Dynamic allocation of servers to jobs in a grid hosting environment

C Kubicek[†], M Fisher, P McKee and R Smith[†]

As computational resources become available for use over the Internet, a requirement has emerged to reconfigure servers to an optimal allocation to cope with an unpredictable flow of incoming jobs. This paper describes an architecture that allows the dynamic reconfiguration of servers to process incoming jobs by switching servers between conceptual pools. The system makes use of heuristic policies to make close-to-optimal switching decisions. A prototype system which builds on existing resource management software has been developed to demonstrate some of the concepts described.

1. Introduction

There has been considerable activity recently aimed at developing a computing grid — this may be defined as a distributed system on an Internet scale that supports collaboration, data sharing, and resource sharing across disparate domains and platforms [1]. To enable the submission of jobs to remote sites in a transparent fashion, technologies such as a generic system of describing jobs [2, 3] and defining service level agreements (SLAs), relating to issues such as security, performance and quality of service (QoS), exist [4].

A resource management system (RMS) offers facilities that collectively allow users to run jobs on the pool of computational resources it manages. The pool may be made up of donated resources not being used by their owners at a given time, or resources allocated to the pool on a full-time basis [5]. The layout of most RMSs involves a central manager that monitors the state of all the machines in its pool, allocates submitted jobs to resources using scheduling algorithms and holds queues of jobs. Other RMS facilities include job prioritisation, local resource configuration and accounting. The scheduling facility selects the most suitable computational resource on which to run a submitted job, based on resource descriptions obtained from the pool, and individual job requirements. An RMS may be used to run jobs submitted by users in a local domain, or jobs submitted by users from remote sites using grid technology.

We are interested in systems which offer services of different types, requiring dedicated servers. A server in this context is defined as a computational resource capable of processing jobs. Consider a collection of M (possibly distributed) servers partitioned into n subsets, such that subset i is dedicated to serving jobs of type i ($i = 1, 2, k$). Examples of different job types are Web page requests, databases queries and streamed media. To avoid ambiguity, we shall refer to the entire collection as a 'cluster', and to the dedicated subsets as 'pools'.

If the allocation of servers to pools is fixed, changes in demand may cause some queues to become very long, while other servers remain underutilised. This is clearly inefficient. It is desirable, therefore, to develop a policy whereby servers may be re-allocated from one type of service to another. That is the object of this paper.

A reconfiguration policy must take into account both the characteristics of demand (arrival rates and average service times of different types), and costs associated with keeping jobs waiting, and with server reallocation.

Although work in the academic research field is limited on this subject, various commercial products [6, 7] have addressed the problem of server utilisation with claims that the number of servers in an organisation may be reduced by 35—50% using policy-driven resource management to increase utilisation. Reconfiguration of a server is possible due to current networking technology which can transfer data at high speeds and allow data sets and software to be installed from remote locations automatically. Software also exists to automatically install an operating system from a remote network location, allowing a single server to be reconfigured to run a different operating system if needed [8].

[†] North East Regional e-Science Centre, Newcastle.

A method of dynamic server reconfiguration has been studied [9] in which servers are switched between services to cope with demand. Groups of servers or virtual clusters are monitored by a virtual cluster manager which makes decisions on when to switch a server from one virtual cluster to another. A request is made to add a new server to a virtual cluster when its queue has reached a given size, and the virtual cluster manager selects an idle server from another virtual cluster, or a pool of unconfigured machines to import. This approach only takes into account queue sizes for a service when making a decision to switch a server, but other factors are just as important. The time taken to run a job may differ between services, as may the time it takes for a server to be reconfigured. There may also be costs associated with keeping jobs waiting if the services in the system are honouring SLAs — so there are clearly factors other than queue sizes which must be considered when deciding to reconfigure a server.

Scalability is a concern in a system that has a centralised processing point, and an organisation could easily have thousands of servers partitioned into pools. With a central component processing data from all pools, there is potential for a bottle-neck. This issue has been considered and we build on the system described to provide a potential hierarchical solution to the scalability problem.

We present the architecture and prototype implementation of a dynamic pooling system for reconfiguring servers to improve the efficiency of a resource hosting environment. The system partitions computational resources into pools and uses heuristics with current and past system information to make reconfiguration decisions which result in servers that are part of underutilised pools being switched to over-utilised pools.

2. Pool partitioning architecture

The goal of the system architecture is to allow servers partitioned into conceptual pools to switch from one pool to another, where the decision to switch is based on heuristics and system metrics gathered over time. One cluster of servers in an organisation may form numerous conceptual pools, allowing servers to be reallocated to another pool without needing any hardware alterations. Networking technology allows an RMS to control servers all over a large area network, and therefore a conceptual pool may be made up of heterogeneous servers in geographically disparate locations. As members of conceptual pools do not need to physically exist in one location, they are able to switch to other pools, providing all servers run the same reconfiguration software. A component to make and co-ordinate switching decisions which has access to all pools in a cluster is also needed. The main components needed to build the pool partitioning architecture are described below; one machine may play the role of all three at any time.

- The server manager

  Each individual computational server runs the RMS software, and a daemon that allows the server to be instructed to switch to another pool.

- The pool manager

  The component managing a conceptual pool runs an instance of the RMS in a central manager mode, queuing and scheduling jobs on servers in its control. The pool manager keeps track of the servers in its pool, participates in co-ordinating switches of resources to and from other pools, and queries the RMS's central manager for data such as queue length to send to the cluster manager.

- The cluster manager

  Dispatches submit jobs to the appropriate pool based on job type, and stores information about job arrival to be used in reallocation heuristics. The cluster manager receives information from each pool manager which is used, together with stored data, to make reallocation decisions. The cluster manager acts as a co-ordinator between two pools during a server switch, and does not need to run an instance of the RMS.

Jobs of type i arrive at the cluster manager and are dispatched to pool i. Each pool has k_i servers dedicated to run job type i, where the total servers in the system is given as:

$$k_1 + k_2 + \ldots + k_n = M$$

Figure 1 illustrates how instances of the tree components are organised to process jobs.

The different job types the system processes are registered with the cluster manager during initialisation. Job types may correspond to a request for a particular service or to a job dependency such as the required operating system; system administrators may define what a job type represents. Information about the dependencies needed for each job type must be specified so each server in a pool is configured correctly. The cluster manager has a list of addresses which correspond to pool managers to which jobs are to be sent; when jobs arrive at the cluster manager their arrival time and type is noted for future use before they are sent for processing.

Dispatching a job to a pool is a trivial task and requires no scheduling as jobs are queued and scheduled at each pool manager by the RMS, so queues

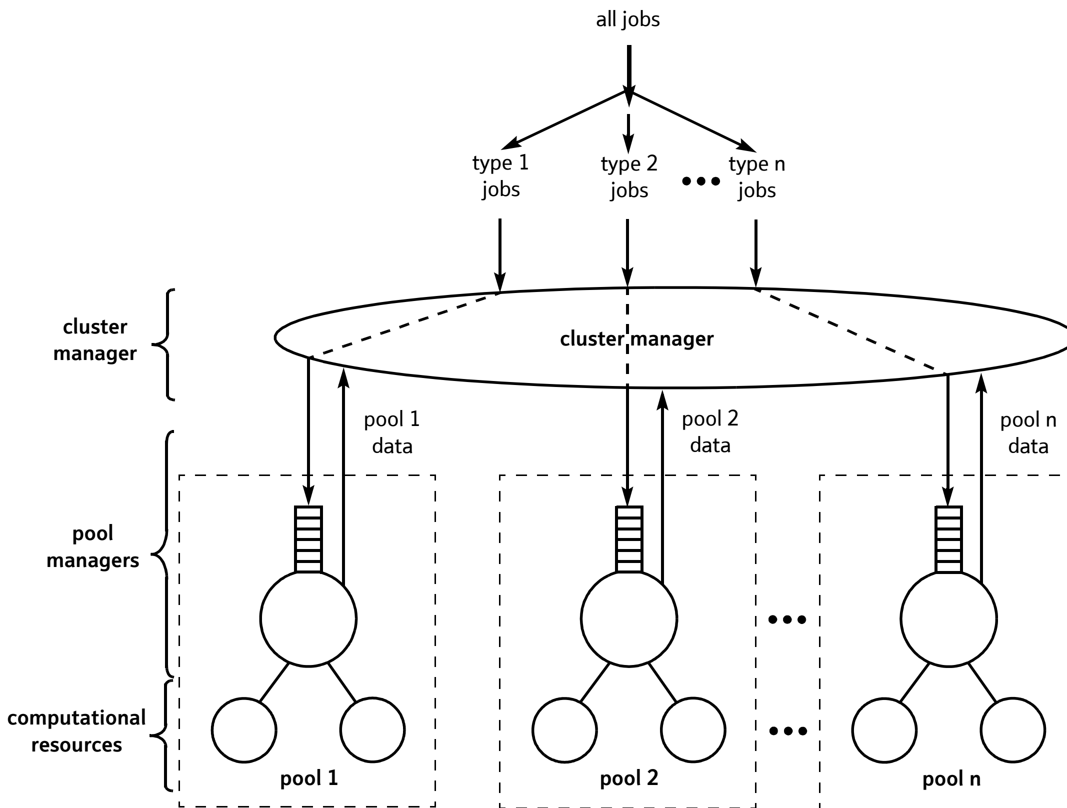

Fig 1 Architecture of a cluster partitioned into pools.

are not likely to build up at the cluster manager. The distributed queuing and scheduling among pool managers helps to disperse load on the whole system. The pool manager communicates with the RMS through a proxy, allowing different RMSs to be used in different pools. Together the cluster manager and pool manager constantly monitor events in the system including the arrival times and processing time of each job type, which allows the cluster manager to make informed decisions on server switching between pools. Different switching policies may be used by the cluster manager provided the necessary system metrics can be gathered for the policy. Data may be obtained by the cluster manager from pool managers in an event-driven or polling fashion. Events that may trigger the sending of data may include a change in the job queue or the completion of a server switch which means switching heuristics are calculated only when necessary. Each pool manager may also be polled at set intervals to send its current state to the cluster manager. A polling approach may not be efficient if a request causes a pool manager to run many monitoring systems, but if monitoring is performed continuously and saved, a poll from the cluster manager only involves reading a value from a data store.

The diagrams in Fig 2 show the communication between the components involved in a server switch. To make a switch the source pool manger (pool manager 1) is informed of the decision to remove one of its servers, and it is also told of the destination pool (pool manager 2) to which the server is to be reallocated (Fig 2(a)). The pool manager may then choose the most appropriate server to be removed, for example a currently idle machine. Any jobs(s) currently running on the server are halted and put back into the queue; then the source pool manager initiates the configuration alteration which may include loading a different RMS to match the destination pool. Information received by the source pool manager in the switch request from the cluster manager contains information about which software packages are to be installed on the server to match the requirements of the destination pool, and the necessary packages may then be downloaded from a software server. Upon successful reconfiguration of the server the source pool sends details of the server that is to be switched to the cluster (Fig 2(b)); the details are then forwarded on to the destination pool manager which can make the necessary adjustments to allow the resource to join the pool.

Adjustments made by a pool manager to accept a server include updating a list of current pool members and any execution of operations needed to instruct the RMS to accept the new resource. The source pool manager updates its list of servers upon successful completion of the switch.

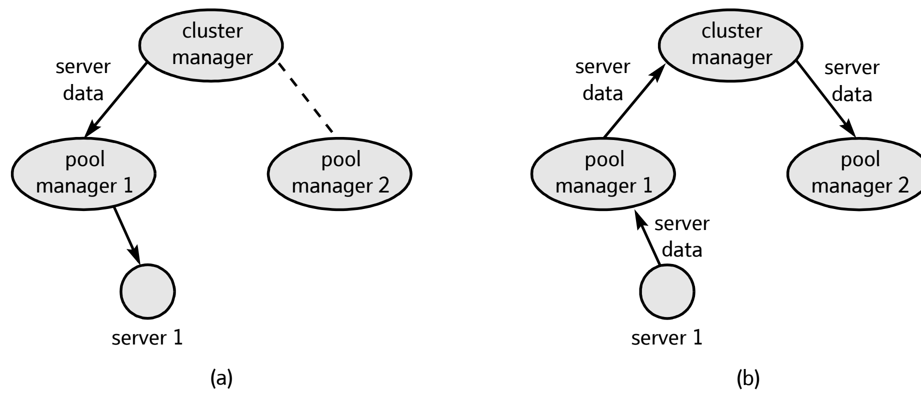

Fig 2 Communication during the switching of a server between two pools.

3. Prototype system design

A system has been developed to demonstrate server switching between pools. The system implements enough functionality of the server manager, pool manager and cluster manager components described above to show servers being dynamically reallocated to pools based on a heuristic. As the main aim of the prototype is to show how resources may be reallocated, a complex job description language such as RSL or JDSL has not been used. Instead Condor class-ads have been wrapped in an XML document which contains file transfer information and the job type. A simple file transfer mechanism has been developed, and Condor is used as a resource manager. Two pools are used to demonstrate the features of the system, as the heuristic used at the time of development supported two pools. The switching that occurs involves a server being taken from one pool, and added to another pool, with the only reconfiguration taking place on the server being adjustments to instances of Condor. To fully demonstrate server switching a graphical interface has been constructed, with a system that allows a user to set system variables to observe the outcome. As the system makes use of potentially unreliable distributed components, some fault-tolerance measures are incorporated into the system.

The main components of the cluster manager implementation are shown in Fig 3. The components represent interfaces which allow different implementations to be plugged in, and therefore the simple file transfer protocol may be replaced by a system such as GridFTP.

The pool communication layer contains components that are used to invoke pool managers. The job submission component sends a job description to a pool manager which will then process the job. Before a job can actually be processed, the file transfer component transports files associated with the job from the cluster manager to the jobs target pool.

An administrator starts the initialisation of the cluster manager during which each pool is registered using an administrator interface. Instructions can then be given to add servers to pools via the administrator interface, although no individual server data is stored by the cluster manager. After pool initialisation the cluster manager is started and starts processing job submissions. While the cluster manager is capable of storing the times of job submissions and job processing times, in the prototype the user sets the average arrival

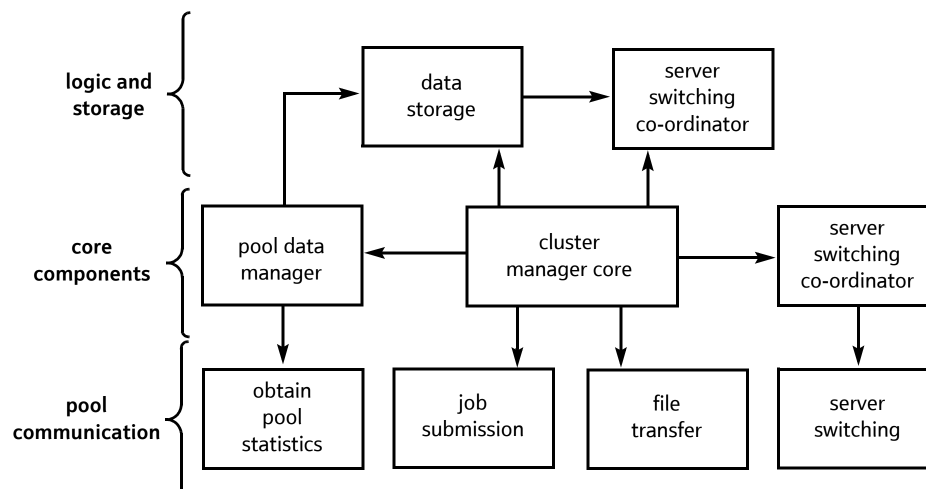

Fig 3 Cluster manager main components.

rate of the two job types via the graphical interface, and therefore the measured job arrival rate and processing time is not used. The only measured system metric that is used is the queue size obtained from each pool. All data that is obtained either from system measurements or the user is inserted into a database for use by the switching policy.

In the prototype, a simple polling pattern is employed by the cluster manager to obtain data from each pool. The cluster manager makes a call twice every second to each pool manager which returns their current state, that consists of queue details. Independently of the cluster manager, each pool manager also uses the polling approach to query the RMS to extract pool data, which is saved and then retrieved when a call comes from the cluster manager. Saving RMS data, instead of making RMS queries upon a cluster manager call, avoids blocking the cluster manager should a fault occur during the RMS monitoring. When a polling cycle completes and each pool has returned its current state, the switching policy is invoked with the necessary data from the database and the heuristic is calculated. If the result determines that a server switch needs to take place, a switching protocol is invoked.

The pool manager is started by an administrator. Once running, the pool manager begins polling the RMS it uses for information about the queue, and is ready to receive commands from the pool manager. The cluster manager may then initialise the pool by informing the pool manager of the job type it is to process and the address of the cluster manager. The cluster manager then registers individual servers with the pool manager which adds each server to the list of its current members. The underlying RMS is also informed of the new servers. The main components of the pool manager can be seen in Fig 4.

The pool manager communicates with both the cluster manager and each server in its pool. Other RMS utilities used by the pool manager include functions that are specific to the reconfiguring of servers.

The server component runs on each server that is part of a pool, and its only function is to allow a server to switch between pools. The server communicates with the RMS to make use of utilities that are involved in the reconfiguration. The server is started by an administrator, but is only put into use after it has been initialised by the pool manager of the pool it is to join. Each server component has stored the necessary configuration needed to allow it to be reconfigured to join any of the two pools in the system.

The Condor RMS [10] has been used in each pool to provide queuing and scheduling facilities. Condor was chosen because of its flexibility in handling different pools of servers, and its ability for servers to switch from processing jobs in one pool to another quickly and without extensive reconfiguration. Because of its flexibility, all jobs running in the source and destination pool are not disturbed by a server switch and may continue being processed as a switch is taking place. A job currently running on a server must be interrupted and placed into the queue so that the server can join the new pool and accept a queued job as soon as possible. This interruption may mean the job starts again when it gets processing time, or it may be check-pointed so it

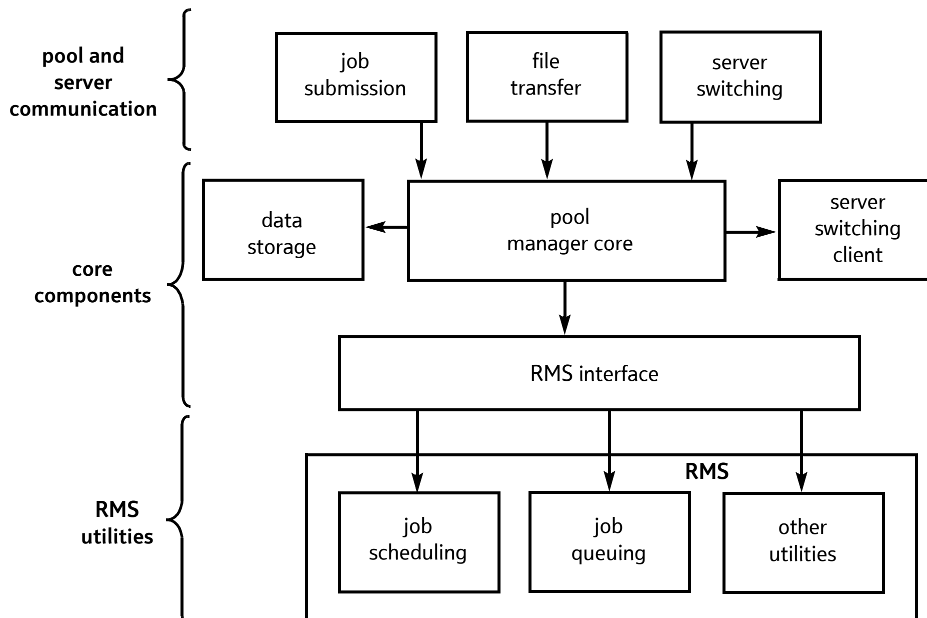

Fig 4 Pool manager components.

can start from where it left off at the time of interruption. Another reason for using Condor is that it is fairly straightforward to take a cluster of servers running the Condor daemons and partition it into multiple Condor pools by restarting some servers as pool managers.

Figure 5 shows the integration of Condor with the rest of the system, the area labelled 'Condor managed domain' represents components and communications internal to Condor which the prototype system employs. The pool manager and server components communicate with Condor through a proxy made up of various scripts that are called to complete an operation. A Condor pool has one server running as a central manager which queues and then matches job submissions to available servers. The actual dispatching of jobs from the pool manager to servers, including any file transfer is handled by Condor. The server component communicates with Condor only when a server is to be switched.

The integration of Condor into the system proved to be a point of failure during development. This was due to adverse effects of small configuration settings, the failure of pools to successfully release and accept servers, job submission failures, and using Condor by programmatically running scripts. Although most of these issues have been dealt with, fault-tolerance measures have been put into place to deal with any failures in a graceful manner.

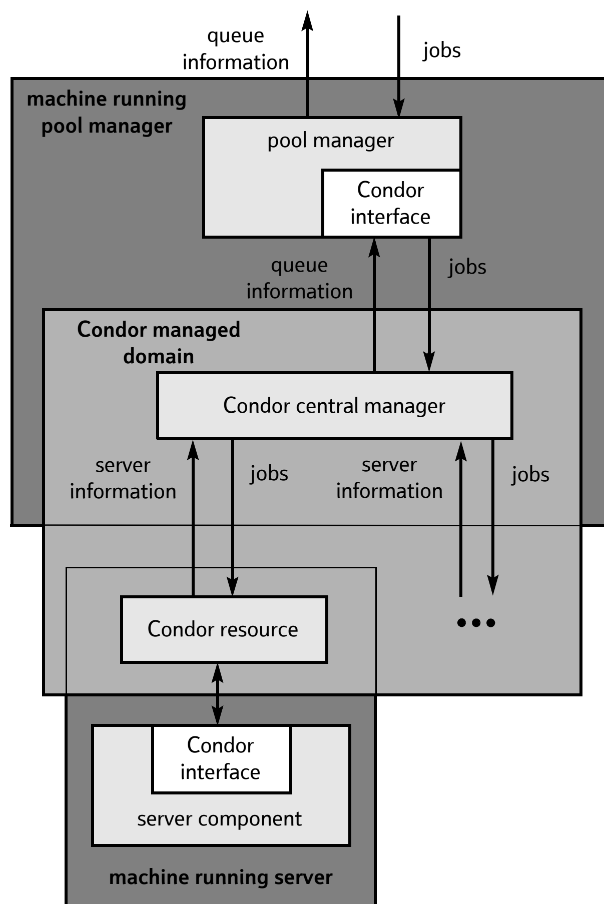

Fig 5 Integration of Condor with the pool manager and server components.

Although the Condor documentation does not discuss the switching of servers between pools, its architecture is flexible enough to allow it to do so. To switch a server into another pool the main Condor configuration file is re-written to instruct a server to point to a new pool manager. Each resource must contain multiple Condor configuration files, each of which maps on to a particular job type allowing a server to join a different pool. Once the decision has been made to switch a server, the cluster manager follows a protocol which aims to ensure that a server leaves its source pool and joins the destination pool in a consistent fashion. The cluster manager starts a co-ordinator which, during a switch, gives instructions to each pool manager to perform actions that will result in a server switch; the actions must happen in a specific sequential order to avoid an inconsistent switch and subsequent failure and removal of the server. Figure 6 shows the messages sent in the protocol.

In the prototype there must be at least one server in each pool which runs the pool manager component, as without the pool manager the pool does not exist. Once the pool manager has ascertained there is more than one server in the source pool, the switch protocol can begin.

The co-ordinator requests the source pool manager to find a server in its pool that can be reallocated to serve a different given job type, requiring that the job(s) currently running on the server are put back into the queue. The server chosen to leave the source pool is informed by its pool manager of the new job type it is to serve, and the server is re-configured to point to the new pool. At this point any other reconfiguration may take place, such as the installation of a new operating system or data set. Upon successful reconfiguration the details of the server are sent to the co-ordinator, indicating the source pool is prepared to complete the switch. The co-ordinator then sends the server information received from the source pool to the destination pool, preparing the destination pool manager to accept the resource into its pool. The destination pool then sends a response indicating it is ready to add the server to its Condor pool, and to its list of resources.

In the final stage of the protocol the co-ordinator issues a concurrent call to both pool managers. Each pool manager then makes the final reconfiguration necessary to complete the switch, resulting in the

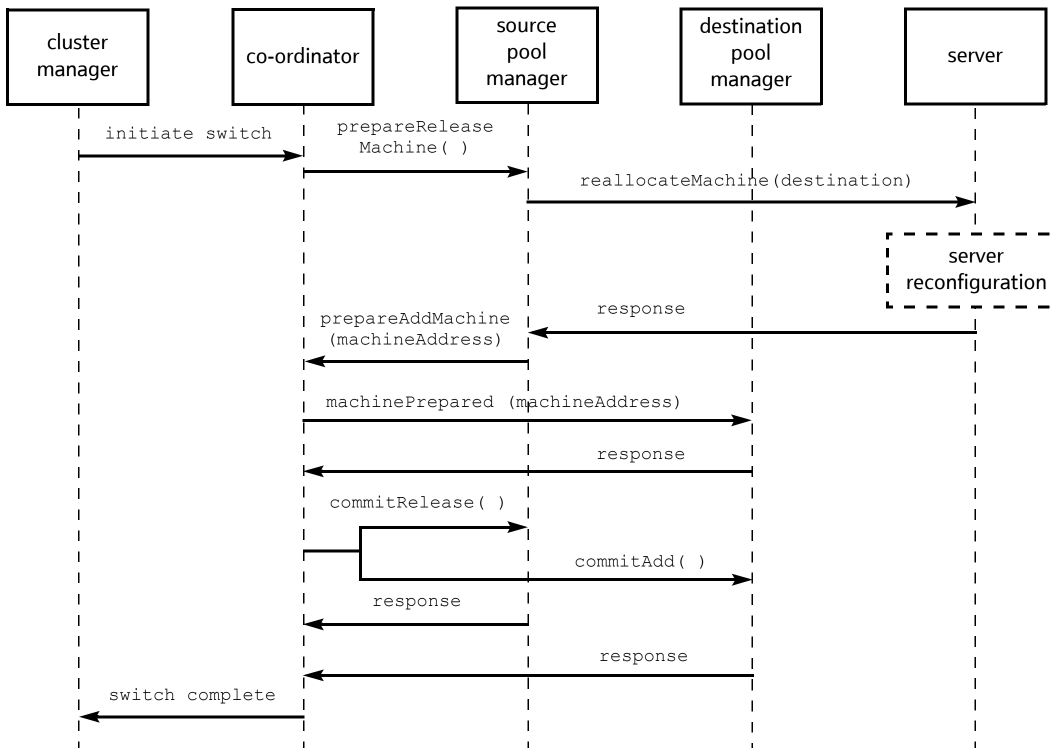

Fig 6 A sequence diagram representing the switching protocol.

reconfigured server becoming a part of the destination pool.

If there is a problem, such as the resource cannot configure itself to serve the new job type, the co-ordinator will cancel the switch and the system will continue as if no switch was attempted. Should a problem arise during the final phase, such as the destination pool manager does not accept the new server, the co-ordinator will roll back the switch so that the resource reverts back to its old configuration and joins the source pool again. If a more serious error occurs and the server becomes 'stranded', unable to join any pool, the server is removed from view and the appropriate manual action can be taken. Through the use of the switching protocol, servers may be reliably switched allowing the system to run continuously, despite faults.

A graphical user interface has been developed for the cluster manager to show a user the states of various parts of the system. The GUI shows each pool currently running, with a queue of submitted jobs and a number of servers in each pool. Various cluster manager plug-ins have been developed for testing and demonstration purposes, including a component which submits a stream of jobs at a configurable rate of each type to the cluster manager. The average length of time a job takes to complete can also be set, as can the time it takes to switch a server. To introduce randomness into the rate of job arrival and job length, an exponential formula is applied to the given values. The cost of holding a job of a given type may also be altered, the holding cost is a variable used in the heuristic and is discussed in section 4. The GUI uses green squares to represent servers actively processing jobs, and a red square to represent a server that is currently being reallocated to another pool (as shown in Fig 7). Two pools are active in the screenshot, and a resource is currently being reconfigured to switch to serve job type 2. Allowing the user to set values means changes in the system (such as a change in the arrival rate), that may otherwise take minutes or hours to occur if the system were taking job arrival and processing time measurements, can be made instantly and the results observed.

4. Design and justification of heuristic

The prototype system uses a heuristic policy [11] which calculates a close to optimal allocation of servers to pools in a two-pool system with M servers. A reallocation decision is made using the average arrival rate and average job processing time of each job type and the time taken to switch a server between pools. Using the average values means past measured values may be taken into account, and so the heuristic produces a more meaningful result. The heuristic takes into account the cost of holding a job which may be measured in monetary terms, but serves to represent relative importance between the two job types. For example, if all measurements are equal but the holding

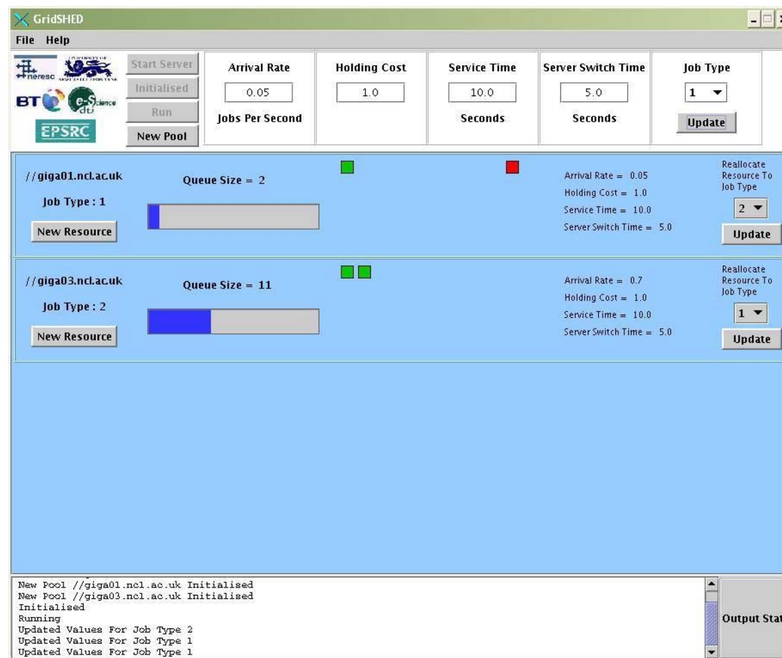

Fig 7 The GridSHED prototype GUI.

cost of job type 1 is greater than a type 2 job, more servers will be allocated to serve type 1 jobs than jobs of type 2. The process of switching a resource also has an associated cost as the server is unable to process jobs while in a state of reconfiguration. A method of calculating the optimal server allocation for the system has been found, but its computation is complex. Therefore a heuristic policy is used which is practical and easy to implement, and has been shown to be almost as good as the optimal policy in simulations.

5. Hierarchical organisation of clusters

Because a resource-hosting environment may have thousands of servers readily available to process a large, varied and unpredictable stream of jobs, an infrastructure must be in place to cope with such a demand effectively. The central manager component of an RMS must receive continuous updates from each server and must also perform job-to-server matches. In an RMS which administers thousands of servers there is the potential for a bottle-neck to form at the central manager due to a high network volume of updates from the servers, and due to the expensive task of server-to-job matching when a large number of servers are present. The pool-portioning approach described in this paper may solve the problem to some degree by distributing job-to-server matching among the pool managers in a cluster, but as the cluster manager is a centralised component it too has the potential to become a bottle-neck.

To avoid performance issues caused by a centralised approach, numerous clusters may be used to share the work. Each cluster presents an interface for jobs to be submitted, and for the cluster to be queried for information such as the current load and current number of servers in the cluster. The interface allows a number of clusters operating in the same resource management environment to interact with each other. An approach to the organisation of clusters is one of a tree structure, which allows the querying of child nodes and the flowing of jobs down the nodes until the optimal cluster is found to run the job. The data received from a cluster query may be formulated to obtain the 'cost' of running a job on a cluster, the cost could also take into account potential server-switching options. Providing a generic interface means a cluster of servers does not have to implement server switching, and therefore computational resources from other domains could dynamically become a part of the environment if the system cannot cope with an unanticipated large arrival rate. As a tree can be made up of many levels, the process of cost querying may travel down a tree to the leaf nodes; a cost value presented by a tree node therefore takes into account the cost of running the job on a child node.

Two diagrams showing the layout of the tree structure are shown in Fig 8. The diagram illustrates how the tree is organised, with job requests sent down the tree and responses back up in Fig 8(a), and how the job is sent to the optimal node in Fig 8(b).

It is not necessary for each node of the tree to contain a cluster of servers that run jobs — a node may be in place to route jobs to the most optimal cluster

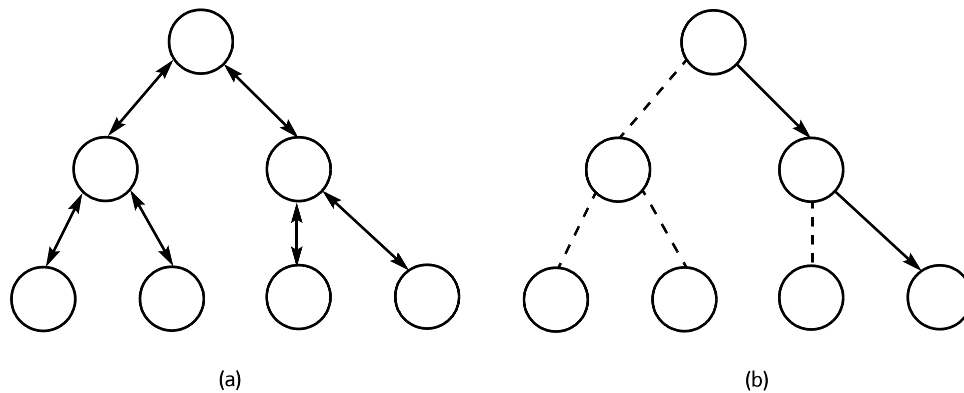

Fig 8 Tree structure querying and job propagation.

while only the tree leaf nodes run jobs. A tree may have a flat structure with only one root node with leaf nodes directly below, minimising the cost of queries going up and down the tree but increasing the load on the tree node. If the tree structure is flexible, nodes may be added or relocated within the tree as necessary to fulfil a requirement such as a high demand for a certain job type.

The hierarchical organisation of a cluster is an option for future work.

6. Conclusions and further work

This paper presents a method of reconfiguring servers in a resource-limited hosting environment in a close to optimal fashion to cope with varied and unpredictable demand. By allocating servers in clusters to conceptual pools and applying a heuristic policy to past and present system information, decisions are made to reallocate servers to different pools to cope with demand. A system architecture has been designed to collect data and use switching policies to act upon the data collected to make switching decisions.

A demonstration system has been developed to show how a close-to-optimal allocation policy can be applied to past and real-time data obtained from various parts of the system to make reallocation decisions; the system then performs the decisions on servers between pools in a cluster to achieve an efficient allocation of servers and a higher throughput of jobs.

Future work will include more sophisticated means of monitoring and gathering data from resource pools to produce more details that may be fed into switching policies. The data could include the average amount of time for which particular resources are idle and how often resources are switched. With detailed information, a system for providing provisions to enable quality of service guarantees may be developed.

References

1 Foster I, Kesselman C, Tuecke S and Nick J: ‘The physiology of the grid: an open grid service architecture for distributed systems integration’, Globus Project Report (2002).

2 The Job Submission Description Language Working Group — http://www.epcc.ed.ac.uk/~ali/WORK/GGF/JSDL-WG/

3 Czajkowski K, Foster I, Karonis N, Kesselman C, Martin S, Smith W and Tuecke S: ‘A resource management architecture for meta-computing systems’, Proc IPPS/SPDP ’98 Workshop on Job Scheduling Strategies for Parallel Processing, pp 62—82 (1998).

4 Czajkowski K, Foster I, Kesselman C, Sander V and Tuecke S: ‘SNAP: a protocol for negotiating service level agreements and coordinating resource management in distributed systems’, 8th Workshop on Job Scheduling Strategies for Parallel Processing, (July 2002).

5 Livny M and Raman R: ‘High throughput resource management’, in Kesselman C and Foster I (Eds): ‘The Grid: Blueprint for a New Computing Infrastructure’, pp 311—339, Morgan Kaufmann (July 1998).

6 IBM on-demand business — http://www.ibm.com/e-business

7 Sychron — http://www.sychron.com/

8 RedHat Linux KickStart — http://wwwcache.ja.net/dev/kickstart/

9 Chase J S et al: ‘Dynamic virtual clusters in a grid site manager’, 12th IEEE International Symposium on High Performance Distributed Computing (2003).

10 Condor Resource Management System — http://www.cs.wisc.edu/condor/

11 Mitrani I and Palmer J: ‘Dynamic server allocation in heterogeneous clusters’, The First International Working Conference on Heterogeneous Networks, Ilkley, UK (July 2003).

Charles Kubicek received an MSc degree in System Design for Internet Applications from the University of Newcastle upon Tyne in 2002.

He now works as a Research Associate at the North East Regional E-Science Centre at Newcastle on the collaborative GridSHED project.

His main interests include the design and development of distributed systems software, Grid and Web Services technologies.

Paul McKee is a team leader in the Distributed Computing and Information Systems research group at BT's Adastral Park. He currently manages projects including collaboration with a number of universities. His research is focused on large-scale distributed systems, particularly policy-based management and high-performance event-based architectures for capturing and processing management information. He joined BT in 1989 and initially worked on high-resolution optical devices before moving to a distributed systems group where he worked on autonomous replication and low overhead consistency protocols. He has published over 35 papers and is a member of the IEEE Computer Society.

Mike Fisher leads the Distributed Computing Research group in BT Exact at Adastral Park.

He joined BT in 1988 after completing a PhD in Physics at the University of Surrey and initially worked in long-term research on semiconductor optical materials and devices.

His current research interests include the management of complex systems, particularly active networks and services.

Rob Smith is Technical Director of the North East Regional e-Science Centre. He is a researcher on a number of e-Science projects, including the EPSCR e-Science Pilot Project GOLD. Before joining the e_Science Centre, he was a software engineer, then consulting contractor, in the IT and telecommunications industry, counting among his successes the roll-out of NTL's narrow- and broad-band ISP offerings in the UK and Ireland, the redesign and deployment of Orange's multimedia and text messaging services infrastructure, and the development of Ericsson's 3G services for the Japanese market.