EPCC-SS-2000-11

Investigating Globus and Grid Technologies

Golestan Radwan

Abstract

A new class of advanced network-based applications is emerging, distinguished from today's Web browsers and other standard Internet applications by a coordinated use of not only networks but also endsystem computers, data archives, various sensors, and advanced human computer interfaces. These applications require services not provided by today's Internet and Web: they need a "Grid" that both integrates new types of resource into the network fabric and provides enhanced "middleware" services such as quality of service within the fabric itself. Clearly, design and operation issues associated with implementing the Grid will differ in many ways from those involved in the design of conventional networks.
This project investigates various Grid-related technologies and packages, evaluates them, and suggests ways of improving their current functionality.

Table of contents

Introduction

The scientific problem-solving infrastructure of the 21st century will be a heterogeneous complex of advanced networks, computers, storage devices, display devices, and scientific instruments that collectively we term the "Grid". The applications that use this infrastructure will range from tomorrow's equivalent of today's "secure shell" and Web browsers to more sophisticated collaborative tele-immersive engineering, distributed Petabyte data analysis, and real-time instrument control systems. These applications will share a common need to couple devices that have not traditionally been thought of as part of the network. This need will motivate the development of a broad set of new services beyond those provided by today's Internet. This "middleware" will provide the security, resource management, data access, instrumentation, policy, accounting, and other services required for applications, users, and resource providers to operate effectively in a Grid environment. Globus is a multi-institutional research and development project focussed on enabling the applications of grid concepts to scientific and engineering computing. It was designed to provide this middleware to the Grid. In this respect, it would be of interest to EPCC to explore Globus, identify its limitations, and try to work out ways of over coming them.
This project is organised around 4 main areas:
Research - which targets challenges in areas such as communication, scheduling, security, information, data access and fault detection.
Tools - focussed on the development of robust prototype software to run on a wide range of interesting and important platforms. This "bag of services" known as the Globus Toolkit forms a central element of the Globus system and defines the basic services and capabilities required to construct a computational grid.
Testbeds - which involves the construction of large-scale, prototype computational grids or testbeds using the basic technology and tools developed.
Applications - focussed on developing large-scale grid-enabled applications in collaboration with application scientists.
EPCC have carried out initial testing with the Globus Toolkit on a number of internal resources including a number of the Ultra-5s, the Beowulf cluster (BOBCAT) and the Cray T3E. Although these tests have had limited success, further research and investigation is required to fully understand and appreciate the "bag of services" on offer from the Globus Toolkit.
The project should achieve two goals:
Investigation of a metacomputing application that makes use of Globus. A good example would be Cactus, as this would be considered worthwhile and valuable given EPCC's involvement within EGRID activities.
Research into and possibly implementation of a layer on top of Globus that allows Globus to run on arbitrary machines. Making choices about which resources to use is a task that the "broker" is responsible for in the Globus architecture. However, no broker is included in the Globus Toolkit. Brokers do exist though, apparently in Nimrod-G and Condor-G, where they have constructed a layer on top of Globus to perform this functionality. The idea would be to investigate and report on these applications and then possibly implement a layer pending the outcome of the research.
The rest of this document is organised as follows: Section 2 presents an overview of Grid technologies and their related design issues. Section 3 presents Globus, a metacomputing infrastructure toolkit. Sections 4 and 5 take a look at two Grid-enabled packages, Nimrod and condor. Section 6 introduces Cactus, a problem-solving environment. Finally, some suggestions are made for future work on Globus and the Grid.

The Grid

The concept of the Grid involves interconnecting supercomputers and resources to form what is called a “Metacomputer”. Specifically, we use the term metacomputer to denote a networked virtual supercomputer, constructed dynamically from geographically distributed resources linked by high-speed networks. Scientists and engineers are just beginning to explore the new applications enabled by networked supercomputing. Among other things, possible metacomputing applications include:

Desktop supercomputing. These applications couple high-end graphics capabilities with remote supercomputers and/or databases. This coupling connects users more tightly with computing capabilities, while at the same time achieving distance independence between resources, developers, and users.

Smart instruments. These applications connect users to instruments such as microscopes, telescopes, or satellite downlinks that are themselves Smart instruments. These applications connect users to instruments such as microscopes, telescopes, or satellite downlinks that are themselves coupled with remote supercomputers. This computational enhancement can enable both quasi-real-time processing of instrument output and interactive steering.

Collaborative environments. A third set of applications couple multiple virtual environments so that users at different locations can interact with each other and with supercomputer simulations.

Distributed supercomputing. These applications couple multiple computers to tackle problems that are too large for a single computer or that can benefit from executing different problem components on different computer architectures. We can distinguish scheduled and unscheduled modes of operation. In scheduled mode, resources, once acquired, are dedicated to an application. In unscheduled mode, applications use otherwise idle resources that may be reclaimed if needed; Condor is one system that supports this mode of operation. In general, scheduled mode is required for tightly coupled simulations, particularly those with time constraints, while unscheduled mode is appropriate for loosely coupled applications that can adapt to time-varying resources.

Understandably, there are a number of issues arising from trying to fulfil the needs of each of these applications. Metacomputers have much in common with both distributed and parallel systems, yet also differ from these two architectures in important ways. Like a distributed system, a networked supercomputer must integrate resources of widely varying capabilities, connected by potentially unreliable networks and often located in different administrative domains. However, the need for high performance can require programming models and interfaces radically di_erent from those used in distributed systems. As in parallel computing, metacomputing applications often need to schedule communications carefully to meet performance requirements. However, the heterogeneous and dynamic nature of metacomputing systems limits the applicability of current parallel computing tools and techniques.

These considerations suggest that while metacomputing can build on distributed and parallel software technologies, it also requires significant advances in mechanisms, techniques, and tools. General observations about desired metacomputing systems characteristics include:

- *Scale and the need for selection.* To date, most metacomputing experiments have been performed on relatively small testbeds. In the future we can expect to deal with much larger collections, from which resources will be selected for particular applications according to criteria such as connectivity, cost, security, and reliability.
- *Heterogeneity at multiple levels.* Both the computing resources used to construct virtual supercomputers and the networks that connect these resources are often highly heterogeneous. Heterogeneity can arise at multiple levels, ranging from physical devices, through system software, to scheduling and usage policies.

- *Unpredictable structure.* Traditionally, high-performance applications have been developed for a single class of system with well-known characteristics, or even for one particular computer. In contrast, metacomputing applications can be required to execute in a wide range of environments, constructed dynamically from available resources. Geographical distribution and complexity are other factors that make it difficult to determine system characteristics such as network bandwidth and latency a priori.
- *Dynamic and unpredictable behaviour.* Traditional high-performance systems use scheduling disciplines such as space sharing or gang scheduling to provide exclusive and hence predictable access to processors and networks. In metacomputing environments, resources especially networks are more likely to be shared. One consequence of sharing is that behaviour and performance can vary over time. For example, in wide area networks built using the Internet Protocol suite, network characteristics such as latency, bandwidth and jitter may vary as traffic is re-routed. Large-scale metasystems may also suffer from network and resource failures. In general, it is not possible to guarantee even minimum quality of service requirements.
- *Multiple administrative domains.* The resources used by metacomputing applications often are not owned or administered by a single entity. The need to deal with multiple administrative entities complicates the already challenging network security problem, as different entities may use different authentication mechanisms, authorisation schemes, and access policies. The need to execute user-supplied code at different sites introduces additional concerns.

Fundamental to all of these issues is the need for mechanisms that allow applications to obtain real-time information about system structure and state, use that information to make configuration decisions, and be notified when information changes. Required information can include network activity, available network interfaces, processor characteristics, and authentication mechanisms. Decision processes can require complex combinations of these data in order to achieve efficient end-to-end configuration of complex networked systems.

Several attempts have been made to build a robust metacomputing infrastructure toolkit that would facilitate Grid interaction. One of them, Globus is discussed in the next section.

Globus: A metacomputing infrastructure toolkit

The goal of the Globus project is to provide basic infrastructure that can be used to construct portable, high-performance implementations of a range of such services. To this end, it is focused on (a) the development of low-level mechanisms that can be used to implement higher-level services, and (b) techniques that allow those services to observe and guide the operation of these mechanisms. If successful, this approach can reduce the complexity and improve the quality of metacomputing software by allowing a single low-level infrastructure to be used for many purposes, and by providing solutions to the configuration problem in metacomputing systems.

The Globus toolkit comprises a set of modules. Each module defines an interface, which higher-level services use to invoke that module's mechanisms, and provides an implementation, which uses appropriate low-level operations to implement these mechanisms in different environments.

Currently identified toolkit modules are as follows. These descriptions focus on requirements.

- *Resource location and allocation.* This component provides mechanisms for expressing application resource requirements, for identifying resources that meet these requirements, and for scheduling resources once they have been located. Resource location mechanisms are required because applications cannot, in general, be expected to know the exact location of required resources, particularly when load and resource availability can vary. Resource allocation involves scheduling the resource and performing any initialisation required for subsequent process creation, data access, etc. In some situations for example, on some supercomputers location and allocation must be performed in a single step.

- *Communications*. This component provides basic communication mechanisms. These mechanisms must permit the efficient implementation of a wide range of communication methods, including message passing, remote procedure call, distributed shared memory, stream-based, and multicast. Mechanisms must be cognisant of network quality of service parameters such as jitter, reliability, latency, and bandwidth.
- *Unified resource information service*. This component provides a uniform mechanism for obtaining real-time information about metasystem structure and status. The mechanism must allow components to post as well as receive information. Support for scoping and access control is also required.
- *Authentication interface*. This component provides basic authentication mechanisms that can be used to validate the identity of both users and resources. These mechanisms provide building blocks for other security services such as authorisation and data security that need to know the identity of parties engaged in an operation.
- *Process creation*. This component is used to initiate computation on a resource once it has been located and allocated. This task includes setting up executables, creating an execution environment, starting executable, passing arguments, integrating the new process into the rest of the computation, and managing termination and process shutdown.
- *Data access*. This component is responsible for providing high-speed remote access to persistent storage such as files. Some data resources such as databases may be accessed via distributed database technology or the Common Object Request Broker Architecture (CORBA). The Globus data access module addresses the problem of achieving high performance when accessing parallel file systems and network-enabled I/O devices such as the High Performance Storage System (HPSS).

Together, the various Globus toolkit modules can be thought of as defining a metacomputing virtual machine. The definition of this virtual machine simplifies application development and enhances portability by allowing programmers to think of geographically distributed, heterogeneous collections of resources as unified entities.

A lot can be said about Globus and its implementation issues. However, this is beyond the scope of this project. For more information, please refer to the list of references about Globus. So far, Globus has one severe limitation: resource selection is not transparent to the user, i.e. the user has to not only specify their job but also specify which machine to run that job on. This, in many people's opinions, kills the concept behind having the Grid in the first place. They don't want to have to worry about where their job is being run. They want the Grid to determine the most suitable set of resources, allocate them, run the job on them, and return the results. Specifically, EPCC has got two large computing facilities, the Cray T3E and the Sun-HPC cluster. Jobs should be submitted to the Grid interconnecting the two, and the Grid infrastructure should determine which one is more suitable for that job.

One way to solve this problem would be to implement a "broker" layer on top of Globus, which will handle resource selection and management. To this end, two job management packages, Nimrod and Condor have been investigated, to determine whether either of them could provide the desired functionality. Both are discussed in turn in the following two sections.

Nimrod: A tool for performing parameterised simulations

Nimrod is a tool for performing parameterised simulations over networks of loosely couples workstations. After the user specifies the simulation parameters, Nimrod takes the cross product of these parameters and generates a job for each set. It then manages the distribution of the various jobs to machines, and organises the aggregation of results. By implementing log facilities Nimrod can be restarted at any time throughout an experiment. It is also possible to run multiple copies of Nimrod concurrently, allowing simultaneous execution of experiments with different parameter settings, or even different modelling applications. Rather than assuming a shared file system, Nimrod copies files between systems before and after execution of the program.

Nimrod Architecture

There are two separate servers, one providing file transfer between the client host and the server host, and the other allowing remote execution of the processes on the server host. Control of a run begins on the client side at the Job Organisation Tool (JOT). This module provides the user interface, allowing the user to create, monitor, and direct jobs running on many hosts. The Job Distribution Manager (JDM) queues the jobs that are to be run. The JDM interacts with the two servers on each remote host. The client communicates with the server through the use of user agents. User agents are client side modules working on behalf of remote servers. When a job is started on a remote host, any required files are transferred to the host. The remote execution server is used to start a shell on the server host and this shell subsequently starts the executable program. On completion, output files are transferred from the remote host to the originating host.

Nimrod File Transfer

Before a job can be executed, the remote system must hold a copy of the executable image and any input files. In a parameterised run, some of these files may be different for each job, while other files may be the same for all jobs. Nimrod can generate specific data files for each job based in the selection of parameters for that job. Nimrod uses a file transfer server and requires explicit copying of the executable and all data files to the remote host. This has general applicability since any host running a file transfer server can accept jobs. In addition, if necessary, and NFS mount can be used for efficiency reasons. The file transfer service consists of three modules. The File Transfer Manager contains the server side functions that perform server activity related to file transfer. The File Transfer Server performs server initialisation and listens for requests. The File Transfer User Agent provides a high-level interface to the functions of the File Transfer Server. The Remote Execution Server is responsible for starting a process on a remote host, and is partitioned using the same principles as the FT server. Processes can be started with arbitrary command arguments and execution environments.

Terminal I/O can be redirected t a file, obtained through further RPCs, or ignored completely. Processes can be queried for status by clients such as termination and suspension.

In order to control the load on any given machine, a server only accepts jobs until various thresholds are exceeded.

The remote execution server provides some basic information about the status of the job being executed. A remote shell is used to provide more detailed status information to be obtained than that provided by the RX server. When a new job is started the RX server is used to run a remote shell, which is controlled using calls to the RX server's I/O RPCs. The shell can be used to start the job and wait for termination of the job. In addition, a customised shell can monitor the intermediate progress of the job. A simple way of doing this is to monitor the size of a particular output file.

The job distribution manager performs a similar task to that provided by existing systems such as Condor and LSF. Given a set of jobs, it runs them on under-utilised hosts on the network, and provides basic status monitoring. The task of distributing the jobs can be broken down into a number of tasks. For each job, the subtasks to be performed are:

1. Find a suitable remote host
2. Perform any setup necessary to allow the job to run on the host (typically file transfer)
3. Start the job running on the remote host
4. Wait for job termination
5. Perform any cleanup necessary

The task of finding suitable hosts is accomplished by connecting to a trading service and specifying the requirements for a suitable host. Typical requirements are for a host with a particular architecture with a low current CPU load. The trading service then returns the names of hosts. This means that it is not necessary to know which hosts are present before Nimrod is started.

Hosts may become available and unavailable during the simulation. The JDM uses Tcl as a scripting language.

The Job Organisation tool controls generation of a run. It provides a user interface, which is related to the problem being solved, rather than to computational issues. The user selects the domains over which parameters of the problem will vary, and the system generates one job for each combination of parameters. Once the jobs are generated and the run is started, the JOT allows monitoring of the progress of the run.

The JOT allows the run to be directed by the user. This includes changing the priorities of jobs, and terminating and restarting jobs. The JOT is also responsible for selection of parameter domains and controlling scripts, both of which are dependent on the problem being solved. A customised JOT is generated for controlling a particular system.

Limitations of Nimrod and the migration to Nimrod/G

Nimrod contains no mechanisms for scheduling the computation on the underlying resources. Consequently, users would not have any idea when an experiment might complete. Hence, Nimrod/G, a grid-enabled version of Nimrod, was introduced. Using Globus, it is possible for Nimrod users to specify time and cost constraints on computational experiments. Globus provides mechanisms for estimating execution time and waiting delays when using networked queued supercomputers. Nimrod/G will then use these to schedule the work in a way which meets user specified deadlines and cost budgets. In this way, multiple Nimrod users can obtain a quality-of-service from the computational network. In order to understand how Nimrod/G works, it is first necessary to be familiar with the concept of the “Grid Resource Broker”, which is introduced in the following section.

The Grid Resource Broker

The Grid Resource Broker (GRB) is responsible for resource selection, binding of software (application), data, and hardware resources, initiating computations, adapting to the changes in grid resources and presenting the grid to the user as a single, unified resource.

Resource broker component include:

1. *Job Control Agent (JCA).* This component is a persistent component responsible for shepherding a job through the system. It takes care schedule generation, the actual creation of jobs, maintenance of job status, interacting with clients/users, schedule advisor and dispatcher.
2. *Schedule advisor*. This component is responsible for resource discovery (using grid explorer), resource selection, and job assignment. Its key function is to select those resources that meet user requirements such as meet the deadline and minimise the cost of computation while assigning jobs to resources.

3. *Grid Explorer.* This is responsible for resource discovery by interacting with grid-information server and identifying the list of authorised machines, and keeping track of resource status information.
4. *Trade Manager.* This works under the direction of resource selection algorithm (schedule advisor) to identify resource access costs. It interacts with trade servers and negotiates for access to resources at low costs.
5. *Deployment Agent.* This is responsible for activating task execution on the selected resources as per the scheduler's instruction. It periodically updates the status of task execution to JCA.

The Nimrod/G Resource Broker

The Nimrod/G resource broker is a tailored global scheduler for running parametric applications on computational grid. It is developed using Globus toolkit services and can be easily extended to operate with any other emerging grid middleware services. It uses MDS services for dynamic resource discovery and GRAM APIs to dispatch jobs over wide-area distributed grid resources. It allows scientists and engineers to model whole parametric experiments and transparently stage the data (using GASS) and program (using GEM) at remote sites, and run the program on each element of a data set on different machines and finally gather results from remote sites to the user site. The user need not worry about the way in which the complete experiment is set up, data or executable staging, or management. The user can also set the deadline by which the results are needed and the Nimrod/G broker tries to find the cheapest computational resource available in the grid and use them so that the user deadline is met and cost of computation is kept to a minimum. However, the grid resources are shared and their availability and load varies from time to time. When scheduler notices that it cannot meet the deadline with the current resource set, it tries to select the next cheapest resource and continues to do this until the completion of task farm application meets the (soft) deadline. The Nimrod/G uses static cost model (stored in a file) for resource access cost trade-off with the deadline.

Nimrod/G system architecture

On the local site, the origin process operates as the master of the whole system. The origin process exists for the entire length of the experiment, and is ultimately responsible for the execution of the experiment within the time and cost constraints. The user is either a person or process responsible for creation of the experiment and interacts with the origin process through a client process. Each remote site consists of a cluster of computational nodes. Before submitting any jobs to a cluster, the origin process uses the Globus process creation service to start a Nimrod resource broker on the cluster. The NRB is a different entity to the resource manager. It provides capabilities for file staging, creation of jobs from the generic experiment, and process control beyond that provided by the Globus Resource Manager.

The client process performs several tasks. First, the client assists the user in setting reasonable constraints. By using the Globus Metacomputing Directory Service, the client determines a probability of particular time and cost constraints being achieved for the user's experiment. The user may then decide to modify the constraints to achieve a particular outcome, trading a later deadline for a lower cost, or a higher cost for more chance of completion within the deadline. Second, once the user is satisfied with a particular combination of constraints, the client sends the experiment to the origin process along with the deadline and cost constraints. In effect, this begins the execution of the experiment. The final task of the client is to provide feedback to the user about the progress of the experiment. This information is obtained from the origin process, which as the master of the whole experiment, must keep status on the progress of the experiment.

For each cluster, the origin process starts a local handler thread. The first task of this thread is to start a Nimrod Resource Broker on its cluster through the Globus process creation service (part of the Nexus module). It then interrogates the cluster for an up-to-date estimate of its computational

capacity. This estimate is determined heuristically from information provided by the Globus resource manager and any other NRB's executing on the cluster. Once a thread has received the estimate from its cluster, it removes a number of jobs from the common pool. The number of jobs is the number required to fill the cluster up to the deadline.
The Nimrod Resource Broker ensures that all jobs it handles complete by their deadline. It monitors the probability of successful completion of all jobs it is locally responsible for using information supplied by the Globus Resource Manager. A job submitted to NRB is not necessarily submitted immediately to the local resource manager. If the probability of successful completion of a job on a particular cluster falls below a threshold, the origin removes jobs from the NRB until the probability of successful completion of the remaining jobs is above the threshold. The origin process returns jobs to the common pool, where they are redistributed, either to clusters performing ahead of schedule, or to new clusters.

Nimrod/G and Globus

Services offered to Nimrod by Globus:

1. Resource allocation and process management (GRAM).
2. Unicast and multicast communications services (Nexus)
3. Authentication and related security services (GSI)
4. Distributed access to structure and state information (MDS)
5. Monitoring of health and status of system components (HBM)
6. Remote access to data via sequential and parallel interfaces (GASS)
7. Construction, caching and location of executables (GEM)
8. Advanced resource reservation (GARA)

The resource trading services are offered by the middleware infrastructure, GRACE. The local resource manager is responsible for managing and scheduling computations across local resources such as workstations and clusters. They are even responsible for offering access to storage device, databases, and special scientific instruments such as a radio telescope.
GRACE can co-exist with Globus. It offers services that help resource brokers in dynamically trading (cheap) resources to support computational economy. The components of GRACE infrastructure are:

1. A trade manager (it is actually a GRACE client and a component of the resource broker).
2. It is a client that uses GRACE trading APIs to interact with trade servers and negotiate for access to resources at low cost. It works under the direction of resource selection algorithm (schedule advisor) to identify resource access costs.
3. Trading protocols and APIs.
4. The protocols define the rules and format for exchanging commands and messages between GRACE client (trade Manager) and Trade Server.
5. A trade server (it uses pricing algorithms defined by the resource owner and interacts with Resource Usage Accounting and Billing Systems). It is a resource owner agent that negotiates with resource users and sells access to resources. It aims to maximise the resource utility and profit its owner (earn as much money as possible). It uses pricing algorithms as defined by the resource owner that may be driven by the demand and supply. It also interacts with the accounting system for recording resource usage that bills the user.

The trade manager contacts trade server with a request for quote/bid. The TM specifies resource requirements in Deal Template (DT), which can be represented by a simple structure (record) with its fields corresponding to deal items or by a "Deal (Template) Specification Language" similar to the ClassAds mechanism employed by Condor. Contents of DT include CPU time units, expected usage duration, storage requirements, etc., along with its initial offer or leave it

blank. The Tm looks into DT and updates its contents with price etc, and sends back to TS. This negotiation continues until one of them says that its offer is final. Then it is up to the other party to decide to whether to accept or reject the deal.
APIs can be used by grid tools and application programmers to develop software supporting the computational economy. The trading APIs are C-like functions that GRACE clients can use to communicate with trading agents:

```
grid_trade_connect(resource_id, tid)
grid_request_quote(tid, DT)
grid_trade_negotiate(tid, DT)
grid_trade_confirm(tid, DT)
grid_trade_cancel(tid, DT)
grid_trade_change(tid, DT)
grid_trade_reconnect(tid, resource_id)
grid_trade_disconnect(tid)
```

tid = trade identification code
DT = deal template

New suggested architecture for Nimrod/G

The key components of Nimrod/G system are Client/User Station, a Persistent Parametric/Taskfarming Nimrod Engine, Scheduler, and Dispatcher. One of the key components of proposed Nimrod/G architecture is trading manager. The TM will also explore the advance resource reservation during trading. Developers are hoping that the new Nimrod/G will be able to answer questions like: "I am willing to pay $$$, can you complete this job by deadline D?" This ability means, users can trade-off the deadline against the cost and decide the manner in which computations are to be performed.

A conclusion about Nimrod

Obviously, Nimrod is not very suited to the type of application required at EPCC. It is definitely a good simulation tool and can be used for many of the simulation work at EPCC. However, to act as a Grid resource broker, the application must be more generic than just a simulation tool. In addition, it should allow the users to create their own jobs and specify their own parameter combinations.
Therefore, Nimrod cannot be used as a broker element on top of Globus. The next section discusses Condor.

Condor: A hunter for idle workstations

What is Condor?

Condor is a software system that creates a High Throughput Computing (HTC) environment by effectively harnessing the power of a cluster of UNIX workstations on a network. Instead of running a CPU-intensive job in the background on their own workstation, users submit their job to Condor. Condor will then find an available machine on the network and begin running the job on that machine. When Condor detects that a machine running a Condor job would no longer be available, Condor checkpoints the job and then migrates it over the network to a different machine which would otherwise be idle. If no machine on the network is currently available, then the job is stored in a queue on disk until a machine becomes available.

Matchmaking in Condor

To handle job allocation to idle machines, Condor uses the ClassAds paradigm. ClassAds work in a fashion similar to the newspaper classified advertising want ads. All machines in the Condor pool advertise their resource properties, such as available RAM memory, CPU type and speed, virtual memory size, physical location, current load average, and many other static and dynamic properties, into a resource offer ad. Likewise, when submitting a job, users can specify a resource request ad, which defines both the required and desired set of resources to run the job. Similarly, a resource offer ad can define requirements and preferences. Condor then acts as a broker by matching and ranking resource offer ads with resource request ads, making certain that all requirements in both ads are satisfied. During this match-making process, Condor also takes several layers of priority values into consideration: the priority the user assigned to the resource request ad, the priority of the user which submitted the ad, and desire of machines in the pool to accept certain types of ads over others.
Matchmaking uses a semi-structured data model-the classified advertisements data model- to represent the principles of the system and folds the query language into the data model, allowing entities to publish queries (i.e., requirements) as attributes. The paradigm also distinguishes between matching and claiming as two distinct operations in resource management: A match is an introduction between two compatible entities, whereas a claim is the establishment of a working relationship between the entities.

Matchmaking framework components:

1. The ClassAd specification, which defines a language for expressing characteristics and constraints, and a semantics of evaluating these attributes.
2. The advertising protocol, which defines basic conventions regarding what a matchmaker expects to find in a classAd if the ad is to be included in the matchmaking process, and how the matchmaker expects to receive the ad from the advertiser.
3. The matchmaking algorithm, which defines how the contents of ads and the state of system relate to the outcome of the matchmaking process.
4. The matchmaking protocol, which defines how matched entities are notified and what information they are given in case of a match.
5. The claiming protocol, which defines what actions the matched entities take to enable discharge of service.

Differences b/w matchmaking and conventional resource allocation models:

1. Conventional resource management systems only allow customers to impose constraints on the type of services they require. Our mechanism also allows service providers to express constraints on the customers they are willing to serve.
2. The semantics of a matchmaker identifying a match between entities A and B is not "allocating A to B". Rather, a match results in a mutual introduction of the two entities, who may activate a separate claiming protocol, not involving the matchmaker, to complete the allocation. Either entity may choose to not proceed further and reject the introduction altogether. Thus, a match is to be construed as a hint. A beneficial consequence of this approach is that the matchmaker is a stateless service, which simplifies recovery in case of failure.

Novel aspects of ClassAds:

1. ClassAds use a semi-structured data model, so no specific schema is required is required by the matchmaker to work naturally in a heterogeneous environment.

2. The classAd language folds the query language into the data model. Constraints may be expressed as attributes of the classAd.
3. ClassAds are first-class objects in the model. They can be arbitrarily nested, leading to a natural language for expressing resource aggregates or co-allocation requests. A classAd is a mapping from attribute names to expressions. Attributes may be simple integer, real, or string constants, or they may be more complicated expressions constructed with arithmetic and logical operators and record and list constructors.

Matchmaking and claiming:

Providers and customers construct ClassAds describing themselves and send them to the Matchmaker. These ClassAds must be constructed to conform to the advertising protocol specified by the matchmaker, which attaches a meaning to some attributes. The advertising protocol also specifies how the entities send the ClassAds to the matchmaker. The matchmaker then invokes a matchmaking algorithm by which matches are identified. To perform the match, the matchmaker evaluates expressions in an environment that allows each classAd to access attributes of the other. After the matching phase, the matchmaker invokes a matchmaking protocol to notify the two parties that were matched and sends them the matching ads. The matchmaking protocol could also include the generation and hand-off of a session key for authentication and security purposes. The customer then contacts the server directly, using a claiming protocol to establish a working relationship with the provider. It is important to note that identifying a match and invoking the matchmaking protocol does not immediately grant service to a customer. Rather, the match is a mutual introduction to the advertising entities.
Resources in the Condor system are represented by Resource-owner agents (RAs), who are responsible for enforcing the policies stipulated by resource owners. An RA periodically probes the resource to determine its current state, and encapsulates this information in a classAd along with the owner's usage policy. Customers of Condor are represented by Customer Agents (CAs), which maintain per-customer queues of submitted jobs, represented as lists of ClassAds. RAs and CAs periodically send ClassAds to a Condor pool manager, describing the resources and job queues respectively. The resource ClassAds and the request ClassAds conform to an advertising protocol that states that every classAd should include expressions named Constraint and Rank. The protocol also requires the advertising parties to include "contact addresses" with their ads, and allows an RA to include an "authorisation ticket" with its ad. Periodically, the pool manager enters a negotiation cycle. This phase invokes the matchmaking algorithm, which determines which CAs require matchmaking services, obtains requests from these CAs, and matches them with compatible RA ads. Since the notion of "compatible" is completely determined by Constraint expressions, ClassAds may be matched in a general manner. In addition, Rank expressions are used as goodness metrics to identify the more desirable among the compatible matches. The matchmaking algorithm also uses past resource usage information to enforce a fair matching policy. When the pool manager determines that two ClassAds match, it invokes the matchmaking protocol to contact the matched principals at the contact addresses specified in their ClassAds and send them each other's ClassAds. The manager also gives the CA the authorisation ticket supplied by the RA. The CA then performs the claiming protocol by contacting the RA and sending the authorisation ticket. The RA accepts the resource request only if the ticket matches the one that it gave the pool manager, and the request matches the RA's constraints with respect to the updated state of the request and resource, which may have changed since the last advertisement. If the request is accepted, the workstation runs the customer's job. The RA may also send an ad when it starts running the job, indicating that although the workstation is currently busy, it is still interested in hearing from higher priority customers. The specification of what constitutes "higher priority" is completely under the control of the RA. ClassAds are used for other purposes in Condor as well. All entities are represented with ClassAds, as are queries

submitted by various administrative and user tools. "One-way matching" protocols are used to find all objects matching a given pattern. For example, there are tools to check on the status of job queues and browse existing resources.

Condor mechanisms

1. A mechanism for determining when a workstation is not in use by its owner, and thus should become part of the pool of available machines. This is accomplished by measuring both the CPU load of the machine, and the time since the last keyboard or mouse activity. The default is to consider a machine idle when the CPU average as measured by UNIX is less than 0.3, and the keyboard and mouse have been idle for at least 15 minutes. Individual workstation owners can customise each of these parameters, as they deem appropriate.
2. A mechanism for fair allocation of these machines to users who have queued jobs. This task is handled by a centralised "machine manager". The manager allocates machines to waiting users on the basis of priority. The priority is calculated according to the up-down algorithm. This algorithm periodically increases the priority of those users who have been waiting for resources in the recent past.
3. Condor provides a remote execution mechanism, which allows its users to run remotely the same programs that they had been used to running locally after only a re-linking step. File I/O is redirected to the submitting machine, so that users don't need to worry about moving files to and from the machines where execution actually takes place.
4. The fourth mechanism is responsible for stopping the execution of a Condor job upon the first user activity on the hosting machine. As soon as the keyboard or mouse becomes active, or the CPU load on the remote machine rises above a specified level, a running Condor job is stopped. This provides automatic return of the use of the hosting workstation to its owner.
5. Condor provides a transport checkpointing mechanism which allows it to take a checkpoint of a running job, and migrate that job to another workstation when the machine it is currently on becomes busy with non-Condor activity. This allows Condor to return workstations to their owners promptly, yet provide assurance to Condor users that their jobs will make progress, and eventually complete.

How to use Condor

1. Job preparation: first, you will need to prepare your job for Condor. This involves preparing it to run as a background batch job, deciding which condor runtime environment to use, and possible relinking your program with the Condor library via the `condor_compile` command.
2. Submit to Condor: next, you'll submit your program to Condor via the `condor_submit` command. With `condor_submit` you'll tell Condor information about the run, such as what executable to run, what filenames to use for keyboard and screen data, and where to send email when the job completes. You can also tell Condor how many times to run a program; many users may want to run the same program multiple times with multiple different data files. Finally, you'll also describe to Condor what type of machine you want to run your program.
3. Condor runs the job: once submitted, you'll monitor your job's progress via the `condor_q` and `condor_status` commands, and/or possibly modify the order in which Condor will run your jobs with `condor_prio`. If desired, Condor can even inform you every time your job is checkpointed and/or migrated to a different machine.
4. Job completion: when your program completes, Condor will tell you the exit status of your program and how much CPU and wall clock time the program used. You can remove a job from the queue prematurely with `condor_rm`.

Priorities in Condor:

Job priority. Job priorities allow you to assign a priority level to each of your jobs in order to control their order of execution.

User priority. Machines are allocated to users based upon that user's priority.

User priorities in Condor can be examined with the `condor_userprio` command, and Condor administrators can set and edit individual user priorities with the same utility. Condor continuously calculates the share of available machines that each user should be allocated. This share is inversely related to the ration between user priorities; for example, a user with a priority of 10 will get twice as many machines as a user with a priority of 20. The priority of each individual user changes according to the number of resources he is using. Each user starts out with a priority of 0.5 (the best priority allowed). If the number of machines the user currently has is greater than his priority, the priority will numerically increase (worsen) over time, and if it is less the priority will automatically decrease (improve) over time. The long-term result is fair-share access across all users.

Conclusion about Condor

Condor definitely seems like a good choice for a queue-management tool. Its main functionality is resource selection and allocation, which is precisely what is required by EPCC. However, Condor lacks the necessary infrastructure required for Grid operation. As its developers promise, this should be remedied by the introduction of Condor-G, which should perform all of Condor's functions on the Grid. Since Condor-G does not exist as yet, the choice is now either to wait for it, or start implementing a standalone Grid broker from scratch.

Cactus

One layer above the Grid broker on the Grid pyramid is the application environment. Clearly, to write good Grid-enabled applications, users need to have an application environment that could interact well with the underlying layers. In this section, we discuss Cactus, an open-source problem-solving tool, which can be used, either on its own or on top of Globus.

Overview of Cactus

Cactus is an open source problem-solving environment designed for scientists and engineers. It consists of a central core (called the "flesh") which connects to application modules (called "thorns") through an extensible interface. Thorns can either be custom developed to serve a specific domain or general utilities for the public domain, such as parallel I/O, data distribution, or checkpointing. Cactus runs in many different architectures, thus allowing seamless running of applications across multiple platforms.

Why use Cactus?

Cactus application areas include:

- Performing parallel programming in an easy but powerful manner using several languages: F77, F90, C, C++.
- Running applications on different architectures and operating systems.
- Code portability: developing code on one machine and moving it to a supercomputer to run it straight away.
- High performance cluster computing or turning a networked PC pool into a computing cluster.
- Collaborating with others on the same code without having to fragment programs.

Thorns and Arrangements

Thorns are grouped into arrangements that can be collected into general applications. Cactus comes with a set of read-made arrangements serving a wide variety of purposes. For a list of all arrangements and their thorns, please refer to:
http://www.cactuscode.org/cactus_cgi-bin/viewthorn.pl

Installing Cactus

1. Log into the cactus repository via:
   ```
   cvs -d :pserver:cvs_anon@cvs.cactuscode.org:/cactus login
   ```
   when prompted for a password, type: anon.
2. To obtain a fresh copy of cactus, type:
   ```
   cvs -z9 -d :pserver:cvs_anon@cvs.cactuscode.org:/cactus checkout
   Cactus
   ```
   This creates a directory called Cactus and installs the code into it.
3. To check out (install) an arrangement or thorn, move to the arrangements directory and type:
   ```
   cvs -z9 checkout <arrangement-name¦arrangement-name/thorn-name>
   ```
4. For a list of all available arrangements and thorns, type:
   ```
   cvs -z9 -d :pserver:cvs_anon@cvs.cactuscode.org:/cactus checkout
   -s
   ```
5. To update an existing Cactus checkout, got o the Cactus directory and type:
   cvs –z9 update
6. The Cactus directory structure is as follows:
   CVS: the CVS bookkeeping directory, present in every subdirectory
   doc: Cactus documentation
   lib: contains libraries
   src: contains the source code for cactus
   arrangements: contains the cactus arrangements
   When cactus is first compiled it creates a new directory called /`configs` which will contain all the source code, object files and libraries created during the build process.

Compiling Cactus

You can compile cactus on its own, but it doesn't actually do anything. In order to be able to test cactus and run several applications, you need to install one or more thorns. Some of them come with a testsuite that you can run to see whether your installation was a success.
Cactus can be compiled using different configurations. This allows you to have different arrangements for different architectures based on a single source code. You can also use that to compare different compiler options or different thorn collections.
There are two ways to create a configuration:
a. Add the options to a configuration file and use:
```
    gmake <config>-config options=<filename>
```
The format of the filename should be in the form: <option>[=]...
Any unspecified options will take default values.
b. Pass the options individually on the command line:
```
    gmake <configuration name>-config <option-name>=<value>
```
For the various configuration options, please consult the Cactus Users' Guide.
To build a configuration:
```
    gmake <config-name>
```
For several build options of `gmake`, please refer to the Cactus Users' Guide.

Notes for Solaris 2.6

This was the first time Cactus has been compiled on Solaris 2.6. Most of the default settings work ok. However, it fails to locate the c preprocessor (cpp).
To solve that, go to:
`../Cactus/configs/<config-name>/config-data/make.config.defn`
and change the CPP value from echo to the location of the local cpp.
Some thorns require environment variables to be set. A way around this is just to remove these thorns from the thorn list:

```
      gmake <config-name>-thornlist
```

and to make sure they will not be included in future builds, remove them from the arrangements directory as well.
Some Fortran libraries may be missing. To remedy this, either remove all thorns that contain Fortran code (The CactusWave/WaveToyC and CactusWave/IDScalarWaveC thorns are written in C and come with a testsuite to test Cactus operation), or add the required libraries to the LIBS line in

```
     make.config.defn.
```

General Notes

Before making any changes to the `make.config.defn` file, you should clean all dependencies first, using:

```
      gmake <config-name>-cleandeps
```

To use the testsuite, you need to add the `CactusPUGHIO/IOASCII` thorn to display test outputs. I have tried running the testsuite. It performed both the Regression and the Portability tests and passed both.

Conclusion and Future Work

This project has explored various Grid technologies, including a metacomputing infrastructure toolkit, Globus, two Grid-enabled job management packages, Nimrod and Condor, and a Grid-enabled programming environment, Cactus. Among all of these, Nimrod seems to be of less use to EPCC than the others. Condor is a good queue-management tool, which can be used locally at EPCC to link the Cray T3E and the Sun-HPC cluster. To have a Grid-enabled resource broker, however, a more powerful tool will be needed. Condor-G might be a good choice, if it appears soon, and if it fulfils all its inventors' promises. Otherwise, writing a standalone broker element should not be that hard, given the information and experience that EPCC now has about Globus and the Grid.
Another interesting area would be writing an XML job specification layer on top of Condor. With a GUI-based interface, this layer can provide the user with the capability of specifying their job and run parameters easily and quickly. The event handler would then divide this XML script into two separate files, readable by Condor, the job specification file and the job ClassAd. In future, this layer can be used to create job specification for any underlying application other than Condor that would handle job management in the future.

References

1. Abramson, D., and Giddy, J., “Scheduling Large Parametric Modelling Experiments on a Distributed Meta-computer”
2. Buyya, R., Abramson, D., and Giddy, J., “An Economy Driven Resource Management Architecture for Global Computational Power Grids”: www.csse.monash.edu.au/~davida/papers/GridEconomy.pdf
3. Litzkow, M., and Livny, M., “Experience with The Condor Distributed Batch System”
4. Abramson, D., Sosic, R., Giddy, J., and Hall, B., “Nimrod: A Tool for Performing Parameterised Simulations using Distributed Workstations”
5. Foster, I., Czajkowski, K., Kesselamn, K., Tuecke, S., Smith, W., and Martin, S., “A Resource Management Architecture for Metacomputing Systems”
6. The Globus Website: www.globus.org
7. The Condor Website: www.cs.wisc.edu/condor/
8. The Nimrod Project Website: www.csse.monash.edu.au/~davida/nimrod.html/
9. The Condor Manual: www.cs.wisc.edu/condor/manual/
10. www.csse.monash.edu.au/~davida/papers/hpcasia.ps.Z
11. www.csse.monash.edu.au/~davida/papers/nimrodg.ps.Z
12. Raman, R., Livny, M., and Solomon, M., “Matchmaking: Distributed Resource Management for High Throughput Computing”

Biography

I have just finished a 5-year undergraduate degree in Computer Engineering at Cairo University. Now I’m going to do a postgraduate course at the same University for another year and then possibly go abroad for a Ph.D. My main computing interests include Embedded Systems and Artificial Intelligence. Outside college and work, I enjoy playing several sports (Basketball, Squash, Swimming…) and shopping. I also like to travel and, when I feel like it, I would go to the theatre or play the flute.

Finally, I would like to thank my supervisors, Paul Graham and Connor Mulholland for dedicating the time to me whenever I needed it, and for allowing me to do things that I really like and am interested in. I would also like to thank Dr Rob Baxter for taking on my project on such a short notice and for coming up with the idea for the XML layer on top of Condor. I would also like to thank the EPCC staff for making my second SSP experience even more interesting than the first.