Global Computing Systems

C. Germain, G. Fedak, V. Néri, F. Cappello

LRI - CNRS and Paris-Sud University

Abstract. Global Computing harvest the idle time of Internet connected computers to run very large distributed applications. The unprecedented scale of the GCS paradigm requires to revisit the basic issues of distributed systems: performance models, security, fault-tolerance and scalability. The first parts of this paper review recent work in Global Computing, with particular interest in Peer-to-Peer systems. In the last section, we present XtremWeb, the Global Computing System we are currently developing.

1 Introduction

Global Computing is a particular modality of MetaComputing targeting massive parallelism, Internet computing and cycle-stealing. The key idea of Global Computing Systems (GCS) is to harvest the idle time of Internet connected computers, which may be widely distributed across the world, to run a very large and distributed application. All the computing power is provided by volunteer computers, which offer some of their idle time to execute a piece of the application. Thus Global Computing extends the cycle stealing model across the Internet. With more than 93 millions of Internet connected computers, and the revolutionary expansion of the mobile and handheld devices, the challenge is to harness so many unused computing resources to build a *Very Large Parallel Computer*. Due to poor network performance, GCS target mainly applications that can be broken down into coarse grain tasks, either independent or scarcely communicating. From some computer graphics programs to multi-parameter simulations in astrophysics or biology, such applications are sufficiently pervasive to motivate a high research and even commercial interest in GCS.

Over the past years, the popular success of cryptographic key cracking challenges and the SETI@Home [4] program have aggregated huge computing power, in the TeraFlop order. Extremely popular software such as Gnutella or Freenet [32] have shown that Internet-based data storage and retrieval is realistic.

GCS drastically depart from usual computer usage at a psychological and economic level [17]. Scientific and technical issues are less disrupted: performance models, security, fault-tolerance and scalability are part of the distributed systems framework. Nevertheless, the unprecedented scale of the GCS paradigm requires to revisit them all. In the next parts of this paper, we review recent work in Global Computing following these axes. In the last section, we present XtremWeb, the Global Computing System we are currently developing.

2 Architectures

A GCS is logically organized as a 3-tier system. The request layer submits a job. The broker layer marshals the request, then maps and schedules work, and the service layer actually computes. While the 3-tier organization is fairly common, GCS have two major originalities. First, the physical architecture does not map the physical one. Requesting, servicing and ultimately brokering are provided by the same resources, the Internet links and the collaborating computers. Second, the resources are highly volatile and users untrustworthy. Computers can come and go freely, and the same is true for users; the bandwidth, latency and security of Internet connections is highly variable.

All GCS do not fully implement this program. Implementing the service layer over non-dedicated computers (the *workers*) is the minimum. All commercial GCS and specialized ones (such as SETI@Home) do not allow for public access to the request layer, thus working in a Master-Slave mode. The so-called Peer-to-Peer (P2P) mode could tentatively be defined as allowing such access, with the hugely increased security and privacy problems implied. P2P systems are the focus of the current GCS research. Ultimate P2P systems would shift the brokering level itself to volatile resources. This issue has been much less explored for GCS than for data storage [32]. In this area, P2P systems offer a continuum of broker layer architectures, from centralized organizations such as Napster to fully decentralized ones such as Freenet. However, there is a commonality in these architectures : a brokering functionnality is offered, either by dedicated machines (centralized systems), or by the particpating machines themselves. In both cases, scalability is a major issue. For centralized systems, classical hierarchical organizations derived from the LDAP protocol are under investigation [26]. For decentralized implementations of the broker layer, only very preliminary results on the scalability issue are currently available [32].

3 Performance

Scheduling presumably is the key for application performance in the context of a GCS environment.

There is a general consensus about the fact that static information is inadequate for the development of efficient schedulers [12]. The obvious reason is that a Global Computer is essentially a shared resource, with external (with respect to the scheduler) users of the computing power and externally generated network traffic. Dynamic information on all resources of the Global Computer must be embodied in the performance modeling scheme, so as to provide forecasts that are one of the inputs of adaptive schedulers. Monitoring and prediction tools such as NWS [43] have been developed in the framework of MetaComputing systems, but not yet for GCS.

3.1 Scheduling

Predictions about processor and network workload will be used to map and schedule the tasks on a GCS. Due to the long-term unpredictable nature of this system [42], scheduling should be a dynamic process iterated many times as external conditions change. Scalability of the scheduling algorithm itself is a main concern, both with respect to the number of tasks and the size of the GCS.

The theoretical foundation for the analysis of scheduling have been laid in [35, 36, 14]. The basic model is that launching a remote task has a fixed and high overhead, regardless of data transfers. The overhead is an incentive to supply large chunks of work at a time, to amortize the overhead; the risk of loosing work in progress when a worker leaves suggests to supply a sequence of small chunks. An optimal schedule will balance these pressures in a way that maximizes the expected output, given the expected distribution of idle time on the workers. It has been shown in [36] that such optimal schedules do exist for a large class of distributions, and in [35] that they can be computed efficiently and dynamically. Unfortunately, the heavy-tailed distribution, which has been frequently verified as being typical distribution, does not fall into this class. However, [35] defines computationally simple schedules for the heavy-tailed distribution, which can be tuned to have expected work output that is arbitrarily close to the optimum.

Recent work has included the cost of data access in the mapping-scheduling problem. Computationally independent task may share large input files [18, 13]. Thus the workers must be clustered following their sharing of a storage resource. A special case of shared file is the job code itself. A very important question for schedulers is thus their ability to capture locality properties. The *Sufferage* heuristic defined in [28] captures host locality: the sufferage index of a task is the difference between its best and second-best execution time for a given scheduling; tasks with higher sufferage index take precedence. [18] shows how to evolve the Sufferage heuristic to capture cluster locality.

Tasks which exhibit a data-dependent execution time cannot be scheduled once for all. Distributed workstealing scheme following the Cilk model [16] are the main scheduling policy of Atlas [9] and Javelin++ [29]. In the original shared memory multithreaded framework of Cilk, a thread pushes work to be done to a stack, where other idle thread can pick it. The main issue in extending this scheme to Global Computing is scalability. Atlas and Javelin++ achieve scalability through a tree-structured selection of the host to steal. Alternatively, Javelin++ proposes a policy of random choice with tables of known hosts that matches the P2P information storing structure.

3.2 Performance Models

In a production environment, the predictions got from a monitor/predictor such as NWS will drive a scheduler. Research and benchmarking on scheduling raise another issue. Effective investigation and objective comparison of scheduling algorithms would require a performance evaluation system that allows analysis and

comparison of these algorithms under a reproducible, configurable and controlled environment.

One model has been defined in the Bricks project [39]. The entities of the model are clients, servers and network links. Servers and network links are modelled by queuing systems, clients by arrival rates and tasks by the volume of data they have to transfer to/from the servers and the number of instruction they execute. Tasks and data transfers include those issued by clients as well as those invoked by external processes. Storage performance is not explicitly modelled, but could probably be included. Experiments against real monitoring/prediction tools report Bricks to be very accurate, but the simulator is not publicly available.

4 Fault Tolerance

GCS largely stretch the concept of Fault Tolerance. The issue becomes how to compute efficiently in an environment where faults are normal, not exceptional, events.

Fault tolerance is an issue both at the broker and the service level. When physically distributed, the broker level should maintain a consistent view of a distributed data space, which is a classical problem. Full P2P systems devoted to file storage and retrieval have implemented broker faut-tolerance based on redundancy. Failure-resilient distributed data space at the programming level have been defined in [8] and in JavaSpaces.

4.1 Volatile Workers

At the worker level, a GCS has to ensure that the computation will make some progress, at long as functional resources are available. However, defining what is a functional resource is somehow blurred in such systems. The most traditional way is to consider only *online* resource, that is computers currently registred in the system. As soon as they do no more appear as registred, they are declared faulty, their assigned task is lost, and must be restarted from scratch, except for checkpointing. At the other end of the spectrum, one may consider that resources come and go, and that is does not make sense to base the policy of them, but only of the tasks to perform. If there exists a registration system, this allows computations to carry on *offline*, when a computer is technically faulty.

With offline systems, the problem id now how to deal with truly faulty (never coming back) resources. One solution has been proposed in the framework of parallel computing, with the concept of *eager scheduling* [6]. Eager scheduling is not a specific scheduling policy, but a layer over such policy. When all available work has been assigned, unfinished tasks are re-assigned to workers which become idle. This principle has been implemented in Charlotte [10], Bayanihan [37] and Javelin++ [29]. Fault-tolerance is then only a modality of scheduling, a faulty worker being viewed a an infinitely slow one.

Eager scheduling is well-adapted to embarrassingly parallel problems, where there are clear synchronization points at which work must be completed before the next step can proceed. At this point, there is no harm in rescheduling unfinished computations. A GCS which targets continuous computing of independent individual tasks, e.g. a very large multi-parameter application, never fulfills the condition of having no further work to assign. Thus, the eager scheduling algorithm would have to be augmented with decision about when to start a re-scheduling phase.

4.2 Checkpointing

Another important issue is long-running tasks. To guarantee their progress in a volatile environment, some form of checkpointing must be implemented. In essence, checkpointing is any technique that allows for saving the state of the computation so as to restart it from the reached point. Long-running applications generally include a simple form of checkpointing through files. In the online scheme, these files should be saved through the network, while in the offline scheme they could be written only locally.

Checkpointing through file saving leaves the all the burden to the application. Ninflet [38] has proposed a more elaborate solution. The programmer is responsible for inserting calls to a checkpoint method at appropriate places, to skip over what could have been computed when the task resumes, and to take into account checkpoint limitations. The checkpoint method itself is provided by the Ninflet environment. The method uses Java Serialization to save the task object to stable storage. Finally, automatic scheduling of checkpoints [20] merges with task scheduling in the model of [35].

Another possibility is thread or process migration [5,3]. However, deployment of native process migration, even on a cluster, yet suffers from limitations, especially for I/O and network access [3]. Java-based Global Computing systems are in a much better position to this respect, with already deployed mobile technologies such as ObjectSpace Voyager and Aglets.

The major problem with checkpointing, is that the state of a long-running computation is often much larger than the final result. In an online scheme, the pressure on network bandwidth may be very large. Thus checkpointing would probably be more convenient in an offline scheme, or would have to build on existing P2P data storage and retrieval technology.

5 Security

In the P2P scheme, workers will run completely untrusted code, which may be malicious or erroneous. Encryption techniques, such as SSL, provide reliable data and code transmission, but this is only a small, if mandatory, step, to security. Moreover, the privacy of the host running the worker must be guaranteed. This level of protection requires the worker to be run in an environment that isolates it from the physical host resources.

Sun's Java has been the first integrated and modular sandboxing solution. A pointer-safe language executed in a virtual machine with extended dynamic type and array-bound checking protects against malicious or erroneous use of processor resources. Security models allow to limit access to the network and peripherals (displays and files), in a configurable way since Java 1.2. Many Global Computing projects have used the most secure Java framework, namely Applets [19, 10, 33, 31], or Java applications with adequately configured security policy [29, 38].

Sandboxing native code execution has been recently developed, both at compile time and at run-time. Software Fault Isolation [41] restricts an object code from writing or jumping to a memory address outside of a separate portion of application's address space called the *fault domain*. This is done by inserting in the binary code some run-time check before the store or jump instructions. This has approach has the drawback of inducing an overhead proportional to the execution time. Self Certifying Code [30] is an attempt to avoid this overhead. The execution site provides a safety policy expressed as a set of rules and according to it the code producer creates a formal *safety proof* that the untrusted code respects this policy. Before being executed the proof is validated and this step is the only needed overhead. Extension of the compiler [40, 21] can analyze the code to enforce a safe use of the commonly exploited function of the C library (scanf(), strcpy()...), to prevent attack buffer overrun attacks.

In the run-time approach, the key idea is to monitor the execution of a process and allowing only safe operations. A safety mechanism is interposed between the process and the operating system. This interposition mechanism can be sited either at the C library calls level (libsafe [11]) or at the system calls level (Janus [27], Consh [2], MAPbox [1]). The owner of machine expresses in a safety policy the resources the application is allowed to use, mainly network and file usage.

Extending this principle of interposition, the User-Mode-Linux [22] project offers a complete virtual machine dedicated to the execution of the native code. The safety policy is provided by the configuration of the virtual machine, for instance mapping the virtual file system to a specific file of the execution site.

Finally, workers are not only volatile, but also potentially malicious. In the SETI@Home experiment, some workers have replaced the original version of the worker code by a patched one. The altered code was supposed to be faster, but the results were both formally acceptable by the server and erroneous from the application point of view. While this may appear as an instance of very classical problems of distributed computing, checking and correcting the results of a computation operated by massive parallelism and independent tasks cannot for instance be naturally modelled as a consensus problem.

6 XtremWeb

The XtremWeb project [25, 23] aims at building a platform for experimenting *high performance* GCS. XtremWeb1.0 is currently available for download at [34],

allowing any institution to setup its own GCS. Two applications are currently run under XtremWeb. The first one is a large simulation in the field of high-energy particles physics, in collaboration with the Auger experiment [7]. The other one is a PovRay computation, with most stringent requirements for both security and data transfer.

6.1 Architecture

The XtremWeb architecture falls in the P2P model, with a broker layer implemented on non-volatile and trustworthy machines. The architecture is strictly pull-based: all activities and communications are initiated by the service or the request layer, and go to the broker layer. This allows an easier large-scale deployment because firewalls may be block communications in the reverse direction.

Because XtremWeb targets high performance, the applications are native binaries. The request layer is based on standard protocols and tools (PHP, MySQL). The broker layer is organized as a set of queues of submitted and activated tasks. The scheduling policy of the activated tasks can be freely configured on the fly, while a broker is running. Electing submitted task to the activated task queue is also fully configurable. A monitoring/prediction tool is under development.

The service layer describes what resources the worker may use and enforces this policy. The availability of a given machine depends on the User presence (detected through the keyboard or mouse activity), the presence of non-interactive tasks (detected through the CPU, memory and I/O usage) and other conditions like night and day for instance. Resource utilization is continuously monitored by the worker. An interface to the resources is provided by Operating System features, e.g. the /proc directory for the Unix OSes. A User defines an availability policy simply by indicating for each resource a threshold above which the computer is usable for a Global Computation and a threshold that provokes the interruption of the computation.

Controlling the resources used by the Global Computation can be tuned. In the current and simplest scheme, the global computation obtains none of the resources of an used machine and all the resources of a unused machine. However, we are currently working on an integrated solution for allowing a User to limit selected resources consumed by the worker (e.g. disk or memory usage), or conversely to allow the global Computation to share some resource even when the computer is used, for instance by nicing the global application instead of stopping it.

The broker-worker interface have been implemented in Java. However, the design is deliberately independent of Java specificity, especially dynamic class loading. The first reason is that some performance bottlenecks may be created by Java, and we wanted to be free to rebuild the critical parts with other tools. For instance, the service/broker protocol is currently implemented over RMI. RMI calls create threads, and Linux thread creation may scale poorly. Experiments are ongoing to show if the performance degradation remains acceptable.

6.2 Service/Broker Protocol

A worker is a machine identified by its name and its owner. The protocol between a worker and the broker consists of four requests detailed below:

The first request *hostRegister* goes to the last contacted broker or to the root broker. This first connection authenticates the broker to the worker. The broker sends back what is called a communication vector. The communication vector specifies the list of brokers that may provide tasks to the worker and the communication layer (protocol and port) on which they may be contacted. In the simplest case, the broker may return its own address.

Next, the worker asks for a job from the broker, through the *workRequest* request. The worker provides a description of its runtime environment (e.g. operating system, architecture, etc.) and the list of the binaries previously downloaded and stored in a local cache directory. According to this information, the broker selects a task, and sends back to the worker a description of the task, the task inputs, the binary of the application corresponding to the runtime of the worker if necessary, and the address of a broker that is able to store the results.

During the computation the worker periodically invokes *workAlive* to signal its activity to the broker. The broker continuously monitors these calls, to implement a timeout protocol. When a worker has not called for a sufficient long time, the worker is considered down and its task may be rescheduled to another worker.

At the end of the computation the worker sends back results to the specified address, through the *workResult* call. This call is echoed back to the broker which has provided the work, so as to signal the completion of this piece of work.

7 Conclusion

In a recent paper [24], I. Foster attempted to define the general organization underlying a Grid architecture. Layers and protocols are defined, and exemplified on the Globus system. Research in Global Computing systems has not yet reached this maturity. Contrasting with the modular ("hourglass") model of Grid architectures, most GCS are vertically integrated. In the first sections of this paper, we have sketched some aspects of the parameter landscape for GCS along some major axes (scheduling, coping with volatility, checkpointing, security) and situated some major GCS projects in this landscape. In the last part, we have presented our own GCS system, as focused on experimentation rather than in-depth exploration of one point of the landscape: while no system can pretend to offer the possibility to combine freely all the possible choices, XtremWeb is an attempt to provide a testbed for a significant class of GCS.

References

1. A. Acharya and M. Raje. MAPbox: Using parameterized behavior classes to confine applications. Technical Report TRCS99-15, 31, 99.

2. A. Alexandrov, P. Kmiec, and K. Schauser. Consh: A confined execution environment for internet computations. In *USENIX Ann. Technical Conf.*, 99.
3. Y. Amir, Awerbuch B., Barak A., Borgstrom R. S., and Keren A. An Opportunity Cost Approach for Job Assignment in a Scalable Computing Cluster. *IEEE Trans. Parallel and Distributed Systems*, 11(7):760–768, 2000.
4. D. Anderson and al. A New Major SETI Project Based on Project Serendip Data and 100,000 Personal Computers. In *5th Intl. Conf. on Bioastronomy*, 1997. http://setiathome.ssl.berkeley.edu/.
5. G. Antoniu, Luc Bougé, and Raymond Namyst. An efficient and transparent thread migration scheme in the PM2 runtime system. In *3rd Workshop on Runtime Systems for Parallel Programming*. LNCS 1586.
6. Y. Aumann, Z. M. Kedem, K. Palem, and M. Rabin. Highly efficient asynchronous execution of large grained programs. In *34th IEEE Symp. on Foundations of Computer Science*, 93.
7. The Auger project. http://www.auger.org.
8. D. E. Bakken and R. D. Schlichting. Supporting fault-tolerant parallel programming in Linda. *IEEE Trans. on Parallel and Distributed Systems*, 6(3):287–302, 95.
9. J. E. Baldeschwieler, R. D. Blumofe, and E. A. Brewer. Atlas: An Infrastructure for Global Computing . In *7th ACM SIGOPS*, 96.
10. A. Baratloo, M. Karaul, Z. Kedem, and P. Wyckoff. Charlotte: Metacomputing on the Web. *Future Generation Computing Systems*, 15:559–570, 99.
11. A. Baratloo, T. Tsai, and N. Singh. Transparent run-time defense against stack smashing attacks. In *USENIX Ann. Technical Conf.*, 2000.
12. F. Berman. High-performance scheduling. In *The Grid: Blueprint for a New Computing Infrastructure. I.Foster and C.Kesselman Eds.* Morgan Kaufmann, 97.
13. M. Beynon, T. Kurc, A. Sussman, and J. Saltz. Design of a Framework for Data-Intensive Wide-Area Applications. In *9th Heterogeneous Computing Workshop*, 2000.
14. S. N. Bhatt, F. Chung, F. T. Leighton, and A. L. Rosenberg. On Optimal Strategies for Cycle-Stealing in Networks of Workstations. *IEEE Trans. on Computers*, 46(5), 97.
15. M. Blum and H. Wasserman. Software reliability via run-time result-checking. *Journal of the ACM*, 44(6):826–849, 97.
16. R. D. Blumofe and al. Cilk: an efficient multithreaded runtime system. In *5th PPoPP*, pages 205–216, 95.
17. R. Buyya, D. Abramson, and J. Giddy. Economy Driven Resource Management Architecture for Computational Power Grids. In *International Conference on Parallel and Distributed Processing Techniques and Applications*, 2000.
18. H. Casanova, A. Legrand, D. Zagorodnov, and F. Berman. Heuristics for Scheduling Parameter Sweep Applications in Grid Environments. In *9th Heterogeneous Computing Workshop*, pages 349–363, 2000.
19. B.O. Christiansen and al. Javelin: Internet-based parallel computing using Java. *Concurrency: Practice and Experience*, 9(11):1139–1160, 97.
20. E.G. Coffman, L. Flatto, and A. Y. Kreinin. Scheduling saves in fault-tolerant computations. *Acta Informatica*, 30:409–423, 93.
21. C. Cowan and al. StackGuard: Automatic adaptive detection and prevention of buffer-overflow attacks. In *USENIX Ann. Technical Conf.*, 98.
22. J. Dike. A user mode port of the Linux kernel. In *4th Ann. Linux Showcase*, 2000.
23. G. Fedak, C. Germain, V. Néri, and F. Cappello. XtremWeb: A Generic Global Computing System. In *IEEE Int. Symp. on Cluster Computing and the Grid*, 2001.

24. I. Foster. The Anatomy of the Grid: Enabling Scalable Virtual Organizations. In *IJSA*, 2001
25. C. Germain, V. Néri, G. Fedak, and F. Cappello. XtremWeb : Building an Experimental Platform for Global Computing. In *1st IEEE ACM Intl. Workshop Grid 2000*, 2000.
26. I. Kuz, M. van Steen, H.J. Sips. The Globe Infrastructure Directory Service. In *Proc. 7th ASCI Conference* May 2001, pp. 115-122.
27. I. Goldberg, D. Wagner, R. Thomas, and Eric A. Brewer. A secure environment for untrusted helper applications – confining the wily hacker. In *6th USENIX Security Symposium*, 96.
28. M. Maheswaran and al. Dynamic matching and scheduling of a class of independent tasks on heterogeneous computing systems. In *8th Heterogeneous Computing Workshop*, 99.
29. M. Neary and al. Javelin++: Scalability Issues in Global Computing. In *ACM Java Grande'99 Conf.*, 99.
30. G. C. Necula and P. Lee. Safe kernel extensions without run-time checking. In *2nd Symp. on Operating Systems Design and Implementation.* pages 229–243,96.
31. N. Nisan, S. London, O. Regev, and N. Camiel. Globally distributed computation over the internet-the POPCORN project. In *18th Int. Conf.on Distributed Computing Systems*, 98.
32. A. Oram. *Peer-to-Peer: Harnessing the power of disruptive technologies.* O'Reilley, 2001.
33. H. Pedroso, L. M. Silva, and J. G. Silva. Web-based metacomputing with JET. In *ACM 97 PPoPP Workshop on Java for Science and Engineering Computation.*
34. The XtremWeb Project. http://www.xtremweb.net.
35. A. L. Rosenberg. Optimal schedules for data-parallel cycle-stealing in networks of workstations. In *12th ACM SPAA*, pages 22–29, 2000.
36. A. L. Rosenberg. Guidelines for Data-Parallel Cycle-Stealing in Networks of Workstations, I; on maximizing expected output. *JPDC*, 59:31–53, 99.
37. L. F. G. Sarmenta, S. Hirano, and S. A. Ward. Towards Bayanihan: builing an extensible framework for Volonteer Computing using Java. In *ACM Workshop on Java for High-Performance Network Computing*, 1998.
38. H. Takagi and al. Ninflet: A Migratable Object Framework using Java. In *ACM Workshop on Java for High-Performance Network Computing*, 98.
39. A. Takefusa and al. Overview of a performance evaluation system for global computing scheduling algorithms. In *8th Int. Symp. on High Performance Distributed Computing*, pages 97–104, 99.
40. D. Wagner, J. S. FOster, E. A. Brewer, and A. Aiken. A first step towards automated detection of buffer overrun vulnerabilities. In *Networking and Distributed System Security Symp.*, 2000.
41. R. Wahbe, S. Lucco, T. E. Anderson, and S. L. Graham. Efficient software-based fault isolation. *ACM SIGOPS Operating Systems Review*, 27(5):203–216, 93.
42. R. Wolski, N. Spring, and J. Hayes. Predicting the CPU Availability of Time-shared Unix Systems on the Computational Grid. In *8th Int. Symp. on High Performance Distributed Computing* , 99.
43. R. Wolski, N.T. Spring, and J. Hayes. The Network Weather Service: A Distributed Resource Performance Forecasting Service for Metacomputing. *Future Generation Comp. Sys.* , 99.