Programming for Dependability in a Service-based Grid

Win Bausch, Cesare Pautasso, Gustavo Alonso
Dept. of Computer Science
Swiss Federal Institute of Technology (ETHZ)
ETH Zentrum, 8092 Zürich, Switzerland
{bausch,pautasso,alonso}@inf.ethz.ch

Abstract

Service-based Grid infrastructures emphasize service composition rather than sharing of low level resources. The idea is to build Grid applications out of computational services provided by the different sites of the Grid. Recent developments in the area of Web services have strengthened this idea by standardizing procedures like service description, publication and invocation. What is still missing is the infrastructure necessary to support the complete life cycle of applications running on service based Grids, i.e., suitable programming paradigms, execution infrastructure, and the ability to monitor and control such computations. Moreover, once computations are made of composable services, dependability becomes a key issue that needs to be addressed by the infrastructure as it cannot be addressed separately by each individual service. To address these concerns, we have developed the BioOpera Grid computing platform. BioOpera is a process support system for dependable cluster computing that has been extended with additional functionality to provide adequate support for service-based Grids. In this paper we describe how BioOpera can be used to develop, execute, and administer highly dependable computations over service-based Grids.

1. Introduction

Grid Systems are the infrastructure necessary for the generic and large-scale inter-operation of a broad variety of distributed computing resources. A common approach to build such an infrastructure is to provide a world wide single system image of all the resource involved (or to get as close as one can to this ideal). This requires mechanisms for sharing low level resources such as CPU cycles or data storage space, mechanisms that are grouped into middleware and development platforms like the Globus toolkit [15].

Useful as these platforms are, there are a few factors that speak for alternative solutions. First, sharing of low level resources is an attractive option only for experienced grid users and programmers. The vast majority of scientific users are overwhelmed by the complexities of developing and managing a computation running on such a Grid. Second, it is not always the case that the owners of the distributed resources want to share or grant access to raw CPU power or storage capacity. Offering concrete computational services (e.g., searches on a specialized database or running a proprietary algorithm over user supplied data) is a value added proposition that may make more sense from a business perspective (and of which there are already several interesting examples: [12, 13]).

These and similar observations have led to computing Grids where what is shared are not computational resources but *computational services*. It has even been suggested that only this approach will allow scientist to fully leverage the Grid as next generation computing platform [16]. In such *service-based* Grids, the emphasis is shifted from resource management to service composition, a shift that affects not only the infrastructure but also the programming model used for building Grid applications.

Following this idea, in this paper we present BioOpera, a process support system that addresses the entire life cycle of software applications on service-based Grids. For reasons of space, we ignore the problem of interoperability and related standards such as Web services [22, 24, 32] and OGSA [17]. Instead we focus on the core functionality of BioOpera. In the first place, BioOpera provides a programming paradigm particularly suited for service composition. A paradigm that lends itself quite well to visual composition, thereby greatly simplifying the development of complex Grid applications by inexperienced users. In the second place, BioOpera can be deployed in a variety of configurations to adapt it to different types of Grids. It can be used hierarchically (a BioOpera server invoking other BioOpera servers), peer-to-peer (several BioOpera servers invoking each other), or as execution engines that invoke services located at remote sites that do not run BioOpera. In the third place, BioOpera provides a comprehensive set of user interfaces for monitoring and administration of service-based Grid computations. These interfaces can be use directly by humans or as an *Application Programming Interface* to build additional tools, e.g., for debugging. When extended with support for Web services standards, all these features provide a complete development and run time environment for service-based Grids.

We believe that all these features are *per-se* valuable contributions as they contain generic mechanisms that can be used in a wide range of settings. Our goal, however, is even more ambitious. We are interested in providing an *autonomic* Grid computing platform [23]. Computational services are ideal for this purpose and several similar research efforts are underway in the area of electronic commerce [20, 21]. BioOpera has been designed from the very beginning with this type of autonomic behavior in mind. In the paper we show how the programming paradigm supported by BioOpera, the architecture of the system, and the different possible configurations can be used not only to develop and run service-based Grid applications but also to make them highly dependable. For instance, the experiments discussed in the paper show how BioOpera uses up to four different clusters to keep a computation alive without requiring any manual intervention in spite of dynamic changes in availability.

The paper is organized as follows: in Section 2 we introduce our programming model based on service composition. Section 3 presents the extensions to the Architecture of BioOpera relevant to grid computing. In section 4 we show some results about an experimental grid computation performed on clusters at four different locations. Section 5 briefly covers related work. Section 6 concludes the paper.

2. Service composition in the Grid

BioOpera [7] was originally a workflow engine used to implement experiments in virtual laboratories [2]. It was later extended to become a cluster computing platform for coarse-grained applications [4]. Before discussing how it has now been modified for Grid computing, we will describe the scenarios we have in mind for the new platform.

2.1. Service model

The service model we follow is similar to that used in the context of Web services [22, 24] (Figure 1). The basic idea is for *service providers* (sites B, C and D in the figure) to construct a series of computational services that they make publicly available either to the world in general or to a selected community of users. These services will be the basic building blocks for that particular computational Grid. There is no restriction on the nature of such services. They may be an application running on a single computer or fully-fledged Grid applications that involve several other Grid sites. Composition of services should be possible and transparent to the end user of the service.

We assume the services are described in a standard manner (e.g., using the *Web Services Description Language*, WSDL) and published in some repository shared by all members of that Grid. We will refer to these repositories as *service catalogs*. The catalogs might be centralized or they might be replicated across all participants so that each participant has its own local copy. In the current version of BioOpera we follow the latter approach as we are at the moment more interested in closed user communities. There is nothing, however, that would prevent the design from being extended to use a centralized catalog. BioOpera supports two types of catalogs: *public* and *private*. The public ones are made available to interested and authorized partners. The private ones can only be used within the domain that owns the catalog (e.g., a laboratory or a company).

The catalogs are part of the development environment provided by BioOpera. Application developers can pick a computational service from the catalog and *drag and drop* it into the *process* that describes the application (see below). Then they can use BioOpera to compile the process into an executable and run as well as monitor the progress of the Grid application using the BioOpera engine.

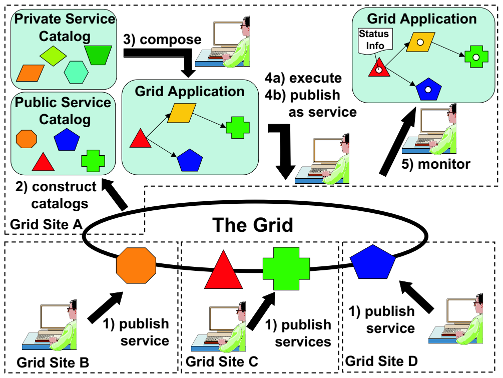

Figure 1. Service composition in the Grid

2.2. Programming model

To implement a Grid application in BioOpera, services must be composed into a *process*. In this aspect, BioOpera is not different from the growing number of cluster computing tools that use a similar approach [1, 10, 26, 29]. A process describes a set of service invocations as well as the dependencies between them. A process is a *Directed Acyclic Graph* (DAG), the nodes of which describe the computational services that make up the computation, whereas the edges represent either data or control flow dependencies between these services. BioOpera provides a visual programming tool (Figure 2) for service composition using a component library. For use with Grid services, this library has been extended to include the public and private catalogs just mentioned. Grid applications are therefore built by selecting services (and local computational components) from the library and establishing the flow of data and control between these components.

When the process is compiled, the result is a *process template*. These templates are uploaded as new library components and, if so desired, made available as a computa-

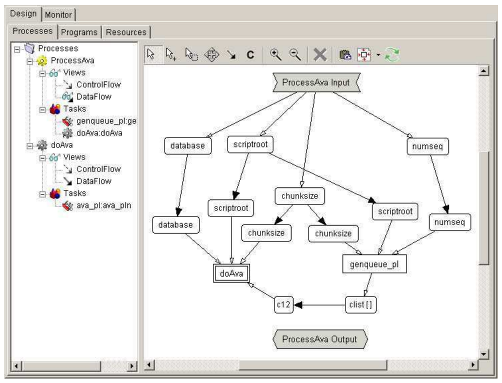

Figure 2. Visual programming tool

tional service to other Grid users. Unlike in other process based systems, templates in BioOpera only represent data and control flow. They do not contain information about the location or methods of invocation related to each service. These are determined at run time by the BioOpera *scheduler* by looking at the information available in the catalogs. This feature is very important for dependability purposes since it delays to the latest possible moment the decision of where to run a job. Upon starting the execution of a process, the process template is copied into a *process instance*. The process instance incorporates runtime state information concerning that particular execution. Instances of the same or different templates may be run concurrently by the same runtime environment.

2.3. Setting up services

Public and private services are set up in a very similar fashion. We will limit ourselves to describe the procedure only for public services. This procedure can be manual or automatic. The manual approach is needed to incorporate new library components that run in the local environment. The process is automatic when incorporating processes as components or importing into the catalog computational services provided by other sites in the Grid.

The first step in setting up a service for use in a process is to register the *resources* that provide the service. A *resource definition* consists of an IP address identifying the Grid site where that service can be found. BioOpera supports nesting of resource definitions by allowing to use *resource groups*. A resource group can be used, for instance, to identify subclusters of a bigger cluster. It can also be used to identify a set of single resources that provide the same service. Resource groups need to have a unique name, in analogy to individual resources. The set of all resource definitions describes all possible service access points in that particular Grid.

A service is made available by registering its *service definition*. This is done by uploading the service into a catalog. A service description consists of (1) the *service interface*, (2) a list of the resources that provide it, (3) the name of the appropriate execution subsystem to use to control the application implementing the service and (4) the service's *access method*.

The service interface is the list of the input and output parameters for that service. The list of resources that provide the service is used by the BioOpera dispatcher to determine at run time where to invoke that service. If more than one resource is eligible, BioOpera uses whatever load balancing policy is in place to select one. Also, note that this list is maintained dynamically by the BioOpera engine. If a resource fails or becomes unavailable, it will be removed from the list. When it becomes available again, it is added back to the list. This mechanism allows BioOpera to adapt the execution to dynamic changes in the configuration of the Grid.

The *execution subsystem name* identifies the BioOpera runtime system interface to be used to start the service. Through the execution subsystem mechanism, BioOpera may start services using a variety of different execution platforms (Sun Grid Engine [30], Condor [25], Portable Batch System, [5], CORBA [27], RPC [28], Web services [32], Condor-G [18]).

The access method encapsulates all information required by the service execution platform itself to start the service. For example, the access method for a Condor-G job may be a script. Using a Condor-G execution subsystem, BioOpera sends the script to a Condor submission node for execution. Access methods may be parameterized by service input parameters, which will be provided at runtime and will be used to complete the access method before calling a service.

3. BioOpera architecture

A description of BioOpera's architecture can be found in [4], in here we present only aspects relevant for this paper.

3.1. System components

As illustrated in Figure 3, BioOpera is designed as a three tiered client-server system. The front tier contains user interface tools for process design and monitoring. The middle tier groups all components related to process enactment support. The back tier incorporates the execution platform for the computational services.

In the front tier, BioOpera provides a variety of client tools, in an effort to offer appropriate support for different Grid users. The *Web Monitor* may be used to build Science Grid Portals [19] for specific applications and Grid communities. The *Process Design and Monitoring GUI* may be used by Grid programmers to design, develop, test and optimize their Grid applications in an integrated, visual development environment. Additionally, the *Command Line Client* opens up the system for building extensions such as

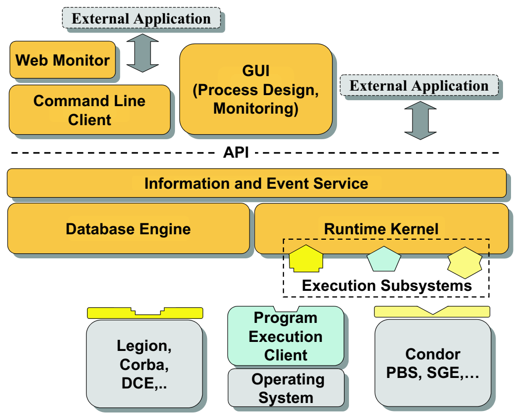

Figure 3. BioOpera architecture

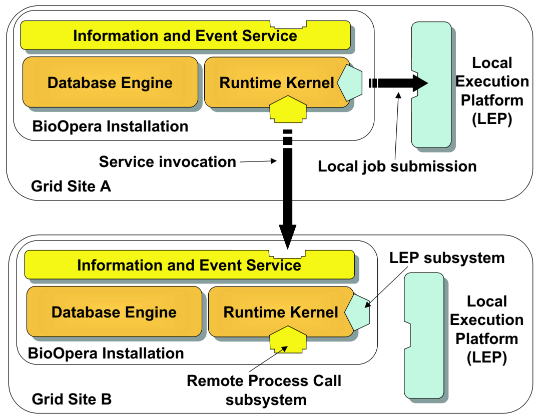

Figure 4. Remote service invocation

history mining tools, or for integrating calls to BioOpera processes into other applications.

The middle tier components are the *Information and Event Service* which provide a uniform point of access to the rest of the system, the *Runtime Kernel* dedicated to process execution, and a *Database* used to store process templates and instances as well as resource and service descriptions. The database is used by the information service to efficiently generate different views on the computation for the various clients. It is also used to keep consistent copies of the process instances and to guarantee their transparent recovery in case of failures. The database allows BioOpera to resume computations even after complete failures of all the system components [2].

The back tier consists of various task execution platforms. BioOpera provides mechanisms to run shell commands and scripts in a cluster environment, to execute processes on remote BioOpera systems and also supports a variety of execution platforms. Different execution subsystems may be plugged into the kernel at the same time, allowing processes to use computational services implemented on different execution platforms. Adding support for a new platform amounts to implementing a kernel plug-in that maps basic job submission, job status notification and load monitoring services to the specific mechanisms of the platform.

3.2. Remote service invocation

For Grid computing purposes, the BioOpera kernel has been extended with a remote process execution subsystem. For the purposes of this paper, this subsystem implements a protocol to start processes at a remote BioOpera server and a scheduler interface to gather load information about the remote site. With this subsystem, we can implement hierarchical grids where a master BioOpera server uses the computational services provided by other BioOpera servers. We can also implement peer-to-peer settings where each engine invokes computational services provided by the other engines. We are currently in the process of implementing additional subsystems and extending the existing ones, including extensive support for SOAP, the *Simple Object Access Protocol* [32].

Figure 4 shows how two BioOpera systems interact through the remote service invocation mechanism. To start a remote process, the local site A sends its name along with the input data to the remote site B. In return, it receives a token identifying the process instance running on the remote site. Whenever the remote process instance terminates, the remote event service notifies the local execution subsystem. In case of a successful termination, the result data from the remote site is downloaded and the kernel may proceed with the next task in the process. If the remote process fails, no results need to be retrieved and the local kernel decides how to handle the failure, either by failing the local process or by retrying the call.

3.3. Scheduling remote services

In the current version of the system, load information is propagated using a *pull* model: clients interested in a particular service query the providers about the load at their site. To collect load information from a remote site, the local runtime's scheduler periodically queries the remote site through the execution subsystem. This mode of operation can be combined with a *push* model by simply piggybacking load information onto job status notification messages. This would allow the local scheduler to make informed choices concerning to which remote Grid site a job should be sent to.

Currently, load information used by the remote process execution subsystem consists of a real number in the range from 0 to 1 indicating whether the remote site is willing to accept more service invocations. The local scheduler will submit jobs to the remote site until the index reaches its maximum value. Using this relatively simple load index, we assume that service providers are neither able nor willing to expose the details of their computing environment to clients. Each site may decide on its own how to calculate the load index. In the current implementation, for instance, the load index is proportional to the amount of jobs waiting on the remote site to be executed. Sites that usually experience high fluctuations in available computing power may prefer to have more jobs waiting in the queue to optimally react to sudden increases in available processing power. Sites with a more predictable computing environment may desire to keep the waiting queue as short as possible to allow clients to optimally use alternative Grid resources.

The approach we use is a reasonable compromise between obtaining useful information and keeping the necessary interfaces as simple as possible. Eventually, load information will be propagated using the notification mechanisms available in SOAP and WSDL.

3.4. Handling of service failures

In an effort to improve dependability of Grid computations, BioOpera also provides fault tolerance guarantees with respect to service failures. The local execution subsystem detects such failures on repeated unsuccessful connection attempts to the remote site. When that happens, we assume that all jobs submitted to the unavailable site have failed and the local scheduler will automatically reschedule all of these jobs. In this way, BioOpera guarantees that the computation adapts to changing service availability without any work being lost at any time, except for possible duplicated work done by remote jobs running at the time of the failure. The assumption that all jobs at a failing site also fail may not be appropriate in all cases. For instance, only a site's service access point may have failed, or a network partition may have occurred, implying that the jobs running on-site are still healthy, although it may be impossible to check their progress.

There are two possible approaches to handling this situation: if the site detecting the failure has access to a failure management infrastructure as proposed in [31], it may be used to investigate the cause of the failure before deciding how to react to it. If there is no such infrastructure, one can adopt a speculative strategy, and resubmit the jobs running at the failed site to an alternative location. Whenever the failed site becomes available again, a termination protocol needs to be run to identify redundant jobs and abort them.

4. Experiments

To test BioOpera as a computing platform for service-based Grids, we have performed an extensive set of experiments. In the following we present two of them. The first experiment shows how BioOpera executes a parallel application by sharing workload between redundant services provided by different Grid sites. The second experiment shows how BioOpera improves dependability of a computation despite service-level failures by exploiting service redundancy.

4.1. Grid test application

To conduct our experiments, we use a hierarchical configuration with a BioOpera server running a master process that invokes subprocesses at a number of other BioOpera servers. The computation involves a cross comparison of the SwissProt [3] protein database, a typical large scale Bioinformatics computation. In fact, this computation is one of the reasons to extend BioOpera to support Grid computing rather than just cluster computing. In the past BioOpera has been used to successfully run the computation [8] using a collection of medium size local clusters (amounting to a total of 192 CPUs) [2]. The latest version of SwissProt (v40.23) currently contains 110'236 entries, a cross comparison thus results in the order of 10^9 individual protein-protein comparisons. At this size, our local cluster starts to be insufficient to process the data, therefore we have been setting up the same computation using the computational services offered by clusters at other universities. Eventually, each one of these clusters will host different computational services and become part of a BioOpera based computational Grid. We also have plans to perform similar cross comparisons on demand and provide such comparisons as Web services implemented of top of BioOpera in its cluster computing version (essentially implementing the services currently available through a manual web interface and E-mail [12]). Then we will use the Grid version of BioOpera to create processes that combine the computational services offered by the participating universities.

For the experiments, each of the BioOpera servers implements and makes available a service that takes as input a set of proteins, runs the cross comparison on that set, and returns the matching sequences to the master BioOpera server. The data is exchanged using SOAP over HTTP. To study the ability of BioOpera to dynamically adapt to changes in service availability, the same comparison service has been implemented at the three sites. The process at the server site creates the jobs to execute by partitioning the SwissProt database and uses the computational services of the other BioOpera servers to execute these jobs.

To provide enough flexibility for running experiments of different duration, the partition sizes, as well as the fraction of the input data to use for a run may be provided by the user when starting the experiment. For both experimental runs, we chose these parameters so that the master site would generate 300 partitions, each requiring in the order of 10 minutes of CPU time to complete.

The Grid used for the experiments is described in Table 1. It consists of 4 clusters at 3 different locations in Switzerland and Canada. Each site runs its own BioOpera server, which requires two nodes. The remaining nodes are used for the computation. For the first experiment, we have

Table 1. Experimental Grid resources

Grid Site	Location	service type	nodes	CPU (Mhz)	RAM (MB)	OS
ETHZ	Zurich, CH	master	3	dual P-III (700)	512	LINUX V2.4.18
ETHZ	Zurich, CH	compare	12	dual P-III (1000)	1024	LINUX V2.4.17
EPFL	Lausanne, CH	compare	12	P-III (700)	128	LINUX V2.4.9
McGill	Montreal, CA	compare	12	P-III (1000)	512	LINUX V2.4.15

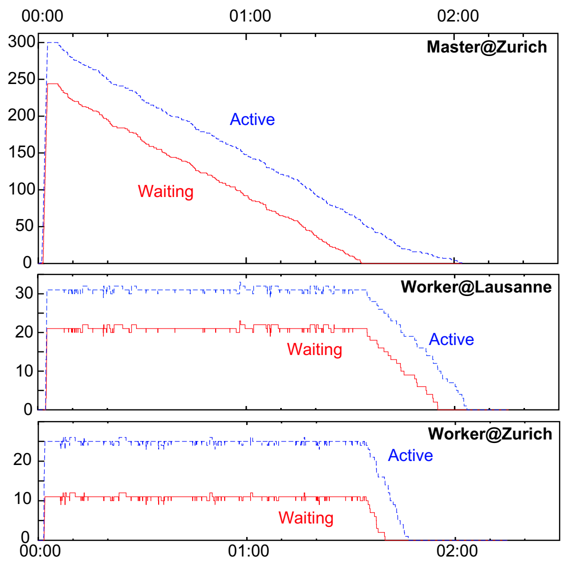

Figure 5. Load sharing among 2 Grid sites

only been using the clusters located in Zurich and Lausanne. The second experiment involves clusters in Zurich, Lausanne and Montreal. Note that each one of these clusters is managed by the local BioOpera server. Communication between the sites only happens in terms of the service being invoked through the BioOpera servers.

4.2. Load sharing among 2 Grid sites

The first experiment involves the master site as well as two clusters. In this experiment, we test the behavior of BioOpera as a Grid computing platform in a hierarchical configuration. We were particularly interested in the efficient distribution of jobs and an appropriate interactions between the BioOpera servers for invoking services, returning results and propagating load information. The results of the experiment are shown in Figure 5. The top part of the Figure shows the master, whereas the bottom part displays data measured from the two workers. For each site, the bottom curve displays the amount of jobs waiting in the scheduler queue (*waiting*), while the top one shows the total amount of jobs that are being processed by that site (or *active* jobs, which include those waiting and those running). On the y axis, the difference between the top and bottom curves represents the amount of jobs that have been submitted for execution either to the remote Grid site (for the master), or to the local execution platform (for the workers). The x-axis denotes time, the experiment took 2 hours to complete.

The experiment starts with the master generating the work partitions and inserting them into the scheduler's queue. Then it submits the jobs to the worker sites, using a minimum load placement policy. As jobs finish and no site failures occur, the amount of active jobs at the master site continually decreases. The bottom graphs show that each worker keeps its queue of waiting jobs at the desired length, since the master sends more jobs whenever it detects that the worker's waiting job queue length drops below a given threshold. At the master, this causes the amount of waiting jobs to drop proportionally to the amount of active jobs. At the worker sites, it makes the number of active jobs remain relatively constant throughout the computation. Imbalance may occur at the end of the computation, when the master has distributed all the jobs to the workers. Here, unfortunate placement decisions may have a negative impact on the overall completion time of the computation. In extreme cases, this effect may be reduced by forcing the master to balance the load by manually aborting jobs allocated to one site and putting them back into its scheduling queue. The BioOpera tools for process monitoring make such operation straightforward.

4.3. Adaptability to service failures

In the second experiment we wanted to text the ability of BioOpera to adapt to changes in the availability of the computational services used in a process. Of course, in processes where there are no alternatives to a given service, the failure of that service would halt the execution of the process. In many settings, however, it is possible to provide redundant services at different locations and let BioOpera deal with the availability changes. For these experiment we used three clusters in Zurich, Lausanne and Montreal.

The results of the experiment are shown in Figure 6. During the experiment the clusters are subject to site failures, which were triggered by shutting down and restarting the local BioOpera systems at the site. As in the previous experiment, the top part of the figure shows the number of waiting and active jobs for the master site. The lower graph of the Figure shows the availability for each of the three worker sites throughout the experiment. Time is again represented on the x-axis, the experiment took about 3 hours.

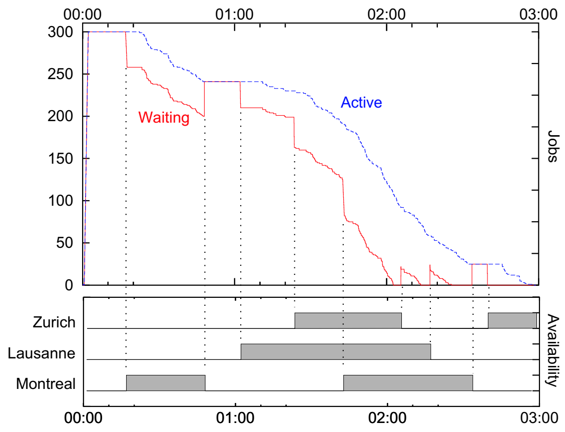

Figure 6. Adaptability to service failures

At the very beginning of the run, none of the worker sites is available. This is why all jobs are waiting in the master's scheduler queue: the *Waiting* and *Active* curves coincide. Then the Montreal site becomes available, and the master starts submitting jobs to it until the returned load information indicates that the site is busy. During this phase the amount of waiting jobs makes a sharp drop. After some time, remote jobs finish, which causes also the *Active* curve to drop. Again, since the master is supplying the workers with new jobs whenever needed, the *Waiting* curve keeps dropping. The first failure occurs at the Montreal site, which goes off line at about 45 minutes into the computation. This causes all active jobs that were allocated to Montreal to be rescheduled by the master, as indicated by the *Waiting* jobs curve joining again the *Active* curve. Since all services are unavailable, no work is done. The computation picks up again when the Lausanne site becomes available. As all clusters gradually go online, the slope of the progress curves increases. Toward the end of the computation, the master's job queue is empty, since all jobs have been submitted to some of the workers. Workers failing at this point may cause the queue to get filled again if the remaining workers do not have enough capacity to process the failed site's work right away. Such condition can be observed three times at the end of the computation.

The most important aspect of this experiment is that there is no manual intervention whatsoever, thereby proving the ability of BioOpera to exhibit some basic autonomic behavior. When combined with the already proven capabilities of each individual BioOpera server to maintain persistent state and recover computations even after total failures [2], the result is an extremely reliable platform for service-based Grid computing. Moreover, with these functionality already in place and working, we are now in a position to explore more sophisticated autonomic features that will greatly help with long lived, complex Grid computations. Examples thereof are *quality of service* guarantees for processes, tools for helping administrators in planning system outages, and load balancing and resource allocation policies across computational services.

5. Related work

The use of high-level programming environments for specifying Grid computations is advocated in [19] as enabling factor for doing science on the Grid. Several visual programming environments for developing Grid applications exist, e.g., Symphony [26], SCIRun [29]. They mainly focus on simplifying distributed application development and steering for non-expert users. GridFlow [9] exploits high-level metainformation about a computation to improve scheduling decisions at runtime. Much in the same way that BioOpera provides support for distributed computations that can be modeled as processes, AppLes [11] or Nimrod/G [14] provide support for programming and tuning special classes of applications for the Grid. The GrADS [6] initiative aims at identifying generic techniques for building dependable and performing Grid applications. The Open Grid Services Architecture [17] leverages existing Web service standards to promote interoperability in the Grid. Computational tasks as well as the Grid infrastructure itself are modeled as Web services. The OGSA specification addresses two dependability related issues in more detail: upgradeability of individual services at runtime and service instance lifetime management. Similar to OGSA, BioOpera enables service upgradeability by dynamically resolving the endpoint for a call to a specific service at runtime. Concerning lifetime management, OGSA uses a soft-state approach to deal with failures of service clients. The infrastructure does however not provide support for increasing the fault tolerance of service clients themselves. In contrast, BioOpera gives recoverability guarantees concerning every component involved in a computation. In that regard, it can be seen as a complement to existing implementations of the OGSA specification.

6. Conclusions

In this paper we have presented BioOpera, a Grid engine that supports the complete life cycle of a Grid application. BioOpera is particularly suited to service-based Grids. It emphasizes composition and provides powerful graphical tools for developing Grid applications. It also provides an execution engine that can be used in a variety of configurations to enact different forms of Grid computations. A significant advantage of BioOpera is that it has been designed for users who may not have an extensive expertise in parallel and Grid computing. For instance, BioOpera is capable of dealing with many failures and changes in the Grid configuration without requiring manual intervention. The experiments described in the paper demonstrate this basic autonomic behavior and open up exciting research opportunities that will yield more reliable, dependable, and user friendly Grid engines.

Acknowledgments Part of this work is supported by grants from the *Hasler Foundation* (DISC Project No. 1820) and the *Swiss Federal Office for Education and Science* (ADAPT, BBW Project No. 02.0254 / EU IST-2001-37126). We would also like to thank Prof. Mike Hallett and Prof. Robert Kearney at McGill University, and Prof. Andre Schiper at EPFL for allowing us to use their clusters.

References

[1] E. Akarsu, G. C. Fox, W. Furmanski, and T. Haupt. Webflow - high-level programming environment and visual authoring toolkit for high performance distributed computing. In *Proceedings of Supercomputing'98*, Orlando, FL, Nov. 1998. ACM SIGARCH and IEEE. Syracuse University.

[2] G. Alonso, W. Bausch, C. Pautasso, M. Hallett, and A. Kahn. Dependable Computing in Virtual Laboratories. In *Proc. of the 17th International Conference on Data Engineering (ICDE2001)*, Heidelberg, Germany, 2001.

[3] A. Bairoch and R. Apweiler. The SWISS-PROT protein sequence data bank and its supplement EMBL. *Nucleic Acids Research*, 27:49–54, 1999.

[4] W. Bausch, C. Pautasso, R. Schaeppi, and G. Alonso. Bioopera: Cluster-aware computing. In *Proceedings of the 4th IEEE International Conference on Cluster Computing*, 2002.

[5] A. Bayucan, R. Henderson, C. Lesiak, B. Mann, T. Proett, and D. Tweten. *Portable Batch System External Reference Specification*. MRJ Technology Solutions, May 1999.

[6] F. Berman, A. Chien, K. Cooper, J. Dongarra, I. Foster, D. Gannon, L. Johnsson, K. Kennedy, C. Kesselman, J. Mellor-Crummey, D. Reed, L. Torczon, and R. Wolski. The GrADS Project: Software Support for High-Level Grid Application Development. *International Journal of High Performance Computing Applications*, 15(4):327–344, 2001.

[7] BioOpera. *Process Support for Bioinformatics*. www.inf.ethz.ch/department/IS/iks/project_home_pages/bio-opera/.

[8] G. Cannarozzi, M. Hallett, J. Norberg, and X. Zhou. A cross-comparison of a large gene dataset. *Bioinformatics*, 16:654–655, 2000.

[9] J. Cao, S. A. Jarvis, S. Saini, and G. R. Nudd. Gridflow: Workflow management for grid computing. In *Proceedings of the 3rd IEEE International Symposium on Cluster Computing and the Grid*, Tokyo, Japan, 2003.

[10] H. Casanova and J. Dongarra. NetSolve: A Network Server for Solving Computational Science Problems. In *Proceedings of the 1996 ACM/IEEE Supercomputing Conference*, 1996. http://www.cs.utk.edu/netsolve.

[11] H. Casanova, G. Obertelli, F. Berman, and R. Wolski. The AppLeS Parameter Sweep Template: User-Level Middleware for the Grid. In *Proceedings of the Supercomputing*, 2000.

[12] CBRG. *Computational Biology Research Group*. http://cbrg.inf.ethz.ch/.

[13] Entigen. *BioNavigator*. http://www.bionavigator.com.

[14] R. B. et al. Nimrod/G: An architecture for a resource management and scheduling system in a global computational grid. In *Proc. 4th Int. Conf. on High Performance Computing in Asia-Pacific Region (HPC Asia 2000)*, Los Alamitos, California, USA, 2000. IEEE CS Press.

[15] I. Foster and C. Kesselman. Globus: A metacomputing infrastructure toolkit. *International Journal of Supercomputing Applications*, 11(2):115–128, 1997.

[16] I. Foster and C. Kesselman, editors. *The Grid: Blueprint for a New Computing Infrastructure*. Morgan Kaufmann, 1999.

[17] I. Foster, C. Kesselman, J.Nick, and S. Tuecke. The physiology of the grid: An open grid services architecture for distributed systems integration. Technical report, Service Infrastructure Workgroup, Global Grid Forum, 2002. http://www.globus.org/research/papers/ogsa.pdf.

[18] J. Frey, T. Tannenbaum, I. Foster, M. Livny, and S. Tuecke. Condor-G: A Computation Management Agent for Multi-Institutional Grids. *Cluster Computing*, 5:237–246, 2002.

[19] D. Gannon, R. Bramley, G. Fox, S. Smallen, A. Rossi, R. Ananthakrishnan, F. Bertrand, K. Chiu, M. Farrellee, M. Govindaraju, S. Krishnan, L. Ramakrishnan, Y. Simmhan, A. Slominski, Y. Ma, C. Olariu, and N. Rey-Cenvaz. Programming the Grid: Distributed Software Components, P2P and Grid Web Services for Scientific Applications. *Cluster Computing*, 5(3):325–336, 2002.

[20] M. Gillmann, J. Weienfels, G. Weikum, and A. Kraiss. Performance and availability assessment for the configuration of distributed workflow. In *Proceedings of EDBT*, 2000.

[21] M. Gillmann, W. Wonner, and G. Weikum. Workflow management with service quality guarantees. In *Proceedings of the ACM SIGMOD Conference*, 2002.

[22] K. Gottschalk, S. Graham, H. Kreger, and S. J. Introduction to Web services architecture. *IBM Systems Journal*, 41(2):170–177, 2002.

[23] IBM. Autonomic Computing: IBM's Perspective on the State of Information Technology, 10 2001. http://www.research.ibm.com/autonomic.

[24] T. Jeweel and D. Chappell. *Java Web Services*. O'Reilly, 2002.

[25] M. J. Litzkow, M. Livny, and M. W. Mutka. Condor—a hunter of idle workstations. In *Proc. of the 8th Int'l Conf. on Distributed Computing Systems*, pages 104–111, 1988.

[26] M. Lorch and D. Kafura. Symphony - a java-based composition and manipulation framework for computational grids. In *Proceedings of the 2nd IEEE/ACM International Symposium on Cluster Computing and the Grid*, 2002.

[27] Object Management Group (OMG). *CORBA: Common Object Request Broker Architecture*. http://www.corba.org/.

[28] Open Group. *DCE: Distributed Computing Environment*. http://www.opengroup.org/dce/.

[29] S. G. Parker and C. R. Johnson. SCIRun: A Scientific Programming Environment for Computational Steering. In *Proceedings of the 1995 ACM/IEEE Supercomputing Conference*, 1995.

[30] SUN microsystems. *Sun Grid Engine*. http://www.sun.com/software/gridware/.

[31] W. Vogels. World wide failures. In *Proceedings of the ACM SIGOPS European Workshop*, Connemara, Ireland, Sept. 1996.

[32] W3C. *Simple Object Access Protocol (SOAP) 1.1*, 2000. http://www.w3.org/TR/SOAP.