Scalable Computing: Practice and Experience
Volume 6, Number 1, pp. 33–43. http://www.scpe.org
ISSN 1895-1767
© 2005 SWPS

THE ROLE OF XML WITHIN THE WEBCOM METACOMPUTING PLATFORM

JOHN P. MORRISON[*], PHILIP D. HEALY[*], DAVID A. POWER[*] AND KEITH J. POWER[*]

Abstract. Implementation details of the Nectere distributed computing platform are presented, focusing in particular on the benefits gained through the use of XML for exoressing, executing and pickling computations. The operation of various Nectere features implemented with the aid of XML are examined, including communication between Nectere servers, specifying computations, code distribution, and exception handling. The area of interoperability with common middleware protocols is also explored.

Key words. Nectere, WebCom, XML, Distributed Computing, Condensed Graphs

1. Introduction.

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 [1] and DCE [2] have been joined by a variety of new middleware standards and architecture independent development platforms. CORBA [3], 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 [4] fulfills a similar role in Windows environments. Sun Microsystems' RMI [5] 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. Microsoft have recently launched a similar, competing platform in the form of their .NET initiative [6].

Although the various platforms and middlewares described above facilitate distributed application development, their use of binary protocols hinders interoperability and the ability of humans to understand the data being communicated. Because of these limitations, such middlewares are often unsuitable for applications distributed over the Internet, such as Web Services and Business to Business applications [7]. These factors, along with the realization that proprietary binary protocols can lead to "vendor lock-in", have led to a demand for an open, standardized, text-based format for exchanging data. XML [8], 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) [9], developed by the W3C is a lightweight protocol for exchange of information in a decentralized, distributed environment. XML-RPC [10] is a simple remote procedure calling protocol. Microsoft's BizTalk Framework [11] 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) [12]. These technologies, due to their open, text-based nature, can utilize existing Internet protocols such as HTTP and SMTP, bypassing firewalls and enabling Internet based applications.

The field of *Metacomputing*, defined as *"the use of powerful computing resources transparently available to the user via a networked environment"* [13], has attracted considerable interest as a means of further simplifying distributed application development. Although this definition is quite vague, in practice metacomputing has come the mean the use of middlewares to present a collection of potentially diverse and geographically distributed computing resources transparently to the user as a single virtual computer. Examples of metacomputing environments are the Globus Metacomputing Toolkit [14], Legion [15] and WebCom (the focus of this paper). WebCom is a metacomputing environment that allows computations expressed as *Condensed Graphs* (see Section 2) to be executed on a variety of platforms in a secure, fault-tolerent manner. Load-balancing is also performed over the computing resources available without requiring any intervention on the part of the programmer. Originally designed as a means of creating *ad hoc* metacomputers from Java applets embedded in web pages, WebCom has since been developed into a general-purpose distributed computing environment suitable for the creation of grids. An extended version of WebCom, entitled WebCom-G [16], allows for interoperability with other Grid Computing platforms and includes support for legacy applications.

[*]Computer Science Dept., University College Cork, Ireland.

The organization of the remainder of this paper is as follows: Section 2 describes the Condensed Graphs model of computation that provides the underlying execution model for WebCom. Section 3 provides an overview of the design and operation of the WebCom metacomputer itself. The XML file format developed for specifying WebCom applications is presented in Section 4. The pickling of live WebCom computations is discussed in Section 5. A web service interface to WebCom is presented in Section 6. Some empirical results pertaining to the bandwidth savings achievable through the use of compressed XML compared to compressed serialized Java objects are provided in Section 7. Finally, conclusions and a discussion on future work is presented in Section 8.

2. Condensed Graphs. While being conceptually as simple as classical data-flow schemes [17, 18], the Condensed Graphs ($\mathcal{CG}$) model [19] 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 conveying 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, acting as input to successor nodes. Operand sets are used as 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 [20].) 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 at 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 complimentary 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 multinode $\mathcal{CG}$ 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, multinode $\mathcal{CG}$ operands represent partially evaluated expressions. The processing of condensed node and multinode 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 multinode $\mathcal{CG}$.

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. Fig. 2.1 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 $\mathcal{CG}$ firing rule takes accounts 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.

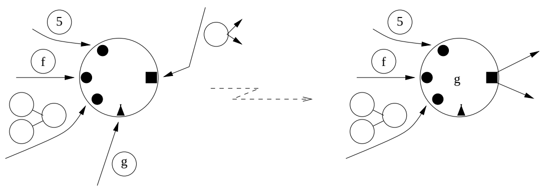


FIG. 2.1. *cgs congregating at a node to form an instruction.*

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 multinode $\mathcal{CG}$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 multinode $\mathcal{CG}$ 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 specialized 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 node 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 [21] but is not restricted to a purely lazy framework.

$\mathcal{CG}$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 multinode $\mathcal{CG}$ 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 already had 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 Fig. 2.2.

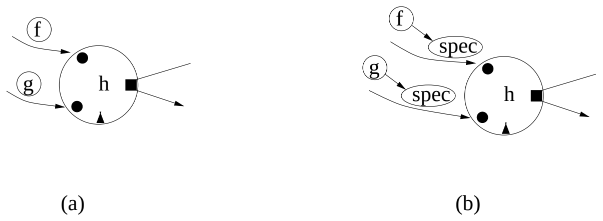


FIG. 2.2. *Increasing parallelism by speculatively evaluating operands.*

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 operands 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. Fig. 2.3 illustrates the process of fetching a stored structure.

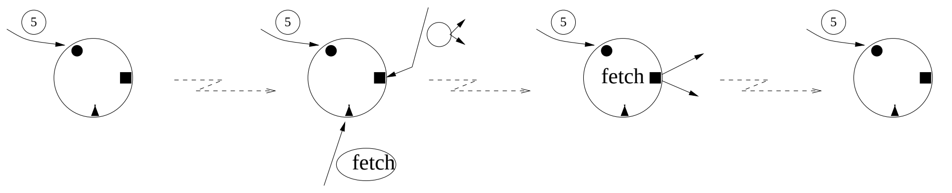


FIG. 2.3. *Sequence of events showing the fetching of a stored structure and subsequent node deconstruction.*

By statically constructing a $\mathcal{CG}$ to contain operators and destinations, the flow of operand $\mathcal{CG}$s sequence 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 $\mathcal{CG}$ models result from being able to exploit all of these evaluation strategies in the same computation, and dynamically move between them, using one single, uniform, formalism.

3. WebCom. Problem solving for parallel systems traditionally lay in the realm of message passing systems such as PVM and MPI on networks of distributed machines, or in the use of specialised variants of programming languages like Fortran and C on distributed shared memory supercomputers. The WebCom System [22, 23, 24, 25] detailed here relates more closely to message passing systems, although it is much more powerful. Message passing architectures normally involve the deployment of a codebase on client machines, and employ a master or server to transmit or *push* "messages" to these clients. WebCom supports this level of communication, but is unique in bootstrapping its codebase by *pulling* it from the server as required.

Technologies such as PVM, MPI and other metacomputing systems place the onus on the developer to implement complete parallel solutions. Such solutions require a vast knowledge on the programmer's part in understanding the problem to be solved, decomposing it into its parallel and sequential constituents, choosing and becoming proficient in a suitable implementation platform, and finally implementing necessary fault tolerance and load balancing/scheduling strategies to successfully complete the parallel application. Even relatively trivial problems tend to give rise to monolithic solutions requiring the process to be repeated for each problem to be solved.

WebCom removes much of these traditional considerations from the application developer; allowing solutions to be developed independently of the physical constraints of the underlying hardware. It achieves this by employing a two level architecture: the computing platform and the development environment. The computing platform is implemented as an Abstract Machine (AM), capable of executing applications expressed as Condensed Graphs. Expressing applications as Condensed Graphs greatly simplifies the design and construction of solutions to parallel problems. The Abstract Machine executes tasks on behalf of the server and returns results over dedicated sockets. The computing platform is responsible for managing the network connections, uncovering and scheduling tasks, maintaining a balanced load across the system and handling faults gracefully. Applications developed with the development environment are executed by the abstract machine. The development environment used is specific for Condensed Graphs. Instructions are typically composed of both sequential programs (also called atomic instructions) and Condensed nodes encapsulating graphs of interacting sequential programs. In effect, a Condensed Graph on WebCom represents a hierarchical job control and specification language. The same Condensed Graphs programs execute without change on a range of implementation platforms from silicon based Field Programmable Gate Arrays[26] to the WebCom metacomputer and the Grid.

Imposing a clear separation between the abstract machine and the development environment has many advantages. Fault tolerance, load balancing and the exploitation of available parallelism are all handled implicitly by WebCom abstract machines without the need for programmer intervention. Also, different strategies such as load balancing strategies can be used without having to redesign and recode any applications that may execute on the system. Removing the necessity for developers to implement the features mentioned previously, which can be considered as core to any distributed computing system, greatly simplifies and increases the rate of development of parallel applications.

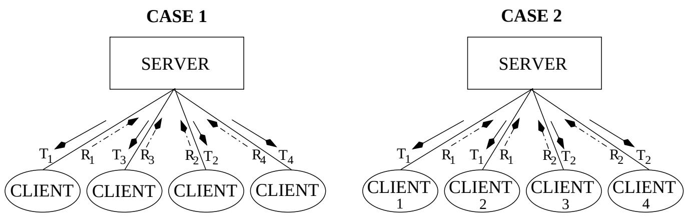


FIG. 3.1. *Traditional Server/Client Metacomputers consist of a two tier hierarchy. The server sends tasks to the clients.*

Physical compute nodes connect over a network to form a hierarchy. Metacomputing systems typically consist of one server (or master) and a number of clients. Tasks are communicated from the server to the clients, and results returned when task computation is completed. This topology represents a two tier unidirectional connectivity hierarchy. Tasks are only transmitted in one direction from the server to the clients and results of task execution transmitted back, as illustrated in Figure 3.1. Case 1 is reminiscent of the *"...@home"* projects like seti@home and genome@home. With these projects, a generic task is installed apriori on each computer by its local administrator. These tasks register with a server which subsequently supplies input data and gathers results on an ongoing basis. Fault tolerance is achieved by issuing the same input data to multiple tasks.

Case 2 depicts a more sophisticated arrangement. Clients in this architecture represent codebases which may be invoked with specific parameters by the server. Each client may comprise a different codebase, thereby requiring the server to target instructions. These systems typically invoke the same task on multiple available clients in an effort to minimise response time: the earliest result received for each task is used and all others are discarded.

Two tier systems have limited scalability, particularly when distributing fine grained tasks: the server will eventually become inundated with client connections and will expend a disproportionate effort in connection management. The seti@home project alleviates this problem by allowing clients to disconnect after receiving substantial data blocks that can take days to process. An advantage of two tier hierarchies is simple fault tolerance and load balancing strategies can be easily implemented.

WebCom also utilises a server/client model and World Wide Web technologies. In contrast, A WebCom client is an *Abstract Machine*(AM) whose codebase is populated on demand. In certain instances client machines may act as servers capable of distributing tasks to clients of its own. This feature gives rise to a multi-tier bidirectional topology. When employing the Condensed Graphs model of computation, the evolution of this topology reflects the unfolding of the Condensed Graph description of the application. An example WebCom connection topology is illustrated in Figure 3.2.

In certain circumstances a client may request its server to execute one or more tasks on its behalf. This mechanism allows maximum exploitation of available compute resources and gives rise to the evolution of peer to peer connectivity. Within a WebCom deployment, peer to peer connections may be established within sections of the hierarchy, resulting in the creation of a moderated peer to peer network. For deployments consisting of a small number of clients, a fully connected peer to peer topology may evolve. An example evolution of a moderated peer to peer network hierarchy is illustrated in Figure 3.3.

A deployment of WebCom consists of a number of AMs. When a computation is initiated one AM acts as the root server and the other AMs act as clients or *Promotable Clients.* The promotability of a client is determined by the deployment infrastructure. A promoted client acting as a server accepts incoming connections and executes tasks. Promotion[27, 28] occurs when the task passed to the client can be partitioned for subsequent distributed execution (i.e., when the task represents a Condensed Graph). It is possible for the server to be redirected or to act as a client of a promotable client. This situation normally results in the evolution of a peer

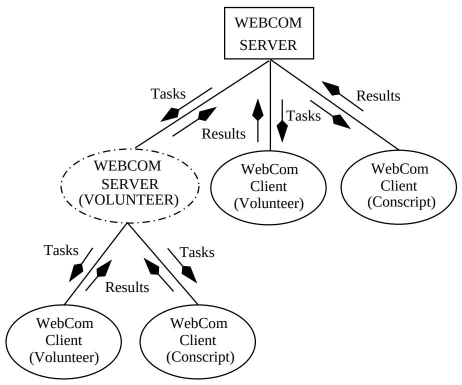

WEBCOM SERVER
Tasks
Results
Tasks
Results
WEBCOM SERVER (VOLUNTEER)
WebCom Client (Volunteer)
WebCom Client (Conscript)
Tasks
Results
Tasks
WebCom Client (Volunteer)
WebCom Client (Conscript)

Fig. 3.2. *Typical WebCom Client/Server connectivity. Each WebCom Abstract Machine can either volunteer cycles to a WebCom server, or be conscripted as a WebCom client through Cyclone.*

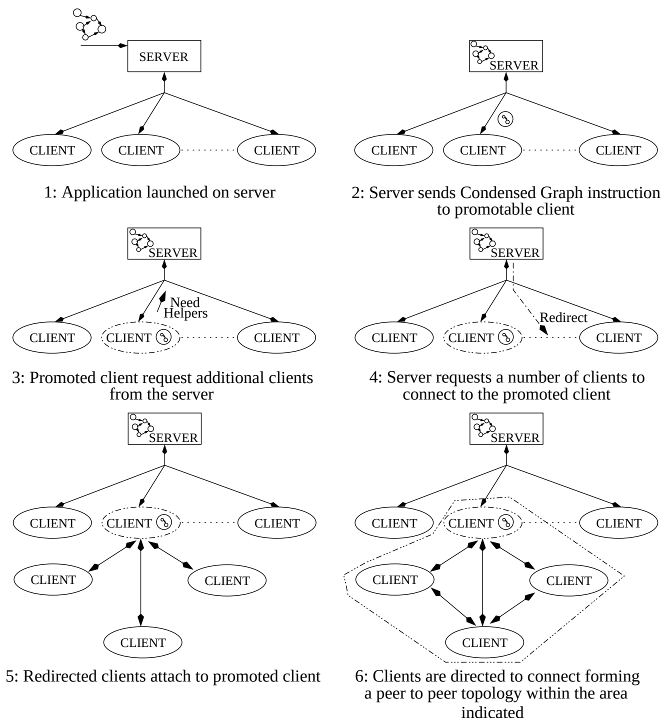

SERVER
CLIENT
CLIENT
CLIENT
1: Application launched on server
2: Server sends Condensed Graph instruction to promotable client
Need Helpers
3: Promoted client request additional clients from the server
Redirect
4: Server requests a number of clients to connect to the promoted client
5: Redirected clients attach to promoted client
6: Clients are directed to connect forming a peer to peer topology within the area indicated

Fig. 3.3. *Evolution of a Moderated Peer to Peer network. 1: Computation begins with 2 tier hierarchy. Graph execution begins on the server. 2: Server sends a Condensed Graph task to a client, causing it to be promoted. 3. The promoted client requests additional clients to be assigned to it by the server. This request may not be serviced. If not, the client continues executing the graph on its own. 4: The server issues a redirect directive to a number of its clients. 5: Clients redirected from the server connect as clients of the promoted client. 6: The promoted client directs its new clients to connect to each other forming a local peer to peer network.*

to peer topology. The return of a result may trigger the demotion of a promoted client, thus causing it to act once more as a normal client.

Furthermore, WebCom clients are uniquely comprised of both volunteers and conscripts. Volunteers donate compute cycles by instantiating a web based connection to a WebCom server. The AM, constrained to run in the browser's sandbox, is automatically downloaded and establishes communications with the server. Due to the constraints associated with executing within the sandbox, such as prohibiting inbound connections, volunteer clients are not promotable: they will only be sent primitive tasks for execution. Clients that have the WebCom AM pre-installed may also act as volunteers. These clients are promotable, as they are not constrained by the restrictions imposed by executing within a browser.

Intranet clients must be conscripted to WebCom by the Cyclone[29] server. Cyclone directs machines under its control to install the WebCom AM locally thereby causing them to attach to a predefined server and to act as promotable clients. These clients may be further redirected by that server.

The Abstract Machine(AM) consists of a number of modules including a Backplane and modules for Communications, Fault Tolerance, Load Balancing and Security. For better tuning to specific application areas, the AM allows every component to be dynamically interchangeable. The pluggable nature of each WebCom module provides great flexibility in the development and testing of new components. Certain components maybe necessary, while others could be highly specialised to particular application environments. A skeletal installation is composed of a Backplane module, a Communication Manager Module and a number of module stubs as illustrated in Figure 3.4.

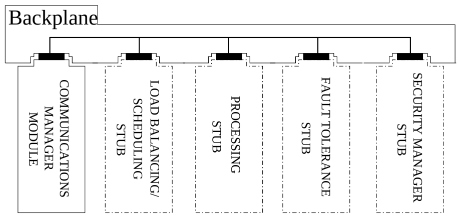


Fig. 3.4. *A minimal WebCom installation consists of a Backplane Module, a Communication Manager Module and a number of module stubs for Processing, Fault Tolerance, Load Balancing and Security.*

The Backplane acts as an interconnect and bus that co-ordinates the activities of the modules via a well defined interface. Inter module communication is carried out by placing a task on the Backplane. The Backplane interrogates the task to determine whether it should be routed to a local module or to another AM via the communications manager. This mechanism provides great flexibility especially as the mechanism can be applied transparently across the network: transforming the metacomputer into a collection of networked modules. This allows an arbitrary module to request information from local modules or modules installed on different abstract machines. For example, when making load balancing decisions, a server's Load Balancing Module can request information from the Load Balancing Module of each of its clients. A high level view of multiple AM's on a network is given in Figure 3.5

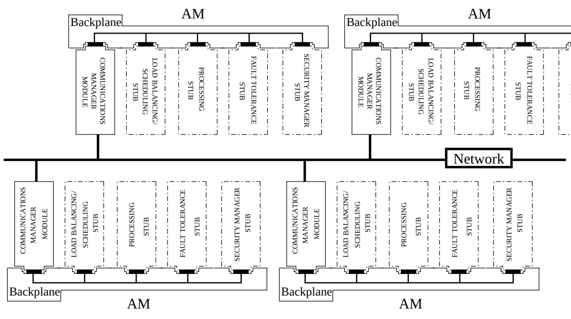


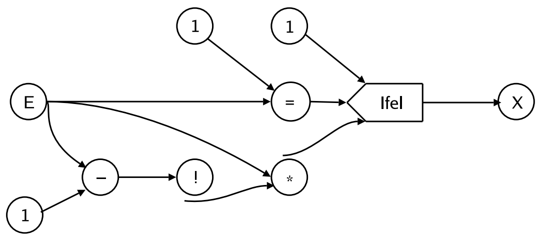
Fig. 3.5. *High level view of abstract machines connected over a network.*

Inter module and inter AM communication is carried out by the Backplane. A *message* representing a task, lies at the core of the communication structure. To summarise, when a module initiates communication with another local or remote module, a message is created specifying the task to be executed along with certain execution environment parameters. The message is placed on the Backplane which interrogates it to determine suitability for distribution. If the message is to be executed remotely it is passed to the Connection Manager Module for subsequent dispatch, otherwise it is passed to the appropriate local module. Upon receipt of a message object, a module immediately processes the associated task.

4. Graph Definition XML Format. Early versions of WebCom required the implementation of a Java class for every graph definition used by an application. As development progressed, it became clear that this approach suffered from a number of significant limitations, such as requiring knowledge of Java and an in-depth understanding of the opearation of the Condensed Graphs model on the part of the application developer. In order to simplify application development, an IDE was developed that allows Condensed Graph definitions to be expressed in a graphical fashion. Initially, the IDE maintained an internal representation of graph definitions that could be converted to Java code and subsequently compiled. However, applications other than the IDE that wished to generate graph definitions could not easily do so.

It became clear that a common method of expressing graph definitions was required that could be utilized directly both by humans and a variety of potential applications using a variety languages and APIs. XML was the obvious choice for this task due its text-based nature, platform independence and the widespread availability of XML tools across practically every platform. XML also offered the advantage of self-validation using DTDs and XML Schema.

Files written according to the XML format are composed of a a number of graph definitions, represented by `graphdef` elements enclosed in a root `definitions` element. Each `graphdef` element is in turn composed of a number `node` elements that specify the operand ports, operands, operator and destinations that comprise the node. A namespace (with `cg` used as the prefix) is used to allow graph definitions to be embedded in other XML documents and *vice versa*. An example graph and its equivalent XML representation is given in Figure 4.1. Live graph instances can be converted to the XML format via pickling (see Section 5, below).



```xml
<?xml version="1.0" standalone="no"?>
<!DOCTYPE cg:graphdefs SYSTEM "http://cuc.ucc.ie/xml/cg.dtd">

<cg:graphdefs xmlns:cg="http://cuc.ucc.ie/xml/cg">
  <cg:graphdef name="Factorial">
    <!-- E -->
    <cg:node name="E">
      <cg:operandport/>

      <cg:operatorport operator="webcom.cgengine.EnterOperator"/>

      <cg:destinationport>
        <cg:destination nodename="Eq0" portnumber="0"/>
        <cg:destination nodename="minus1" portnumber="0"/>
        <cg:destination nodename="mul" portnumber="0"/>
      </cg:destinationport>
    </cg:node>

    <!-- minus1 -->
    <cg:node name="minus1">
      <cg:operandport strictness="strict"/>
      <cg:operandport strictness="strict"/>

      <cg:operand portnumber="1" type="Integer" value="1"/>

      <cg:operatorport operator="webcom.nodes.core.MinusOp"/>

      <cg:destinationport>
        <cg:destination nodename="Factorial" portnumber="0"/>
      </cg:destinationport>
    </cg:node>

    <!-- factorial -->
    <cg:node name="Factorial">
      <cg:operandport strictness="strict"/>

      <cg:operatorport operator="Factorial"/>

      <cg:destinationport/>
    </cg:node>

    <!-- mul -->
    <cg:node name="mul">
      <cg:operandport strictness="strict"/>
      <cg:operandport strictness="strict"/>

      <cg:operand portnumber="1" type="Node" value="Factorial"/>

      <cg:operatorport operator="webcom.nodes.core.MultiplyOp"/>

      <cg:destinationport/>
    </cg:node>

    <!-- Eq0 -->
    <cg:node name="Eq0">
      <cg:operandport strictness="strict"/>
      <cg:operandport strictness="strict"/>

      <cg:operand portnumber="1" type="Integer" value="0"/>

      <cg:operatorport operator="webcom.nodes.core.EqualsOp"/>

      <cg:destinationport>
        <cg:destination nodename="ifel" portnumber="0"/>
      </cg:destinationport>
    </cg:node>

    <!-- ifel -->
    <cg:node name="ifel">
      <cg:operandport strictness="strict"/>
      <cg:operandport strictness="nonstrict"/>
      <cg:operandport strictness="nonstrict"/>

      <cg:operand portnumber="1" type="Integer" value="1"/>

      <cg:operand portnumber="2" type="Node" value="mul"/>

      <cg:operatorport operator="webcom.nodes.core.IfElOp"/>

      <cg:destinationport>
        <cg:destination nodename="X" portnumber="0"/>
      </cg:destinationport>
    </cg:node>

    <!-- X -->
    <cg:node name="X">
      <cg:operandport strictness="strict"/>

      <cg:operatorport operator="webcom.cgengine.ExitOperator"/>

      <cg:destinationport/>
    </cg:node>
  </cg:graphdef>
</cg:graphdefs>
```

FIG. 4.1. *A trivial example of a graph definition that calculates the factorial of its single argument, along with the equivalent XML document.*

5. Pickling Computations. Pickling is the process of creating a serialized representation of objects [30]. In this case, the objects in question comprise an executing WebCom 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 computations 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 computation, two pieces of information are needed: the V-Graph and the set of $\mathcal{CG}$ definitions from which the computation was created. The V-Graph represents the current state of the computation. 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 servers 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 engines 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 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 at the root server. The V-Graph and definition graphs 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.

6. Exposing WebCom as a Web Service. In order to facilitate the utilization of the WebCom platform across a variety of platforms and programming languages, it was decided to expose WebCom as a web service. In effect, this implied making one or more methods remotely available using SOAP and creating descriptions of these methods using WSDL. Due to the text-based nature of SOAP and the ability to use it over common protocols such as HTTP and SMTP, web services can be accessed in a platform-agnostic way, eliminating the need to reimplement APIs on different platforms and languages.

For simplicity, only two methods were exposed: `runGraph` and `runXmlJob`. `runGraph` accepts two parameters (a graph name and a parameter list) and executes the specified graph definition with the supplied parameters. The return value of the graph is returned if the computation finishes succesfully; otherwise an appropriate error message is returned. `runXmlJob` behaves identically to `runGraph` except that it accepts a graph definition document as an additional parameter, with the graph definitions contained in the document added to the computation's definition collection before execution commences. The newly-added definitions are propagated from the root WebCom instance to clients as required via the messaging system.

The Glue web service platform [31] was used to create and deploy the web service. This platform was chosen both for ease of development and deployment. Development is simplified through the use of reflection by the platform to determine the names and types of the methods to be exposed, allowing the web service to be defined with a minimum of coding effort. The corresponding WSDL (Web Services Description Language) descriptions are created automatically, eliminating the need to specify them by hand. Deployment is simplified by the fact that Glue is a standalone platform; there is no requirement for WebCom administrators to install and configure a separate web services container, a requirement of alternative web services platforms such as Apache Axis.

7. Empirical Data on XML Document Sizes. Given the verbosity of XML documents, some empirical data was gathered to determine the overhead imposed by representing $\mathcal{CG}$s as XML. Linear $\mathcal{CG}$s containing 1 to 1,000 nodes were specified as XML documents and loaded into a WebCom server before being converted to their serialized Java equivalents. This allowed the relative sizes of both representations to be compared. Since it would be possible to compress large graphs (such as those created by the pickling process) before transmission, the relative sizes after compression were also compared. The compression schemes tested were JAR, the standard Java archiving tool, and XMill [32], a specialized XML compression utility. Both the serialized Java and XML representations were compressed with JAR, whereas XMill applies only to XML. The number of operands and destinations in the linear $\mathcal{CG}$ nodes were varied so that nodes were not uniform and hence more easily compressed.

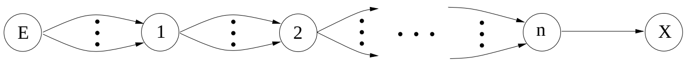
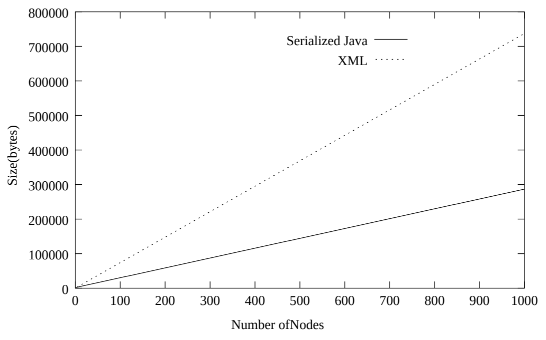


FIG. 7.1. *Structure of the $\mathcal{CG}$s used in the empirical study.*



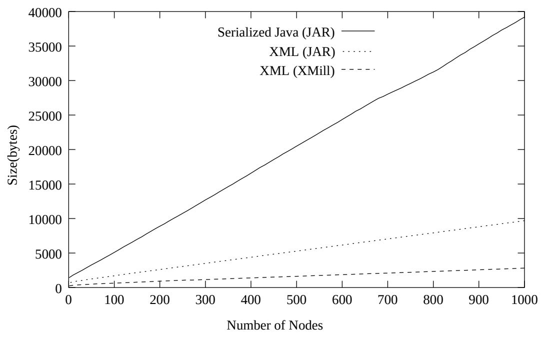


FIG. 7.2. *Graphs showing the relative sizes of $\mathcal{CG}$s represented and compressed using different formats.*

The results of studying 100 graphs with node sizes ranging from 1 to 1000 are illustrated in Fig. 7.2. As expected, graphs represented as XML documents are larger than their serialized Java equivalents, typically by a factor of two. When compressed, however, the XML documents occupy far less space, roughly 1% of their original size for large files, an order of magnitude smaller than the compressed serialized Java files representing the same graphs. These results would indicate that significant storage and bandwidth savings are possible through the utilization of XML compression, at the cost of human readability. A possible application of these results would be to compress extremely large $\mathcal{CG}$ definition documents before transmission and sending them as SOAP attachments rather than in message bodies.

8. Conclusions and Future Work. This paper describes the role of XML when communicating Condensed Graph definitions between WebCom machines. The determining factors in choosing XML for communications are described, and empirical tests illustrate the bandwidth savings observed in using XML instead of Java object serialization. XML has a number of other benefits over object serialization in that it is more extensible; extra graphs can be easily added to an XML file, additional elements and attributes can be added to XML files, without breaking backwards compatability, to support metadata such as layout information particular to the IDE and scheduling policies for nodes.

In the current version of WebCom, only graph definitions and serialized V-Graphs are stored in XML files. Future versions of the $\mathcal{CG}$ XML file format will facilitate the specification of metadata as described above. In addition, other policies can be included that temper the bahaviour of the metacomputer, allowing for different models to be loaded and used for different parts of the graph. For example, a graph could direct specific security manager, load balancing or shceduling modules to be used.

REFERENCES

[1] A. D. Birrell and B. J. Nelson, *Implementing remote procedure calls,* ACM Transactions on Computer Systems, 2(1):39–59, February 1984.

[2] Open Group Product Documentation, *DCE 1.2.2 Introduction to OSF DCE,* November 1997.

[3] The Object Management Group, *The Common Object Request Broker: Architecture and Specification, Revision 2.6,* December 2001.

[4] Microsoft Corporation, *DCOM Technical Overview*, November 1996.

[5] Sun Microsystems, *Java Remote Method Invocation—Distributed Computing for Java, White Paper.*

[6] Andrew Troelsen, *C# and the .NET Platform,* June 2001.

[7] Etham Cerami, *Web Services Essentials,* O'Reilly & Associates, March 2002.

[8] Tim Bray, Jean Paoli, C.M. Sperberg-McQueen, and Eve Maler, *Extensible Markup Language (XML) 1.0 (Second Edition),* October 2000.

[9] Don Box, David Ehnebuske, Gopal Kakivaya, Andrew Layman, Noah Mendelsohn, Henrik Frystyk Nielsen, Satish Thatte, and Dave Winer, *Simple Object Access Protocol (SOAP) 1.1,* May 2000.

[10] Dave Winer, XML-RPC specification.

[11] Microsoft Corporation, *BizTalk Framework 2.0: Document and Message Specification,* December 2000.

[12] Dario Laverde, *Developing Web Services with Java APIs for XML (JAX Pack),* Syngress, March 2002.

[13] Larry Smarr and Charles E. Catlett, *Metacomputing,* Communications of the ACM, 35(6):44–52, 1992.

[14] I. Foster and C. Kesselman, *The Globus Project*, a status report. In Proceedings of the IPPS/SPDP '98 Heterogeneous Computing Workshop, pages 4–18, 1998.

[15] A. Grimshaw and W. Wulf et al, *The Legion vision of a worldwide virtual computer,* Communications of the ACM, 40(1), January 1997.

[16] John Morrison, Brian Clayton, David Power and Adarsh Patil, *WebCom-G: Grid enabled metacomputing,* The Journal of Neural, Parallel and Scientific Computation, Special Issue on Grid Computing, 2004.

[17] J. R. Gurd, C. C. Kirkham, and I. Watson, *The manchester prototype dataflow computer,* Communications of the ACM, 28(1):34–52, January 1985.

[18] Arvind and Kim P. Gostelow, *A computer capable of exchanging processors for time,* In Proceedings of IFIP Congress 77, pages 849–853, Toronto, Canada, August 1977.

[19] John P. Morrison, *Condensed graphs: Unifying availability-driven, coercion-driven and control-driven computing,* October 1996.

[20] Frank Harary, Robert Norman, and Dorwin Cartwright, *Structural Models: An Introduction to the Theory of Directed Graphs,* John Wiley and Sons, 1969.

[21] Rinus Plasmeijer and Marko van Eekelen, *Functional Programming and Parallel Graph Reduction,* Addison-Wesley, 1993.

[22] John P. Morrison, David A. Power, and James J. Kennedy, *An Evolution of the WebCom Metacomputer,* The Journal of Mathematical Modelling and Algorithms: Special issue on Computational Science and Applications, 2003(2), pp 263-276, Editor: G. A. Gravvanis.

[23] John P. Morrison, James J. Kennedy and David A. Power, *Extending WebCom: A Proposed Framework for Web Based Distributed Computing,* Workshop on Metacomputing Systems and Applications, ICPP2000.

[24] John P. Morrison, James J. Kennedy, and David A. Power, *WebCom: A Volunteer-Based Metacomputer,* The Journal of Supercomputing, Volume 18(1): 47-61, January 2001.

[25] John P. Morrison, James J. Kennedy, and David A. Power, *WebCom: A Web-Based Distributed Computation Platform,* Proceedings of Distributed computing on the Web, Rostock, Germany, June 21–23, 1999.

[26] John 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.

[27] David A. Power, *WebCom: A Metacomputer Incorporating Dynamic Client Promotion and Redirection,* M. Sc. Thesis. Submitted to the National University of Ireland, July 2000, July 2000.

[28] John P. Morrison and David A. Power, *Master Promotion & Client Redirection in the WebCom System,* Proceedings of the international conference on parallel and distributed processing techniques and applications (PDPTA 2000), Las Vagas, Nevada, June 26–29, 2000.

[29] John P. Morrison, Keith Power, and Neil Cafferkey, *Cyclone: Cycle Stealing System,* Proceedings of the international conference on parallel and distributed processing techniques and applications (PDPTA 2000), Las Vegas, Nevada, June 26–29, 2000.

[30] Roger Riggs, Jim Waldo, Ann Wollrath and Krishna Bharat, *Pickling state in the Java^{TM} system,* In Proceedings of the 2nd USENIX Conference on Object-Oriented Technologies, 1996.

[31] WebMethods, Inc., *Glue 5.0.2 User Guide,* June 2004.

[32] Hartmut Liefke and Dan Suciu *XMill: an efficient compressor for XML data,* In Proceedings of the 2000 ACM SIGMOD Conference on Management of Data, Dallas, Texas, 2000.

Edited by: Dan Grigoras, John P. Morrison, Marcin Paprzycki
Received: August 15, 2002
Accepted: December 15, 2002