memory access and thus a procedure call. Secondly, memory protection permissions can be used, which allow interrupts to be generated when marked pages are accessed. UNIX conveniently has a set of user-level calls to allow a process to change the memory permissions of each of its pages. This mechanism is used since it only requires work to 'map' (make available) the page when it is not already available on that node.

By using calls such as *mprotect() and munprotect()*, pages can have their access permissions changed to support each node's memory server. The pages in each memory server's page table are marked as "read/write", while the other non-active servers' pages are marked as "no access". The page map of the new node's memory is installed each time the node is changed by the virtual kernel. This means that any code executing on that node can only access the current memory located on the node's emulated real memory. When accesses are made to memory outside the current map, the current memory server is informed of the access by the kernel, when the interrupt is generated. It then decides which page to replace and requests the kernel to unmap the old page and map the new page. The memory server updates its page table and execution continues.

To simulate the time that a page fetch would normally incur, the page fault code waits for an interval according to a configurable normal distribution before returning execution. This simulates an average network access time for fetching fixed size pages.

A.2.3 Kernel memory and User memory

Clearly not all memory used by the ERA OS can be in user-space, since access to memory requires money and access to the memory server. For example, the memory system runs as a server process, but it cannot reside in user space since access to its text or data would require access to itself. Instead, certain processes must exist outside the ERA memory server controlled address space. As implied, this is achieved by divided the address space into kernel memory (access not controlled) and user memory (access controlled). Kernel memory is simply memory obtained off the UNIX process' heap, while user memory is a block of memory allocated off the UNIX process' heap by the virtual kernel and protected with the mprotect() call.

Any memory allocation in user space is performed through the virtual kernel onto the user memory segment. For simple malloc() and free() calls these can be redirected to the virtual kernel with macros. In addition, user space objects such as 'code' objects need to be placed in user memory too. In C++, this can be achieved by overloading the *new* operator with a call to *malloc()* into user space. Likewise *delete* can be overridden. To achieve this, a sub-class of the 'code' class was created which also inherits the methods to override the *new* and *delete* operators.

A.2.4 Memory allocation

In this user memory, memory is allocated in a first-fit fashion, which coalesces neighbouring memory chunks to reduce fragmentation. This is sufficient for simulation purposes.

As discussed in Chapter 5, due to the problem of a process running out of money and dying which can leave shared data structures in an inconsistent state, it is important to keep track of memory shared between ERA processes. To solve this problem, the kernel provides an atomic 'pointer update' primitive to add new elements to shared data structures. This requires that memory be tagged as shared, or private, so that failed updates (which are still private) are deleted when an agent dies, but that added elements are not deleted (since they are now shared).

Since individual processes do not have separate address spaces, which can be removed when the process is deleted, then an alternative is needed. The solution provided in the simulation is to record the blocks which have been allocated to each process: when an element is marked as shared it is removed from the calling process' list; when updates succeed the old shared data is automatically freed. This solution does require a lot of state and more efficient techniques exist, but for simulation purposes it suffices.

A.2.5 Timing support

The emulator includes support for interrupt driven timers, for operations such as page fault timing and timeslice management, working in simulated time. They refer to short time intervals and require a high resolution timer. UNIX provides such high resolution timers through the itimer() system call and the SIGVTALRM signal. The itimer call is suitable since, it is not only high resolution but also is based on the process' execution time (virtual time) rather than real time.

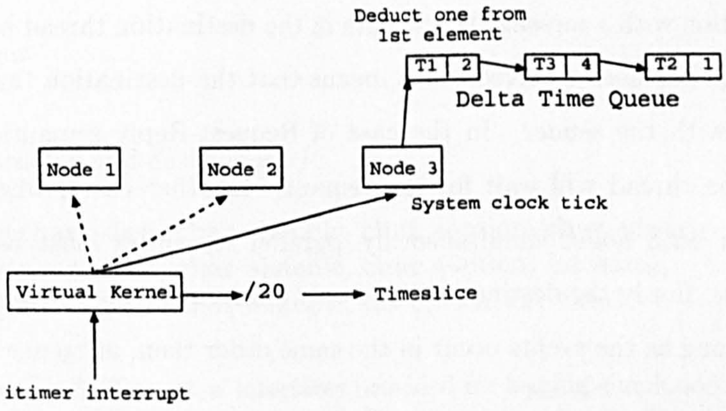


Figure A.7: The use of an 'itimer' to provide per node interrupt timer support for threads

The simulation uses this call to provide per node timer interrupts. The time seen by each process must not be affected by the number of nodes being emulated. The simulation therefore uses itimer to provide an ERA system clock at a configurable quantum. This system clock is used for two things. Firstly, after a certain number of system clock ticks, a virtual kernel timeslice is performed. Secondly, the current node's kernel is called to update its virtual time. This means that time on an emulated node only increases when the node is scheduled by the virtual kernel. This allows a node's kernel to provide timing services to processes.

Processes requiring timer interrupts register their required time interval with their kernel. Each kernel keeps a delta list of local timer events it must service, which is updated on each system clock tick. At each request the kernel creates an entry in the delta timer queue and suspends the calling thread. Events which reach delta time 0 are dispatched, by resuming the suspended calling thread. This structure can be seen in Figure A.7.

A.2.6 Processor coherence

It is important that code simulated as executing in parallel has the same semantics as executed in a real parallel situation. Nodes are designed to run with a share-nothing paradigm between kernels. So that it is only when the parallel entities interact that there is a potential problem. Simultaneously executing code can interact in one of two ways: either by sending messages, or by sharing memory through DSM.

Communication with a message box results in the destination thread being schedulable once the message has been received. This means that the destination thread can execute simultaneously with the sender. In the case of Request-Reply semantics this does not matter, since the thread will wait for the request. In other cases, when the execution can continue on both nodes simultaneously, parallel semantics must be ensured. Two possibilities exist: firstly the destination code only affects separate remote information. In which case as along as the events occur in the same order then, at coarse grain, the result is the same.

The second possibility is that the code accesses information shared with the sender, either through sending other messages or through shared memory. If information is shared through message passing then the resulting ordering is simply the serialisation of the messages, which is the case mentioned above. While, if memory is shared, the resulting semantics are a result of parallel memory accesses. Ensuring parallelism of shared memory at fine granularity carries a high overhead since, at every shared access, the sharing process' memory must be synchronised and this means executing the remote node's code up to the same point in time as the access to the shared variable.

This tight memory coherency is not a problem for the emulation, since the address space is actually shared between simulated nodes, but the parallel execution semantics impose too high an overhead on the simulation. So, a weak coherency model between nodes (Acquire/Release semantics [39]) is used instead. In this model, whenever access to a page is required it must be acquired first. The semantics allows multiple reads but, on a write, the writing process must acquire the page for its sole use. After access the page is released for reading and parallel accesses can again occur. During write access other read copies are invalidated.

To summarise, these mechanisms ensure the emulation has parallel semantics since the acquisition of a page (for either read or write) acts as synchronisation between sharing threads. If a page cannot be acquired, then the thread is blocked until it can be. Therefore, if threads share data, the coarse-grain semantics of the serialised version are the same as the parallel version, because potentially interfering threads are blocked.

```
class Logger
{
public:
   /* constructor and destructor */

   void log(char *dest, char *handle, char *option, char *data);
   void log(char *dest, char *handle, char *option, int data);
   void log(char *dest, char *handle, int option, int data);
}
```

Figure A.8: The set of interfaces provided for logging simulation data

A.3 Instrumenting the simulator

In order to obtain execution data from the simulation, it must be 'instrumented'. This instrumentation applies to all parts of the simulation, from recording modifications to each process' money, to the position of advertising agents in the ERA marketplace. To cater for this wide range of instrumentation needs, a generic logging interface is provided.

The logging interface is provided globally in the simulation and allows a set of data to be recorded in a fixed format log. The logging object provides several alternative typed output formats, see Figure A.8. These range from all string to all numerical arguments. The first argument describes the source of the data. This is usually the name of the process, but it can be the ID of the thread. The second argument describes the type of data that is being logged. This might be "wageUpdate" or "migration", for example. The last two arguments are dependent on the data being logged. Generally they are the operation that is being recorded and its value. For instrumentation, where only a value or operation is required, the last argument is used and the penultimate argument is left NULL.

The logging interface takes these arguments and writes them to the log indicated by the first argument. The data is written to the log in the format: `<current time on node>:arg2:arg3:arg4`. Figure A.9 shows an example log file.

```
2005:WagesUpdated:0:0
2005:payoutFunds:350:350
2005:payoutFunds:350:0
```

Figure A.9: An extract from the log file of a process

A.3.1 Obtaining simulation information

A lot of data routinely computed within the ERA system is logged. This data is useful not only for analysis, but also for debugging the system. This includes the payment of wages, monetary transactions and the length of timeslices used by each process so that information like the fairness of CPU proportion sharing can be monitored. However, overhead information of the operating system itself is not routinely computed.

Ideally, the simulation should measure the amount of processor time which is spent performing the critical ERA algorithms. This would require measuring the time taken by all the algorithms as the system is scaled. These algorithms execute frequently and over short periods of time. To measure their total execution time would require a timer which has sufficient resolution to record the short periods to be totalled. Unfortunately, FreeBSD[3] does not have a timer with sufficient resolution to count the execution of a few hundred or thousand instructions on a processor which can work at many MIPS[4].

A solution is to count the number of instructions and use this count to calculate the time spent in each section. This approach works well on RISC style processors, since on average the processor executes 1 instruction per clock cycle and stalls due to memory latency and branch-prediction are usually explicit with the use of 'NOP' instructions. Therefore, an instruction count in a particular code section can be translated to a timing if the processor clock speed is known. Unfortunately, machines in the SARC environment have Intel Pentium[5] processors, with a CISC architecture which has been greatly adapted to enable super-scalar execution of instructions. A Pentium instruction can take anywhere from 1 clock cycle to tens of clock cycles, thereby preventing direct conversion into timings.

[3]In fact, all versions of UNIX
[4]Million Instructions Per Second
[5]Intel and Pentium are registered trademarks of the Intel Corporation

Thus, to get useful measurements, this thesis makes a few assumptions about the code executed. The first assumption is that the instruction mix in the code is pretty uniform, so that particular parts of the code do not run quicker than others due to the mix of instructions. This is a reasonable assumption since all the code is generated by the same compiler (GCC/G++) without optimisation enabled. The second assumption is that issues of memory references are constant across the code sections. Thus, a particular part of the code does not make any worse memory references than any other. This assumption too is fair, since all the code uses the same style list and array structures as any other and so should exhibit the same spatial locality as any other. Issues of temporary locality still exist, and unfortunately are not measurable in this environment, since it is multi-user/process presenting differing temporal localities over time.

With these assumptions the *relative* proportions of time spent in code sections can be measured, as long as the number of instructions can be counted. The Pentium processor contains several internal profiling registers which contain information such as the number of instructions executed so far. Unfortunately, this information is of little use to the simulation, since it represents the total number of instructions executed by *all* processes executing in the system, rather than just in the simulation. For this information to be of any use, the operating system would require modification in order to store this count at each context switch and restore it appropriately on re-execution. This would enable the instruction count for the simulation's UNIX process to be obtained, which could then be subdivided between internal threads on thread context switches. This approach, however, would require changes on several people's workstations.

The alternative taken in this thesis is to modify the output of a compiler[6] so as to count the execution of each instruction it emits which requires a dedicated register. The Pentium has very few registers for general purpose use anyway. Reducing the available registers to normal processing would reduce the execution speed of the simulator further. So, to minimise the overhead imposed it was decided to use the PIC[7] register, used for

[6]again GCC/G++

[7]Position Independent Code

shared libraries, etc. This means the simulator code can only be statically linked, but as a result there is little extra use of registers. The modification required to the GNU compiler is shown in Figures A.10 and A.11. Figure A.10 shows the modification necessary to remove the PIC register from the tables used by the compiler for parameter passing and computation. This involves two modifications to the init_reg_sets() code in GCC, to remove the register from the table for each type. Together these modifications ensure that the PIC register (%ebx) is never touched by the compiler. Figure A.11 shows the modifications to count each instruction's execution. The output_asm_insn() function is called whenever a new instruction is emitted. To count each instruction's execution, a package of instructions are emitted, to increment the PIC register for each instruction emitted.

The package of instructions outputted to count each instruction executed consists of three instructions. This is because the Pentium has no instructions to increment a register without affecting the condition flags. If these condition flags are not preserved between instructions then all conditional statements will no longer work. This is because the instruction count code can be inserted in between the instructions of an `if (x<4)` {} statement. So, the first instruction saves this state of the processor to the stack, then the PIC register is incremented and the process' state restored from the stack.

To provide access to this information by the simulation, a small library, libICount, was created which provides functions to set and get the current instruction count of the processor, see Figure A.12. When the simulation starts it clears the count with `setICount(0)` and then `getICount()` can be used to monitor the progress of the code in terms of instructions executed.

With this modification to the compiler, to ensure that the PIC register is not modified accidentally during its execution, all the libraries and startup-code used by the compiler in constructing the ERA simulation binary must be recompiled with the modified compiler itself. Also, the UNIX process context saving and resuming calls `setjmp()` and `longjmp()` were modified so as not to save and restore the value of the PIC register. This allows the instruction count of the ERA virtual kernel context switch to be recorded. This can be seen later in Figure A.13.

```
/* Function called only once to initialize the above data on reg usage.
   Once this is done, various switches may override. */

void
init_reg_sets ()
{
  register int i, j;

  /* First copy the register information from the initial int form into
     the regsets. */

  for (i = 0; i < N_REG_CLASSES; i++)
    {
      CLEAR_HARD_REG_SET (reg_class_contents[i]);

      for (j = 0; j < FIRST_PSEUDO_REGISTER; j++)
        if (int_reg_class_contents[i][j / HOST_BITS_PER_INT]
            & ((unsigned) 1 << (j % HOST_BITS_PER_INT)))
          SET_HARD_REG_BIT (reg_class_contents[i], j);
    }

  bcopy (initial_fixed_regs, fixed_regs, sizeof fixed_regs);
  bcopy (initial_call_used_regs, call_used_regs, sizeof call_used_regs);
  bzero (global_regs, sizeof global_regs);

  /* Remove the PIC register from general use, since it */
  /* shouldn't be used for any binary. */

  /* Register shouldn't be used in calling */
  call_used_regs[PIC_OFFSET_TABLE_REGNUM] = 0;

  /* Add register to table of globak 'unusable' registers for */
  /* normal computation use */
  globalize_reg(PIC_OFFSET_TABLE_REGNUM); */

  /* Compute number of hard regs in each class. */
  ...
}
```

Figure A.10: Removing the PIC register from use by parameter passing and computation.

```
void
output_asm_insn (template, operands)
      char *template;
      rtx *operands;
{
  register char *p;

  /* Emit an instruction sequence to count */
  /* the instruction in the PIC reg. %ebx */

  fputs("\tpushf\n", asm_out_file);
  fputs("\taddl $1,%ebx\n", asm_out_file);
  fputs("\tpopf\n", asm_out_file);

  p = template;
  putc ('\t', asm_out_file);

  /* Normal code to output this instruction */
  ...

}
```

Figure A.11: Instruction counting using the PIC register

```
void setICount(unsigned int value)
{
  register unsigned int instnCount asm("%ebx");

  instnCount=value;
}

unsigned int getICount()
{
  register unsigned int instnCount asm("%ebx");

  return instnCount;
}
```

Figure A.12: setICount() and getICount() provide the interface to the count kept in the PIC register

Code sections monitored

The main purpose of the instruction counting described previously is to allow the division of overhead between the ERA operating system and a traditional operating system and user level processes. Two techniques are used to achieve this. Firstly, the instructions executed in the operating system are counted, along with the total number of instructions executed per timeslice. This provides the division between operating system and user level processes as a relative proportion per timeslice. Then, in the operating system code, a global flag exists to indicate whether the various ERA mechanisms should be executed or not. This allows the code required by the ERA architecture to be removed and the proportion counted again. When the ERA code is switched off, traditional techniques, such as round-robin scheduling of processor access, apply.

The simulation distinguishes several sections of code as ERA operating system code, these are:

- The pay round process: wage payment, message box upcalls and processing standing orders.
- The selection of access by server processes: cost assessment, contract processing and client selection.
- The selection of resources by client processes: assessing prices and modifying standing orders.
- The marketplace: placing adverts and advertising with agent processes.
- Acquiring resources through sending contracts to server processes.
- The context switch cost in the virtual kernel.

The number of instructions, counted as executed by the PIC register between entering and exiting these functions, is recorded. In the case of the context switch function in the virtual kernel, instruction count is measured only between entering the call and the tidy up path when 0 is returned as a result by setjmp(). This means that, when a thread is resumed it immediately exits the virtual kernel and continues with user code without the few instructions required being counted.

The logging interface described in Section A.3 is the only code which does not have its instructions counted. As soon as the call is made, the current instruction count is saved, interrupts are disabled and the logging takes place. Once complete, the old instruction count is then restored. This means that no matter how much or little logging is enabled the results are not seriously affected.

A.4 Problems

Several problems were encountered during the construction of the simulation and these slowed its construction considerably. This section details the more serious examples.

A.4.1 Reentrant code

BSD UNIX is not designed to have multi-threaded processes. This means there are potential problems with the reentrant[8] nature of the system libraries. Two such problems were encountered during the development of the ERA simulation.

The first is that `malloc()` and `free()` are not reentrant. These memory calls are used in the simulation in two situations: storing system data structures and by user level threads. Storing system information is performed by system threads with interrupts disabled, so `malloc()` does not need to be reentrant. User level threads must, by definition, use ERA controlled memory. This means that memory allocated must go through the appropriate memory server and on to the virtual kernel. As a solution the virtual kernel implements its own `malloc()` and `free()` calls which are thread-safe.

The second is that the file system calls are not reentrant. This means file logging is also not reentrant. This is not a problem while logging information from system-side

[8]The ability to be reentered at any point while others use the same call. Lack of re-entrance usually results from unprotected access to shared data structures

processing, but there are problems when user-level threads start to log information, such as the progress of the marketplace agent processes. To side-step the problems, the virtual kernel was supplemented by a version of the UNIX file interface which provides mutual exclusion to file access.

A.4.2 Setjmp()

The UNIX calls `setjmp()` and `longjmp()` presented a couple of nasty problems. Due to the nature of these calls, when something goes wrong everything falls to pieces quickly and unpredictably. This makes the tracking and fixing of bugs associated with these calls a real pain.

The first problem involves the state saved by these calls. As soon as a complex user level process was written to calculate a matrix multiplication while it considered prices in the marketplace, the ERA process would randomly stop with a "floating-point division by 0" error. Eventually, this bug was tracked down, not to the user process itself, or memory corruption by the simulator, but to the fact that the UNIX `setjmp()` call was not correctly saving the state of the floating point unit in the Pentium.

On inspection of the code used in BSD UNIX, it appeared that the setjmp() call only saves the status word of the floating point unit and restores it when `longjmp()` is called. This means that all the state of the floating point stack inside the 80587 FPU is lost! Clearly the designers of BSD UNIX felt that saving the state of the floating-point unit was unnecessary and, since it is quite large, would caused additional overhead. To overcome the problem, the `setjmp()` and `longjmp()` calls were modified to save the state of the FPU correctly and the storage required to save this state was increased. Besides this direct problem, 80x87 documentation does not make clear how many bytes are required to save the FPU context. Most books describe the 8087, or 80387 at best, and list it as requiring 63 bytes. Unfortunately, the 587 actually has increased state information (many old 16 and 8-bit values are extended) and 106 bytes are required. But this information had to be deduced rather than obtained!

The second problem was found with `longjmp()`. The `longjmp()` call restores the signal mask, which stops the code being preempted. After restoring the mask, the rest of the state is setup. Unfortunately, this makes it possible for the `longjmp()` call to install a signal mask which leaves the BSD virtual timer still enabled. This can intermittently result in the state of a thread being only partially restored.

This problem stems from the lack of re-entrance (again) of the `setjmp()` and `longjmp()` calls. If the signal mask is not correctly setup before `setjmp()` is called, then an insufficient signal mask can be restored by `longjmp()`. In this scenario, only part of the state may be saved, thereby corrupting execution. The solution to this problem was to ensure the virtual timers were always masked before `setjmp()` was called and also before `longjmp()` was called. This means that the timer signal is always masked while saving and restoring state with these calls. See Figure A.13 for the modified version of `setjmp()`. For brevity `longjmp()` is not shown.

A.5 Configuration

This section describes the format of the configuration file used for the ERA simulation. Each run of the simulation is governed by a configuration file `config` which is usually placed in a separate directory with a `logs` sub-directory. The simulation takes the directory containing the configuration as its parameter. It then reads the specification from the configuration file to configure the distributed system and the processes to run and logs information into its `logs` sub-directory. Within this directory, the simulation creates a log for each process run and any additional logs as soon as they are written to. For example, a log `Overheads` records the instruction count information from the run into an additional log.

A.5.1 Global parameters

The configuration file starts with a section (in a predefined order) of the global parameters of the simulation. These are as follows:

DEBUG={ *YES*| *NO*}

```
ENTRY(setjmp)
/* set param to #0 so not to add to current mask */
        pushl   $0
/* causes signals not to be modified and instead return current sigmask */
        call    PIC_PLT(_sigblock)
/* %edx temporarily holds sigblock result */
        popl    %edx
/* get parameter (ptr to jmpbuf) and put it in %ecx */
        movl    4(%esp),%ecx
/* get %eip from frame to save state of program pc before call */
        movl    0(%esp),%edx
/* store PC in buf[0] */
        movl    %edx, 0(%ecx)
/* don't bother storing the PIC reg, since using it for instn counting */
/*      movl    %ebx, 4(%ecx) */
        movl    %esp, 8(%ecx)          /* store SP in buf[2] */
        movl    %ebp,12(%ecx)          /* store BP in buf[3] */
        movl    %esi,16(%ecx)          /* store SI in buf[4] */
        movl    %edi,20(%ecx)          /* store DI in buf[5] */
        movl    %eax,24(%ecx)          /* store sigmask in buf[6] */
        fnsave  28(%ecx)               /* Save *ENTIRE* FP state (106 bytes) */
        fwait                          /* and wait for it to finish */
 /* Clear eax so that setjmp return value is 0 */
        xorl    %eax,%eax
        ret                            /* return to caller */
```

Figure A.13: The modified version of the BSD 4.4 UNIX setjmp() call.

This variable describes whether debugging output is generated on standard output when the simulation is run. This debugging information is in addition to the information logged by the simulation and includes the run queues of the emulated processors and when timeslices occur.

`MONETARY=`{*YES*| *NO*}

This variable describes whether or not the code for the ERA framework is enabled or not. This option is useful for disabling the ERA framework for comparison of overhead with and without the variable enabled.

`InitialMoney=`*unsigned int*

This is a check variable. If the sum of money later allocated to processes is not equal to this value, then the simulation stops and complains. It does not need to know the global amount of money, but acts as a validity check for the monetary values.

`WageTime=`*timeslices*

This variable describes the number of timeslices between payment of wages to processes.

`EndSimulation=`*unsigned int*

This variable describes, in milliseconds, when the simulation should end. The simulation will end somewhere between this time and time+one timeslice (20ms).

`VirtualMemSize=`*unsigned int*

This variable describes the size of the memory across all nodes in kilobytes. It is used in creating the ERA controlled memory heap onto which the memory servers are mapped.

A.5.2 Node descriptions

Following these global variables are brace enclosed descriptions of each node to be simulated. The node description is opened with:

```
{(0)(100000)Broker0
```

The first parameter is the number of the node. This is used for describing the network links between nodes. The second parameter is the amount of money given to the market server running on this particular node. The final parameter gives the node's name for debugging purposes. This line causes the simulator to create a kernel for the node and create a market process which is given the parameters provided.

This line is then followed by two or more lines describing the processes that are to run on the node. At least two processes must be run: a CPU server, a Memory server, and these are specified first. The general format for each line is as follows:

`At` *time processType(processName, money, ... params ...)*

Some of the types of processes are now listed:

`CPUResource(`*name,money*`)`

`MemResource(`*name,money,memory*`)`

`MemRental(`*name,money,memory*`)`

`MatMult(`*name,money,size,*`Maximise,`*workers*`)`

`MatMult(`*name,money,size,*`Spawn,`*time,childSize*`)`

`Dhrystone(`*name,money*`)`

`SGP(`*name,money*`)`

`MPEG(`*name,money*`)`

etc.

Each node description concludes with a closing brace.

A.5.3 Market links

The links between the market processes, which are uni-directional network links, are now described. The format of the network links is terse, but clear. Each line refers to the nodes in order, starting at 0. A line has the format:

numberOfLinks`@`*destinationNode*$_1$*,...,destinationNode*$_n$ *where n is the number of links.*

Thus a simple line of 4 nodes can be configured:

```
1@1
2@0,2
2@1,3
1@2
```

A.6 Summary

This appendix has described the structure and details of the ERA simulation, which emulates a variable number of nodes executing an ERA architecture style operating system. The operating system implemented in this simulation is a Single Address Space Operating System (SASOS) under which ERA processes run in the emulated environment. The use of a SASOS simplified the implementation of the emulation, by removing problems such as location independence.

The emulated environment was provided by mapping a single UNIX process into a virtualised distributed machine with a shared address space and a *virtual kernel.* This kernel multiplexes ERA kernels, called *node kernels,* in which the real operating system code runs, using the calls provided by the UNIX system and the virtual kernel. Above this level, the ERA user level processes execute. The virtual kernel uses a modified POSIX thread library to provide preemptive multi-processes. These threads, how they are mapped into ERA monetary accountable threads, and how they are grouped into ownership by processes, have been described.

The memory emulation issues were then presented. The difference between kernel and user memory was discussed in detail. Following this, other support features, especially with respect to parallelism, were discussed and issues concerning the provision of the timing support were presented. Then the issue of coherence between emulated nodes, so as to ensure correct parallel semantics in a serial environment, was discussed.

Instrumentation, in order to record data inside the simulation, and compiler modifications in order to give information about the relative execution overhead of the operating system with respect to the entire system, were then described. Finally, problems found during the construction of the simulation were reviewed along with the format of the simulators configuration file.

Taken together, the sections in this appendix have described the details of an accurate representation of a multi node distributed system running an ERA architecture type of operating system. The simulation provides full emulation of an underlying system connected by point-to-point network links. By using the relative instruction counts of the operating system and user level code it has been possible to examine the overhead and scalability issues of the operating system to an acceptable level of accuracy.