Towards Bayanihan: Building an Extensible Framework for Volunteer Computing Using Java

Luis F. G. Sarmenta,[1] Satoshi Hirano,[2] and Stephen A. Ward[1]

[1] MIT Laboratory for Computer Science
Cambridge, MA 02139, USA
lfgs,ward@cag.lcs.mit.edu
[2] Electrotechnical Laboratory
1-1-4 Umezono, Tsukuba, 305, Japan
hirano@etl.go.jp
URL: http://www.cag.lcs.mit.edu/bayanihan

Abstract. Project Bayanihan seeks to develop the idea of *volunteer computing*, which allows people to very easily pool together their computers' processing power and cooperate in solving parallel problems. While the idea of volunteer computing offers many exciting new prospects in global supercomputing and collaboration, its realization involves many challenging research issues that need to be addressed. In this paper, we present a flexible object-oriented framework using Java and HORB that allows programmers to experiment with different approaches to these issues by simply extending and interconnecting basic library components. We also present results from a distributed web-crawling application written using the framework.

1 Introduction

Bayanihan is a Filipino word meaning a community spirit of unity and cooperation that makes seemingly impossible tasks possible through the concerted effort of many people working together on a common goal. Project Bayanihan seeks to bring the *bayanihan* spirit to global computing by developing the idea of *volunteer computing*, which allows "ordinary" people with little technical expertise to easily pool together their computers' processing power and cooperate in solving parallel problems. While the idea of volunteer computing offers many exciting new prospects in global supercomputing and collaboration, its realization involves addressing many challenging issues such as adaptive parallelism, fault-tolerance, computational security, and scalability [1].

To address these issues, we have built a software framework using Java and HORB [2], a distributed object library similar to RMI but not requiring JDK 1.1 (which, unfortunately, is presently not as ubiquitous as JDK 1.0). By allowing programmers to access remote objects without worrying about network communication details, HORB allows us to take full advantage of object-oriented techniques to build an extensible *framework* [3] that lets programmers experiment with different approaches to research issues by simply filling-in appropriate *hot-spots* in the framework.

2 Framework Design

Figure 1 shows a block diagram of a system using the Bayanihan framework.

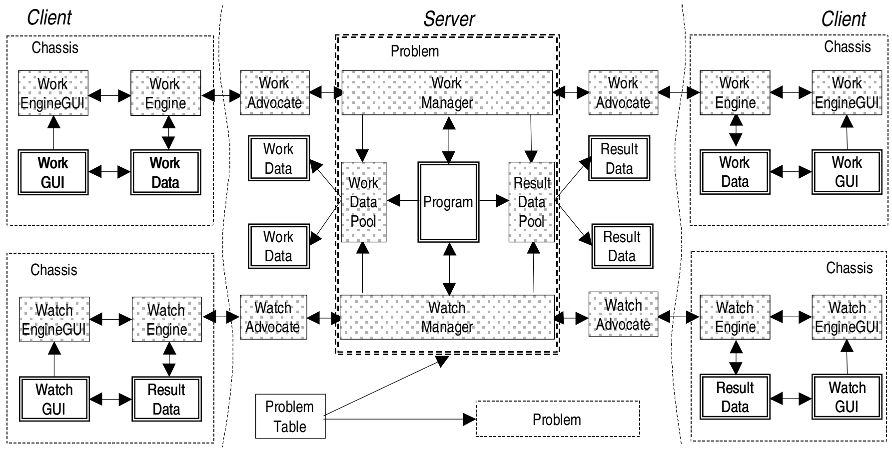

Fig. 1. A Bayanihan system

On the *client* side, an *engine* runs in its own thread, and takes care of receiving and processing *data* objects from its corresponding *manager* on the server. A work engine, for example, would take care of getting work data from the work manager, executing the work, and requesting more work when it is done. Data objects are generally *polymorphic*, and know how to process themselves. Work data objects, for example, implement a `doWork()` method, which the work engine can call. Both the engine and the data have associated *GUI* objects which can be used for user interface. The engine and the GUIs are all contained in a *chassis* object, which can be an applet or application.

On the *server* side, an *advocate* object serves as the engine's representative, forwarding the engine's calls to the manager. The manager has access to several *data pools*, which may be shared by other managers serving other purposes. In Fig. 1, for example, the work and result pools are shared by the work and watch managers, allowing the watch manager to watch the progress of the computation and inform watch engines of such things as new results and statistics. The whole set of associated managers and data pools compose a *problem.* A problem may include a *program* object, which creates and controls managers, and data pools, and fills the latter with data objects. The program may be active (i.e., have its own thread) or it may be passive, only reacting to occasional callbacks from its managers. For example, in our current applications, we implement a simple form of barrier synchronization by having the work manager do a callback when it finishes the current batch of work, and having the program object wait for such a callback before creating the next batch of work. A *problem table* keeps track of independent problems on the same server.

Writing an application using the Bayanihan framework typically involves first selecting and using *generic* components (shown as shaded boxes in Fig. 1) from a library, and then defining *application-specific* components (shown as double-bordered boxes) by extending existing base classes. In this way, high-level application programmers can write a wide variety of applications with the same programming model (e.g., master-worker) by using a common set of pre-defined engine, manager, and data pool objects, and then defining different data, GUI, and program objects according to the application.

The framework also allows programmers and researchers to change generic objects in order to implement new functionality. Researchers, for example, can experiment with performance optimization by writing work manager objects with different scheduling algorithms. Similarly, replication-based fault-tolerance mechanisms can be implemented by extending the manager and data pool objects. Programmers can even implement entirely new parallel programming models by creating new sets of engines, managers, and data pools.

At present, we have implemented a generic set of engines and managers that support master-worker style programming with eager scheduling [4], and have used it for a variety of demo applications including factoring, Mandelbrot set rendering, and distributed web-crawling. We are currently working on extending the basic generic objects to implement performance, reliability, and programmability improvements. In the future, we also plan to implement generic components for supporting Cilk [5] and other adaptively parallel programming models.

3 An Example: Distributed Web Crawling

In [1], we presented the results of a factoring application which demonstrated the potential use of Java-based volunteer computing and the Bayanihan framework for computationally-intensive applications. Here, we present an application which demonstrates another interesting potential use of volunteer computing—multiplying not just computational power, but *communication* power as well.

Consider a web crawler (i.e., an application that recursively follows links in web pages, compiling a list of pages as it goes along). Such an application is extremely communications-bound—i.e., most of its time is spent downloading web pages from the network. Thus, it can take a long time, even on a fast processor, if network bandwidth is limited. This leads to an interesting idea: what if we can use volunteer computing to get *other* machines, with their own (possibly faster) network connections, to do the crawling for us?

To test this idea, we implemented a distributed web-crawling application using the Bayanihan framework, and used it in the setup shown in Fig. 2. We simulated a fast computer with a slow network link by placing the user on a 166 MHz Pentium PC connected to the network through a 28.8Kbps modem. We ran a Bayanihan server on this machine, and a watcher client so the user can see the results. Then, we measured the time it took to find 4000 URLs starting from `http://www.yahoo.com/`, first running the worker applet on the same PC (using `appletviewer` with network restrictions turned off), and then running it on "volunteer" UNIX workstations with slower Java interpreters (i.e., the factoring demo runs about 4 times slower on these machines) but faster network links (i.e., 10Mbps Ethernet).

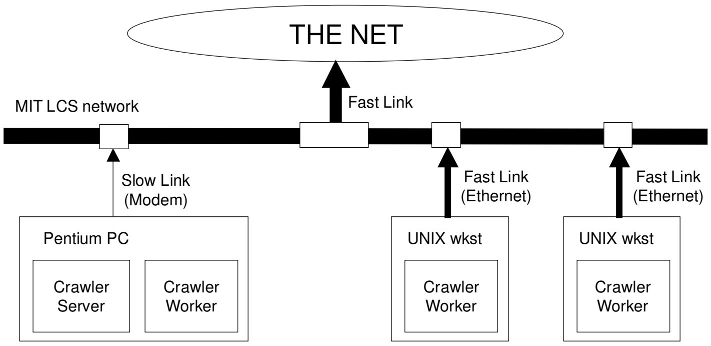

Fig. 2. The distributed web-crawler experiment.

Table 1 shows the results of the experiment using two versions of the application. The first version was based on the same "basic" work engine and work manager objects used in the factoring demo. Although this version produced the desired speedups, we noticed that its full potential was supressed by a communications bottleneck—although the workers were able to download web pages very quickly through their Ethernet links, they were slowed down considerably by the need to report results back to the server (on the other side of the modem link) for *each* work packet before working on the next packet. To remove this bottleneck, we extended the basic work engine and work manager objects to allow the worker to prefetch several packets of work so that it can do new work while simultaneously reporting the results of previously done work. As Table 1 shows, this resulted in up to 80% more speedup over the basic version. Furthermore, the new "multi-work" versions of the work engine and work manager can be used transparently in place of the basic version, allowing us to improve the performance of other applications as well without the need for recoding.

Table 1. Timing Measurements for the distributed web-crawler.

		UNIX-Ethernet					
		Basic			Multi-Work		
measurement	PC-mdm	1	2	3	1	2	3
time for 4000 URLs (s)	817	391	268	213	222	148	166
speed (URL/s)	4.89	10.2	14.9	18.8	18.0	27.0	24.1
speedup (rel. to PC-mdm)	1	2.09	3.05	3.84	3.68	5.52	4.93
speedup (rel. to 1-UNIX)		1	1.46	1.84	1	1.50	1.34
speedup (rel. to Basic)					1.76	1.81	1.28

4 Related, Present, and Future Work

Project Bayanihan joins a growing number of other projects enabling people to use Java for web-based parallel computing. Most early projects such as ATLAS, ParaWeb, JPVM, and many newer projects such as IceT and others presented or referred to in [6] use Java *applications*. These are more powerful and versatile than applets, but require some technical expertise and setup effort from volunteer users. Projects like Bayanihan that support the use of *applets* and Web browsers to allow "ordinary" users to volunteer their computers seem to be fewer, but have also been growing in number. These include simple systems such as DAMPP [7], and more complex and general-purpose *frameworks* such as Charlotte [4], and Javelin [8]. Many of these projects provide the programmer with a way to easily generate different applications, but are usually restricted to following a particular programming model.

Project Bayanihan takes advantage of the flexibility and ease-of-use provided by distributed object technology to build a general-purpose framework for implementing and experimenting not only with different applications, but with different programming models and volunteer computing mechanisms as well. Work remains to be done on implementing more complex programming models, but current results with master-worker style applications demonstrate the framework's future potentials.

References

1. L. F. G. Sarmenta. "Bayanihan: Web-Based Volunteer Computing Using Java," *Worldwide Computing and Its Applications - WWCA'98*, Springer Lecture Notes in Computer Science 1368, 1998. `http://www.cag.lcs.mit.edu/bayanihan`
2. S. Hirano. "Distributed Execution of Java Programs," *Worldwide Computing and Its Applications*, Springer Lecture Notes in Computer Science 1274, 1997.
3. D. Roberts and R. Johnson. "Evolving Frameworks: A Pattern Language for Developing Object-Oriented Frameworks," University of Illinois, 1997. `http://st-www.cs.uiuc.edu/users/droberts/evolve.html`
4. A. Baratloo, M. Karaul, Z. Kedem, and P. Wyckoff. "Charlotte: Metacomputing on the Web," *Proc. of the 9th International Conference on Parallel and Distributed Computing Systems*, Sept. 1996. `http://cs.nyu.edu/milan/charlotte/`
5. R. D. Blumofe, C. F. Joerg, B. C. Kuszmaul, C. E. Leiserson, K. H. Randall, and Y. Zhou. "Cilk: An Efficient Multithreaded Runtime System," in *Proceedings of the 5th ACM SIGPLAN Symposium on Principles of Parallel Programming*, July, 1995, Santa Barbara, California. `http://theory.lcs.mit.edu/~cilk/`
6. *ACM 1997 Workshop on Java for Science and Engineering Computation*. Las Vegas, June 1997. `http://www.npac.syr.edu/users/gcf/03/javaforcse/acmspecissue/latestpapers.html`
7. L. Vanhelsuwe. "Create your own supercomputer with Java," *JavaWorld*, Jan. 1997. `http://www.javaworld.com/jw-01-1997/jw-01-dampp.ibd.html`
8. P. Cappello, B. O. Christiansen, M. F. Ionescu, M. O. Neary, K. E. Schauser, and D. Wu. "Javelin: Internet-Based Parallel Computing Using Java," *ACM Workshop on Java for Science and Engineering Computation*. Las Vegas, June 1997.