Contents

Chapter 1

Metacomputing: Harnessing Informal Supercomputers

Mark Baker[†] and Geoffrey Fox[‡]

[†]Division of Computer Science
University of Portsmouth
Southsea, Hants, PO4 8JF

[‡]NPAC at Syracuse University
Syracuse, NY, 13244

Email: *Mark.Baker@port.ac.uk, gcf@npac.syr.edu*

1.1 General Introduction

The term metacomputing, when first encountered, seems a rather strange and typically "geek" word. Its origin is believed to have been the CASA project, one of several U.S. Gigabit testbeds around in 1989. Larry Smarr, the NCSA Director, is generally accredited with popularizing the term thereafter.

A search through an ordinary dictionary to try and decipher the term would, at the time of writing, be fruitless. So, what does metacomputing mean? There seems to be many, sometimes conflicting, interpretations. Perhaps one should refer back to the Greek word "meta" to help understand the full word. Among the many meanings of "meta", one will often find references to "sharing" and "action in common." These words are the key to understanding the concept of metacomputing. Using these terms one can interpret metacomputing and understand it to be computers sharing and acting together to solve some common problem.

At this point, to reduce potential confusion, it is worth distinguishing between a parallel computer and a metacomputer. The key difference is the behavior of individual computational nodes. A metacomputer is a dynamic environment that has some informal pool of nodes that can join or leave the environment whenever they desire. The nodes can be viewed as independent machines. So, in a slightly confusing sense, a parallel computer, such as an IBM SP2, can be viewed as a "metacomputer in a box." Whereas, an SMP parallel computer, such as Tera MTA or Sun Enterprise 10000, cannot. The difference is that individual computational nodes in an SMP are not independent.

More recently Catlett and Smarr [1] have related the term metacomputing to "the use of powerful computing resources transparently available to the user via a networked environment." Their view is that a metacomputer is a networked virtual supercomputer. To an extent our usage of the term metacomputing still holds true to this definition apart from the need to explicitly refer to "powerful computing resources." Today's typical desktop computing resources can be viewed as powerful resources of yesterday.

The steps necessary to realize a metacomputer include:

- The integration of individual software and hardware resources into a combined networked resource.
- The implementation of middleware to provide a transparent view of the resources available.
- The development and optimization of distributed applications to take advantage of the resources.

1.1.1 Why Do We Need Metacomputing ?

The short answer to this question is that our computational needs are infinite, whereas our financial resources are finite. As we are all well aware, the sophisticated applications that we run on our desktops today seem to require more and more computing resources with every new revision. This trend is likely to continue as users and developers demand, for example, additional functionality or more realistic simulations. The net result is that from desktops to parallel supercompters, users will always want more and more powerful computers. This is where metacomputing comes into its own. Why not try and utilize the potentially hundreds of thousands of computers that are interconnected in some unified way? The realization of this concept is the essence of metacomputing. It should be mentioned here that seamless access to remote resources is an uncontroversial topic. Whereas linking remote resources together to, say, execute a parallel application is more contentious as there are overheads involved. There are some situations where there is a strong case, for example, when linked resources must be geographically distributed, such as in filtering and visualization of scientific data. In general, it is fair to say that metacomputing comes with an efficiency penalty and it is usually better to run separate jobs on components of the metacomputer. However, linking of self contained applications is of growing importance in science (multidisciplinary applications) and industry (linking different components of an organization together).

It is not too bizarre to envision that at some stage in the not so distant future individuals – be they engineers, scientists, students, health-care workers, or business persons – will be able to access the computing resources that they need to run their particular application with the same ease that we switch on a light or turn on a kitchen appliance. Their application may be the simulation of the fluid flow around the after end of a ship, image processing of NMR scans, a Monte-Carlo financial simulation for a stockbroker, or a final year project for a student dissertation. It should be noted that these metacomputing resources can be accessed via a remote laptop or desktop [2].

1.1.2 What is a Metacomputer?

The simplest analogy to help describe a metacomputer is the electricity grid. When you turned on the power to your computer or switched on your television, you probably did not think about the original source of the electricity to drive these appliances. It was not necessary for you to select a generator with adequate capacity or consider the gauge of wire used to connect the outlet or whether the power lines are underground or on pylons. Basically, you were using a national power grid sophisticated enough to route the electrons across hundreds of miles, yet easy enough for a child to use. In the same manner a metacomputer is a similarly easy-to-use assembly of networked computers that can work together to tackle a single task or set of problems.

It is not surprising, therefore, that the terms "The Grid" and "Computational Grids" are being used to describe a universal source of computing power [3]. A Grid can be viewed as the means to provide pervasive access to advanced computational resources, databases, sensors, and people. It is believed that it will allow a new class of applications to emerge and will have a major impact on our approach to computing in the twenty first century. For our purposes, Computational Grids are equivalent to metacomputing environments.

Metacomputing encompasses the following broad catorgories:

- Seamless access to high performance resources.
- "Parameter" studies (embarassingly parallel application, see FAFNER, Section 1.2.2).
- The linkage of scientific instruments, analysis system, archival storage, and visualisation (so called four-way metacomputing, see I-WAY, Section 1.2.2).
- The general complex linkage of N distributed components.

1.1.3 The Parts of a Metacomputer

A metacomputer is a virtual computer architecture. Its constituent components are individually not important. The key concept is how these components work together as a unified resource. On an abstract level the metacomputer consists of the following components:

- *Processors and Memory*

  The most obvious component of any computer system is the microprocessor that provides its computational power. The metacomputer will consist of an array of processors. Associated with each processor will be some dynamic memory.

- *Networks and Communications Software*

  The physical connections between computers turn then from a collection of individual machines into an interconnected network. The link between machines could, for example, be via modems, ISDN, standard Ethernet, FDDI, ATM, or a myriad of other networking technologies. Networks with high bandwidth and low latency that provide rapid and reliable connections between the machines are the most favored. To actually communicate over these physical connections it is necessary to have some communications software running. This software bridges all of the gaps, between different computers, between computers and people, even between different people.

- *Virtual Environment*

  Given that we have an interconnected, communicating network of computers, processors with memory, and there needs to be something like an operating system that can be used to configure, manage, and maintain the metacomputing environment. This virtual environment needs to span the extent of the metacomputer and make it usable by both administrators and individual users. Such an environment will enable machines and/or instruments that may be located in the same building, or separated by thousands of miles, to appear as one system.

- *Remote Data Access and Retrieval*

  In a metacomputing environment there is the potential that multiple supercomputers performing at Gflop/s are interacting with each other across national or international networks streaming GBytes of data in and out of secondary storage. This is a major challenge for any metacomputing environment. The challenge will become ever greater as new data-intensive applications are designed and deployed.

1.2 The Evolution of Metacomputing

1.2.1 Introduction

In this section we describe two early metacomputing projects that were in the vanguard of this type of technology. The projects differ in many ways, but both had to overcome a number of similar hurdles, including communications, resource management, and the manipulation of remote data, to be able to work efficiently and effectively. The two projects also attempted to provide metacomputing resources from opposite ends of the computing spectrum. Whereas FAFNER [4] was capable of running on any workstation with more than 4 MBytes of memory, I-WAY [5], on the other hand, was a means of unifying the resources of large supercomputing centres.

1.2.2 Some Early Examples

FAFNER

Public key cryptographic systems use two keys: a public and private key. A user must keep their private key a secret, but the public key is publicly known. Public and private keys are mathematically related, so that a message encrypted with a recipient's public key can only be decrypted by their private key. The RSA algorithm [6] is an example of a public key algorithm. It is named after its developers Rivest, Shamir, and Adleman, who invented the algorithm at MIT in 1978.

The RSA keys are generated mathematically in part by combining prime numbers. The security of RSA is based on the premise that it is very difficult to factor extremely large numbers, in particular those with hundreds of digits. RSA keys use either 154 or 512-digit keys. The usage of this type of cryptographic technology has led to integer factorization becoming an active research area. To keep abreast of the state of the art in factoring, RSA Data Security Inc. initiated the RSA Factoring Challenge in March 1991. The Factoring Challenge provides a testbed for factoring implementations and provides one of the largest collections of factoring results from many different experts worldwide.

Factoring is computationally very expensive. For this reason parallel factoring algorithms have been developed so that factoring can be distributed over a network of computational resources. The algorithms used are trivially parallel and require no communications after the initial setup. With this setup, it is possible that many contributors can provide a small part of a larger factoring effort. Early efforts relied on Email to distribute and receive factoring code and information. More recently, in 1995, a consortium led by Bellcore Labs., Syracuse University and Co-Operating Systems started a project of factoring via the Web, know as FAFNER.

FAFNER was set up to factor RSA130 using a new numerical technique called the Number Field Sieve (NFS) factoring method using computational Web servers. The consortium produced a Web interface to NFS. A contributor then uses a Web-form to invoke server-side CGI scripts written in Perl. Contributors could, from one set of Web pages, access a wide range of support services for the sieving step of the factorization: NFS software distribution, project documentation, anonymous user registration, dissemination of sieving tasks, collection of relations, relation archival services, and real-time sieving status reports. The CGI scripts produced supported cluster management, directing individual sieving workstations through appropriate day/night sleep cycles to minimize the impact on their owners. Contributors downloaded and built a sieving software daemon. This then became their Web client that used HTTP protocol to GET values from and POST the resulting relations back to a CGI script on the Web server.

Three factors combined to make this approach succeed.

1. The NFS implementation allowed even single workstations with 4 Mbytes to perform useful work using small bounds and a small sieve.

2. FAFNER supported anonymous registration – users could contribute their hardware resources to the sieving effort without revealing their identity to anyone other than the local server administrator.

3. A consortium of sites was recruited to run the CGI script package locally, forming a hierarchical network of RSA130 Web servers which reduced the potential administration bottleneck and allowed sieving to proceed around the clock with minimal human intervention.

The FAFNER project won an award in TeraFlop challenge at SC95 in San Diego. It paved the way for a wave of Web-based metacomputing projects, some of which are described in Sections 1.4 and 1.5.

I-WAY

The Information Wide Area Year (I-WAY) was an experimental high performance network linking many high performance computers and advanced visualization environments. The I-WAY project was conceived in early 1995 with the idea not to build a network but to integrate existing high-bandwidth networks with telephone systems. The virtual environments, datasets, and computers used resided at 17 different U.S. sites and were connected by ten networks of varying bandwidths and protocols, using different routing and switching technologies.

The network was based on ATM technology, which at the time was an emerging standard. This network provided the wide-area backbone for various experimental networking activities at SC95, supporting both TCP/IP over ATM and direct ATM-oriented protocols.

To help standardize the I-WAY software interface and management, the key sites installed point-of-presence (I-POP) computers to serve as their gateways to the I-WAY. The I-POP machines were UNIX workstations configured uniformly and possessed a standard software environment called I-Soft. I-Soft helped overcome issues such as heterogeneity, scalability, performance, and security. The I-POP machines were the gateways into each site participating in the I-WAY project.

The I-POP machine provided uniform authentication, resource reservation, process creation, and communication functions across I-WAY resources. Each I-POP machine was accessible via the Internet and operated within its site's firewall. It also had an ATM interface that allowed monitoring and potential management of the site's ATM switch.

For the purpose of managing its resources efficiently and effectively, the I-WAY project developed a resource scheduler known as the Computational Resource Broker (CRB). The CRB basically consisted of user-to-CRB and CRB-to-local-scheduler protocols. The actual CRB implementation was structured in terms of a single central scheduler and multiple local scheduler daemons – one per I-POP machine. The central scheduler maintained queues of jobs and tables representing the state of local machines, allocating jobs to machine and maintaining state information on the AFS file system.

Security was a major feature of the I-WAY project. An emphasis was made on providing a uniform authentication environment. Authentication to I-POPs was handled by using a telnet client modified to use Kerberos authentication and encryption. In addition, the CRB acted as an authentication proxy, performing subsequent authentication to I-WAY resources on a user's behalf.

I-WAY used AFS to provide a shared file repository for software and scheduler information. An AFS cell was set up and made accessible from only I-POPs. To move data between machines where AFS was unavailable, a version of remote copy (`ircp`) was adapted for I-WAY.

To support user-level tools, a low-level communications library, Nexus , was adapted to execute in the I-WAY environment. Nexus supported automatic configuration mechanisms that enabled it to choose the appropriate configuration depending on the technology being used, for example, communications via TCP/IP or AAL5 when using the Internet or ATM. The MPICH and CAVEcomm libraries were also extended to use Nexus.

The I-WAY project was application driven and defined five types of applications:

- Supercomputer - Supercomputing
- Remote Resource - Virtual Reality
- Virtual Reality - Virtual Reality
- Multisupercomputer - Multivirtual Reality
- Video, Web, GII-Windows

The I-WAY project was successfully demonstrated at SC'95 in San Diego. The I-POP machine was shown to simplify the configuration, usage, and management of this type of wide-area computational testbed. I-Soft was a success in terms that most applications ran, most of the time. More importantly, the experiences and software developed as part of the I-WAY project have been fed into the Globus project described in Section 1.4.2.

A Summary of Early Experiences

The projects described in this section both attempted to produce metacomputing environments by integrating hardware from opposite ends of the computing spectrum. FAFNER was a ubiquitous system that would work on any platform where a

Web server could be run. Typically, its clients were at the low-end of the computing performance spectrum. Whereas I-WAY unified the resources at supercomputing sites. The two projects also differed in the types of applications that could utilize their environments. FAFNER was tailored to a particular factoring application that was in itself trivially parallel and was not dependent on a fast interconnect. I-WAY, on the other hand, was designed to cope with a range of diverse high performance applications that typically needed a fast interconnect. Both projects, in their way, lacked scalability. For example, FAFNER was dependent on quite a lot of human intervention to distribute and collect sieving results, and I-WAY was limited by the design of components that made up I-POP and I-Soft.

FAFNER lacked a number of features that would now be considered obvious. For example, every client had to compile, link, and run a FAFNER daemon in order to contribute to the factoring exercise. Today, one would probably download an already set up and configured Java applet. FAFNER was really a means of task-farming a large number of fine-grain computations. Individual computational tasks were unable to communicate with one another, or with their parent Web-server. Today perhaps, using technology such as Java RMI, tasks would register themselves, ask for work, coordinate their computation, deliver results, and so on, with even less human intervention or interaction.

Likewise, with I-WAY, a number of features would today seem inappropriate. The installation of an I-POP platform made it easier to set up I-WAY services in a uniform manner, but it meant that each site needed to be specially set up to participate in I-WAY. In addition, the I-POP platform created one, of many, single-points-of-failure in the design of the I-WAY. Even though this was not reported to be a problem, the failure of an I-POP would mean that a site would drop out of the I-WAY environment. Today, many of the services provided by the I-POP and I-Soft would be available on all the participating machines at a particular site.

Regardless of the aforementioned features of both FAFNER and I-WAY, both projects were highly successful. Each project was in the vanguard of metacomputing and has helped pave the way for many of the succeeding projects. In particular, FAFNER was the forerunner of projects such as WebFlow (described in Section 1.4.4), and the I-WAY software, I-Soft, was very influential on the approach used to design the components employed in the Globus Metacomputing Toolkit (described in Section 1.4.2).

1.3 Metacomputer Design Objectives and Issues

In this section we lay out and discuss the basic criteria required by all wide area distributed environments or a metacomputer. In the first part of this section we outline the underlying hardware and software technologies potentially being used. We then move on to discuss the necessary attributes of the middleware that creates the virtual environment we call a metacomputer.

1.3.1 General Principles

In attempting to facilitate the collaboration of multiple organizations running diverse autonomous heterogeneous resources, a number of basic principles should be followed so that the metacomputing environment:

- does not interfere with the existing site administration or autonomy
- does not compromise existing security of users or remote sites
- does not need to replace existing operating systems, network protocols or services
- allows remote sites to join or leave the environment whenever they choose
- does not mandate the programming paradigms, languages, tools, or libraries that a user wants
- provides a reliable and fault tolerance infrastructure with no single point of failure
- provides support for heterogeneous components
- uses standards, and existing existing technologies, and is able to interact with legacy applications
- provides appropriate synchronization and component program linkage

1.3.2 Underlying Hardware and Software Infrastructure

As one would expect, a metacomputing environment must be able to operate on top of the whole spectrum of current and emerging hardware and software technologies. An obvious analogy is the Web. Users of the Web do not care if the server they are accessing is on a UNIX or NT platform. They are probably unaware that they are using HTTP on top of TCP/IP, and they certainly do not want to know that they are accessing a database supported by a parallel computer, such as an IBM SP2, or an SMP, such as the SGI Origin 2000. From the client browser's point-of-view, they "just" want their requests to Web services handled quickly and efficiently. In the same way, a user of a metacomputer does not want to be bothered with details of its underlying hardware and software infrastructure. A user is really only interested in submitting their application to the appropriate resources and getting correct results back in a timely fashion.

An ideal metacomputing environment will therefore provide access to the available resources in a seamless manner such that physical discontinuities such as differences between platforms, network protocols, and administrative boundaries become completely transparent. In essence, the metacomputing middleware turns a radically heterogeneous environment into a virtual homogeneous one.

1.3.3 Middleware – The Metacomputing Environment

In this section we outline and describe the idealized design features that are required by a metacomputing system to provide users with a seamless computing environment.

Administrative Hierarchy

An administrative hierarchy is the way that each metacomputing environment divides itself up to cope with a potentially global extent. For example, DCE uses cells and DNS has a hierarchical namespace. The reasons why this category is important stems from the administrative need to provide resources on autonomous systems on a global basis. The administrative hierarchy determines how administrative information flows through the metacomputer. For example, how does the resource manager find its resources ? Does it interrogate one global database of resources or a hierarchy of servers, or perhaps servers configured in some peer-related manner?

Communication Services

The communication needs of applications using a metacomputing environment are diverse – ranging from reliable point-to-point to unreliable multicast communications. The communications infrastructure needs to support protocols that are used for bulk-data transport, streaming data, group communications, and those used by distributed objects.

These communication services provide the basic mechanisms needed by the metacomputing environment to transport administrative and user data. The network services used also provide the metacomputer with important Quality of Service parameters such as latency, bandwidth, reliability, fault-tolerance, and jitter control. Typically, the network services will be built from a relatively low-level communication API that can be used to support a wide range of high-level communication libraries and protocols. These mechanisms provide the means to implement a wide range of communications methodologies, including RPC, DSM, stream-based, and multicast.

Directory/Registration Services

A metacomputer is a dynamic environment where the location and type of services available are constantly changing. A major goal is to make all resources accessible to any process in the system, without regard to the relative location of the resource user. It is necessary to provide mechanisms to enable a rich environment in which information about metacomputing is reliably and easily obtained by those services requesting the information. The registration and directory services components provide the mechanisms for registering and obtaining information about the metacomputer structure, resources, services, and status.

Processes, Threads and Concurrency Control

The term process originates in the literature on the design of operating systems and is generally considered as a unit of resource allocation both for CPU and memory. The advent of shared memory multiprocessors brought about the provision of Light Weight Processes or Threads. The name thread comes from the expression "thread of control." Modern OSs, like NT, permit an OS process to have multiple threads of control. With regards to metacomputers, this category is related to the granularity of control provided by the environment to its applications. Of particular interest is the methodology used to share data and maintain its consistency when multiple processes or threads have concurrent access to it.

Time and Clocks

Time is an important concept in all systems. First, time is an entity that we wish to measure accurately, as it may be a record of when a particular transaction occurred. Or, if two or more computer clocks are synchronized it can be used to measure the interval when two or more events occurred. Second, algorithms have been developed that depend on clock synchronization. These algorithms may, for example, be used for maintaining the consistency of distributed data or as part of the Kerberos authentication protocol.

Naming Services

In any distributed system, names are used to refer to a wide variety of resources such as computers, services, or data objects. The naming service provides a uniform name space across the complete metacomputing environment. Typical naming services are provided by the international X.500 naming scheme or DNS, the Internet's scheme.

Distributed Filesystems and Caching

Distributed applications, more often than not, require access to files distributed among many servers. A distributed filesystem is therefore a key component in a distributed system. From an applications point of view it is important that a distributed filesystem can provide a uniform global namespace, support a range of file I/O protocols, require little or no program modification, and provide means that enable performance optimizations to be implemented, such as the usage of caches.

Security and Authorization

Any distributed system involves all four aspects of security: confidentiality – prevents disclosure of data; integrity – prevents tampering with data; authentication – verifies identity, and accountability – knowing whom to blame. Security within a metacomputing environment is a complex issue requiring diverse resources autonomously administered to interact in a manner that does not impact on the usability of the resources or introduce security holes in individual systems or the environments as a whole. A security infrastructure is key to the success or failure of a metacomputing environment.

System Status and Fault Tolerance

There is a very high likelihood that some component in a metacomputing environment will fail. To provide a reliable and robust environment it is important that a means of monitoring resources and applications is provided. For example, if a particular platform goes out-of-service, it is important that no further jobs are scheduled on it until it becomes in-service again. In addition, jobs that were running on the system when it crashed should be rerun when it is available again or rescheduled onto an alternative system. To accomplish this task, tools that monitor resources and application need to be deployed. So, when a platform is unavailable, information is passed to the directory services, or perhaps, when a job crashes, some part of the system reschedules that job to run again.

Resource Management and Scheduling

The management of processor time, memory, network, storage, and other components in a distributed system is clearly very important. The overall aim is to efficiently and effectively schedule the applications that need to utilize the available resources in the metacomputing environment. From a user's point of view, resource management and scheduling should be almost transparent; their interaction with it being confined to a manipulating mechanism for submitting their application. It is important in a metacomputing environment that a resource management and scheduling service can interact with those that may be installed locally. For example, it may be necessary to operate in conjunction with LSF, Codine, or Condor at different remote sites.

Programming Tools and Paradigms

Ideally, every user will want to use a diverse range of programming paradigms and tools with which to develop, debug, test, profile, run, and monitor their distributed application. A metacomputing environment should include interfaces, APIs and conversion tools so as to provide a rich development environment. Common scientific languages such as C, C++, and Fortran should be available, as should message passing interfaces like MPI and PVM. A range of programming paradigms should be supported, such as message passing and distributed shared memory. In addition, a suite of numerical and other commonly used libraries should be available.

User and Administrative GUI

The interfaces to the services and resources available should be intuitive and easy to use. In addition, they should work on a range of different platforms and operating systems.

Availability

Earlier in this section we mentioned the need to provide middleware that provided heterogeneous support. In particular, we are concerned about issues, such as if a particular resource management system works on a particular operating system, or will the communication services run on top of particular network architecture such as Novell or SNA. The issues that relate to this category are those that relate to the portability of the software services provided by the metacomputing environment. The metacomputing software should either be easily "ported" on to a range of commonly used platforms, or should use technologies that enable it to be platform neutral, in a manner similar to Java Byte-code.

1.4 Metacomputing Projects

1.4.1 Introduction

In this section we map the techniques and technologies that three representative current metacomputing environments use with the aid of the design objectives and issues laid out in the previous section. The main purpose of this template is to help the reader review the methodologies used by each project. The three projects reviewed in this section are: Globus from Argonne National Laboratory, Legion from the University of Virginia, and WebFlow from Syracuse University The reasons why these three particular projects were chosen are:

- `Globus` - provides a toolkit based on a set of existing components with which to build a metacomputing environment.
- `Legion` - provides a high-level unified object model out of new and existing components to build a metasystem.
- `WebFlow` - provides a Web-based metacomputing environment.

1.4.2 Globus

Introduction

Globus [8], [9], provides a software infrastructure that enables applications to handle distributed, heterogeneous computing resources as a single virtual machine. The Globus project is a U.S. multiinstitutional research effort that seeks to enable the construction of computational grids. A computational grid, in this context, is a hardware and software infrastructure that provides dependable, consistent, and pervasive access to high-end computational capabilities, despite the geographical distribution of both resources and users. A central element of the Globus system is the Globus Metacomputing Toolkit (GMT), which defines the basic services and capabilities required to construct a computational grid. The toolkit consists of a set of components that implement basic services, such as security, resource location, resource management, and communications.

It is necessary for computational grids to support a wide variety of applications and programming paradigms. Consequently, rather than providing a uniform programming model, such as the object-oriented model, the GMT provides a bag of services from which developers of specific tools or applications can use to meet their own particular needs. This methodology is only possible when the services are distinct and have well-defined interfaces (API) that can be incorporated into applications or tools in an incremental fashion.

Globus is constructed as a layered architecture in which high-level global services are built upon essential low-level core local services. The Globus toolkit is modular, and an application can exploit Globus features, such as resource management or information infrastructure, without using the Globus communication libraries.

The GMT currently consists of the following:

- Resource allocation and process management (GRAM)
- Unicast and multicast communications services (Nexus)
- Authentication and related security services (GSI)
- Distributed access to structure and state information (MDS)
- Monitoring of health and status of system components (HBM)
- Remote access to data via sequential and parallel interfaces (GASS)
- Construction, caching, and location of executables (GEM)

Administrative Hierarchy

Globus has no obvious administrative hierarchy. Every Globus-enabled resource is a peer of every other enabled resource.

Communication Services

Communication services within Globus are provided by Nexus, a communication library that is designed specifically to operate in a grid environment. Nexus is distinguished by its support for multimethod communication, providing an application a single API to a wide range of communication protocols and characteristics. Nexus defines a relatively low-level communication API that can be used to support a wide range of high-level communication libraries and languages. Nexus communication services are used extensively in other parts of the Globus toolkit.

Directory/Registration Services

The Globus Metacomputing Directory Service (MDS) provides information about the status of Globus system components. MDS is part of the information infrastructure of the GMT and is capable of storing static and dynamic information about the status of a metacomputing environment. MDS uses a Lightweight Directory

Access Protocol [10] (LDAP) server that can store metacomputing-specific objects. LDAP is a streamlined version of the X.500 directory service. The MDS houses information pertaining to the potential computing resources, their specifications, and their current availability.

Processes, Threads and Concurrency Control

Globus works at the process level. The Nexus API can be used to construct communication primitives between threads. There is no concurrency control in Globus.

Time and Clocks

Globus does not mandate the usage of a particular time service, it relies on those already used at each site.

Naming Services

Globus makes extensive usage of LDAP as well as DNS and X.500.

Distributed Filesystems and Caching

The Globus system currently provides three interfaces for remote access of user data:

- Global Access to Secondary Storage (GASS) – provides basic access to remote files. Operations supported include remote read, remote write, and append.
- Remote I/O – The RIO library implements a distributed implementation of the MPI-IO, parallel I/O API.
- Globus Executable Management (GEM) – enables loading and executing a remote file through GRAM using GASS caching calls.

The Remote I/O for Metasystems (RIO) library provides basic mechanisms for tools and applications that require high performance access to data located in remote, potentially parallel file systems. RIO implements the Abstract I/O (ADIO) device interface specification, which defines basic I/O functionality that can be used to implement a variety of higher-level I/O libraries. ROMIO has adopted the parallel I/O interface defined by the MPI forum in MPI-IO and hence allows any program already using MPI-IO to work without unchanged in a wide-area environment. The RIO library has been developed as part of the GMT, although it can also be used independently. ROMIO can be used with Nexus communications, GSI security, and MDS to provide confiugration information.

The GMT data movement and access service, GASS, defines a global name space via URLs, allows access to remote files via standard I/O interfaces, and provides specialized support for data-movement in a wide-area environment. GASS addresses bandwidth management issues associated with repeated access to remote files by providing a file cache: where a “local” copy of a remote file can be stored. Files are moved in to and out of the cache when a file is opened or closed by an application. GASS uses a simple locking protocol for local concurrency control, but does not implement a wide-area cache coherency mechanism.

Security and Authorization

Globus employs an authentication system known as the Generic Security Service API (GSI) using an implementation of the Secure Sockets Layer. This system uses the RSA encryption algorithm and the associated public and private keys. The GSI authentication relies on an X509 certificate, provided by the user in their directory, that identifies them to the system. This certificate includes information about the duration of the permissions, the RSA public key, and the signature of the Certificate Authority (CA). With the certificate is the user’s private key. The certificates can be created only by the CA, who reviews the X509 certificate request submitted by the user, and accepts or denies it according to an established policy.

System Status and Fault Tolerance

Globus provides a range of basic services designed to enable the construction of application specific fault recovery mechanisms. In Globus it is currently assumed detection of a fault is a necessary prerequisite to fault recovery or fault tolerance. The main fault detection service in Globus is the Heartbeat Monitor (GHM) that enables a process to be monitored and periodic heartbeats to be sent to one or more monitors. The Nexus communication library also provides support for fault detection.

Resource Management and Scheduling

The Globus Resource Allocation Manager (GRAM) is the lowest level of Globus architecture. GRAM allows jobs to run remotely and provides an API for submitting, monitoring, and terminating jobs. GRAM provides the local component for resource management and is responsible for the set of resources operating under the same site-specific allocation policy. Such a policy will often be implemented by a local resource management package, such as LSF, Codine, or Condor.

GRAM is responsible for:

- Parsing and processing the Resource Specification Language (RSL) specifications that outline job requests. The request specifies resource selection, job process creation, and job control. This is accomplished by either denying the request or creating one or more processes (jobs) to satisfy the request. The RSL is a structured language that can be used to define resource requirements and parameters by a user.

- Enabling remote monitoring and managing of jobs already created.

- Updating MDS with information regarding the availability of the resources it manages.

Programming Tools and Paradigms

Globus currently supports MPI, Java, Compositional C++, Simple RPC, and Perl. There are ongoing efforts to add a Sockets API, an IDL, Legion, and Netsolve.

User and Administrative GUI

Globus makes extensive usage of the Web and command line interfaces for administration. For example, LDAP can be browsed via the Web. There are also a growing number of Java components that can be used with Globus.

Availability

Globus is available on most versions of UNIX and is currently being developed for NT.

1.4.3 Legion

Introduction

Legion [11], [12] is an object-based metasystem developed at the University of Virginia. Legion provides the software infrastructure so that a system of heterogeneous, geographically distributed, high performance machines can interact seamlessly. Legion attempts to provide users, at their workstations, with a a single, coherent, virtual machine. The Legion system is organized by classes and metaclasses (classes of classes).

In Legion:

- *Everything is an object* - Objects represent all hardware and software components. Each object is an active process that responds to method invocations from other objects within the system. Legion defines an API for object interaction, but not the programming language or communication protocol.
- *Classes manage their instances* - Every Legion object is defined and managed by its own active class object. Class objects are given system-level capabilities; they can create new instances, schedule them for execution, activate or deactivate an object, as well as provide state information to client objects.
- *Users can define their own classes* - As in other object-oriented systems users can override or redefine the functionality of a class. This feature allows functionality to be added or removed to meet a user's needs.
- *Core objects* - Legion defines the API to a set of core objects that support the basic services needed by the metasystem.

Legions has the following set of core object types:

- *Classes and Metaclasses* - Classes can be considered managers and policy makers. Metaclasses are classes of classes.
- *Host objects* - Host objects are abstractions of processing resources, they may represent a single processor or multiple hosts and processors.
- *Vault objects* - Vault objects represents persistent storage, but only for the purpose of maintaining the state of Object Persistent Representation (OPR).
- *Implementation Objects and Caches* - Implementation objects hide the storage details of object implementations and can be thought of as equivalent to executable files in UNIX. Implementation cache objects provide objects with a cache of fequently used data.
- *Binding Agents* - A binding agent maps object IDs to physical address. Binding agents can cache bindings and organize themselves in hierarchies and software combining trees.
- *Context objects and Context spaces* - Context objects map context names to Legion object IDs, allowing users to name objects with arbitrary-length string names. Context spaces consist of directed graphs of context objects that name and organize information.

A Legion object is an instance of its class. Objects are independent, active, and capable of communicating with each other via unordered nonblocking calls. Like other object-oriented systems, the set of methods of an object describes its interface. The Legion interfaces are described in an Interface Definition Language (IDL) .

A Legion object can be in one of two different states, active or inert. An active object runs as a process that is ready to accept function invocations. An inert object is represented by an OPR . An OPR is an image of the object which resides on some stable storage; this is analogous to a process that has been swapped-out to disk. In a similar, manner an OPR contains state information that enables the object to be reactivated. Legion implements a three-tiered naming system.

1. Users refer to objects using human-readable strings, called context names.
2. Context objects map context names to LOIDs (Legion object identifiers), which are location-independent identifiers that include an RSA public key.
3. A LOID is mapped to an LOA (Legion object address) for communication. A LOA is a physical address (or set of addresses in the case of a replicated object) that contains sufficient information to allow other objects to communicate with the object (e.g., an IP address and port number pair).

Administrative Hierarchy

Legion has no obvious administrative hierarchy. Objects distributed about the Legion environment are peers to one another.

Communication Services

Legion uses standard TCP/IP to support communications between objects. Every Legion object is linked with a UNIX sockets-based delivery layer, called the Modular Message Passing System (MMPS) .

Directory/Registration Services

A Binding agent in Legion maps LOIDs to LOAs. A LOID/LOA pair is called a binding. Binding agents can cache bindings and organise themselves in hierarchies and software combining trees.

Processes, Threads and Concurrency Control

Currently Legion has one process per active object and objects communicate via MMPS. There is no concurrency control included in Legion.

Time and Clocks

Legion does not mandate the usage of a particular time service and relies on those already used at each site.

Naming Services

Legion Context objects map context names to LOIDs, allowing users to name objects with arbitrary-length string names. A LOID is mapped to an LOA for communication purposes. A LOA consists of an IP address and port number. It is assumed that Legion uses DNS to translate names to IP addresses.

The Context Manager is a Java GUI that can be used to manage context space. Context space is organized into a series of subcontexts (also called contexts) and each context contains context names of various Legion objects. In the Context Manager all context-related objects such as contexts, file objects, and objects are represented by icons that can be manipulated. Basic context manager commands are Move, Alias, Get Interface, Get Attributes, Destroy, Activate, and Deactivate.

Distributed Filesystems and Caching

Legion provides a virtual filesystem that spans all the machines in a Legion system. I/O support is provided via a set of library functions with UNIX-like file and stream operations to read, write, and seek. These functions provide location independent and secure access to context space and to "files" in the system. Different users can also employs the virtual filesystem to collaborate, sharing data files and even accessing the same running computations.

Legion has a special core object called a vault object. This represents persistent storage, but only for the purpose of maintaining the state of OPRs. The vault object may manage a portion of a UNIX filesystem, or a set of databases.

Security and Authorization

Legion does not require any special privileges from the host systems that run it. The Legion security model is oriented towards protecting objects and object communication. Objects are accessed and manipulated via method calls; an object's rights are centered in its capabilities to make those calls. The user determines the security policy for an object by defining the object's rights and the method calls they allow. Once this is done, Legion provides the basic mechanism for enforcing that policy.

Every object in Legion supports a special member function called `MayI`. An object with no security will have a null `MayI`. All method invocations to an object must first pass through `MayI` before the target member function is invoked. If the caller has the appropriate rights for the target method, `MayI` allows that method invocation to proceed.

To make rights available to a potential caller, the owner of an object gives it a certificate listing the rights granted. When the caller invokes a method on the object, it presents the appropriate certificate to `MayI`, which then checks the scope and authenticity of the certificate. Alternatively, the owner of an object can permanently assign a set of rights to a particular caller or group. `MayI` is responsible for confirming the identity of a caller and its membership of an authorized group, followed by comparing the rights authorized with the rights required for the method call.

To provide secure communication, every Legion object has a public key pair; the public key is part of the object's name. Objects can use the public key of a target object to encrypt their communications to it. Likewise, an object's private key can be used to sign messages. This ensures authentication and integrity. This integration of public keys into object names eliminates the need for a certification authority. If an intruder tries to tamper with the public key of a known object, the intruder will create a new and unknown name.

System Status and Fault Tolerance

Legion does not mandate any fault-tolerance policies; applications are responsible for selecting the level they need. Fault tolerance will be built into generic base classes and applications will be able to invoke methods that provide the functionality that they require. Legion will support object reflection, replication, and check pointing for the purposes of fault tolerance.

Resource Management and Scheduling

Host objects represent processors, and more than one may run on each computing resource. Host objects create and manage processes for active Legion objects. Classes invoke the member functions on host objects in order to activate instances on the computing resources that the hosts represent. Legion provides resource owners with the ability to initiate, manage and control, and kill their resources.

The Legion-scheduling module consists of three components:

- A resource state information database (*Collection*). The *Collection* interacts with resource objects to collect state information describing the system.

- A module which maps requests to resources (*Scheduler*). The *Scheduler* queries the *Collection* to determine a set of available resources that match the Scheduler's requirements. After computing a schedule, or set of desired schedules, the *Scheduler* passes a list of schedules to the *Enactor* for implementation.

- An agent responsible for implementing the schedules (*Enactor*). The *Enactor* then makes reservations with the individual resources and reports the results to the Scheduler. Upon approval by the *Scheduler*, the *Enactor* place objects on the hosts, and monitors their status.

Host objects can be adapted to different environments to suit user needs. For example, a host object may provide an interface to the underlying resource management system, such as LSF, Codine, or Condor.

Programming Tools and Paradigms

Legion supports MPL (Mentat Programming Language) and BFS (Basic Fortran Support). MPL is a parallel C++ language. Legion is written in MPL. BFS is a set of pseudo-comments for Fortran and a preprocessor that gives the Fortran programmer access to Legion objects.

Object Wrapping is used in Legion for encapsulating existing legacy codes into objects. It is possible to encapsulate a PVM, HPF, or shared memory threaded application in a Legion object. Legion also provides a complete emulation of both PVM and MPI with user libraries for C, C++, and Fortran. Legion also supports Java.

User and Administrative GUI

Legion has a command-line and graphical user interface. The Legion GUI, known as the Context Manager, is a Java application that runs context-related commands. The Context Manager uses icons to represent different parts of context space (file objects, subcontexts, etc.) and runs most context-related commands. The Context Manager can be run from the command-line of any platform compatible with the Java Development Kit (JDK) 1.1.3. In addition, there is a Windows 95 client application, called the Legion Server, that allows users to run the Context Manager from Windows 95.

Availability

Legion is available on: x86/Alpha (Linux), Solaris (SPARC), AIX (RS/6000), IRIX (SGI), DEC UNIX (Alpha) and Cray T90.

1.4.4 WebFlow

Introduction

WebFlow [13], [14] is a computational extension of the Web model that can act as a framework for the wide-area distributed computing and metacomputing. The main goal of the WebFlow design was to build a seamless framework for publishing and reusing computational modules on the Web so that endusers, via a Web browser, can engage in composing distributed applications using WebFlow modules as visual components and editors as visual authoring tools. Webflow has a three-tier Java-based architecture that can be considered a visual dataflow system. The frontend uses applets for authoring, visualization, and control of the environment. WebFlow uses servlet-based middleware layer to manage and interact with backend modules such as legacy codes for databases or high performance simulations.

Webflow is analogous to the Web. Web pages can be compared to WebFlow modules and hyperlinks that connect Web pages to intermodular dataflow channels. WebFlow content developers build and publish modules by attaching them to Web servers. Application integrators use visual tools to link outputs of the source modules with inputs of the destination modules, thereby forming distributed computational graphs (or compute-webs) and publishing them as composite WebFlow modules. A user activates these compute-webs by clicking suitable hyperlinks, or customizing the computation either in terms of available parameters or by employing some high-level commodity tools for visual graph authoring.

The high performance backend tier is implemented using the Globus toolkit:

- The Metacomputing Directory Services (MDS) is used to map and identify resources.
- The Globus Resource Allocation Manager (GRAM) is used to allocate resources.
- The Global Access to Secondary Storage (GASS) is used for a high performance data transfer.

WebFlow can be regarded as a high level, visual user interface and job broker for Globus.

With WebFlow, new applications can be composed dynamically from reusable components just by clicking on visual module icons, dragging them into the active WebFlow editor area, and linking them by drawing the required connection lines. The modules are executed using Globus components combined with the pervasive commodity services where native high performance versions are not available.

The prototype WebFlow system is based on a mesh of Java enhanced Web Servers (Apache), running servlets that manage and coordinate distributed computation. This management infrastructure is implemented by three servlets: Session Manager, Module Manager, and Connection Manager. These servlets use URL addresses and can offer dynamic information about their services and current state.

Each management servlet can communicate with others via sockets. The servlets are persistent and application independent.

Future implementations of WebFlow will use emerging standards for distributed objects and take advantage of commercial technologies, such as the CORBA as the base distributed object model.

Administrative Hierarchy

WebFlow has no obvious administrative hierarchy. A WebFlow node is a Web server with a unique URL address, and it is a peer to other nodes.

Communication Services

WebFlow communication services are built on multiple protocols. Applet-Web Server communication uses HTTP; Server-to-Server communication is currently implemented using TCP/IP, soon to be replaced by IIOP. The module developer chooses communications between a backend module and its frontend control panel (Java applet) – typically it is either TCP/IP or IIOP. The modules exchange data (serialized Java objects) via input and output ports. Originally, the port-to-port connection was implemented using TCP/IP. This model is now being changed. The module is a Java Bean, and it interacts with other modules via Java events over IIOP. The data flow paradigm with port-to-port communication is the default that enables users to visually compose an application from independent modules. However, the user is not restricted to this model. A module can be a high performance application to be run on a multiprocessor machine with intramodule communications using any communication service (for example MPI) available on the target system. Also, the modules can interact with each other via remote methods invocation (Java events over IIOP).

Also, WebFlow supports multiple protocols for file transfer, ranging from HTTP to FTP to IIOP to Globus GASS. The user chooses the protocol to be used depending on the file, performance, and security requirements.

Directory/Registration Services

WebFlow does not define its own directory services. The usage of the CORBA naming services and interface repository is planned. It should be noted that WebFlow typically is used in conjunction with Globus and will coordinate with its directory services (MDS).

Processes, Threads, and Concurrency Control

Each module runs as separate Java threads, and all modules run concurrently. It is the user's responsibility to synchronize modules (for example, the user may want the module to block on receiving the input data).

Time and Clocks

WebFlow does not mandate the usage of a particular time service and relies on those already used at each site.

Naming Services

Currently, no specialized naming service other than DNS is used. CORBA services are planned.

Distributed Filesystems and Caching

WebFlow does not offer "a native" distributed filesystem or support caching. This is left to the user. As a part of WebFlow distribution there is a file browser module that allows the user to browse and select files accessible by the host Web server. The selected files can then be sent to a desired destination using HTTP, IIOP, FTP, or GASS. For example, a WebFlow module that serves as the Globus GRAM proxy takes the name of the input file and URL of the GRAM contact as input. This information is sufficient to stage the input file on the target machine and retrieve the output file using GASS over FTP.

Security and Authorization

WebFlow requires two security levels: secure Web transactions between client and the middle tier, and secure access to the backend resources. Secure Web transactions in WebFlow are based on TLS 1.0 and modeled after the AKENTI system; secure access to the backend resources is delegated to the backend service providers, such as Globus. The secure access to resources directly controlled by WebFlow has not yet been addressed.

System Status and Fault Tolerance

The original implementation of WebFlow did not address these issues. The new WebFlow middle-tier will use CORBA mechanisms to provide fault tolerance, including a heartbeat monitor. In the backend, WebFlow relies on services provided by the backend service provider.

Resource Management and Scheduling

WebFlow delegates the resource management and scheduling to the metacomputing toolkit (Globus) and/or a local resource management package such as PBS or CONDOR.

Programming Tools and Paradigms

WebFlow modules are Java objects. Object wrapping is used in WebFlow for encapsulating existing codes into objects. WebFlow test applications include modules

Table 1.1 Metacomputing Functionality Matrix

Design Objective	**Globus**	**Legion**	**Webflow**
Admin. Hierarchy	Peer	Peer	Peer
Comms Service	Nexus - Low-Level	MMPS - Sockets-based	Hierarical - Sockets+MPI
Dir/Reg Services	MDS - LDAP	Via Binding agent	MDS - LDAP
Processes	Process-based	Object/process-based	Process-based
Clock	Not specified	Not specified	Not specified
Naming Services	LDAP + DNS/X.500	Context Manger + DNS	LDAP + DNS
Filesystems & caching	GASS + ROMIO	Custom Legion filesystem	GASS
Security	GSI (RSA + X.509 certs)	Object-based with RSA	SSL
Fault Tolerance	Heart-beat monitor	Not available yet	None
Resource Management	GRAM + RSL + Local	Host object + Local	GRAM-based
Prog. Paradigms	Many and varied	MPL, BFS + wrappers	MPI
User Interfaces	GUI + command-line	GUI + command-line	Applet-based GUI
Availability	Most UNIX	Most UNIX	Most UNIX and NT

with encapsulated Fortran, Fortran with MPI, HPF, C with MPI, Pascal, as well as Java.

User and Administrative GUI

WebFlow offers a visual-authoring tool, implemented as a Java applet that allows the user to compose a (meta-) application from preexisting modules. In addition, a developer can use a simple API to build a custom graphical user interface.

Availability

Since WebFlow is implemented in Java, it runs on all platforms that support JVM. So far it has been tested on Solaris, IRIX, and Windows NT.

Summary and Conclusions

In this section we have attempted to lay out the functionality and features of three representative metacomputer architectures with the design criteria we outlined in Section 1.3. This task in itself has been rather difficult as it has been necessary to map the developer's terminology for components within their environments to those used more commonly in distributed computing. In the final part of this section we summarize the functionality of each environment and conclude by making some observations about the approaches each environment uses.

Functionality Matrix

In the functionality matrix, shown in Table 1, we outline the components within each metacomputing environment that deals with our design criteria.

- *Administrative Hierarchy* - All three environments use a peer-based administrative hierarchy, which makes the services they provide globally scalable and reduces potential administrative bottlenecks and single-points of failure.

- *Communications Service* – Globus uses Nexus to provide its underlying communications services, whereas Legion and WebFlow use sockets-based protocol.
- *Directory/Registration Services* – Both Globus and Webflow use the commodity LDAP service, whereas Legion uses a custom binding agent.
- *Processes* – All three environments are process based - but each has the ability to encompass threads and enable consistency control.
- *Clock* – The three environments do not require special timing services.
- *Naming services* – Globus and WebFlow use LDAP in conjuction with DNS; Legion uses a custom context manager in conjunction with DNS.
- *Filesystems and caching* – Globus makes extensive use of the remote access tool GASS and the parallel I/O interface ROMIO. Legion has a custom global filesystem and the ability to interface with other I/O systems. WebFlow utilizes GASS to provide filesystem services.
- *Security* – All three environments use RSA in some form. Globus uses GSI to provide its security services. Legion uses an Object based system where every object has a security method `MayI`. WebFlow uses SSL for security purposes.
- *Fault Tolerance* – Of the three environments, only Globus provides tools, such as the heartbeat monitor.
- *Resource Management* – Globus implements an extensive resource management and scheduling system, GRAM. Legion has the concept of host object for local resource management. Both Globus and Legion have interfaces to other resource management system, such as Codine and LSF. WebFlow utilizes the services of GRAM.
- *Programming Paradigms* – All three environments provide a raft of tools and utilities to support various programming paradigms.
- *User Interfaces* – Globus and Legion provide both command-line and GUI interfaces. WebFlow uses just a GUI.
- *Availability* – All three environments are available on most UNIX platforms.

Some Observations

Globus is constructed as a layered architecture in which high-level global services are built upon essential low-level core local services. The Globus toolkit is modular. This means that an application can exploit an array of features without needing to implement all of them. Globus can be viewed as a metacomputing framework based on a set of APIs to the underlying services. Even though Globus provides the services needed to build a metacomputer, the Globus framework allows alternative local services to be used if desired. For example, the GRAM API allows alternative resource management systems to be utilized, such as Condor or NQE.

Abstracting the services into a set of standard APIs has a number of advantages. These include:

- the underlying services can be changed without affecting applications that use them
- this type of layered approach simplifies the design of a rather complicated system
- it encourages developers of tools and services, they need to support only one API, making their development and testing cycle shorter and cheaper.

Globus provides application developers with a pragmatic means of implementing a range of services to provide a wide-area application execution environment.

Legion takes a very different approach to provide a metacomputing environment, it encapsulates all its components as objects. The methodology used has all the normal advantages of an object-oriented approach, such as, data abstraction, encapsulation, inheritance, and polymorphism.

It can be argued that many aspects of this object-oriented approach potentially makes it ideal for designing and implementing a complex environment such as a metacomputer. For example, Legion's security mechanism, where each object uses RSA keys and a `MayI` method, seems straightforward and more natural than security mechanisms used in many other environments. In addition, the set of methods associated with each object naturally becomes its external interface and hence its API.

Using an object-oriented methodology in Legion does not come without a raft of problems. It is not obvious how best to encapsulate nonobject-oriented programming paradigms, such as message passing or distributed shared memory. In addition, the majority of real-world computing services have procedural interfaces and it is necessary to produce object-oriented wrappers to interface these services to Legion. For example, the APIs to DNS or resource management systems such as Condor or Codine are procedural.

WebFlow takes a different approach to both Globus and Legion. It is implemented in a hybrid manner using a three-tier architecture that encompasses both the Web and third party backend services. This approach has a number of advantages, including the ability to "plug-in" a diverse set of backend services. For example, currently many of these services are supplied by the Globus Matacomputing Toolkit, but they could be replaced with components from CORBA or Legion. WebFlow also has the advantage that it is more portable and can be installed anywhere a Web server supporting servlets is capable of running.

1.5 Emerging Metacomputing Environments

1.5.1 Introduction

There are a large number and diverse range of emerging distributed systems currently being developed. These systems range from metacomputing frameworks to application testbeds, and from collaborative environments to batch submission mechanisms.

In this section we briefly described and referenced a few of the better know systems (due to space considerations the full text for this section can be found elsewhere [15]). The aim of this section is to bring to the reader's attention not only some of the large number of diverse projects that exist, but also to detail the different approaches used to solve the inherent problems encountered.

1.5.2 Summary

The projects described in this section are a crosssection of those currently undertaken. It is interesting to note that all are using Java and the Web as the communications infrastructure. It is also evident that Java has revolutionized the shape and characteristic of the software environments for heterogeneous distributed systems. It seems that the developers of distributed systems no longer have to focus on aspects such as portability and heterogeneity, by using Java they seem able to concentrate on designing and implementing functional distributed environments. It is not clear, among the raft of projects listed in this section, which environments will succeed. However, each project, in its own way, is contributing to our knowledge of how to design, build, and implement efficient and effective distributed virtual environments.

1.6 Summary and Conclusions

1.6.1 Introduction

In this chapter we have attempted to describe and discuss many aspects of metacomputers. We started off by discussing why there is a need for such environments. We then moved on to describe two early metacomputing projects. Here we also outlined some of the benefits and experiences learned. Having set the scene, we then laid out a design template to map out the critical services that a metacomputing environment needs to encompass. Then, using this template, we mapped the services of three differing environments onto it. This mapping made comparing and contrasting the services that each metacomputing environment provided clearer to understand. Having described three fairly mature environments, we then briefly described some 30-odd emerging distributed environments and tools. Finally, here, we summarize what we have discovered while researching this chapter and conclude by making a few predictions about metacomputing environments of the future.

1.6.2 Summary of the Reviewed Metacomputing Environments

Globus is constructed as a layered architecture in which high-level global services are built upon essential low-level core local services. The Globus toolkit is modular, and as such, an application can exploit an array of features without needing to implement all of them. Globus can be viewed as a metacomputing framework based on a set of APIs to the underlying services. Globus provides application developers with a pragmatic means of implementing a range of services to provide a wide-area application execution environment.

Legion takes a very different approach to provide a metacomputing environment; it encapsulates all its components as objects. The methodology used has all the normal advantages of an object-oriented approach, such as data abstraction, encapsulation, inheritance, and polymorphism. It can be argued that many aspects of this object-oriented approach potentially makes it ideal for designing and implementing a complex environment such as a metacomputer. However, using an object-oriented methodology does not come without a raft of problems, many of these are tied-up with the need for Legion to interact with legacy applications and services. In addition, as Legion is written in MPL, it is necessary to "port" MPL onto each platform before Legion can be installed.

WebFlow takes a different approach to both Globus and Legion. It is implemented in a hybrid manner using a three-tier architecture that encompasses both the Web and third party backend services. This approach has a number of advantages, including the ability to "plug-in" to a diverse set of backend services. For example, many of these services are currently supplied by the Globus toolkit, but they could be replaced with components from CORBA or Legion. WebFlow also has the advantage that it is more portable and can be installed anywhere a Web server supporting servlets is capable of running.

So, in summary, we believe that all three environments have their merits. Fundamentally, the Globus Metacomputing Toolkit is currently the most comprehensive attempt at providing a metacomputing environment. The Globus team has taken a very pragmatic approach to providing the services that are needed in a metacomputer. The design methodology they have used – abstracting the services of some underlying entity into a well thought out API – will give the project longevity, as the entities that provide the service can be updated without changing the fundamental service API. In addition, Globus uses existing standard commodity software components to provide many of its services, for example, LDAP, X.509, and RSA. This has a number of beneficial implications, including code reuse and avoiding the necessity to create all the services from scratch.

Alternatively, Legion is a very ambitious and impressive project. We believe the object-oriented approach they have taken has a lot of merit. A fundamental flaw with Legion currently is the reliance on MPL. If Legion were written in Java, which no doubt the University of Virginia is seriously contemplating, then we would have much more faith in Legion's longevity. Also, perhaps, we would question the use of this system as opposed to one based on the well-known standard CORBA.

WebFlow is still basically an experimental prototype system that is being used to explore a range of new and emerging technologies. It has much merit, particularly in its comprehensive GUI frontend and its ability to utilize standard backend components designed by other organizations.

1.6.3 Some Observations

The Java programming language successfully addresses several key issues that plague the development of distributed environments, such as heterogeneity and security. It also removes the need to install programs remotely; the minimum execution environment is a Java-enabled Web browser. Java has become a prime candidate for building distributed environments.

In a metacomputing environment it is not possible to mandate the types of services or programming paradigms that particular users or organizations must use. A metacomputer needs to provide extensible interfaces to any service desired.

Providing adequate security in a metacomputer is a complex issue. A careful balance needs to be maintained between the usability of an environment and security mechanisms utilized. The security methods must not inhibit the usage of an environment, but it must ensure that the resources are secure from malicious intruders.

1.6.4 Metacomputing Trends

It is very difficult to predict the future. In a field such as computing, the technological advances are moving very fast. Windows of opportunity for ideas and products seem to open and close in the seeming "blink of the eye." However, some trends are evident.

Java, with its related technologies and growing repository of tools and utilities, is having a huge impact on the growth and development of metacomputing environments. From a relatively slow start, the development of metacomputers is accelerating fast with the advent of these new and emerging technologies. It is very hard to ignore the presence of the sleepy giant CORBA in the background. We believe that frameworks incorporating CORBA services will be very influential on the design of metacomputing environments in the future.

Whatever technology or computing paradigm becomes influential or most popular, it can be guaranteed that at some stage in the future its star will wane. Historically, in the computing field, this fact can be repeatedly observed. The lesson from this observation must therefore be drawn that, in the long term, backing only one technology can be an expensive mistake. The framework that provides a metacomputing environment must be adaptable, malleable, and extensible. As technology and fashions change it is crucial that a metacomputing environment evolves with them.

1.6.5 The Impact of Metacomputing

Metacomputing is not only a computing paradigm for just providing computational resources for supercomputing-sized parallel applications. It is an infrastructure that can bond and unify globally remote and diverse resources ranging from meteorological sensors to data-vaults, from parallel supercomputers to personal digital organisers. As such, it will provide pervasive services to all users that need them.

Larry Smarr observes in "The GRID: Blueprint for a New Computing Infrastructure" [3] that metacomputing has serious social consequences and is going to have as revolutionary an effect as railroads did in the American mid-West in the early nineteenth century. Instead of a 30 to 40 year lead-time to see its effects, however its impact is going to be much faster. He concludes that the effects of computational grids are going to change the world so quickly that mankind will struggle to react and change in the face of the challenges and issues they present.

So, at some stage in the future, our computing needs will be satisfied in the same pervasive and ubiquitous manner that we use the electricity power grid. The analogies with the generation and delivery of electricity are hard to ignore, and the implications are enormous.

Acknowledgements

The authors wish to thank Ian Foster (ANL) and Tom Haupt (Syracuse) for information and useful suggestions about their projects. We would also like to thank Wolfgang Gentzsch (Genias) for early access to a FGCS Special Issue on *Metacomputing* [16]. The authors would like to thank Kate Dingley, Tony Kalus, John Rosbottom, and Rose Rayner for proofreading the copy of this chapter.

1.7 Bibliography

[1] C. Catlett and L. Smarr. Metacomputing. *Communications of the ACM*, vol. 35(6), pages 44-52, 1992.

[2] Desktop Access to Remote Resources - `http://www-fp.mcs.anl.gov/~gregor/datorr/`

[3] I. Foster and C. Kesselman, eds. The GRID: Blueprint for a New Computing Infrastructure, *Morgan Kaufmann Publishers, Inc.*, San Francisco, California, 1998. ISBN 1-55860-475-8.

[4] FAFNER - `http://www.npac.syr.edu/factoring.html`

[5] I-WAY - `http://146.137.96.14/`

[6] RSA - `http://www.rsa.com/`

[7] I. Foster, J. Geisler, W. Nickless, W. Smith, and S. Tuecke. Software Infrastructure for the I-WAY Metacomputing Experiment. *Concurrency: Practice and Experience*, vol. 10(7), pages 567-581, 1998.

[8] Globus - `http://www.globus.org/`

[9] I. Foster and C. Kesselman, The Globus Project: A Status Report. *Proceeding IPPS/SPDP '98 Heterogeneous Computing Workshop*, pages 4-18, 1998.

[10] W. Yeong, T. Howes and S. Kille. Lightweight Directory Access Protocol. *RFC 1777*, 28/03/95. Draft Standard.

[11] Legion - `http://legion.virginia.edu/`

[12] A. Grimshaw, W. Wulf, et al. The Legion Vision of a Worldwide Virtual Computer. *Communications of the ACM*, vol. 40(1), January 1997.

[13] WebFlow - `http://osprey7.npac.syr.edu:1998/iwt98/products/webflow/`

[14] T. Haupt, E. Akarsu, G. Fox and W. Furmanski. Web based metacomputing. Special Issue on Metacomputing *Future Generation Computer Systems*, North Holland, to appear in early 1999.

[15] M. Baker and G. Fox Metacomputing: Harnessing Informal Supercomputers. *Portsmouth University preprint*, December 1998. `http://www.dcs.port.ac.uk/~mab/Papers/Cluster-Book/`

[16] *Metacomputing*, Editor, Wolfgang Gentzsch, *Future Generation Computer Systems*, North Holland, due for publication in early 1999.