Grapevine: An Internet Distributed Computing Framework

Kunal Agrawal and Kevin Chu

May 20, 2002

Abstract

Tapping into the unused resources of volunteer computers connected to the internet is not a new idea. Indeed, there are currently many active efforts in this area. While many early distributed computing efforts targeted specific applications, more recent work has focused on the development of general distributed computing infrastructure. Leaders in this field include Parabon Computation, United Devices, the Globus Project and the Alice project. Each of these efforts has developed software components that supports distributed computing.

For our class project, we explored some fundamental and practical issues in distributed computing on volunteered nodes connected to the internet by designing and implementing a simple distributed computing infrastructure. To gain an appreciation for the types of computing applications that are suited for a distributed computing environment, we worked with Ang Huey Ting and Li Guoliang on the Distributed Meta Classifiers project to deploy their algorithm in on the infrastructure.

1 Introduction

Before delving into the detailed design of the Grapevine system, it is important to put the idea of internet, distributed computing into context. First, the key driving forces behind internet, distributed computing are not related to performance or speed. For performance, one should look towards dedicated computing machinery. Rather, the motivations for distributed computing arise from less technical considerations. For example, for some problems, it may be acceptable exchange an increase in computing time for decreased space, hardware or cost requirements. This trade-off becomes especially important for the largest of problems where the cost-benefit curve rises steeply due to the extermely high cost of very large parallel machines. In some senses, the notion of performance need to be widened in order to appreciate internet, distributed computing. Traditional metrics, such as flop count or speed up, are inadequate for understanding and evaluating distributed computing systems.

From a computer science viewpoint, standard parallel computing issues such as communication vs. computation, node reliability, and load balancing must be addressed. These issues are amplified in a distributed computing environment deployed over the internet due to the poor quality of communication over the internet as compared to communication on a dedicated parallel computer. Understanding the implication of these issues in the context of "internet supercomputing" is essential if supercomputing is to progress beyond physical and economic bounds of dedicated machinery.

However, there are fundamental differences between a massively-parallel computer and an internet, distributed computing framework (IDCF). Perhaps the most significant is that the nodes of the latter are *volunteered*. That is, unlike traditional parallel computing, an IDCF cannot be designed to maximize utilization of the volunteered computing resources because the machines are not dedicated to running computation intensive jobs. Rather, it must be designed to harvest only idle compute cycles.

Further differences between dedicated parallel hardware and an IDCF arise from practical considerations. From the perspective of a volunteer node owner, security against malicious code and overuse of computing resources are serious issues of concern. Unfortunately, these concerns are somewhat at odds with users' desire for flexibility and expressiveness in programming and their need for computational resources. In addition, adequate computing resources can only be realized with a sufficiently large base of volunteer nodes. This last difficulty is particularly interesting (and challenging) because successfully addressing it lies, to a large extent, outside of the hands of the developers. Clearly, the design of IDCFs involves compromises between many competing factors.

While we were not able to address all of these concerns in the Grapevine system within the time constraints of a course project, we did explore possible ways to addresss some of these problems.

Communication vs. Computation Because the internet provides such low quality communication, applications requiring a large amount of communication are expected to perform poorly. However, data and task parallelism for embarrassingly parallel computations should run reasonably well. It turns out that a large class of problems demonstrate these features. For example, the Bagging AI application (Distributed Meta Classified project) does not require any communication at all. For these problems, it is natural to distribute to the problem among many independent machines and have the work be done simultaneously.

Having programs efficiently manage the trade-off between computation and communication is primarily the responsibility of the programmer. However, the computing framework must provide sufficient flexibility and expressiveness for this to be possible. In our design, we tried to minimize the constraints placed on the user code by assuming a very general computing model. The details of this model will be described in a later section.

Node Reliability The poor network quality of the internet and the fact that ownership and administration of the volunteer nodes is not centralized makes node reliability a serious issue for IDCFs. For instance, how should the system respond to a node that loses its network connection or is rebooted in the middle of a computation? Or, what if a defective node is producing inaccurate results?

While we did not have time to implement any node reliability protocols in the Grapevine framework, we considered some possible strategies during our design phase. One strategy is to periodically send keep alive messages from volunteer to server. But this method is very unreliable by itself since the server may not get the messages at times. It also puts a lot of pressure on the communication bandwidth as well as the server. So we decided that this was not a very good idea. The strategy we considered most seriously was redundant assignment of tasks to volunteers. Theoretically, redundancy helps mitigate stability issues by increasing the probability that at least one result is returned. It should also help address accuracy issues by allowing results from multiple computations to be compared before accepting a final result to be returned to the user.

One drawback of this strategy is that it complicates the semantics of intertask communication.If more than one volunteer is executing a task, then all the sends will be done by both of them. Anything that is to be received should also be sent to both of them. This problem also exists when they are reporting Results. How this is handled in Grapevine is explained in detail in a later section.

Load Balancing The heterogenous nature of IDCFs makes load balancing a challenge. As with the node reliability, we were unable to implement a load balancing solution in the Grapevine system, but we did spend some time exploring possible solutions. Our main approach to address this problem involved estimating both the capacity of each node and the size of tasks.

To estimation of node capacity, we planned to use a dynamic update scheme. First, as part of the registration process, the volunteer node would send some information about itself, such as processor speed and amount of volunteered disk space, to the server. These data would be used to initially estimate the capacity of each new node when assigning tasks to nodes. As tasks are assigned and completed by the node, statistics would be collected on task size, time to completion, accuracy of results, *etc.* These statistics would then be used to dynamically update the capacity of the node. Note that this scheme allows the load balancing mechanism dynamically adapt variations in capacity of individual volunteer nodes.

Estimating the size of tasks is somewhat more difficult because empirical data is not available to aid in the estimation process. Instead, we had intended for users to provide some resource estimates for their application as part of the job submission process. To avoid gross misestimatation, we had also considered using information about the size of data and executable elements required by a task in estimating its size.

Security Security is a very important issue in an IDCF. The last thing that any volunteer would want is malicious code fragments running on his machine. To minimally address this concern, we leveraged the Java Security model. By granting access to only required computing resources, such as specific parts of the file system, we restricted the amount of damage that a user task could do to the volunteer node. This security policy, however, is quite coarse grained, so it could certainly be improved.

2 The Grapevine Framework

The design of the Grapevine infrastructure is loosely based on the design of the Parabon Computation distributed computing framework and the Globus Resource Allocation Manager system. We extracted the key features of both systems as the foundation for our framework.

2.1 Grapevine Features

In order to constrain the scope of the project, we focused on delivering a framework that supports four main features.

Parametrised operation of tasks One important type of application for parallel computers is parametrised tasks. The same work is done again and again but with different parameters. Furthermore, each individual task is not very expensive and can be run on one machine, so a large number of parameter settings can be tried. This type of job is perfect for distributed computing as there is no communication. Grapevine supports this by allowing the client to independently set the parameters for each task. The Grapevine server then send the parameter list over to the appropriate volunteer.

Intertask communication From the client's point of view, the intertask communication semantics are similar to the MPI interprocessor communications. We give the task full control over what it wants to send and to whom. For this purpose we have some functions in the task class which should not be overridden by the subclass defined by the client. The details are given in the User's Manual.

It is important to consider the semantics of intertask communication. For the following, assume that a task on volunteer A wants to send Object O to a task on volunteer B. For simplicity we will refer to both the task and the volunteer as A or B, but it should be remembered that tasks and volunteers need not be uniquely paired. From the implementation point of view, the process of intertask communications is as follows.

1. Task A invokes method to send O to B.
2. The daemon on A gets this information and tries to send this to the server. Meanwhile, task A can continue with its own processing. Hence, sends are non blocking.
3. When the server gets O, it stores it.
4. When task B comes across a Receive and sees that it is supposed to get something from A, it tells the daemon running on B. This daemon contacts the server asking for it. It keeps trying until the server sends the required object and then it goes on. Thus, receives are non blocking.

We chose this particular way of communication where volunteer nodes initiate all the communications (as opposed to the server sending out O to B as soon as it gets it) for two main reasons. First, as the number of volunteers increase, the burden on the server will increase. To place this load on the server is unreasonable because it will make the server slower. Second, the volunteers are on the internet, so they might not be connected to the server at all times. As a result, the server may have to retry many times before it can get through to the volunteer. On the other hand, we assume that the server is always online (unless there some malfunction in the system or network).

Results reporting The result reporting semantics of Grapevine are very simple. After a task finishes on the volunteer, it can send the results to the server by just calling a function defined in the task base class. The implementation of this feature is very similar to the implementation for intertask communication. In fact, we used common mechanisms for both. After computing the results, the task sends them back to the server, and the server stores them in a table. These are sent to the client when the client asks for them. A similar mechanism is used to provide a facility for intermediate result reporting if the client wants to.

Status reporting The current status reporting in Grapevine is very simple. Whenever the server gets the final results from all the tasks running the particular job, the server sets the status of the job as done. The client can check the status of the job using the JobID and can ask for the full results when the job is done. Also, the client can check the progress of the job at any time to see how many tasks of a given job have been completed and can get the results for those tasks. We planned to implement more complex status reports from tasks but did not have sufficient time. However, we have the basic infrastructure to do this and it can be implemented in future.

2.2 The Choice of Java

In pure performance terms, Java is considered to be inferior to C/C++ and Fortran. However, its cross-platform portability and its large supporting library make it attractive for software development in a heterogenous, distributed computing environemnt. In particular, Java provides easy to use and well-documented networking, file I/O, threads, and remote object libraries. Furthermore, we were able to leverage Java's security model to provide basic protection for volunteer nodes against malicious code.

The theoretical portability of Java byte-code across different platforms is a major advantage of Java over other compiled programming languages where the object code is platform dependent (*e.g.* C or C++). Using such a programming language, would require a certain level of homogeneity among the machines in the distributed computing network. This would drastically limit the ability of the IDCF to harvest idle computing resources on the internet. In contrast, the Java byte-code can run on any machine as long as there is a Java Virtual Machine (JVM) available to interpret the byte-code. Since Sun provide JVMs for all major computing platforms, a large fraction of the machines connected to the internet should be able accessible to the system.

From an implementation standpoint, Java was a very convenient choice because it provides libraries for all the basic building blocks that we required for the Grapevine system. For example, remote method invokation (RMI) pakcage allowed us to use remote object interfaces to facilitate communication over the network rather than having to design, implement, and debug our own network protocols. In addition, the object-oriented nature of Java allwed us to express and enforce class relationships necessary for proper operation of the framework. While C APIs would have sufficed in defining proper relationships between components, verifying and enforcing these relationships at runtime would have been a major development undertaking.

An unfortunate shortcoming of Java as a programming language is its poor performance for numerical computing. While there are currently several efforts underway to develop high-performance numerical libraries for Java, it will be some time before competitive solutions exist (possibly never). However, as noted earlier, the metrics traditionally used to evaluate parallel performance are completely not appropriate for an IDCF, so the lower performing numerical libraries are probably sufficient for IDCF applications.

2.3 Grapevine Design

2.3.1 Grapevine Components

There are two primary components in the Grapevine distributed computing framework: the Grapevine Server and the Grapevine Volunteers.

Grapevine Server The Grapevine Server is the liaison between a user's distributed application and the volunteer nodes. The functionality provided by the Grapevine Server is encapsulated in two objects: a JobManager and a VolunteerManager. The JobManager handles the receipt and distribution of jobs and the delivery of results back to the user. The VolunteerManager keeps track of the availability of volunteer nodes. When a volunteer node starts up, it registers itself with the VolunteerManager. From then on, it can be allocated when the JobManager requests a set of nodes for a job.

Grapevine Volunteers The Grapevine Volunteers are a set of computers whose computing resources have been volunteered to help run jobs submitted to the Grapevine framework. On each of these nodes, a TaskManager keeps track of the tasks running on the volunteer. Whenever a new task is sent to the volunteer, the TaskManager creates a TaskWrapper which controls an individual task.

2.3.2 Developing Applications for Grapevine

To create an application for the Grapevine framework, the developer needs to write a collection of Java classes that carry out individual tasks and a launcher program that defines the job and tasks to be run. In addition, she may optionally write programs to monitor the status of the job or collect the results of the computation.

Task Classes Each user task class in an application must extend the abstract Task base class defined in the framework because the Task class provides some basic functionality required by the system. Specifically, it provides task initialization, results reporting, intertask communication functionality. Of course, classes used by user task classes need not be derived from the Task base class.

There are really very few restrictions placed on user tasks. User tasks have access to most computing resources that they would normally have running on a stand-alone machine. For instance, through the methods in the Task base class, the task can gain access to scratch space on the volunteer node's file system and communicate with other user tasks in the job. Of course, these accesses are restricted for security reasons, but we believe that they are sufficient for a large number of applications.

Launcher Program In addition to the user task classes, the user must also write a launcher program that defines the job in sufficient detail that the Grapevine system can create and run it. A job is defined by the following components:

- a data directory,
- a set of DataElements (data files),
- an executable directory,
- a set of ExecElements (java .class files), and
- a set of TaskDefinitions.

Task definitions are similarly defined by specifying

- a set of DataElements (data files),
- a set of ExecElements (java .class files), and
- the name of the user task class.

Once the job and the tasks it contains are fully defined, the launcher program submits the job to the Grapevine system and starts the job. For simple applications, it is convenient to have the launcher program monitor the job, wait for it to complete, collect the results, and perform any clean up necessary operations. However, the system does not require that the same program that launches the job perform these other duties. It is perfectly legal for a separate program to handle each duty. All that is required to access a job is knowledge of its JobID.

2.3.3 Running the Grapevine System

Starting up the Server and Volunteer To make it easier to start up and configure the Grapevine system, we have provided a startup scripts and simple configuration files can be used to configure Grapevine for a particular system. Running the Grapevine Server and Grapevine Volunteer is simply a matter of modifying the server.config or volunteer.config files to reflect the specifics of your system and running the following commands:

1. server.sh server.config
2. volunteer.sh volunteer.config

To start up multiple volunteers, it is only necessary to repeat the second command on all volunteer nodes once the server has been started. At this point, if all went well, the server and volunteers should be all set to run jobs.

A Typical Job Run This section describes a typical run of a user program in the Grapevine system. This sequence corresponds to the order of events in a simple job launcher. For more details, see the User's Manual.

1. The job is defined by specifying all all the required data and executable files and defining the user tasks.
2. The job launcher starts Grapevine by instantiating a Grapevine object.
3. The job definition is submitted to the Grapevine server. If all goes well, the Grapevine server creates a job which awaits further instructions before proceeding.
4. The job is started.
5. The results of the job are requested by the job launcher. This stage need not be a part of the job launcher program.
6. The job is removed from the server.

3 Conclusions

In the process of designing and implementing the Grapevine framework, we found that distributed computing in an internet environment is very different than traditional parallel computing. Many issues arise that are of a less technical nature. As a result, evaluating the performance of IDCFs must expand beyond considerations of speedup and related metrics.

A significant motivation for IDCFs is the high cost of supercomputing, not only in economic but also energy terms. As greater computing power is needed to address the largest problems, the economic costs of building sufficiently large computer centers grows prohibitively large. For example, Lawrence Livermore's ASCI White machine cost the lab $110M, yet some problems still push the limits of its computational power. Furthermore, there is an huge infrastructure required to operate these machines. These costs cannot be ignored. Beyond economic costs, the energy required to run these supercomputers is enormous. In addition to the electricity necessary to run the machines themselves, energy is needed to cool these systems.

From a slightly different perspective, consider the money and energy wasted by idle personal computers. As the number of these machines connected to the internet grows, it seems reasonable that some effort should be made to improve their economic and energy efficiency by harnessing their computing power for useful work.

In considering these "fuzzy" issues, we are not implying that high performance is not important. But traditional techniques for achieving high performance in parallel computing are not all appropriate in a distributed computing environment. It is important to look beyond data parallelism and communication costs. Thought should be given towards other forms of parallelization, such as task parallelization. In a distributed computing environment, many other factors other than communication time affect performance of the system. Some of these are:

- network traffic levels on undedicated networks,
- randomly varying load levels determined by users,
- software stability and portability across different platforms, and
- security.

Some of the factors, such as security, are particularly challenging because they require social and cultural changes.

One of the main difficulties we ran across in implementing the Grapevine system is debugging. The time it takes to do one iteration of the code-build-debug cycle is much longer than for single processor or parallel programming due to the dependencies between different components of the framework. Moreover, as with parallel programming, the it is often not obvious what the source of bugs is. We spent a large amount of time deciphering exception stack traces which were not always helpful in determining the precise location of the bugs.

In summary, this project has been a wonderful opportunity to explore the issues involved in designing and implementing an internet, distributed computing framework. As distributed computing grows more important as a viable computing option, good solutions to these issues will need to be found. While were only able to consider solutions to a small number of issues within the context of the Grapevine system, our explorations forms a good foundation for future work in distributed computing technologies.

References

[1] Ian Foster, Carl Kesselman, Steven Tuecke. “Anatomy of the Grid: Enabling Scalable Virtual Organizations”, Intl J. Supercomputer Applications, 2001.

[2] The Globus Project. http://www.globus.org/

[3] “The Frontier Application Programming Inteface, Version 0.5.2”, Parabon Computation, Inc., 2000.

[4] “Frontier Software Development Kit”, Parabon Computation, Inc., 2001.