Implementation of a Proactive Load Sharing Scheme

R. Anane
Department of Computer Science
Coventry University
Priory Street
Coventry CV1 5FB, UK
r.anane@coventry.ac.uk

R.J. Anthony
Department of Computer Science
The University of Greenwich
Park Row, Greenwich
London SE10 9LS, UK
r.j.anthony@gre.ac.uk

ABSTRACT
This paper presents a proactive approach to load sharing and describes the architecture of a scheme, Concert, based on this approach. A proactive approach is characterized by a shift of emphasis from reacting to load imbalance to avoiding its occurrence. In contrast, in a reactive load sharing scheme, activity is triggered when a processing node is either overloaded or underloaded. The main drawback of this approach is that a load imbalance is allowed to develop before costly corrective action is taken. Concert is a load sharing scheme for loosely-coupled distributed systems. Under this scheme, load and task behaviour information is collected and cached in advance of when it is needed. Concert uses Linux as a platform for development. Implemented partially in kernel space and partially in user space, it achieves transparency to users and applications whilst keeping the extent of kernel modifications to a minimum. Non-preemptive task transfers are used exclusively, motivated by lower complexity, lower overheads and faster transfers. The goal is to minimize the average response-time of tasks. Concert is compared with other schemes by considering the level of transparency it provides with respect to users, tasks and the underlying operating system.

KEYWORDS
Distributed systems, Load sharing, Proactivity, Tranparency

1. INTRODUCTION
This paper describes the design and implementation of Concert; a rich-information load sharing scheme for loosely-coupled systems. This scheme presents a new emphasis to load sharing in preventing load imbalance [12]. Concert generates information concerning load levels and the nature of tasks ahead of when it is needed. The information is used to schedule tasks in such a way that an acceptable load distribution is maintained. This proactive approach differs from the traditional reactive approach in which load sharing is triggered by load imbalance.

Research into and development of load sharing schemes in distributed systems is motivated by three factors:
1. The differential load between processing nodes can be significant.
2. The unused computing power in a network can be exploited by migrating tasks from overloaded nodes to underloaded ones.
3. The cost of load sharing can be offset by the performance gains that can be achieved by migrating tasks.

Most load sharing schemes perform four main functions: information generation, selection of tasks, selection of location and transfer of tasks. Each of these functions is governed by a specific policy. The information policy is concerned with determining the load index, generating load information (and possibly other information) and disseminating this information. The selection policy is concerned with scheduling and involves the selection of tasks for transfer. The location policy is responsible for determining where to send a task. The transfer policy determines when and how a transfer should take place. The second function of the policy is often redundant, being implicitly defined by the transfer mechanism employed.

In a reactive load sharing strategy, the selection, location and transfer policies react to load imbalance or to a shortage of resources typically determined by comparing load metrics against threshold values. This approach relegates the information policy to a subordinate role. Reactivity is characterised by a tight coupling between the selection policy and the information policy. Another characteristic of reactivity is that migration decisions are often based on a narrow and 'shallow' information base. It is common for the load index to be linked to the level of activity at one resource only, such as the CPU queue length [8, 10, 17] or the number of remote processes allocated to each node [6], and to be generated on demand. Task information is often ignored [6, 8, 10, 15].

Once a significant load imbalance has occurred the system suffers inefficient resource utilization which implies inefficient processing of tasks, and longer response times. The problem can be rectified only be employing (expensive) preemptive task transfers or by waiting for the problem tasks to complete. We consider that prevention is better than cure, where possible.

SAC 2003, Melbourne, Florida, USA

A proactive load sharing strategy seeks to prevent load imbalance by avoiding wasteful transfers and by ensuring that tasks are placed at the most appropriate node for execution. This approach requires a wide information base for making scheduling decisions. The traditional emphasis of the *load* index (i.e. as a measure of load) is less helpful for a proactive strategy. A *resource availability* emphasis is more useful. This should represent a multidimensional analysis of resource availability. A proactive strategy is also marked by the decoupling of the information policy and the selection policy. This approach ensures that the information is made available in advance by the information policy, at the local level, before it is actually required and used by the selection policy. Scheduling decisions by the selection policy can therefore be taken quickly, independently of the information policy.

A load sharing scheduling policy can be either preemptive or non-preemptive. Where scheduling is preemptive the focus is on the selection, transfer and location policies. The scheduling is flexible; tasks can be transferred at any point in their execution. Preemptive scheduling, however, requires a complex mechanism for gathering the state of a process, transferring the state to the target node and re-instating the task. Because the state of a process is distributed throughout various structures of an operating system, the implementation of a preemptive transfer mechanism requires significant kernel modification. Sprite [5] is an example of a load sharing scheme that follows a reactive approach. In this case the scheme is supported effectively by a sophisticated preemptive implementation.

Non-preemptive load sharing puts more emphasis on the information policy. Tasks can only be transferred at task initiation time. The cost of poor transfers is high because task placements are permanent. This promotes a proactive strategy. The roles of the selection policy and the transfer policy are significantly diminished. The selection policy only needs to determine whether a newly arrived task is suitable for transfer. The transfer policy has only to determine whether or not a sufficient load differential exists between nodes to justify a transfer. This is often decided by comparing a simple load index against a threshold value. Non-preemptive load sharing requires a much simpler mechanism than its preemptive counterpart. A process is not created until the transfer is complete; there is no state to collect and transfer. Because of this simplification, non-preemptive load sharing can be implemented entirely at user level. However, this is likely to require program modification and/or re-linking with new libraries. User level implementations, like Utopia [18], usually require special commands to invoke load sharing and are thus generally not transparent to programs nor to users.

The rest of this paper is set out as follows. Section 2 provides an outline of the design of Concert. Section 3 describes the proactive aspects of Concert. Section 4 discusses the performance of Concert. Section 5 gives an evaluation of Concert and other schemes in terms of the transparency achieved, and Section 6 presents some conclusions.

2. CONCERT ARCHITECTURE

Concert is a load sharing scheme with the aim of improving the performance of distributed systems, measured in terms of the average response-time of tasks, by placing tasks at the most suitable node for execution. Concert implements a non-preemptive transfer policy within a proactive load sharing strategy, in order to reduce the occurrence of the symptoms of load imbalance. Under this scheme, the information policy plays a crucial role and the environment acquires a special significance.

2.1 Environment

A proactive strategy postulates the existence of an environment with a specific structure and behaviour. One aspect of the environment of a distributed system is that it exhibits some intrinsic characteristic behaviour that can be monitored. More to the point, however, is the realization that the capacity of that environment to support the role a load sharing scheme can be enhanced. The enrichment process is facilitated by the quality and the quantity of the information that can be extracted from the environment. Three fundamental assumptions form the basis of the proactive strategy in Concert:

1. Tasks are executed more than once on nodes; records of tasks' behaviour can be compiled and stored for future reference.
2. The environment is a rich source of information and is responsive; mechanisms can be set up to probe the environment effectively.
3. The environment, although dynamic, is relatively stable with respect to the nature and behaviour of tasks. This offers the opportunity for the introduction of predictive models.

Proactivity in Concert relies on the active gathering of information about task behaviour and node resource availability in order to enhance the potential ability of the load sharing scheme to achieve its goals. This strategy involves both environment and load sharing scheme in meaningful and timely interactions, with the aim of generating the required information and ensuring that it conforms to the following criteria:

- Availability. Network-wide information must be available locally and immediately. The generation and compilation of information is performed concurrently with processing as an essential activity of the system.
- Relevance. The information must be relevant to, and support the goals of, load sharing. Relevance implies the adoption and determination of adequate load indices, and the choice of correct thresholds.
- Accuracy. Accurate information is a requirement for effective scheduling. Accuracy has implications for the implementation level of the information policy. In Concert it is achieved by accessing kernel-generated information via new system calls.
- Freshness. Up-to-date information can prevent the routing of tasks to nodes that are already overloaded. The system should generate up-to-date information, incorporate mechanisms for dealing with obsolete information, and ensure that updates occur when required.

In Concert the basic environment provided by the Linux operating system is enriched by the information policy of the load sharing scheme. Under this scheme the network is transparent to the user. The main objects of interest are tasks and nodes.

2.2 Design requirements

It was decided at the outset that Concert's focus on proactivity should not be realized at the expense of sound design and engineering principles. To that end, a number of design goals were identified. The design should accommodate the following criteria:

- Decentralization. Nodes should operate autonomously on locally cached information. Centralized design should be avoided as it can lead to the creation of bottlenecks and a single point of failure. Decentralization should be reinforced by the availability of network-wide information at the local level.
- Efficiency. The design should minimize communication between nodes and between the modules themselves. Likewise the processing overheads associated with information generation, use and storage should be minimized through careful design.
- Scalability. There should be support for logical division into clusters. This enhances scalability by reducing inter-node communication.
- Transparency. The system should be transparent to users and to tasks. Users should not be aware of the existence of the load sharing scheme and tasks should not be modified to accommodate the operation of the load sharing scheme.
- Reliability. Processing nodes should not rely on services provided by other nodes, and should be sufficiently autonomous to permit failure independently of each other.

2.3 Architecture

In designing Concert, special emphasis was put on abstraction as an architectural feature in order to support the proactive strategy. This requirement translated into a modular structure.

Conceptually the architecture of Concert conforms to the traditional four-policy structure of a load sharing scheme. The information policy is implemented in the form of the Resource Availability Daemon (RAD) and the Resource Utilization Daemon (RUD). The selection, transfer and location policies are all incorporated in the Distributed Scheduler (DS). The combination of the three policies into a single implementation module increases efficiency by reducing communication between components. In this implementation, the transfer policy is largely redundant since only non-preemptive transfers are supported and transfers can only occur when a new task arrives. Transfers are initiated directly by the Distributed Scheduler. In addition to this fundamental structure, Concert incorporates another module, the Remote Task Manager (RTM), which manages the execution of migrated tasks. The Remote Task Manager acts as an abstraction of the remote kernel to client nodes from where tasks were transferred. The structure is illustrated in Figure 1.

The implementation modules are now discussed.

The Resource Availability Daemon has three functions: 1, measure local load and resource availability, using a new system call to extract kernel-generated information; 2, disseminate load information to other nodes within the cluster; and 3, maintain a table of load information concerning the local and remote nodes. This information is made available to the Distributed Scheduler on demand.

The Resource Utilisation Daemon records the resource usage of tasks. Kernel-generated information is extracted via a new system call and used to update a database of tasks' resource requirements information. The system automatically learns the characteristics of tasks entirely on-line.

The Distributed Scheduler is a global task scheduler operating at the cluster level. A task may be scheduled for execution at any processing node in the cluster. The Distributed Scheduler decides where to execute tasks (locally or at a specific remote site). This decision is based on locally held information concerning the load level at processing nodes (obtained from the Resource Availability Daemon) and concerning the resource requirements of tasks (obtained from the Resource Utilisation Daemon database). The Distributed Scheduler directly dispatches the task for execution. When a task is to be executed remotely, the Distributed Scheduler sends a task-transfer message to the Remote Task Manager at the appropriate remote node. If the transfer is accepted, the Distributed Scheduler creates a shadow task locally to handle I/O forwarding.

The Remote Task Manager manages the execution of transferred tasks. In so doing, it acts as an abstraction of the remote kernel for the Distributed Scheduler at the sending node. The Remote Task Manager receives task-transfer requests and dispatches the tasks for execution on its local kernel. I/O and signals are forwarded to remotely executing tasks via the Remote Task Manager.

2.4 Implementation

Concert is implemented in C and integrated with the Linux operating system. The implementation consists of four user-space modules, kernel modifications and bash shell modifications.

Communication between the user-space modules within a node is achieved through remote procedure calls and local access to a shared data file (the RUD database).

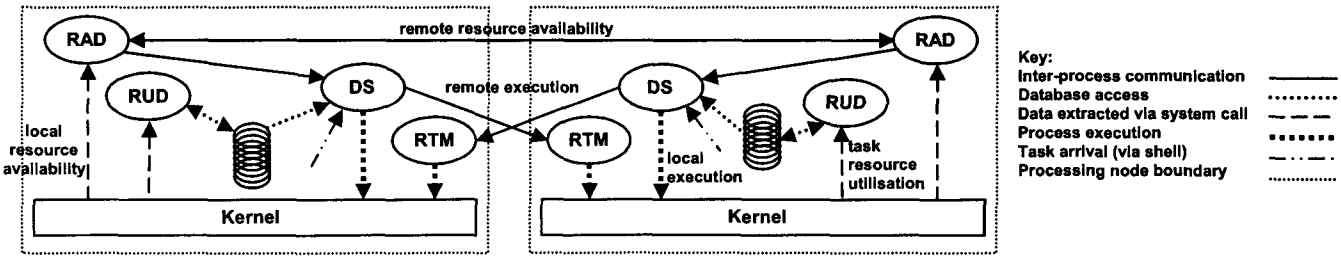

Figure 1. Architecture of Concert

Communication between modules at different nodes is achieved through message passing, implemented using UDP. Communication between user-space modules and the operating system is achieved via new system calls. In each case the choice of communication mechanism is motivated by the need to create an efficient and robust system.

For efficiency and low latency the modules that implement the information policy, the Resource Utilisation Daemon and the Resource Availability Daemon, are integrated with the kernel. This is accommodated by the modification of key system calls including *sys_exit* and *sys_exec* to generate the information and new system calls to provide the communication interface.

3. PROACTIVITY IN CONCERT

The proactive approach adopted in the design of Concert has determined to a large extent the variety and the nature of the information generated by the system, as well as the emphasis of the scheduling process on avoiding load imbalance. In Concert, proactivity involves the search for the node most capable of handling additional load. This approach has influenced some design decisions in response to concerns about the efficiency of its implementation.

3.1 Information

Through a continual interaction between the information policy modules and the environment, Concert is actively engaged in gathering, generating and disseminating information that should be readily available, relevant, accurate, and up-to-date. This service is provided by a number of modules that support directly the scheduling policy, and isolate it from the network configuration.

The generation of load metrics is central to load sharing and encapsulates the need to provide accurate and relevant information. A load index is a metric against which the load at each node in the system is measured. Many different load indices have been used in load sharing schemes. Condor [11], for instance, considers a workstation as idle if no user activity is detected over a short interval; Stealth [9] uses CPU utilization as its load index. Utopia [18], on the other hand, makes use of a load vector which incorporates CPU queue length, free memory, disk transfer, swap space and number of concurrent users.

Concert relies on a wide information base for its scheduling. An investigation of several load measuring methods for various resources was conducted as part of the search for a suitable load index [2]. In Concert many aspects of the information are covered:

- Information base. The shift of emphasis that proactivity entails is reflected in Concert in the use of a *resource availability index* instead of a *load index*. It includes: CPU queue availability, CPU utilization availability and amount of primary memory available. This information is recorded by the Resource Availability Daemon. Along with a few schemes such as History [16] and Stealth [9], Concert keeps a record of resource utilization and behaviour of tasks. Concert identifies these by the name of the executable. This information is used to avoid unnecessary transfers and to provide support for predictive load sharing. Information recorded by the Resource Utilisation Daemon includes CPU time requirement, I/O activity, memory usage, and access to any device-special files. This aspect is dealt with in more detail elsewhere [1,3].
- Heterogeneity. Accuracy of information is also required when performance heterogeneity occurs in a network, namely when nodes have the same architecture but different levels of resource. The components of the resource availability index are normalized before dissemination. CPU speed is normalized by converting to a ratio against the speed of the slowest CPU in the system.
- Information update. Up-to-date information is vital to the implementation of a proactive strategy. Scheduling decisions based on obsolete information may lead to an increase in load imbalance. Concert uses timestamping on receipt of messages for dealing with obsolescence of information about resource availability. This has the added advantage that it reinforces autonomy and enhances accuracy. With respect to tasks, resource utilization and behaviour are updated each time the task executes.
- Information access. This approach is also expressed in the equal status granted by the load sharing scheme to the nodes, and the lack of centralized control. This is manifest in the autonomy of the nodes in scheduling decisions, and in the symmetry of their relationship in terms of access to local and remote information. The replication of information enables each node to participate actively in the scheme. The lack of a centralized control ensures that the system is fault-tolerant, and that nodes contribute autonomously to the generation of information and to the execution of tasks.
- Predictions. This aspect of the design of Concert is one of the most evident manifestations of the proactive strategy, and is aimed primarily at enhancing the capacity of the environment to support load sharing. Prediction of resource requirement, based on previous behaviour of similar tasks, is combined with load information to achieve prediction of response-time. It is designed to determine which node will produce the shortest time for a given task. This heuristic approach is, however, non-optimal and may lead to errors.

3.2 Scheduling

As stated earlier, proactivity in Concert puts more emphasis on avoiding load imbalance. The implication of this bias is that the scheduling policy should identify the node that is *most capable of* handling additional load (specifically the new task), rather than the node with the *least load*. The use of both resource availability information and resource utilization information enables the global scheduling policy to focus on the prevention of resource shortages at nodes, by jointly selecting the tasks to be migrated and the nodes to which they will be transferred. This helps to ensure that tasks are placed at the most appropriate node, and more importantly that unnecessary migration of tasks is avoided. Only carefully selected tasks are transferred. Experimental results support this approach and show that keeping more tasks locally leads to better system-wide performance [16].

The scheduling process is triggered by the entry of a task into the system. Only the Distributed Scheduler is involved since the RUD information base, for previously known tasks, is already set up by the information policy. Unknown tasks are always executed locally; an entry in the RUD database is created when they

terminate. Two conditions need to be fulfilled before a load transfer is considered:
1. The existence of a significant resource availability differential between nodes to ensure that a load transfer will actually be beneficial.
2. The availability of some threshold amount of resource at the target node. For example, scheduling decisions that would deplete primary memory can be disallowed.

This approach relies heavily on the quantity and the quality of the information in order to avoid poor decisions and wasteful transfers.

3.3 Efficiency and Overheads

The overheads arising from the generation and use of large amounts of information could potentially outweigh the performance gains achieved by load sharing. The implementation of a proactive strategy, with its emphasis on a wide information base, is bound to lead to an increase in the number of messages across the network. Decentralized control in Concert requires replication of information at node level. There is, however, a trade-off between information policy requirements and scheduling policy processing needs. Wider, readily available, relevant and up-to-date information leads to quicker and more accurate scheduling decisions. Furthermore, as was observed in Chorus, the flexibility of the policies that can be adopted in a load sharing scheme is enhanced by the range of the information provided by the information modules [14]. The concern about the overheads associated with proactivity has been born in mind during the design of Concert which has purposely minimized overheads. Examples of ways this has been achieved include: careful, informed task transfer decisions help ensure that only beneficial transfers occur; very low communication levels between nodes, partly due to the use of a non-preemptive task transfer mechanism; and the modules that realize the load sharing scheme in Concert are inactive most of the time. The average processing overhead of the load sharing scheme with was found to be 2.83%.

Awareness of potential inefficiency at node level has also shaped the implementation of the load sharing scheme in its structure and its interaction:

- Scheduling policy implementation. The adoption of a proactive policy with non-preemptive scheduling, allowed for an efficient implementation of the selection policy. As the transfer policy was effectively redundant, the selection and location policies were merged into one single module, the Distributed Scheduler.
- Kernel modification. Although it is possible to implement the information policy at user-level, it was decided to opt for a selective modification to the kernel to support an efficient implementation of the information modules.
- Load metric generation. The periodic activity for the load-metric generation was carefully considered. A metric-generation period of ten seconds was chosen as a compromise between minimizing the costs of generation and dissemination and minimizing the obsolescence of information.
- Scheduling. Global scheduling decisions are made at the node level by the Distributed Scheduler without the need to interact with other nodes. The replication of information leads to quicker scheduling decisions.

At the network level various design decisions were motivated by the need to reduce the communication traffic between nodes:

- Task migration. Transfer of tasks at task initiation time is inherently efficient, compared to those schemes that implement preemptive policies such as checkpointing.
- Information dissemination. The dissemination method adopted in Concert is aimed at reducing the exchange of unnecessary messages. It is a state-change policy, which is governed by a minimum period between transmissions in order to limit the effect of frequent state changes.
- Heterogeneity. The normalization of resource availability reduces the amount of information transmitted across the network, simplifies the role of the Distributed Scheduler and thus leads to a faster, less intrusive system.
- Cluster support is provided in order to minimize network traffic between widely spread nodes and to enhance scalability. Nodes are grouped logically into clusters, and message exchanges are localized.
- A connectionless protocol, UDP, was chosen in order to minimize overheads and, in particular, network traffic.

4. PERFORMANCE

Where Concert is concerned, the load sharing policy comprises the location, and selection policies, and uses some or all of the information gathered by the information policy. For each arriving task a decision must be made as to where to execute it; either locally or at a specific remote location.

A very wide range of load sharing policies can be supported by Concert. A policy can be simple, based only on CPU queue length for example, or complex, taking advantage of a rich information base. The performance that can be achieved is dependent on a number of factors: choice of load sharing policy, workload characteristics, workload intensity, system configuration and extent of performance heterogeneity present. Extensive evaluation of Concert has been carried out along each of these dimensions.

The performance results are briefly presented here in terms of the speedup improvements achieved using several different load sharing policies. These policies differ in the type and amount of information they use and in the ways they use the information. The information available to load sharing policies can be divided into three categories: load information (load and resource availability metrics), System resource information (node's performance heterogeneity) and task information (resource requirements). Numerous load sharing policies have been evaluated. A subset of the policies are described:

- Dummy: This policy always executes tasks at their arrival site. It is used as a control to measure absolute performance of the other policies.
- Random: A processing node whose CPU run-queue length is above a threshold value selects a target node randomly from the set of processing nodes whose CPU run-queue length is below the threshold value.
- CPU Queue1: The processing node with the lowest CPU run-queue length is chosen.
- CPU Queue2: As CPU Queue1, but the CPU run-queue lengths are adjusted to account for differences in processing speed. The CPU Queue1 and CPU Queue2 policies are

popular in low information-use load sharing schemes and in queuing theory models.

- MemSat: As CPU Queue2, but with a pre-condition that the target node must have sufficient primary memory available to satisfy the requirements of the specific task to be transferred.
- Prediction: Using detailed information concerning both the resource requirements of the specific task to be scheduled and the load levels on each resource at processing nodes, the response-time of the task is predicted for each processing node. The processing node with the lowest response-time prediction for the specific task is chosen. The prediction algorithm takes into account performance heterogeneity and has been described in detail in [1].

The information use of the policies is compared in Table 1.

Each of the policies are evaluated under consistent circumstances. Background workloads consist of a mixture of real and artificial tasks specifically devised to cover a wide space of task behaviour and intensity and thus designed to exert a wide range of load levels on the various resources at each processing node. Scripts are used to simulate users executing a wide range of different tasks and run continually in a loop. Each task is followed by a short random delay to simulate think-time. These background tasks are kept local. Foreground tasks, which can be migrated, are executed by further scripts and their response times are automatically recorded. The workstation system configuration is used. Each processing node has its own arrival stream of tasks and will transfer these away to less loaded nodes if possible.

Results are presented in Table 2 in terms of the mean response-time speedup achieved by each of the policies. The Dummy policy is used as a baseline. The proactive approach involves moving tasks to avoid resource depletion where possible. The policies that use little of the information generated (e.g. Random) loose the advantages of the proactive approach that Concert offers. Policies that are more proactive in nature, the Prediction policy for example, have been found to perform better.

The most obvious characteristic of the results is the differential between the performance of Random and CPU Queue1, on the one hand, which are only marginally better than that of Dummy, and that of CPU Queue2, MemSat and Prediction. This performance differential maps directly onto the division between those policies that assume all processing nodes to have the same CPU performance, and those which make adjustment for CPU performance heterogeneity.

CPU Queue2 does not consider memory availability and has been observed to cause memory depletion. This is because a node that has a memory shortage can appear a good target in terms of its CPU load.

MemSat uses task's memory requirement to ensure that tasks are scheduled to avoid the overheads associated with paging where possible. The extent of the benefit of this approach is sensitive to the workloads used, and does not show up clearly in the results presented here. Additional tests have confirmed that when a significant proportion of tasks are highly memory intensive, the MemSat approach pays dividends.

Prediction uses detailed information concerning load levels, performance heterogeneity and the resource requirements of tasks. Thus it has higher overheads. It is successful because it implicitly matches the resource requirement of a task to availability when selecting an execution site. In nearly all cases the prediction of most appropriate execution site is correct, this has the effect of minimising task's response time whilst increasing resource utilisation. In cases when a task's behaviour changes dramatically from execution to execution the predictions of response-time are usually incorrect. However, even in these cases it is generally found that the *ranking* of predictions is correct (i.e. the best target node is still chosen).

Load sharing policy	Mean response-time speedup
Dummy	1.00
Random	1.40
CPU Queue1	1.43
CPU Queue2	2.06
MemSat	1.87
Prediction	2.15

Table 2 Response-time speedups over all workload intensities

The load sharing policies described in this section, together with several others, and a very detailed evaluation of their results can be found in [1]; also a detailed analysis of the overheads is provided.

5. TRANSPARENCY IN CONCERT

One of the design goals of Concert is to achieve a high degree of transparency. Program and user transparency are considered the most important aspects of transparency as they directly affect usability. Users should not have to learn new commands or understand the way load sharing works. It is not always convenient or possible to recompile program code.

Policy	Load metrics used	Metrics normalized for performance heterogeneity	Task characteristics taken into account
Random	CPU run queue	none	none
CPU Queue1	CPU run queue	none	none
CPU Queue2	CPU run queue	CPU	none
MemSat	CPU run queue, memory availability	CPU, memory availability	memory requirement
Prediction	full load signature	CPU, disk, memory availability	full task signature

Table 1 Comparison of the information use of the load sharing policies

Concert employs kernel and shell modifications to achieve transparency to users and programs, i.e. operating system transparency has been traded for the other two forms.

An alternative approach, as employed in Utopia [18] for example, is a user-space implementation requiring no kernel modification, thus achieving portability. However this approach looses transparency to users, programs or both.

In Concert, transparency to users is achieved by incorporating the Distributed Scheduler into the bash shell, thus there are no changes to the user interface. Also, all of the policies: information, selection, transfer and location, are automated; there is no user involvement.

Transparency to tasks is achieved in three ways:

1. NFS is used to provide a consistent namespace across the system for loading executables and accessing data files.
2. There is no difference between local and remote execution in terms of results. This execution transparency is a fundamental requirement of any task transfer mechanism.
3. The interface to existing system calls is unchanged. Programs do not need modification, re-linking or registering.

The implementation is split into user level and kernel level. The requirements for a wide information base and for efficiency led to a tight coupling between the implementation of the information policy and the Linux kernel. This is achieved by the modification of some existing system calls and by the addition of two new system calls, one to interface with the Resource Availability Daemon and one to interface with the Resource Utilisation Daemon. This renders the information policy directly dependent on the kernel. Partial transparency to the operating system is achieved however, by confining the kernel dependent modules to information policy implementation only. The selection, transfer and location policies are implemented in the Distributed Scheduler which is incorporated into the bash shell at the user level.

Concert's provision of some transparency to users, programs and the operating system, is compared with that of a number of load sharing schemes, in Table 3. It is not possible to simultaneously achieve all three forms of transparency. A load sharing scheme that is transparent to the operating system has to introduce a new interface for task submission. This has to involve either new commands or new program functionality, requiring re-linking and/or recompilation.

In its implementation, Concert strikes a balance between two extremes. Utopia and Freedman are examples of load sharing schemes which present fully user-level implementations with no modification to the kernel. Condor, on the other hand requires significant modifications to the kernel so that checkpointing can be supported. Transparency and abstraction are closely linked concepts in Concert. Whilst transparency is a horizontal concept that characterizes the behaviour of the load sharing scheme, abstraction is a vertical architectural mechanism.

Load sharing scheme	Transparency to the operating system	Transparency to users	Transparency to programs
Utopia [18]	Yes, user-level implementation.	No, users invoke load sharing indirectly via the use of special load sharing applications that include a load sharing shell. E.g. *lsrun <command>*	Partially. Some programs need re-linking with a new library, LSLIB. Others use special load sharing applications (e.g. a load sharing shell) as an execution environment.
DAWGS [4]	No, user-level implementation with kernel changes.	No, load sharing is user-invoked: *dawgs <command>* User is sent email when their remote task completes.	Yes.
Freedman [13]	Yes, user-level implementation.	No, load sharing is user-invoked: *lbrun <command>* Programs must be pre-registered for load sharing.	Yes.
Condor [11]	No, user-level implementation with kernel modifications to support checkpointing.	Yes.	Yes. But it is very inefficient for tasks which make frequent system calls because each call is forwarded back to the originator node. This can impact on run-time performance.
Stealth [9]	No, user-level implementation with kernel changes (new and modified system calls).	Partially. Off-line training is required for new task-types.	Yes.
GENESIS [7]	No, it is a specialized platform incorporating its own micro-kernel operating system and kernel servers.	Yes	Partially (for PVM applications). Programmers must adapt to the enhanced GENESIS-PVM interface. New applications can also use the GENESIS-DSM facility
Concert	No, user-level implementation with kernel changes (new and modified system calls).	Yes.	Yes.

Table 3 Comparison of the transparency achieved by seven load sharing schemes

Transparency is largely supported by a set of abstractions. Abstraction is used in two ways in achieving remote execution:

1. The remote node offers to the sending node an abstraction of the kernel through its Remote Task Manager.
2. The shadow process provides an abstraction of the remote execution of the task.

Abstraction is used in information gathering. The Resource Availability Daemon and the Resource Utilisation Daemon abstract away kernel details and thus provide a high-level information base to the Distributed Scheduler.

6. CONCLUSION

In this paper, proactivity in load sharing was defined and contrasted with the reactive approach. The emphasis is put on avoiding load imbalance rather than reacting to its detection. A rationale for the adoption of a proactive approach in distributed systems was also outlined. A load sharing scheme, Concert, was presented as an illustration of a proactive approach to load imbalance. This scheme offers full transparency to users and tasks, and partial transparency to operating system. It also shows that support for an efficient implementation of proactivity, need not be realized at the expense of sound design and engineering principles. Design and implementation decisions were motivated by the need to satisfy stated requirements.

Results indicate that the level of intrusion of the load sharing scheme on the system is outweighed by the benefits of the proactive strategy. Concert's implementation, with its emphasis on abstraction and its separation of mechanism and policy in particular, offers the scope for a wider investigation of the impact of the various scheduling policies on the load sharing process.

This is an on-going project, and further work will be concerned with making this scheme more adaptive, by exploiting the wide information base that it generates. Additionally, a cluster load management service, based on Concert's innovations, but targeting the most-popular desktop operating systems, is under development at Greenwich.

7. REFERENCES

[1] Anthony R, Load Sharing in Loosely-Coupled Distributed Systems: A rich information approach, D.Phil. thesis, Computer Science, University of York, UK, March 2000

[2] Anthony R, Goodeve D, A New Metric for expressing CPU Load, *Proc 4th International Conference on Computer Science and Informatics*, North Carolina, 3, 229-234, October 1998, Association for Intelligent Machinery

[3] Anthony R, Goodeve D, A Model of Process Behaviour, for Predictive Load Sharing, *Proc 17th International Conference on Applied Informatics*, Innsbruck, 226-230, February 1999, IASTED

[4] Clark H, McMillin B, DAWGS - A Distributed Compute Server Utilizing Idle Workstations, *Journal of Parallel and Distributed Computing*, 175-186, 1992, Academic Press

[5] Douglis F, Ousterhout J, Transparent Process Migration: Design Alternatives and the Sprite Implementation, *Software - Practice and Experience*, 21(8), 757-785, 1991, John Wiley

[6] Goscinski A, Towards an operating system managing parallelism of computing on clusters, *Future Generation Computer Systems*, 17, 293-314, 2000, Elsevier Science

[7] Goscinski A, Hobbs M, Silcock J, GENESIS: an efficient, transparent and easy to use cluster operating system, *Parallel Computing*, 28, 557-606, 2002, Elsevier Science

[8] Hao Y, Liu J S, Kim J, An All-Sharing load-Balancing Scheme on the CSMA/CD Network and its analysis, *The Computer Journal*, 37(4) 779-794, 1994

[9] Krueger P, Chawla R, The Stealth Distributed Scheduler, *Proc 11th Intl conference on distributed computing systems*, 336-343, 1991, IEEE

[10] Le P, Srinivasan B, A migration tool to support resource and load sharing in heterogeneous computing environments, *Computer Communications*, 20, 361-375, 1997, Elsevier Science

[11] Litzkow M J, Livny M, Mutka M W, Condor - A Hunter of Idle Workstations, *Proc 8th International Conference on Distributed Computing Systems*, 104-111, June 1988, IEEE

[12] Milojici D.S, Douglis F, Paindeveine Y, Wheeler R, Zhou S, Process Migration, *Computing Surveys*, 32(3), 241-299, September 2000, ACM

[13] Nuttall M, A brief survey of systems providing process or object migration facilities, *Operating Systems Review*, 28(4), 64-80, 1994, ACM

[14] O'Connor M, Tangney B, Cahill V, Harris N, Micro-Kernel support for migration, *Distributed Systems Engineering*, 1, 212-223, 1994, BCS

[15] Sureswaran R, Samaka M, Knaggs J, LOADIST: A Distributed Processing Environment Based on Load Sharing, *Proc Singapore Intl Conf on Networks / Intl Conf on Information Engineering*, 518-522, 1995, IEEE

[16] Svensson A, History, an Intelligent Load Sharing Filter, *Proc 10th Intl Conference on Distributed Computing Systems*, 546-533, 1990, IEEE

[17] Thomas A, Neilsen M, A Load Sharing System for a Network of Independent Workstations, *Proc Intl Conf on Intelligent Information Systems*, 97-100, 1995, ISMM-ACTA Press

[18] Zhou S, Zheng X, Wang J, Delisle P, Utopia: A Load Sharing Facility for Large Heterogeneous Distributed Computer Systems, *Software - Practice and Experience*, 23(12), 1305-1336, 1993, John Wiley

Richard Anthony received his first degree from the University of East London in 1988 and his D.Phil. from The University of York (dept. Computer Science) in 2001. He is currently a senior lecturer at the University of Greenwich. Research interests include: dynamic reconfiguration of distributed systems, cluster management, load sharing, and election algorithms.

Rachid Anane is a senior lecturer in Computer Science at Coventry University. He holds a BSc. in Computer Science from the University of Manchester, an MSc. and a PhD in Computer Science from the University of Birmingham. Research interests include Software Engineering, Distributed Systems and the Modeling and analysis of historical data.