

WebCom-G: Middleware to Hide the Grid

John P. Morrison, David A. Power, Brian Clayton,
Adarsh Patil, Philip Healy, James J. Kennedy
Centre for Unified Computing,
Dept. Computer Science,
University College, Cork,
Ireland

Abstract

Current Grid enabling technologies consist of stand-alone architectures. A typical architecture provides middleware access to various services at different hierarchical levels. Services exposed at these levels may be leveraged by the application programmer. However, the level at which the service appears in the hierarchy determines both its richness and the complexity of its use. Thus, benefits gained by using these services are defined by the manner in which they are accessed. Generally, choosing to use a particular service inclines the application programmer to use the associated middleware suite as it is difficult to cherry-pick services across middlewares. Interoperability and independent service access are not easily facilitated in current middlewares.

If it is accepted that Grid computing will be an important technology in the near to long term (indeed it can be credibly argued that it is already a very important technology to a select few), then wide spread acceptance will depend on making that technology easily accessible to general exploitation. In practice, this will not only involve interoperability and inclusiveness of key features, but, most importantly, it will require that non specialists be facilitated in constructing grid independent applications that run efficiently on the dynamic architecture that constitutes the Grid. The original problems of programming parallel and distributed systems still hold true: they are notoriously hard to program, since programmers usually have responsibility for synchronizing processes and for resource management. Solutions must be developed to free programmers from the low level details whose consideration gives rise to these problems. In effect, grid programming environments must evolve to a point where grid (and, in general, parallel) programs are freed from architecture details such as data locality, machine availability, inter-task synchronisation, communication topologies, task load-balancing, and fault tolerance - in the same manner as present day sequential programmers are freed from explicit memory management, disk access protocols and process scheduling. At that point in the evolution of the Grid, the grid middleware will adopt the character of a grid operating system and many, if not all, of the issues that make grid programming difficult will migrate out of grid application programs. When this is achieved, the vision of hiding the Grid will have been realised and exploitation of the technology can begin in earnest.

WebCom-G is a fledgling Grid Operating System, designed to provide independent service access through interoperability with existing middlewares. It offers an expressive programming model that automatically handles task synchronisation – load balancing, fault tolerance, and task allocation are handled at the WebCom system level - without burdening the application writer. These characteristics, together with the ability of its computing model to mix evaluation strategies to match the characteristics of the geographically dispersed facilities and the overall problem-solving environment, make WebCom a promising grid middleware candidate.

Keywords - WebCom-G, Grid, Middleware, Interoperability, Co-Existence

1 Introduction

The computing power available in current desktop machines is equivalent to or supersedes that of past high-performance computers (HPCs). Together with advances in networking, HPCs can be built by harnessing the computing power of commodity computers distributed around the world. These HPCs are typically owned by multiple heterogeneous organisations and Grids are constructed from the amalgamation, sharing and selection of networked services by these organisations. Such services are made available under common sharing and security policies.

Much research is also focused on exploiting the computing resources of geographically distributed volunteers using the Internet and user-level middlewares such as SETI@Home and Distributed.Net.

Approaches such as Globus[25, 26, 67], Legion[38, 65], JXTA[55], Hydra[14] etc, exploit core level middleware technologies. These provide a fundamental software infrastructure to build Grid technologies that aim to find ways to make computing easier, faster, and more accessible to programmers and users. Some of these core middleware technologies provide a “*bag of services*” [67], toolkits or integrated service architectures. Other projects: Condor-G[21], PBS[52], LSF[51] provide independent job managers that interact with these core middleware services. A brief outline of some common grid systems is given in Section 2.

Once the services provided by these systems are known, they are aggregated and presented to the application developer by the Grid resource broker. The application developer must possess a knowledge of both the middleware being used and the underlying computational hardware. Using this information, task dependent libraries and binaries can be produced. These are typically managed by the user, who also has to possess some knowledge of the target architecture. This makes the process of application deployment both time consuming and error prone. In these systems, large tasks are executed by distributing sub-tasks to cooperating machines. Task dependent libraries and binaries are typically provided and managed by the user, thus making the process of developing, deploying and execution applications both time consuming and error prone.

WebCom-G, the focus of this chapter, is an application execution environment operating across the Internet or on intranets. Applications executed by WebCom-G are specified as Condensed Graphs (\mathcal{CG}), in a manner which is independent of the execution architecture, and this platform independence facilitates computation in heterogeneous environments.

In addition to its expressive programming model, WebCom-G automatically handles task synchronisation, load balancing[46], fault tolerance[46], and task allocation without the need for these decisions to be propagated to the application developer. These characteristics, together with the ability of the \mathcal{CG} model to mix evaluation strategies to match the characteristics of geographically dispersed facilities and overall problem-solving environment, make WebCom-G a promising Grid middleware candidate[44].

The WebCom-G Operating System is proposed as a Grid Operating System. It is modular and constructed around a WebCom-G kernel, offering a rich suite of features to enable the Grid. It will utilise the tested benefits of WebCom-G and will leverage existing grid technologies such as Globus and MPI. The aim of the WebCom-G OS is to hide the low level details from the programmer while providing the benefits of distributed computing.

WebCom-G hides the low-level details of distributed computing such as Fault Tolerance, Load Balancing, Scheduling etc. from the application developer. WebCom-G OS hides Grid development and runtime issues, such as resource availability (both hardware and software), task staging etc.

The remainder of this chapter is broken up as follows: Section 2 presents a brief survey of the current state of Grid Computing, since it is beyond the scope of this chapter to present a more complete survey. Section 3 describes WebCom and its various components. Again, brief descriptions of these components are given and the reader is directed to further publications where appropriate. Section 4 promotes WebCom-G, Webcom Grid, as a system capable of exploiting and interoperating with different Grid middlewares. This section describes the additional components to WebCom and presents their interactions with Globus as an example grid middleware. Section 5 describes the representation of applications within WebCom-G and different application support features. Finally, looking into the future, Section 6 presents some goals of the WebCom-G project.

2 Current Grid Space

2.1 Systems Architecture

Figure 1 depicts the current Grid space. On the right hand side of the picture, the grid hardware is shrouded by various service layers, which offer successive levels of abstraction without enforcing any specific programming paradigm. The Sun Grid Engine and Legion, towards the bottom of the picture offer vertical integrated, although somewhat insular, platforms. The other category represents the vast array of initiatives gathered under the generic label of metacomputing.

2.1.1 Globus

Globus is an integrated toolkit of grid services facilitating the creation of Grids, enabling high speed coupling of people, computers, databases, and instruments. Globus can be characterized as a "sum of services" architecture; including grid information services, schedulers, authentication, and file transfer. Applications can use grid services without having to adopt a particular programming model. Globus

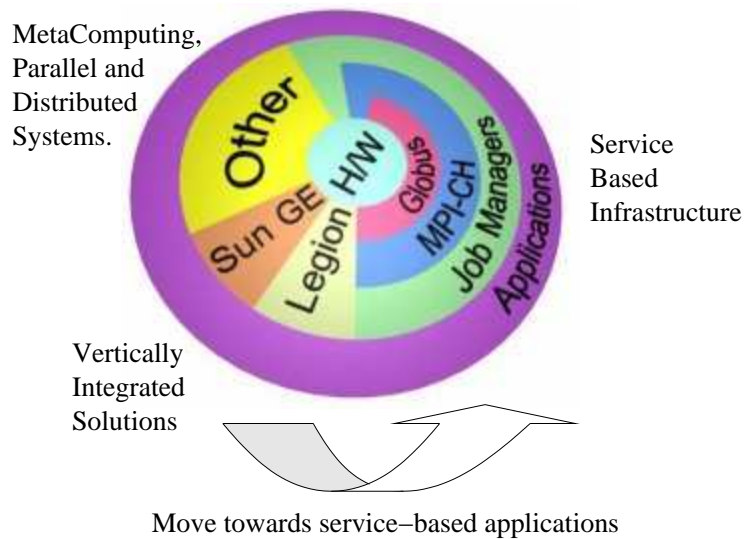


Figure 1: *A representation of present day grid space*

is the basis of the vast majority of grid projects and is now undergoing a major transition to a web services model.

2.1.2 Legion

Legion is a distributed high-performance metacomputing system, arising from Grimshaw's work at the University of Virginia, aiming to find ways to make metacomputing easier, faster, and more accessible to programmers and users. In contrast to Globus, Legion is an integrated architecture based on an abstract object model. It is also a commercial system.

2.1.3 Sun Grid Engine (SGE)

Sun ONE Grid Engine software is Sun's standard solution for managing clusters, a descendant of Gantz's CODINE (<http://www.hu-berlin.de/rz/compsv/codine.html>). It provides transparent resource access, distributed resource management, high utilization and increased computing throughput. Feature includes single point access, resource balancing and scalability. The SGE and Legion are fully featured integrated solutions. They are also insular in that they provide a low level (or no) interoperability with other systems. In contrast to Globus, however, they provide abstraction layers that eases programmability; offering fault masking and static load balancing. Job management is based on target machines and target machine states, controlled from a central server. In both of these systems, the programmer must explicitly handle parallelism and inter-task communication.

2.1.4 MPI

MPI is a library for message passing, proposed as a standard by a broadly based committee of vendors, implementers, and users. Its features includes point-to-point communication, applications oriented process topologies, profiling, and error control. Although this library offers invaluable functionality, the application programmer is

responsible for explicit task management including task synchronisation and fault survival. A grid-enabled library, MPICH-G , is available.

2.2 Job Managers

2.2.1 Condor

Condor is an opportunistic resource management facility that exploits idle resources. To avail of Condor, each participating machine runs daemons to negotiate with a central manager to determine where and when jobs can be remotely executed. Fault tolerance is achieved through check pointing and job migration based on user intervention in response to system emails. Users must have access to single-threaded object files or source code of the program to use Condor check-pointing.

2.2.2 Load Sharing Facility, LSF

LSF is a general purpose distributed computing system from Platform Computing Corporation. It unites a group of Unix and/or NT computers into a single system to make better use of the resources on a network. LSF supports sequential and parallel applications running as interactive and batch jobs. It also allows new distributed applications to be developed through C programming libraries and a tool kit of programs for writing shell scripts. LSF manages job processing by providing a transparent, single view of all hardware and software resources, regardless of systems in the cluster, making use of idle workstations and servers.

2.2.3 Portable Batch System (PBS)

PBS is a flexible batch queuing and workload management system originally developed by Veridian Systems for NASA. It operates on networked, multi-platform UNIX environments, including heterogeneous clusters of workstations, supercomputers, and massively parallel systems. Key features of PBS include portability, configurability, adaptability, expandability, flexibility, and usability.

2.3 MetaComputing Systems

The concept of using the Internet as a ready-made platform for computing parallel programs in a distributed manner has existed for many years and is given the name Metacomputing. These are partitioned into Volunteer Computing Systems that allow non-expert participants to donate compute cycles by installing binaries and following a simple set up procedure. They have the advantage of resulting in very large networks of workstations. However, they are crude instruments whose successful management relies heavily on the altruism of its participants. Examples include distributed.net's Bovine RC5 and SETI at home (SETI@home). Another category is the Java-based volunteer computing system. These consist of "One-click computing", have minimal or no set up and are platform independent. They are also secure in that they restrict execution to the sandbox of the JVM. These applications tend to be custom built clients and are usually problem specific. They require specialist-programming skills to construct and rely on custom fault tolerance and load balancing-built in at the

application level. Again cycles are donated altruistically. A third, more sophisticated, member of the group will be referred to as general-purpose metacomputing systems. In addition to Peer-to-Peer systems like Napster (<http://opennap.sourceforge.net/>) and Gnutella (<http://www.gnutella.com/>), this category also contains 2-tier client server systems, such as Bayanihan[23, 61, 62, 63], Charlotte[8, 33], Javelin[15], SuperWeb[4, 5], and DJM[37]. In operation, these systems distribute work to client machines via Java applets and different applet types are employed for different tasks. These applets, having been loaded onto different client machines, can subsequently communicate directly with each other by remote method invocation (RMI)[42] and Object Request Brokering (ORB)[49, 60]. These systems provide object libraries for the application developer to build distributed systems. The onus is on the programmer to uncover the parallelism in the application and to manage it explicitly within the program.

2.4 Programming on Distributed Architectures

2.4.1 Traditional Imperative Approach

Parallel systems, and by extension grid systems, are notoriously hard to program, since programmers usually have responsibility for synchronizing processes and for resource management. Efficient exploitation of parallelism requires some form of runtime optimisation because the target architecture and resource availability may not be known at compile time. The non-portability of much parallel code is testament to the fact that many optimisations have only been performed statically or that the code has been written for a particular architecture.

2.4.2 Functional and Dataflow Language Approach

Functional languages have been the focus of much research for the programming of parallel systems. They offer a natural model for implicit parallelism and a flexible mapping of evaluations to underlying architectures. Earlier work in data-driven computing, addressed the problem of programming parallel systems but with limited success: granularity was too fine and there was no natural mechanism for handling data structures. Subsequent work including, and addresses these problems with more success. The semantics of these languages may include side effects and specify either strict or non-strict operators. Haskell exemplifies a side-effect free, non-strict language whereas ML is strict with side effects. Operators can be evaluated using either a lazy or an eager strategy. Lazy evaluation attempts to determine a result by evaluating an operator before any of its operands, the evaluation of an operand is forced only if it is required. In contrast, eager evaluation computes the values of operands before attempting to evaluate their operator. An abstract machine may be used to fully exploit dynamic optimisations and facilitate executing a single Intermediate Representation on many architectures. A contemporary example of this approach is the Java Virtual Machine that executes byte-code. Unfortunately it is difficult to dynamically optimise byte-code since it contains less structure than the corresponding Java source code. High level information is thus seen to be important for aggressive optimisations such as making full use of registers, reducing cache line collisions and for the analysis of inter block dependencies. The Tree and Graph based approaches of and

preserve high-level program structure, making it easier to perform code optimisations. To date, these approaches appear to have only been applied to sequential programs and architectures. An ideal intermediate language would allow control over locality, would have low communication and administration overhead, allow various levels of granularity, incorporate speculative computations, and be garbage free. It would need to be efficiently and dynamically optimised without programmer intervention and executable on all machines. The representation should also have a mutable store and allow for many different orders of evaluation. An abstract machine used to execute such a language should have mechanisms for controlling parallelism, matching this parallelism to the machine and preventing it from saturating its resources. Abstract machines can be used to encapsulate various architectures and to provide cost models, and primitives, for communication and synchronization . Rather than using an abstract machine, the bulk synchronous parallel (BSP) computer model provides the programmer with an abstraction for all machines in terms of four performance parameters. In this way it attempts to provide a simple, unified framework viewing all architectures as BSP machines. BSP may be implemented as a library on top of standard programming systems, as well as in novel languages such as GPL which provide the programmer with access to three of these parameters. In doing this, the programmer still has control over, and presumably responsibility for, synchronization and resource management. The advantage is that only a single architecture needs to be considered and source code portability can be achieved.

3 WebCom MetaComputer

Metacomputing systems were developed to harness the power of geographically distributed computing resources. Such resources generally consisted of machines connected to intranets, the Internet and World Wide Web. Different projects in this area of computing range from those harnessing the power of closely coupled networks of workstations, such as Cilk-NOW[39, 59] and Treadmarks[22], to projects such as Charlotte, Bayanihan, Javelin which utilize the processing power of the Internet.

These systems typically use the server/client model for task distribution. Clients normally consist of stand alone applications, or Java applets with different applets being used for different tasks. The stand-alone application client communicates with the server via proprietary mechanisms, while applet based clients typically communicate using Remote Method Invocation(RMI), Object Request Brokers (ORB's) or Object Serialization.

Distributed applications are typically constructed by using Application Programming Interfaces (API's) provided by the chosen system. Here the onus falls on the programmer to explicitly determine the parallelism of the problem as well as having to implement fault tolerance, load balancing and scheduling algorithms.

WebCom separates the application and execution environments by providing both an execution platform, and a development platform. Applications are specified as Condensed Graphs (\mathcal{CG} s), in a manner which is independent of the execution architecture. The independence provided by separating these two environments facilitates computation in heterogeneous environments; the same \mathcal{CG} programs run without change on a range of implementation platforms from silicon based Field Programmable Gate Arrays[48] to the WebCom metacomputer. Fault tolerance, load

balancing, scheduling and exploitation of available parallelism are handled implicitly by WebCom without explicit programmer intervention.

WebCom uses a server/client model for task distribution. Clients consist of Abstract Machines(AM's) that can be either pre-installed or downloaded dynamically from a WebCom server. AM's are uniquely comprised of both volunteers and conscripts. Volunteers donate compute cycles by instantiating a web based connection to a WebCom server and dynamically downloading the client abstract machine. These clients, constrained to run in the browser's sandbox, will execute tasks on behalf of the server. Task communication is carried out over dedicated sockets. Pre-installed clients, also communicate over dedicated sockets. Upon receipt of a task representing a \mathcal{CG} (the task can be partitioned for further distributed execution), such clients are promoted to act as other WebCom servers. The returning of a result causes a promoted AM to be demoted, and act as a simple client once more.

The execution platform consists of a network of dynamically managed machines, each running WebCom. WebCom can assume a traditional client server connection model or the more contemporary peer to peer model, Figure 2.

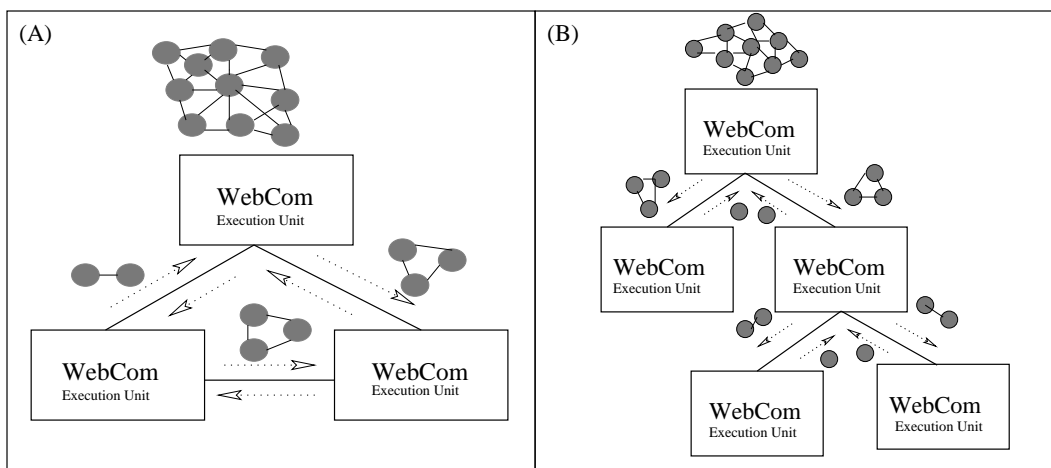


Figure 2: *WebCom can be configured as both a peer to peer hierarchy (A) and a traditional server client hierarchy (B). Each unit can receive instructions that comprise of either Condensed Graphs or “atomic” instructions. Atomic instructions are generally value transforming instructions and can be of arbitrary grainsize.*

3.1 Condensed Graphs - Overview

While being conceptually as simple as classical dataflow schemes [6, 31], the Condensed Graphs (\mathcal{CG}) model is far more general and powerful. It can be described concisely, although not completely, by comparison. Classical dataflow is based on data dependency graphs in which nodes represent operators and edges are data paths which convey simple data values between them. Data arrive at *operand ports* of nodes along input edges and so trigger the execution of the associated operator (in dataflow parlance, they cause the node to *fire*). During execution, these data are consumed and a resultant datum is produced on the node's outgoing edges. This result acts as input to successor nodes. In cyclic dataflow graphs, nodes may fire more than once and so operands belonging to different executions are distinguished

from each other using labels. When data with the same label are present on every operand port, a *complete operand set* is formed. Operand sets are used at the basis of the firing rules in data-driven systems. These rules may be *strict* or *non-strict*. A strict firing rule requires a complete operand set to exist before a node can fire; a non-strict firing rule triggers execution as soon as a specific proper subset of the operand set is formed. The latter rule gives rise to more parallelism but also can result in overhead due to remaining packet garbage (RPG).

Like classical dataflow, the \mathcal{CG} model is graph-based and uses the flow of entities on arcs to trigger execution. In contrast, \mathcal{CG} s are directed acyclic graphs in which every node contains not only operand ports, but also an operator and a destination port. Arcs incident on these respective ports carry other \mathcal{CG} s representing operands, operators and destinations. Condensed Graphs are so called because their nodes may be condensations, or abstractions, of other \mathcal{CG} s. (Condensation is a concept used by graph theoreticians for exposing meta-level information from a graph by partitioning its vertex set, defining each subset of the partition to be a node in the condensation, and by connecting those nodes according to a well-defined rule [27].) Condensed Graphs can thus be represented by a single node (called a *condensed node*) in a graph at a higher level of abstraction. The number of possible abstraction levels derivable from a specific graph depends on the number of nodes in that graph and the partitions chosen for each condensation. Each graph in this sequence of condensations represents the same information but in a different level of abstraction. It is possible to navigate between these abstraction levels, moving from the specific to the abstract through condensation, and from the abstract to the specific through a complementary process called *evaporation*.

The basis of the \mathcal{CG} firing rule is the presence of a \mathcal{CG} in every port of a node. That is, a \mathcal{CG} representing an operand is associated with every operand port, an operator \mathcal{CG} with the operator port and a destination \mathcal{CG} with the destination port. This way, the three essential ingredients of an instruction are brought together (these ingredients are also present in the dataflow model; only there, the operator and destination are statically part of the graph).

A condensed node, a node representing a datum, and a multi-node \mathcal{CG} can all be operands. A node represents a datum with the value on the *operator* port of the node. Data are then considered as zero-arity operators. Datum nodes represent graphs which cannot be evaluated further and so are said to be in *normal form*. Condensed node operands represent unevaluated expressions. They cannot be fired since they lack a destination. Similarly, multi-node \mathcal{CG} operands represent partially evaluated expressions. The processing of condensed node and multi-node operands is discussed below.

Any \mathcal{CG} may represent an operator. It may be a condensed node, a node whose operator port is associated with a machine primitive (or a sequence of machine primitives) or it may be a multi-node \mathcal{CG} .

The present representation of a destination in the \mathcal{CG} model is as a node whose own destination port is associated with one or more port identifications. The expressiveness of the \mathcal{CG} model can be increased by allowing any \mathcal{CG} to be a destination but this is not considered further here. Figure 3 illustrates the congregation of instruction elements at a node and the resultant rewriting that takes place.

When a \mathcal{CG} is associated with every port of a node it can be fired. Even though the

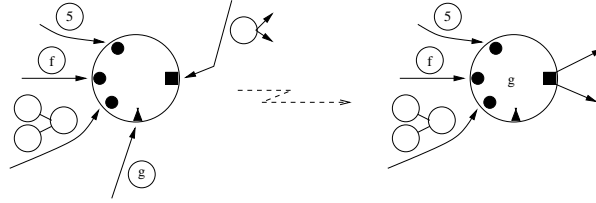


Figure 3: \mathcal{CG} s congregating at a node to form an instruction

\mathcal{CG} firing rule takes account of the presence of operands, operators and destinations, it is conceptually as simple as the dataflow rule. Requiring that the node contain a \mathcal{CG} in every port before firing prevents the production of RPG. As outlined below, this does not preclude the use of non-strict operators or limit parallelism.

A *grafting* process is employed to ensure that operands are in the appropriate form for the operator: non-strict operators will readily accept condensed or multi-node \mathcal{CG} s as input to their non-strict operands. Strict operators require all operands to be data. Operator strictness can be used to determine the strictness of operand ports: a strict port must contain a datum \mathcal{CG} before execution can proceed, a non-strict port may contain any \mathcal{CG} . If, by computation, a condensed or multi-node \mathcal{CG} attempts to flow to a strict operand port, the *grafting* process intervenes to construct a destination \mathcal{CG} representing that strict port and sends it to the operand.

The grafting process thus facilitates the evaluation of the operand by supplying it with a destination and, in a well constructed graph, the subsequent evaluation of that operand will result in the production of a \mathcal{CG} in the appropriate form for the operator. The grafting process, in conjunction with port strictness, ensures that operands are only evaluated when needed. An inverse process called *stemming* removed destinations from a node to prevent it from firing.

Strict operands are consumed in an instruction execution but non-strict operands may be either consumed or propagated. The \mathcal{CG} operators can be divided into two categories: those that are “value-transforming” and those that only move \mathcal{CG} s from one node to another in a well-defined manner. Value-transforming operators are intimately connected with the underlying machine and can range from simple arithmetic operations to the invocation of sequential subroutines and may even include specialised operations like matrix multiplication. In contrast, \mathcal{CG} moving instructions are few in number and are architecture independent. Two interesting examples are the condensed node operator and the **filter** node. Filter nodes have three operand ports: a Boolean, a **then** and an **else**. Of these, only the Boolean is strict. Depending on the computed value of the Boolean, the node fires to send either the **then** \mathcal{CG} or the **else** \mathcal{CG} to its destination. In the process, the other operand is consumed and disappears from the computation. This action can greatly reduce the amount of work that needs to be performed in a computation if the consumed operands represent an unevaluated or partially evaluated expression. All condensed node operators are non-strict in all operands and fire to propagate all their operands to appropriate destinations in their associated graph. This action may result in condensed node operands (representing unevaluated expressions) being copied to many different parts of the computation. If one of these copies is evaluated by grafting, the graph corresponding to the condensed operand will be invoked to produce a result. This result is held local to the graph and returned in response to the grafting of the other copies. This mechanism is reminiscent

of parallel graph reduction [50] but is not restricted to a purely lazy framework.

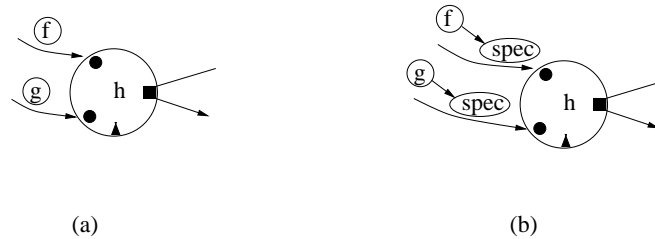


Figure 4: *Increasing parallelism by speculatively evaluating operands*

\mathcal{CG} s which evaluate their operands and operator in parallel can easily be constructed by introducing `spec` (speculation) nodes to act as destinations for each operand. The `spec` node has a single operand port which is strict. The multi-node \mathcal{CG} operand containing the `spec` node is treated by non-strict operand ports in the same way as every other \mathcal{CG} , however, if it is associated with a strict port, the `spec` node's operand is simply transferred to that port. If that operand had already been fully evaluated, it could be used directly in the strict port, otherwise, it is grafted onto the strict port as described above. This is illustrated in Figure 4.

Stored structures can be naturally represented in the \mathcal{CG} model. \mathcal{CG} s are associated with the operand ports of a node which initially contain no operator or destination. These operand structures can then be fetched by sending appropriate `fetch` operators and destinations to these nodes. These fetch operators also form part of the \mathcal{CG} moving operators and so are machine independent.

The power of the operators in the \mathcal{CG} model can be greatly enhanced by, associating with each, specific *deconstruction semantics*. These specify if \mathcal{CG} s can be removed from the ports of a node after firing. In general, every node will be deconstructed to remove its destination after firing. This renders a node incomplete and prevents it from being erroneously refired. The deconstruction semantics of the `fetch` operator cause the operator and the destination to be removed after firing. This leaves the stored structure in its original state, ready for a subsequent fetch. Figure 5 illustrates the process of fetching a stored structure.

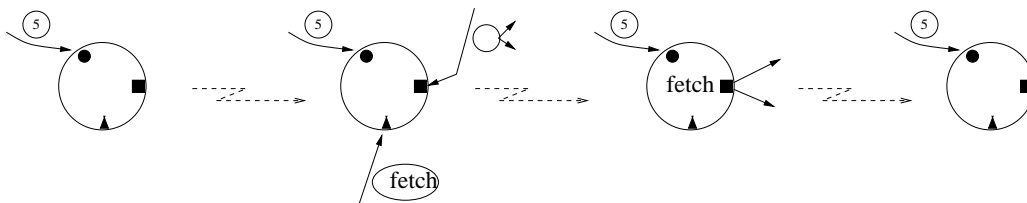


Figure 5: *Sequence of events showing the fetching of a stored structure and subsequent node deconstruction*

The \mathcal{CG} moving instructions are extended to include assignment which, with appropriate deconstruction semantics, implement mutable structures. Side-effect computations are thus encapsulated in graphs representing mutable, stored structures. Condensed nodes representing these structures may flow on graphs linearised to ensure that the side-effects are carried out in the correct order. Those parts of the computation not affected by the side-effects can proceed as normal, in parallel.

This can be likened to the monadic approach for introducing side-effects into pure, non-strict languages: where the monad structure[43] is used to encapsulate the side-effects and ensure correct sequencing[32].

By statically constructing a \mathcal{CG} to contain operators and destinations, the flow of operand \mathcal{CG} s sequences the computation in a dataflow manner. Similarly, constructing a \mathcal{CG} to statically contain operands and operators, the flow of destination \mathcal{CG} s will drive the computation in a demand-driven manner. Finally, by constructing \mathcal{CG} s to statically contain operands and destinations, the flow of operators will result in a control-driven evaluation. This latter evaluation order, in conjunction with side-effects, is used to implement imperative semantics. The power of the \mathcal{CG} model results from being able to exploit all of these evaluation strategies in the same computation, and dynamically move between them, using a single, uniform, formalism.

3.2 WebCom Architecture

WebCom sees each abstract machine as a “unit”. To construct a fully functional metacomputer, core modules that provide fault tolerance[7, 28, 35], load balancing, scheduling [9, 68], compute engine, security and communications have to be installed, Figure 6. Each of these modules are plugins to a core module called the Backplane. Communications between WebCom units use a messaging system. Plugins can send messages between themselves on the local unit or to any plugin on other connected units. The default engine module is capable of executing Condensed Graphs applications. Initial configuration will specify where these modules are to be found: on a local file server or WebCom host. In execution, these modules may be obtained and installed locally, thus dynamically bootstrapping and configuring each host.

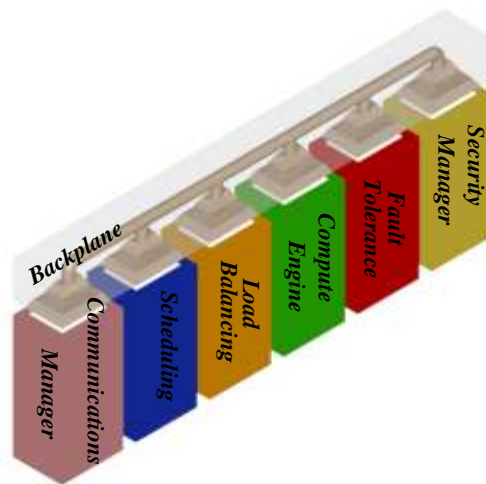


Figure 6: A WebCom Unit consists of a collection of plugins. One each for the fault tolerance, load balancing, compute engine, connection manager, security and scheduling. The backplane is used to bootstrap the whole system and handles communication between the plugins.

A WebCom execution is initiated by a user on a single unit launching a \mathcal{CG} application. As nodes become available for execution they are formed into messages and passed to the scheduler. The scheduler, in conjunction with the load balancing,

fault tolerance, communications and security plugins decide where the node is to be executed. If it is to be executed locally it is passed back to the engine. For remote execution, information about the unit selected to execute the node is incorporated into the message, and the message is then passed to the communications manager plugin for distribution to the selected unit.

Once a node has completed its operation, a result message is created and returned to the unit where the corresponding instruction originated. This unit then incorporates the returned result into its graph and the execution proceeds. If at any time an executing node fails to complete its task, the fault tolerance plugin will cause the node to be rescheduled to an alternative compatible unit. If no such unit is available, the node is retained for subsequent assignment.

WebCom possesses a node targeting mechanism. For example, if a particular node represents a COM instruction, that instruction is sent to a WebCom unit with a COM engine plugin capable of executing the instruction. This mechanism is the same for other technologies such as DCOM, EJB and Corba. One implementation of this system (Anyware) was used to create enterprise wide applications using middleware integration. This philosophy can be applied at higher levels of abstraction.

3.2.1 Communications Manager Module

WebCom contains two Communication Manager Modules. One that possesses the minimum functionality required to configure a “traditional” metacomputing environment and a second more advanced Communication Manager Module that is used to facilitate the dynamic bootstrapping of the metacomputer from a specified codebase on the network.

The standard Communication Manager Module maintains two distinct lists of connection descriptors: one for servers and one for clients. In this context, a descriptor represents a connection to a client. One descriptor exists for each client and the descriptor facilitates the bi-directional communication mechanism between the server and client.

The connection manager operates by listening for incoming connections on a specific port. All inbound connections are from client machines. By convention, if machine M_1 connects to machine M_2 , M_1 creates a server descriptor indicating that M_2 will serve work to M_1 . M_2 will subsequently create a client descriptor indicating that M_1 will act as a client of M_2 . Although this convention indicates the normal flow of work, it does not prevent client machines from sending tasks to their associated servers for execution. In that situation a WebCom deployment takes on the character of a peer to peer system.

The standard Communication Manager Module contains housekeeping methods to detect and handle faults and to remove expired connections. Fault handling is made easy since each connection is represented by an independent process which can handle faults autonomously. This approach, however, also has disadvantages. In particular, it has limited scalability. When the number of connections exceeded 30, client utilisation dramatically fell off regardless of the amount of available parallelism. The reason is thought to be the Operating System overhead in managing process switching.

The advanced Communication Manager Module[54] consists of only two *connection*

descriptors: one maintaining an arbitrary number of incoming connections and the other maintaining an arbitrary number of outgoing connections. A connection represents a unidirectional channel between two host machines.

To allow communication of tasks and results between machines, connections occur in pairs. Where tasks and results are communicated, the number of incoming and outgoing connections on each machine will typically be the same. However, this may not be the case when a machine acts as a data source or data sink.

Associated with each connection is a *connection state*. This is consulted when managing the connections between machines. For example, it detects faults, intentional disconnections (which can be described as being either *hard* or *soft*) and the availability status of the machines. The status of a machine will be described by the *Available*, *Busy* and *Unavailable* flags. An *Available* flag indicates tasks can be scheduled to a machine. It will be flagged as *busy* if it is executing a task. It will be marked as *unavailable* if it is not willing to accept tasks. A machine can be both *busy* and *available* or *busy* and *unavailable*. For example, a gateway of firewall may be flagged as *busy* and *available* provided it is executing a task or at least one of its clients is *busy*, and it is willing to accept a task or at least one of its clients is *available*.

Scheduled tasks are transferred from server to client using either a *push* or a *pull* mechanism. The push mechanism allows a server to transmit tasks to a client provided the *available* flag of that client is set. This mechanism assumes that sufficient buffer space exists on the client to accept incoming tasks. The pull mechanism requires a server to maintain a queue of scheduled tasks for each attached client and clients are responsible for fetching tasks from their associated queues. The pull mechanism can only be used by certain clients as a result of sandbox restrictions. For example, it is not possible to push tasks to a Java applet.

The push mechanism dynamically commits a task to a client. In general, it may not be possible to undo this commitment if a better scheduling decision is subsequently determined. In contrast the scheduling queues on the server associated with the pull mechanism may be rebalanced to reflect better scheduling decisions. For example, if an increase in the rate at which the client is pulling tasks of equivalent complexity from its queue is observed, the scheduler may infer that the client's local load is increasing, or that there is a problem with the client server communication. In either case, the server may decide to redistribute the tasks in that clients queue to ensure faster throughput.

A machine intending to gracefully depart from WebCom will perform a hard intentional disconnect. This process causes any clients of the machine to be redirected to a specified server and informs the machines server, if it exists, of its intention to leave. The appropriate connections are closed and removed from the respective descriptors. When a connection closes, any pending tasks or results waiting to be transmitted over that connection will be handled either by the Load Balancing or Fault Tolerance Module.

A soft intentional disconnection may be issued when a machine decides to process tasks scheduled to it offline. A soft disconnection indicates that a client is temporarily unavailable. It could, of course, be fully available from the perspective of other connections. A connection marked as a soft disconnection is not physically removed from its associated descriptors. Tasks may still be scheduled for transmission over

these connections in the expectation of them being restored at a future time. A connection that changes its disconnection mode from soft to hard will result in any pending tasks being handled by the Load Balancing and Fault Tolerance Module as before. Both hard and soft intentional disconnections are issued only by outgoing connections.

The availability status of a connection is decided at the outgoing descriptor and is propagated to the machine to which it is connected. The machine is then obliged to take this information into account in communicating over its associated return connection. A client can indicate different status conditions on each connection. Therefore, a client may be free from the perspective of one machine, busy from the perspective of a second and unavailable from the perspective of a third. Indeed, since availability is determined per connection, two or more connections between two machines could all indicate a different availability status. This can be used to dynamically build priority into the communication channels.

3.2.2 Compute Engine Module

The Compute Engine Module facilitates a Computing Engine's interaction with WebCom. Any Computing Engine may be employed, provided its implementation adheres to a specific interface.

Messages destined for the Compute Engine take one of two formats: They are either results of existing tasks or requests to execute a new task. Using the Condensed Graphs computing engine, requests to execute a new task cause an associated instruction to be executed. When the instruction to be executed represents a Condensed Graph, the appropriate Condensed Graph is allocated and begins execution. At the time of graph allocation, it is possible that the Condensed Graph representing the task is not available. This will raise an error condition and cause the execution to be terminated. To ensure this condition does not arise, the codebase for all graph instructions has to be preinstalled. Where graph definitions are not available, the use of an advanced Communication Manager Module triggers the fetching of the codebase from the remote machine identified in the message. Using this mechanism ensures the correct configuration of the metacomputer at all times.

WebCom facilitates the integration of partial results into a computation. Prior to executing the task, a list of partial results is examined to determine if an exact replica of the instruction has been executed previously. If so, the result of the previously executed instruction is incorporated. This mechanism is only applicable to instructions that do not exhibit non deterministic behaviour.

When a node becomes ready to execute the Computing Engine places the associated node on the Backplane for subsequent scheduling and possible distribution. All executable tasks uncovered by the Computing Engine are passed to the Backplane for scheduling. The Load Balancing Module determines if the task is to be executed locally or remotely. If it is to be executed locally, the task is resent to the Compute Engine via a message. This mechanism moves the control of task execution to the Load Balancing and Scheduling modules. The only requirement of the Compute Engine Module is to present tasks for execution.

When a graph has finished executing the Computing Engine Module requests the backplane to send the result to a non local address, via the communications manager

module. Subsequently, certain cleanup operations have to be performed. All memory occupied by the graph is freed for subsequent use. If the graph was executing within a self contained process, that process is destroyed to free up CPU resources. Where the graphs are executing within a shared process, a different cleanup is provided. Graph memory is still freed up, but once all graphs have completed execution, the process itself is only suspended. This allows subsequent graphs that may be executed to resume execution of the suspended process.

3.2.3 Fault Tolerance Module

An integral component of WebCom is its Fault Tolerance Module. Faults can be detected and corrected easily. Benefits of the Condensed Graphs model of computation and WebCom's fault tolerance mechanism include the utilisation of referential transparency, unique processor replacement strategies for failed processors, detailed logging of instructions and results. For more information of WebCom's fault tolerance mechanisms the reader is directed to [34, 45].

3.2.4 Load Balancing Module

The Load Balancing Module will decide where tasks are executed. WebCom requires the Load Balancing Module to communicate directly with the Backplane and relies on it to provide information on the status of the network by interacting with the Communication Manager Module. Different Load Balancing Module may employ a number of separate strategies taking account of issues such as security restrictions, specialised resource locations and access reliability (less reliable clients may be chosen to execute speculative tasks[47], whose completion or otherwise may not affect the critical path of the computation).

To maximise its effectiveness, a Load Balancing Module should exploit profile information[46] of attached clients. For example, in suggesting a location to execute a particular task which is an x86 binary, it would consider not only those x86 clients but also those capable of interpreting the code with sufficient speed. In addition to these generic considerations different Load Balancing modules can be employed to implement specific load balancing algorithms.

3.2.5 Scheduling Module

The Scheduling Module decides when tasks are to be executed once they are uncovered by the Processing Module. Currently the functionality of this module is trivial since each task is scheduled for execution as soon as it is uncovered. The Condensed Graphs model, however provides some unique opportunities for implementing scheduling schemas. Critical path analysis may be used, speculative computations may be assigned, reduced priorities and targeted instructions can be sequenced to optimise available resources. For more information on WebCom's scheduling and load balancing mechanisms the reader is directed to [53].

3.2.6 Security Manager Module

The default implementation of WebCom provides a *stub* implementation of the Security Manager Module. This allows for the load balancing, scheduling and fault tolerance

modules the freedom of not having to deal with security related issues. When a task is to be scheduled to a client the Security Manager Module is requested to return a list of candidate clients. The load balancer module then decided on a suitable client from this pool. In the case of the stub implementation, all available clients are returned as potential candidates. Other Security Manager Modules supporting the Keynote[24] standard have also been implemented. Different WebCom units in the connectivity hierarchy can each employ their own specific or localised security manager module, thus providing a flexible and extensible approach to supporting security at the system level.

4 WebCom as Grid Middleware



Figure 7: Overview of Applications Execution using WebCom-G

WebCom as described in Section 3 provides the basis for the unification of existing middlewares such as Corba, EJB, COM, DCOM. Interaction with each middleware is

handled through appropriate compute engine plugins. WebCom provides a general plugin interface. By adhering to this interface plugins can be easily constructed for each of the modules outlined previously. Services offered by the different middlewares are interrogated and provided on a palette for the application developer to incorporate into their application. During application execution WebCom dynamically targets the execution of different middleware services to a WebCom unit that has an appropriate compute engine installed.

In a Grid environment specific compute engine plugins can be created to leverage existing Grid technologies such as Globus, for example. Section 4.1 briefly describes task execution within the Globus environment and Section 4.2 details how WebCom marshals Globus tasks. Similar strategies may be employed for other Grid Middlewares that are in existence.

4.1 Globus Execution

Globus provides the basic software infrastructure to build and maintain the Grid. As dynamic networked resources are widely spread across the world, information services play a vital role in providing Grid software infrastructure, discovering and monitoring resources for planning, developing and adopting applications. A crucial part of every Grid is the information services. The onus is on the information service to support the initial discovery and subsequent use of resources and services.

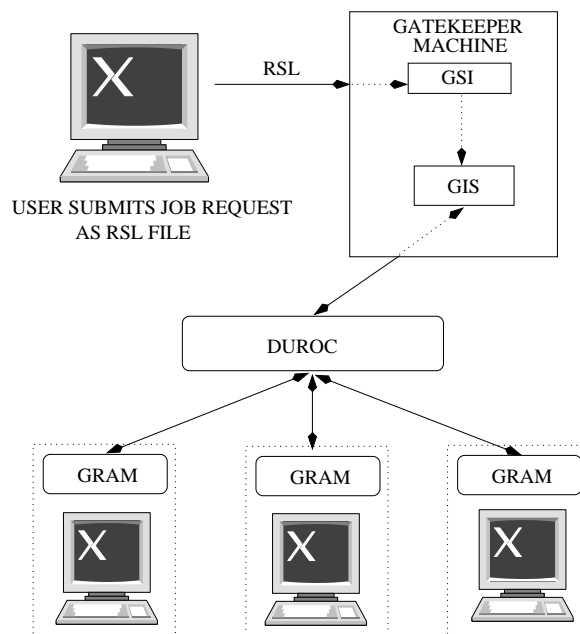


Figure 8: *Globus Execution model. A user generates an RSL script and submits it to the gatekeeper. The gatekeeper, in conjunction with DUROC and GRAM facilitate the distributed execution of requested services on the underlying nodes.*

An organisation running Globus will host its resources in the Grid Information Service (GIS) which is running on its gatekeeper machine constituting the organisations entry point to the grid.

The GIS will typically contain both static and dynamic information reflecting the transient and heterogeneous nature of the grid platform. Static information includes

details of numbers and configurations of compute nodes. Dynamic features of the grid are captured in machine loads, storage and network information. Machines running Globus may use a default scheduler or more advanced schedulers such as those provided by Condor, LSF, PBS and Sun Grid Engine.

Typically, users authenticated via the Grid Security Infrastructure (GSI) create a Resource Specification Language (RSL) script representing the job they wish to execute on Globus. This script is then executed on the Gatekeeper machine, specifying the application to run and the physical node(s) the application should be executed on and any other relevant information. The gatekeeper contacts a job manager service which in turn decides where the application is executed. For distributed services, the job manager negotiates with the Dynamically Updated Request Online Co-allocator (DUROC) to find the location of the requested service. DUROC makes the decision as to where the application is executed by communicating with each machine's lower level Grid Resource Allocation Manager (GRAM). This information is communicated back to the job manager. The job manager will schedule the execution of the application according to its own policies. If no job manager is specified, then a default service is used. This is usually the "fork" command, which returns immediately. A typical grid configuration is shown in Figure 8.

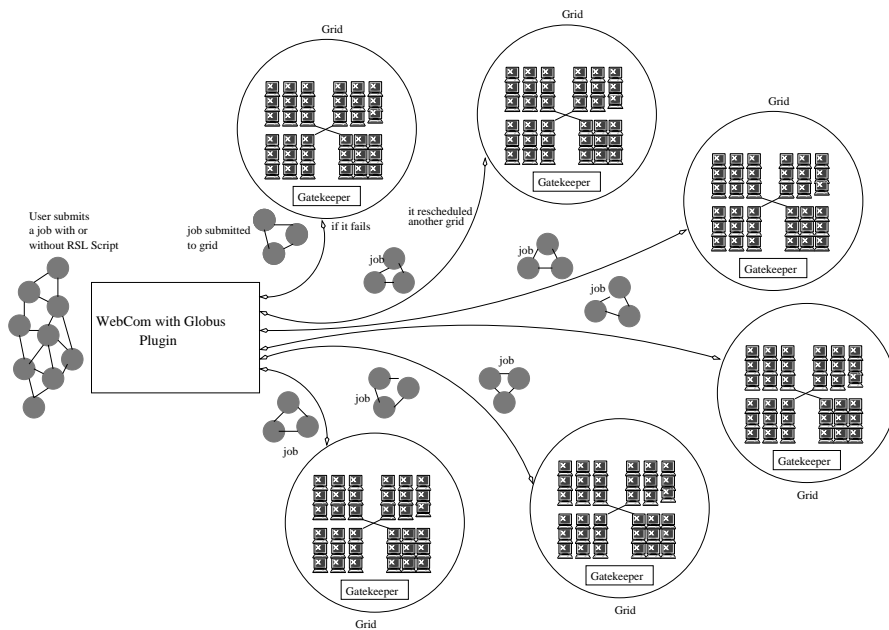


Figure 9: *WebCom-G: WebCom with a Globus/Grid plugin facilitates the execution of grid applications from within a condensed graph. Failures are detected and rescheduled using WebCom's fault tolerance and load balancing strategies.*

There are some disadvantages to using RSL scripts. Most notably, in a distributed execution if any node fails, the whole job fails and will have to be re-submitted by the user at a later time. Furthermore, there is no diagnostic information available to determine the cause of failure. Only resources known at execution time may be employed. There is no mechanism to facilitate job-resource dependencies. The resource must be available before the job is run, otherwise it fails. Within Globus, there is no checkpointing support, although some can be included programmatically or by specifying an appropriate job manager. This may not be feasible due to the

particular grid configuration used. Also, RSL is only suited to tightly coupled nodes, with permanent availability. If any of the required nodes are off-line, the job will fail.

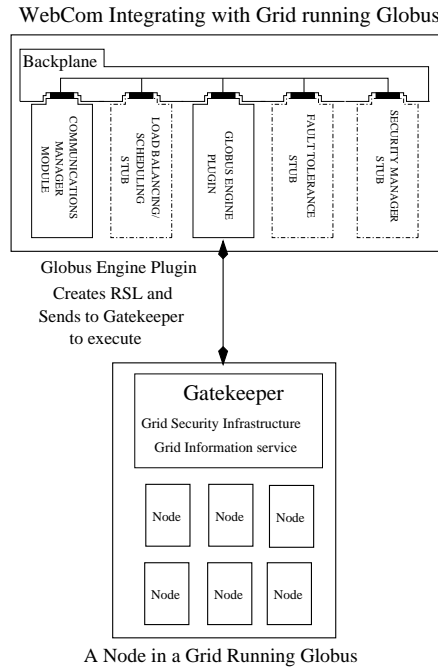


Figure 10: *WebCom-G: The Condensed Graphs compute engine plugin is replaced by the grid compute engine plugin. A targeted grid instruction is received by the compute engine. This dynamically creates the RSL file. The compute engine then causes the RSL script to be executed on the gatekeeper.*

4.2 Marshaling Globus

Due to the highly dynamic nature of resource availability of Globus, it is possible that a job is scheduled with the required resources, but at execution time some resources may no longer be available. This may be caused by hardware failure for example. This unavailability of resources will cause the job to fail. Thus, the scheduling of a large number of jobs to a remote site becomes hard to achieve successfully.

By including jobs as nodes in a Condensed Graph application, WebCom's node targeting mechanism can be used to send them to any existing job manager. Furthermore, the actual job configuration becomes dynamic: the complete configuration may be generated at runtime as a result of the execution of other nodes in the associated Condensed Graph, Figure 9. Even the name of the job could be dynamically uncovered. WebCom can interrogate the targeted grid to ensure the required resources are available before the node is sent for execution, or it may request the targeted grid to notify it when the required resources are ready. Alternatively, the node can be sent and put in a wait state until all resources are available. Using WebCom to schedule grid jobs in this manner also allows the utilisation of its fault tolerance mechanisms[45]. This is used to reschedule any applications targeted to the grid that may have failed. Therefore, once WebCom initiates a job it will complete provided the resources eventually become available.

By using WebCom, a whole grid can be viewed as a single WebCom unit with a specific computation engine plugin. This evolution of WebCom is the first step of producing a grid-enabled middleware: WebCom-G.

Multiple grids are themselves viewed as independent WebCom-G units. When an instruction is sent to WebCom-G all the information is supplied to either dynamically create and invoke an RSL script or to execute the job directly.

When a WebCom-G unit receives an instruction it is passed to the grid engine module. This module unwraps the instruction, creates the RSL script and directs the gatekeeper to execute it. Once the Gatekeeper has completed execution, the result is passed back to the unit that generated the instruction, Figure 10. As WebCom-G uses the underlying grid architectures, failures are detected only at the higher level. In this case WebCom-G's fault tolerance will cause the complete job to be re scheduled.

A WebCom compute engine plugin is implemented as a module that interacts with WebCom via a well defined interface. This gives a platform independent grid compute engine. Although a Condensed Graph may be developed on a platform that is not grid enabled, the execution of grid operations will be targeted to grid enabled platforms.

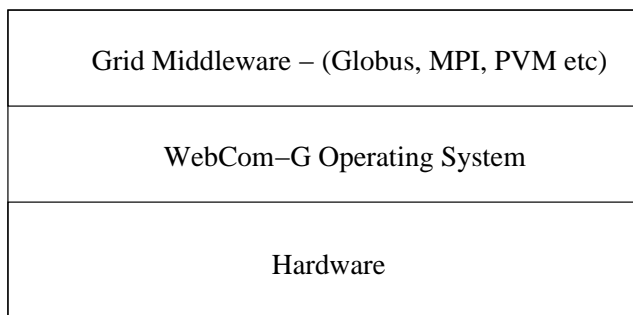


Figure 11: *WebCom-G extends the operating system functionality of the native hardware, interfacing with traditional Grid middleware.*

4.3 WebCom-G OS

The WebCom-G Operating System is proposed as a Grid Operating System. It is modular and constructed around a WebCom kernel, offering a rich suite of features to enable the Grid. It will utilise the tested benefits of the WebCom metacomputer and will leverage existing grid technologies such as Globus and MPI. The aim of the WebCom-G OS is to hide the low level details from the programmer while providing the benefits of distributed computing.

Resources in a heterogeneous environment like the Grid join and leave at times of their choosing. Gathering information and analyzing resources is important for resource management, scheduling, load balancing and fault tolerance. To build strict models (process management models, programming models, service oriented models) for quality of service, a proper Grid Operating System is needed. As with most operating systems there will be various components to provide the required functionality.

This section describes how WebCom-G retrieves resource information from a heterogeneous environment and how it will use this information in conjunction with

its resource management techniques to provide a home for different models which promise Quality of Service. Different methods for gathering and analyzing resource information to improve the Quality of Service requirement are discussed. A statistics analyser is introduced, which forms the basis for evaluating the costs of executing jobs on the grid. Finally, a comparison with Globus approach to static and dynamic information gathering is made.

The WebCom-G Operating System (Figure 11) is designed to operate between the system hardware and any installed grid middleware. It will interoperate with Globus (Versions 2.4 & 3) and with other middlewares such as PVM[29, 64], MPI[40, 41] and will work with standalone products such as Distributed.Net[19] and Seti@Home [12] clients.

The WebCom-G Operating System, illustrated in Figure 12, is designed to be modular. This design allows WebCom-G to be used in a number of different contexts – e.g., where WebCom-G is the only grid middleware installed, or to co-exist with say Globus, or MPI or both. If the system is configured to consist of multiple middlewares, each with its own information provider service, WebCom-G will automatically choose between them based on specific requirements of the application. However, the decision can be overridden by the programmer. WebCom-G will be able to treat different middlewares and users as Virtual Organisations, giving it control over task priorities.

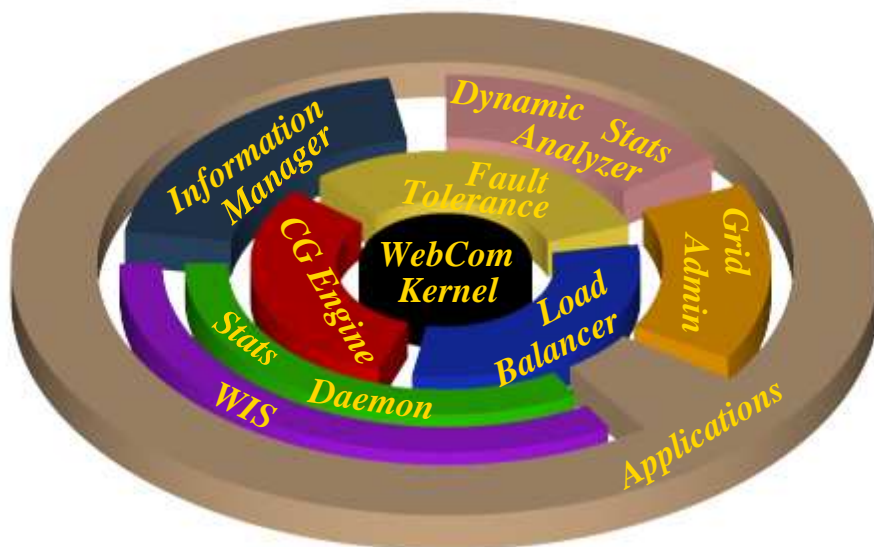


Figure 12: *WebCom-G OS architecture*

4.3.1 Components of the WebCom-G OS

Economy Status Analyser

The Economy Status Analyzer is a plug-in which gets the overall resource status of the machines comprising the grid from either the stats daemon or the information manager in a WebCom-G system or from the WIS if other information gathering middlewares (such as GRIS & GIIS & Ganglia in case of Globus) are present. This module will use various algorithms to evaluate independent cost and total cost of utilization. Scheduling compute intensive, time critical and data intensive jobs depends

on accurate and timely updates of resources. Based on this information, the ESA calculates the cost of executing the job. The quality of this service depends critically on the reported status information.

Stats Daemon

The Stats Daemon is a Linux daemon or Windows service, and uses standard system calls to retrieve system information - the Linux `sysinfo` kernel command or standard MFC function calls on Windows. It logs system usage to the hard drive, using one of several strategies e.g., fifo fixed filesize logging.

WebCom-G Information Module

The WebCom-G Information System (WIS) is the information gathering module within the WebCom-G OS. It consists of three parts, a low-level stats daemon to run directly on the hardware, a higher-level Information Manager, which can be aware of multiple hosts, and the WIS proper, which is capable of communicating with a single stats daemon, with an Information Manager or with Globus via the GIIS.

Grid Administration

The Grid Administration tool will allow administration by user or middleware. It will be possible to dynamically renice processes when unfair CPU allocation occurs, or to give priorities to particular jobs. Processes launched across the grid will be monitored and recorded, allowing the cluster manager to charge accordingly, or to prove quality of service or even to renice the processes on demand (allowing the user to purchase more CPU time or a higher priority).

4.4 WebCom-G IS

The WebCom-G Information Gathering Module (Fig. 13) consists of two components: The Stats Daemon and The Information Manager.

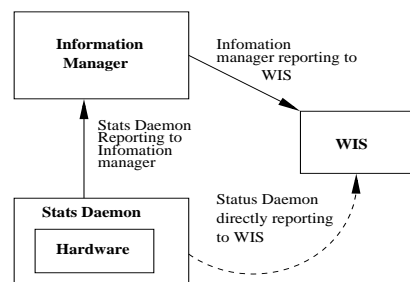


Figure 13: *Architecture of the WebCom-G Information Gathering Module*

The Stats Daemon sits on the hardware and reports to the Information Manager. The WIS sits above the Information Manager and so the WIS can choose to communicate with either the Stats Daemon directly (if it's only talking to a single machine) or it can talk to an Information Manager (which is a promoted stats daemon and so would have information about more than one machine) or to a third party information service provider such as a Globus GIIS. (see Figure 14)

4.4.1 The Stats Daemon

The Stats Daemon is an independent entity residing on the hardware, which gathers system information. It records information on the CPU load, the RAM (total+used),

Pagefiles (total+used), Hard Drives (total+used), Architecture, Processes running, Users, Networking and Kernel version. The Stats Daemon records system information to log files stored on the localhost and provides them over TCP/IP in XML or raw binary format.

4.4.2 The Information Manager

The Information Manager module provides detailed runtime and system information about a number of machines. It communicates with one or more Stats Daemons, gathering stats on each machine. It then sorts and partitions the data as required - e.g., giving an average machine load over a cluster for the Grid Administration module. It is possible to partition the information returned by middleware, person or uid and by machines or sub-clusters. This information can then be used by other WebCom-G modules, such as the Load Balancer or by the ESA. Retrieval of information at a given time period (e.g., given a time scale) is essential to determine the nature and health of connected resources and to execute jobs. The system of collection of data at fixed times may not be suitable for heterogeneous environments, which consist of dynamically changing resources. However, this method can be suitable for homogeneous nodes. It is also advantageous where the network load is high, since this model does not use any of the Globus services. Moreover, this information enables the WebCom Scheduler to execute a (less critical) job request on a more suitable resource.

WIS The WIS is the highest-level component of the WebCom-G Information Gathering Module. It will be able to query Globus to retrieve information (see Figure 14) supplied by Globus through the GIIS and Ganglia (i.e. CPU loads, memory usage etc). It will do this through one of two methods - either by building a new information provider to work with the GRIS or through the use of Commodity Grid kits (e.g., Java Cog Kit), which allow access to the information via the Globus Framework.

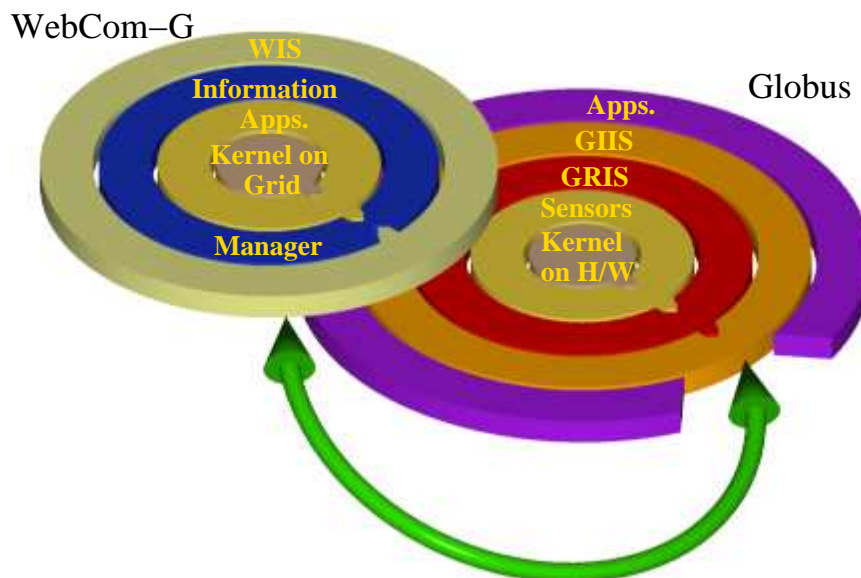


Figure 14: *WIS interoperability with Globus*

Each WebCom-G enabled machine will have the WebCom-G stats daemon running. The daemon will provide similar functionality to Globus, but will not rely on a single centralised stats server (GRIS, GIIS). Instead, it will use the client promotion feature of WebCom. This allows any client to be promoted to a master. Any Stats Daemon can be promoted to become an Information Manager. This Information Manager can then retrieve information about the cluster in which it is situated. This will eliminate bottlenecks and provide multiple entry points for clusters. Also the user can specify a “sub-cluster” within in a cluster and only retrieve information from and interact with those listed machines.

4.4.3 Globus Information Gathering

The Grid is a large distributed computing system formed by tightly or loosely coupled computers following a common set of rules for sharing, authentication and job execution thus enabling the creation of virtual organizations within the Grid environment. The Grid is build up of many virtual organizations; resource information about individual systems building the Grid plays an important role in load balancing, resource allocation & re-allocation and execution of jobs within them.

In the case of tightly coupled computers such as Clusters of Workstations there is central information such as the Network Information Service, which stores information about all the individual nodes forming the cluster. In the case of the Grid, information is hosted through information services. Information is the most crucial part of any Grid or meta-computing system. Information about the Grid system can be acquired through information services hosted at different levels in the Grid systems.

Globus is a collection of services toolkit (reference1) used to build computational Grids. The Grid built using Globus toolkit is formed of three supporting layers (reference 2) (1) Information service (2) Data Management services (3) Management services all under one security service. Information services provide information about Grid resources using utilities such as MDS, GRIS, and GIIS.

MDS, at the top level pinnacle provides information directory services for a Grid built using the Globus toolkit. It can be queried to discover the properties of the machines, (architectures, networks, processor availability, bandwidth and disk space). GRIS is a standard information service, which runs on all resources; it interfaces with LDAP and provides information about the resources. GRIS, the core information provider for MDS provides the resource information of the local system which includes platform type and instruction set architecture, OS version and type, CPU information, Memory, Network interface information and file system summary. Each compute node on the Globus grid runs a GRIS acting as white pages. The GIIS acts as a caching service for searching. Resources register with a GIIS, which in turn publishes the information when requested by the client. MDS uses an LDAP server and (via LDAP protocol and schema) interface for querying, generating, publishing, storing, searching and displaying of such middleware information. Thus the MDS provides tools necessary to build an LDAP based information tree for computational grids. GIIS provides a level of combining individual GRIS services to provide a single system image forming aggregate index service. Thus the GIIS forms yellow pages providing collective indexing and searching function of all the computational resources available in a Grid environment and across multiple virtual organizations

forming the Grid.

4.4.4 Comparing WIS and Globus

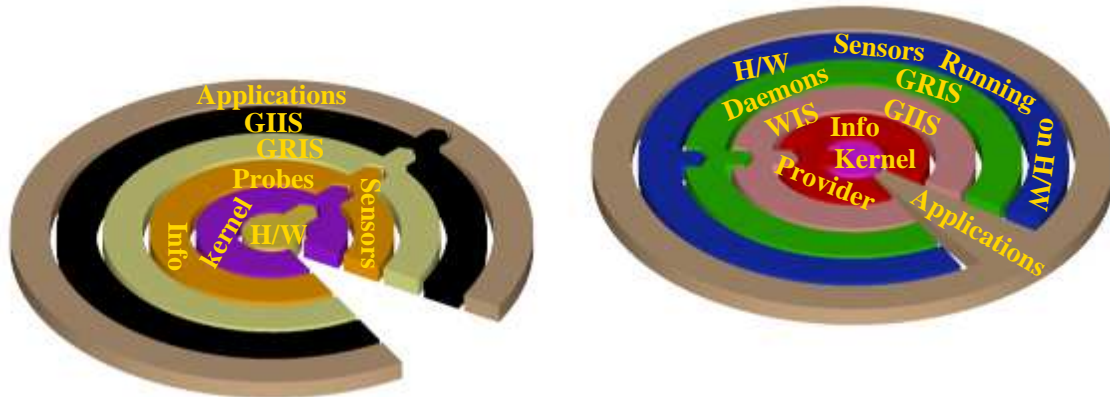


Figure 15: *Flow of control and lookup initiation in WebCom-G and Globus*

The application programmer must explicitly use the services provided by the Globus Toolkit to schedule his job. This must be done this at compile time - where the application is coded. Within the WebCom-G system, control is handed over to the WebCom-G kernel, which will automatically seek out necessary information from available services at runtime and will schedule the job accordingly. The application programmer does not need to specify how this is to be done, nor does he need to know how to do it.

Globus and WIS differ in ‘lookup initiation’ and flow of control(see Figure 15). In Globus the application programmer specifies the services to be invoked whereas in WIS the kernel will initiate lookups and choose the service to be invoked.

Globus relies on a single server - the GIIS to coordinate information gathering. This can lead to bottlenecks and provides a single point-of-failure. The WIS approach of promoting Stats Daemons to Information Managers avoids the problem of single point-of-failures.

5 Condensed Graphs Applications

5.1 XML

The development of distributed applications has been greatly simplified in recent years by the emergence of a variety of standards, tools and platforms. Traditional solutions such as RPC [10] and DCE [20] have been joined by a variety of new middleware standards and architecture independent development platforms. CORBA [30], a middleware standard developed by the Object Management Group, has seen widespread acceptance as a means for constructing distributed object-oriented applications in heterogeneous environments, using a variety of programming languages. Microsoft’s DCOM [18] fulfills a similar role in Windows environments. Sun Microsystems’ RMI [42] has been developed as a means of creating distributed

applications with the Java platform. The Java platform itself facilitates development for heterogeneous environments due to its architectural independence.

Although the various platforms and middlewares facilitate distributed application development, their use of binary protocols hinders interoperability and the ability of humans to understand the data being communicated. Due to the presence of firewalls, such middlewares are often unsuitable for applications distributed over the Internet, such as Web Services and Business to Business applications [16]. These factors, along with the realisation that proprietary binary protocols can lead to “vendor lock-in”, have led to a demand for an open, standardised, text-based format for exchanging data. XML [13], a restricted form of SGML developed by the World Wide Web Consortium (W3C), fills this role. XML simplifies the task of representing structured data in a clear, easily understandable (by both machines and humans) format. Furthermore, Document Type Definitions (DTDs) and XML Schemas can be used to enforce constraints on the structure and content of XML documents.

XML forms the basis for several distributed computing protocols and frameworks. SOAP (Simple Object Access Protocol) [2], developed by the W3C is a lightweight protocol for exchange of information in a decentralized, distributed environment. XML-RPC [69] is a simple remote procedure calling protocol. Microsoft’s BizTalk Framework [17] provides an XML framework for application integration and electronic commerce. Sun have provided a rich framework for the construction of Java/XML based distributed applications, including XML processing (JAXP), XML binding for Java objects (JAXB), XML messaging (JAXM), XML registries (JAXR) and remote procedure calls with XML (JAX-RPC) [36]. These technologies, due to their open, text-based nature, can utilise existing internet protocols such as HTTP and SMTP, bypassing firewalls and enabling Internet based applications.

WebCom-G utilizes XML for a variety of purposes, most notably as a means of specifying computations and also for communicating instructions and results. The advantages of using XML for these tasks are manifold. Firstly, XML can be generated and manipulated easily by both humans and machines, allowing users working on the same project to create and edit definitions using a variety of tools or by editing the XML files themselves. Secondly, the validity of generated documents can be easily verified using a predefined schema, greatly simplifying the code necessary to generate and parse definition documents. Finally, XSLT (XSL Transformations) [3], a rule-based framework for selecting and processing XML document content and transforming it into new XML content, allows for the easy translation of documents to and from other DTDs and schemas.

5.1.1 Defining \mathcal{CG} s

A \mathcal{CG} definition document consists of a set of \mathcal{CG} definitions and an optional set of data data structures, both of which are expressed in XML. Although generalised graph modelling schemas such as XGMML (eXtensible Graph Markup and Modelling Language) [56] or GXL (Graph Exchange Language) [57], it was felt that a custom \mathcal{CG} definition schema would provide a cleaner, more concise interface. In any case, XSLT filters could be written to convert to/from other formats if required. The use of XML namespaces was also rejected in order to preserve clarity. Again, namespace support could be added via XSLT filtering should the need arise.

\mathcal{CG} s are represented by a list of nodes, where each node is comprised of a set of input ports, a set of output ports and an operator. The structure of the document reflects the fact that input ports may have operands associated them, and similarly output ports may have destinations associated with them. The schema attempts where possible to impose structural constraints on the graph. For example, the schema ensures that each \mathcal{CG} has exactly one enter node and one exit node, and that these appear as the first and last nodes in the list respectively. Input and output ports may be statically typed using an optional element within the `<inputport>` and `<outputport>` elements respectively, allowing type consistency to be checked before a graph is executed.

The data structure definitions allow structured data to be communicated between WebCom-G units in a completely platform independent fashion. The creation and manipulation of WebCom-G data structures in Java is simplified through the use of JAXB (Java XML Binding) [66], which provides a schema compiler and runtime framework to support a two-way mapping between XML documents and Java objects. This solution allows for the dependency of WebCom-G of Java technology to be broken without forcing developers to write cumbersome marshalling/unmarshalling code for data structures expressed as XML. Presently, JAXB will only compile DTDs, but support for XML Schemas is planned in the next release.

5.1.2 Exception Handling

Exceptions [11], allow special cases that may arise during the execution of a computation to be dealt with in a more natural and flexible way than the rigorous checking of return codes. A WebCom-G exception is simply an XML document describing the category of the exception, the name of the node that raised the exception, and a description. Exceptions in WebCom-G can be returned by instructions in place of a result if an unusual situation arises that the node is unable to deal with. Exceptions may also be raised by the Triple Manager, for instance when an instruction fails to catch a Java exception and is forced to exit.

WebCom-G provides a facility for parts of a computation to be rolled back if necessary. Nodes may be declared in a \mathcal{CG} definition file with an extra `<undoable/>` element. This signifies to the Compute Engine that the implementation of the node provides an `undo()` as well as an `execute()` method. A special `begintagging` node directs the Compute Engine to begin storing execution details subsequent of undoable instructions, and storing them with a specified tag, until a corresponding `endtagging` instruction is executed. Another predefined node, `rollback`, will undo the execution of all instructions stored with a specified tag. Instructions may be assigned more than one tag, allowing for `begintagging` and `endtagging` nodes to be nested, but each instruction's corresponding `undo()` method will only be called once in the event of multiple executions of `rollback`. This simple mechanism allows for \mathcal{CG} s that produce side-effects to be speculatively executed.

Exception handling is performed by the `try` \mathcal{CG} which accepts two arguments: the first is the \mathcal{CG} to be executed, the second is an instance of the `ExceptionHandler` data type. The \mathcal{CG} argument is executed speculatively using a predefined `execute` instruction. If a node raises an exception during the execution of the \mathcal{CG} argument, `execute` will halt the computation and return the exception. If the \mathcal{CG} executes

successfully, the result of the execution will be returned. If `execute` returns an exception, the undoable instructions generated by the sub-computation are rolled back and the execution of the appropriate exception handler is coerced.

5.1.3 Pickling Computations

Pickling is the process of creating a serialized representation of objects [58]. In this case, the objects in question comprise an executing WebCom-G computation, and the serialized representation will consist of an XML document. Essentially, this allows the state of a computation to be preserved. The computation could be, for example, stored in a database and reactivated at a later date (a feature that could prove useful during the execution of workflows that block for long periods of time). Alternatively, the computation could be transported to another environment and executed there, if the necessary resources were available.

In order to reconstruct the state of a WebCom-G computation, three pieces of information are needed: the V-Graph, the set of \mathcal{CG} definitions from which the computation was created, and the set of tagged undoable instructions executed so far. The V-Graph represents the current state of the computation. The undoable instructions set is required in case a situation arises where parts of the computation need to be rolled back, such as when an exception is raised. The graph definitions are necessary to progress the computation; if a condensed node is executed, it cannot be evaporated unless its definition is available. These pieces of information are usually not contained in any single location; rather they are distributed throughout the various WebCom-G units partaking in the computation and must be reconstructed.

Before a computation can be pickled, it must be frozen. This can be partially achieved by instructing the Compute Engine of all WebCom-G units participating in the computation to cease scheduling new instructions. The situation is complicated by the problem of deciding what to do with partially executed instructions, particularly instructions that may take indefinite periods of time to complete execution. To allow for this problem, three types of halting messages may be sent to the Compute Engines: The first instructs the Compute Engines to report their state immediately and to terminate any currently executing instructions. The second instructs the Compute Engines to wait indefinitely for all instructions to finish execution. The third allows for a timeout period to be specified before instructions are terminated prematurely.

Once all the participating clients have responded, a complete representation of the computation is now available in the root WebCom-G unit. The V-Graph, the definition graphs and the set of undoable instructions are then converted to an XML document. Since the V-Graph is itself a \mathcal{CG} it is stored according to the standard \mathcal{CG} definition schema, as are the definition graphs. The undoable instructions set contains the instruction name, tag list, and arguments for each undoable instruction.

5.1.4 Code Distribution

The code necessary to execute an instruction is contained in a Java bytecode (class) file, which may in turn depend on other class files. Sets of related class files are stored in WebCom-G libraries. A library consists of a set of class files and an XML document describing the instructions contained therein. A \mathcal{CG} definition document may also be

included. Libraries are distributed in cryptographically signed Java Archive (JAR) files.

A small collection of commonly used libraries is included with the default WebCom-G installation. Other libraries are installed as needed from WebCom-G Code Repositories, which are centralized storage locations for WebCom-G libraries. When a WebCom-G unit is assigned an instruction, instructions are specified by a library name and an instruction name, it checks if the required library has already been installed. In the event that the required library is not installed, the unit will connect sequentially to a configuration-dependent list of repositories and attempt to retrieve it. The first repository contacted would most likely be a server in local environment containing locally developed private libraries and mirrors of commonly used publically available libraries. Alternative repositories could be globally accessible collections of publically available libraries, similar to PERL's CPAN (Comprehensive Perl Archive Network)[1] system. The authenticity and integrity of libraries downloaded from untrusted sources can be verified using their cryptographic signatures. Different versions of libraries are stored and referenced separately, so backwards compatibility does not need to be maintained between versions. A facility is available for updating libraries so that bugs, security vulnerabilities, etc. can be fixed without forcing migration to a newer version and potentially breaking binary compatibility with other libraries. An outline of a library description file is as follows:

```
<webcomglibrary>
  <libraryname>Example Library</libraryname>
  <libraryversion>1.0</libraryversion>

  <instruction name="Instruction 1">
    <instructiontype>atomic</instructiontype>
    <classname>com.example.Instruction1</classname>
  </instruction>

  <instruction name="CG 1">
    <instructiontype>condensed</instructiontype>
    <graphfile>orderprocessing.xml</graphfile>
  </instruction>
  .
  .
  .
</webcomglibrary>
```

6 Looking Forward

It is our opinion that the Grid must evolve to a point where it is invisible. From a programmers perspective this means that all architecture details are handled implicitly by a Grid Operating System. From the users point of view, apart from extended functionality and capability, the execution of a Grid Application should be indistinguishable from the same code executed on a local machine. For this to happen,

grid middleware must adopt the character of a grid operating system and many, if not all, of the issues that make grid programming and grid use difficult will be mitigated.

This chapter presents the evolution of WebCom, from a metacomputer to middleware, to a Grid middleware, to a Grid Operating System, and details our initial investigations into marshalling third party middleware such as Globus. The underlying philosophy of the WebCom-G System is to hide the Grid, by providing a vertically integrated solution from application to hardware while maintaining interoperability with existing Grid technologies. Available services will be exploited, increasing functionality and effecting interoperability.

The prime focus of the WebCom-G activity in the coming years will be to support application programmers and computational scientists working in the virtual organisations of Grid-Ireland. These include CosmoGrid and MarineGrid whose activities require both compute intensive and data intensive support.

The measure of success of the WebCom-G system will be the extent to which it allows its programming community to concentrate on application problem specification rather than problem implementation.

Acknowledgements

This work is partly funded by Science Foundation Ireland under the WebCom-G project and the Higher Education Authority under the Cosmogrid project.

References

- [1] Comprehensive Perl Archive Network. <http://www.cpan.org>.
- [2] Simple Object Access Protocol (SOAP) 1.1. W3C Note 08 May 2000.
- [3] XSL Transformations (XSLT) Version 1.0. W3C Recommendation 16 November 1999.
- [4] A. D. Alexandrov, M. Ibel, K. E. Schauser, and C. J. Scheiman. SuperWeb: Research Issues in Java-Based Global Computing. *Concurrency: Practice and Experience*, June 1997.
- [5] A. D. Alexandrov, M. Ibel, K. E. Schauser, and C. J. Scheiman. SuperWeb: Towards a Global Web-Based Parallel Computing Infrastructure. In *11th International Parallel Processing Symposium*, April 1997.
- [6] Arvind and Kim P. Gostelow. A Computer Capable of Exchanging Processors for Time. Information Processing 77 Proceedings of IFIP Congress 77 Pages 849-853, Toronto, Canada, August 1977.
- [7] A. Avizienis. Fault-Tolerance: The survival attribute of digital systems. In *IEEE, Vol 66, No. 10*, pages 1109–1125, 1978.
- [8] Arash Baratloo, Mehmet Karul, Zvi Kedem, and Peter Wyckoff. Charlotte: Metacomputing on the Web. 9th International Conference on Parallel and Distributed Computing Systems, 1996.

- [9] Wolfgang Becker. Dynamic Load Balancing for Parallel Database Processing. Faculty Report No. 1997/08, Institute of Parallel and Distributed High-Performance Systems(IPVR), University of Stuttgart, Germany.
- [10] A.D. Birrell and B.J. Nelson. Implementing Remote Procedure Calls. *ACM Transactions on Computer Systems*, 2(1):39-59, February 1984.
- [11] Alexander Borgida. Language Features for Flexible Handling of Exceptions in Information Systems. *ACM Transactions on Database Systems*, 10(4):565-603, 1985.
- [12] S. Bower, J. Cobb, D. Gedye, D. Anderson, W.T. Sullivan(III), and D. Werthimer. A new major SETI project based on Project Serendip data and 100,000 personal computers. In *Proceedings of the Fifth International Conference on Bioastronomy*, 1997.
- [13] Tim Bray, Jean Paoli, C.M. Sperberg-McQueen, and Eve Maler. Extensible Markup Language (XML) 1.0 (Second Edition). W3C Recommendation 6 October 2000.
- [14] Marian Bubak and Pawel Paszczak. HYDRA - Decentralised and Adaptive Approach to Distributed Computing. Workshop on Applied Parallel Computing, Bergen, Norway, June 18-21, 2000.
- [15] P. Cappello, B. O. Christiansen, M. F. Ionescu, M. O. Neary, K. E. Schauer, and D. Wu. Javelin: Internet-Based Parallel Computing Using Java. In Geoffrey C. Fox and Wei Li, editors, *ACM Workshop on Java for Science and Engineering Computation*, June 1997.
- [16] Etham Cerami. *Web Services Essentials*. O'Reilly & Associates, March 2002.
- [17] Microsoft Corporation. BizTalk Framework 2.0: Document and Message Specification. December 2000.
- [18] Microsoft Corporation. DCOM Technical Overview. November 1996.
- [19] Distributed.Net.
<http://www.distributed.net>.
- [20] Open Group Product Documentation. DCE 1.2.2 Introduction to OSF DCE. November 1997.
- [21] D. H. J Epema, Miron Livny, R. van Dantzig, X. Evers, and Jim Pruyne. A Worldwide Flock of Condors: Load Sharing among Workstation Clusters. *Journal on Future Generations of Computer Systems*, Volume 12, 1996.
- [22] Cristiana Amza et al. TreadMarks: Shared Memory Computing on Networks of Workstations. *IEEE Computer*, Pages 18-28, Vol. 29, No. 2, February 1996.
- [23] Luis F. G. Sarmenta et al. Bayanihan Computing .NET: Grid Computing with XML Web Services. Workshop on Global and Peer-to-Peer Computing at the 2nd IEEE International Symposium on Cluster Computing and the Grid (CCGrid 02), Berlin, Germany, May 2002.

- [24] M. Blaze et al. The Keynote Trust Management System, Version 2. Internet Request for Comments 2704. September 1999.
- [25] I. Foster and C. Kesselman. Globus: A Metacomputing Infrastructure Toolkit. *International Journal of Supercomputer Applications*, 11(2):115-128, 1997.
- [26] I. Foster and C. Kesselman. The Grid: Blueprint for a New Computing Infrastructure. Published by Morgan Kaufmann Publishers inc. ISBN:1-55860-475-8.
- [27] Robert Norman Frank Harary and Dorwin Cartwright. *Structural Models: An Introduction to the Theory of Directed Graphs*. John Wiley and Sons, 1969.
- [28] J. L. Gaudiot and C.S. Raghavendra. Fault-Tolerance and Data-Flow Systems. In *5th international conference on Distributed Computing Systems*, Denver, Colorado, May 13-17 1985.
- [29] Al Geist, Adam Beguelin, Jack Dongarra, Weicheng Jiang, Robert Manchek, and Vaidyalingam S. Sunderam. *PVM: Parallel Virtual Machine: A Users' Guide and Tutorial for Networked Parallel Computing*. MIT Press, Cambridge, MA, USA, 1994.
- [30] The Object Management Group. The Common Object Request Broker: Architecture and Specification. Revision 2.6, December 2001.
- [31] J.R. Gurd, C. C. Kirkham, and I. Watson. The Manchester Prototype Dataflow Computer. *Communications of the ACM*, 28(1):34-52, January 1985.
- [32] Simon Peyton Jones and Philip Wadler. Imperative Functional Programming. 20th ACM Symposium on Principles of Programming Languages (POPL'93), Charleston, Jan 1993, pp71-84.
- [33] Mehmet Karul. Metacomputing and Resource Allocation on the World Wide Web. Phd thesis, New York University, May 1998.
- [34] James J. Kennedy. *Design and Implementation of an N-Tier Metacomputer with Decentralised Fault Tolerance*. PhD thesis, University College Cork, National University of Ireland, 2004.
- [35] Chen L. and Avizienis A. N-Version Programming: A Fault-Tolerance Approach to Reliability of Software Operation. *Proceedings of FTCS 8*, 1978, pp.3-9.
- [36] Dario Laverde. *Developing Web Services with Java APIs for XML(JAX Pack)*. Syngress Publishing, March 2002.
- [37] Kwong-Sak Leung, Kin-Hong Lee, and Yuk-Yin Wong. DJM: A Global Distributed Virtual Machine on the Internet. *Software Practice and Experience*, 28(12), October 1998.
- [38] Michael J. Lewis and Andrew Grimshaw. The Core Legion Object Model. *Proceedings of the fifth IEEE International Symposium on High Performance Distributed Computing*.

- [39] M. O. Rabin M. A. Bender. Online Scheduling of Parallel Programs on Heterogeneous Systems with Applications to Cilk. *Theory of Computing Systems Special Issue on SPAA '00*, 35: 289-304, 2002.
- [40] Message Passing Interface Forum. MPI: A Message-Passing Interface Standard. *The International Journal of Supercomputer Applications and High-Performance Computing*, 8, 1994.
- [41] Message Passing Interface Forum. MPI-2: Extensions to the Message-Passing Interface. Draft proposal, March 1996.
- [42] Sun Microsystems. Remote Method Invocation. <http://www.javasoft.com/products/jdk/1.1/docs/guide/rmi/>.
- [43] Eugenio Moggi. Computational Lambda-calculus and Monads. Tech. Report ECS-LFCS-88-66, Edinburgh Univ., 1988.
- [44] John P. Morrison, Brian Clayton, and Adarsh Patil. Comparison of WebCom in the context of Job Management Systems. *International Symposium of Parallel and Distributed Computing (ISPDC 2002)*, Iasi, Romania, July 17-20, 2002.
- [45] John P. Morrison and James J. Kennedy. Fault Tolerance Mechanism Exploiting Referential Transparency in the Condensed Graphs Model. *Proceedings of the international conference on parallel and distributed processing techniques and applications (PDPTA 2001)*, Las Vegas, Nevada, June 25 - 28 2001.
- [46] John P. Morrison, David A. Power, and James J. Kennedy. Load balancing and fault tolerance in a condensed graphs based metacomputer. *The Journal of Internet Technologies. Special Issue on Web based Programming*, Vol. 3(4), PP. 221-234, December, 2002.
- [47] John P. Morrison and Martin Rem. Speculative Computing in the Condensed Graphs Machine. *proceedings of IWPC'99: University of Aizu, Japan*, 21-24 Sept 1999.
- [48] J.P. Morrison, Padraig J. O'Dowd, and Philip D. Healy. Searching RC5 Keyspaces with Distributed Reconfigurable Hardware. *ERSA 2003*, Las Vegas, June 23-26, 2003.
- [49] omg. Common Object Request Broker Architecture, July 1995. <http://www.omg.org>.
- [50] Rinus Plasmeijer and Marko van Eekelen. *Functional Programming and Parallel Graph Reduction*. ISBN: 0-201-41663-8 Addison-Wesley Publishers Ltd.
- [51] Platform Computing: Load Sharing Facility. <http://www.platform.com>.
- [52] Portable Batch System. <http://www.openpbs.org>.

- [53] David A. Power. *A Framework for: Heterogeneous Metacomputing, Load Balancing and Programming in WebCom*. PhD thesis, University College Cork, National University of Ireland, 2004.
- [54] David A. Power and John P. Morrison. Nectere: A General Purpose Environment for Computing on Cluster, Grids and the Internet. Proceedings of the International Conference on Parallel and Distributed Processing Techniques and Applications (PDPTA 2001), Las Vegas, Nevada, U.S.A., June 25-28, 2001.
- [55] Project JXTA.
<http://www.jxta.org>.
- [56] John Punin and Mukkai Krishnamoorthy (Editors). XGMML 1.0 Draft Specification. <http://www.cs.rpi.edu/puninj/XGMML/draft-xgmml.html>.
- [57] Richard C. Holt and Andreas Winter and Andy Schürr. GXL: Toward a Standard Exchange Format. In 7th Working Conference on Reverse Engineering, IEEE Computer Soc., 2000.
- [58] Riggs, Roger, Jim Waldo, Ann Wollrath, and Krishna Bharat. Pickling State in the JavaTM System. The 2nd USENIX Conference on Object-Oriented Technologies, 1996.
- [59] Philip A. Lisiecki Robert D. Blumofe. Adaptive and Reliable Parallel Computing on Networks of Workstations. *Proceedings of the USENIX 1997 Annual Technical Symposium*, January 1997.
- [60] Hirano S. HORB: Extended Execution of Java Programs. Proceedings of 1st int. conf. on World Wide Computing and its Applications (WWCA97), March 1997.
- [61] Luis F. G. Sarmenta. Bayanihan: Web-Based Volunteer Computing Using Java. 2nd International Conference on World-Wide Computing and its Applications (WWCA'98), Tsukuba, Japan, March 3-4, 1998.
- [62] Luis F. G. Sarmenta and Satoshi Hirano. Bayanihan: Building and Studying Web-Based Volunteer Computing Systems using Java. Future Generation Computer Systems Special Issue on Metacomputing. Elsevier, 1999.
- [63] Luis F. G. Sarmenta, Satoshi Hirano, and Stephen A. Ward. Towards Bayanihan: Building an Extensible Framework for Volunteer Computing Using Java. ACM 1998 Workshop on Java for High-Performance Network Computing, Palo Alto, California, Feb. 28 - Mar. 1, 1998.
- [64] Richard Sevenich. Parallel Processing Using PVM. *Linux Journal*, 45, January 1998.
- [65] Geoff Stoker, Brian S. White, Ellen Stackpole, T. J. Highley, and Marty Humphrey. Toward Realizable Restricted Delegation in Computational Grids. European High Performance Computing and Networking. Amsterdam, June 25-27, 2001.

- [66] Inc. Sun Microsystems. Java Architecture for XML Binding User's Guide. Early-Access Draft, May 2001.
- [67] The Globus Project.
<http://www.globus.org>.
- [68] Jerrell Watts and Stephen Taylor. A Practical Approach to Dynamic Load Balancing. Scalable Concurrent Programming Laboratory, Syracuse University.
- [69] Dave Winer. XML-RPC Specification. <http://www.xml-rpc.org/spec>.