Bayanihan: Building and Studying Web-Based Volunteer Computing Systems Using Java

Luis F. G. Sarmenta

MIT Laboratory for Computer Science, Cambridge, MA 02139, USA
lfgs@cag.lcs.mit.edu, http://www.cag.lcs.mit.edu/bayanihan/

Satoshi Hirano

Electrotechnical Laboratory, 1-1-4 Umezono, Tsukuba, Ibaraki 305, Japan
hirano@etl.go.jp

Abstract

Project Bayanihan is developing the idea of *volunteer computing*, which seeks to enable people to form very large parallel computing networks very quickly by using ubiquitous and easy-to-use technologies such as web browsers and Java. By utilizing Java's object-oriented features, we have built a flexible software framework that makes it easy for programmers to write different volunteer computing applications, while allowing researchers to study and develop the underlying mechanisms behind them. In this paper, we show how we have used this framework to write master-worker style applications, and to develop approaches to the problems of programming interface, adaptive parallelism, fault-tolerance, computational security, scalability, and user interface design.

Key words: metacomputing, parallel and distributed computing, network of workstations, heterogeneous computing, Java

1 Introduction

Bayanihan (pronounced "buy-uh-*nee*-hun") is the name of an old Filipino countryside tradition wherein neighbors would help a relocating family by gathering under the family's house and carrying it to its new location. As such, the word *bayanihan* has come to mean a spirit of communal unity and cooperation which makes seemingly impossible tasks possible through the concerted effort of many people with a common goal and a sense of unity. Project Bayanihan seeks to bring the *bayanihan* spirit to the realm of global computing by developing the idea of *volunteer computing*, a form of metacomputing that seeks to make it easy for even "ordinary people" with little technical knowledge to cooperate in solving parallel problems by *volunteering* their computers' processing power. By minimizing the effort and expertise required to add worker nodes, volunteer computing maximizes the potential worker pool size and minimizes setup time, making it possible to build world-wide computing networks much larger and much more quickly than possible with other forms of metacomputing.

The potentials of volunteer computing have been demonstrated by the success of projects like `distributed.net`, which solved the RSA RC5-56 challenge by employing over 4,000 volunteer teams world-wide, with a combined power equivalent to that of about 26,000 high-end PCs [5]. Project Bayanihan aims to take volunteer computing even further by developing *web-based* volunteer computing systems where programmers can write platform-independent parallel applications in Java and post them on the Web as applets so that volunteers need only a web browser and a few mouse clicks to join a computation.

The possible benefits of such web-based volunteer computing systems are many, ranging from the local to the global [13]. Being much easier and faster to install than existing metacomputing systems, these systems can allow more organizations – including companies and universities that have so far lacked the necessary expertise and time – to pool their existing workstations to provide inexpensive supercomputing facilities for research and teaching. By volunteering their resources to each other, organizations can share and barter trade processing power, creating new possibilities in global collaboration. With the appropriate economic models and mechanisms, barter trading systems can eventually turn into commercial systems where computing power becomes a commodity that people can buy, sell, or trade. When Internet set-top boxes and other *information appliances* become widely available, volunteer computing principles can also be used to build NOIAs – networks of information appliances – that take advantage of the under-utilized processing power of millions of CPUs sitting idle in users' homes. Ultimately, web-based volunteer computing has the potential to harness the computing power of the millions of computers on the Internet, and use them towards solving hard computational problems for worthy causes that serve the common good of local communities and the world.

In order to make these potentials a reality, however, many challenging issues need to be addressed. In this paper, we identify some of these issues, and present a flexible software framework that fully exploits Java's object-oriented features to make it easy not only for programmers to build applications, but also for researchers to study various issues and develop approaches to them. We then show how we have used this framework in these two ways, and discuss our results.

2 Research Issues

Implementing and deploying real volunteer computing systems involves many interesting and challenging technical questions and problems. These include:

- **Accessibility.** In order to maximize the potential work pool and minimize setup time, a volunteer computing system must be accessible to as many people as possible. It must be platform-independent, and must require as little technical knowledge from volunteers as possible.
- **Programmability.** A volunteer computing system should provide a flexible and easy-to-use programming interface that allows programmers to implement a wide variety of parallel applications easily and quickly.
- **Adaptive Parallelism.** Since volunteer nodes can have different kinds of CPUs, and can join and leave a computation at any time, parallel programming models for volunteer computing systems must be *adaptively parallel*. Unlike traditional models, they cannot assume the existence of a fixed number of nodes, or depend on static timing information about the system.
- **Fault-Tolerance and Computational Security.** Like all large-scale metacomputing systems, volunteer computing systems must be able to tolerate faults such as data loss or corruption from random hardware, software, and communication failures. Unlike other metacomputing systems, however, volunteer computing systems must also be resilient against *intentional attacks* from malicious nodes. These include *sabotage* by nodes submitting erroneous results, and *spying* on confidential data in commercial volunteer systems.
- **Performance and Scalability.** To be useful, a Java-based volunteer computing system must ultimately provide its users with speedups better than, or at least comparable to, other available metacomputing technologies. Potential obstacles to this goal include Java's traditionally slow execution speed, communication overhead, and the lack of server scalability.
- **User Interface Design.** Unlike most conventional parallel systems, volunteer computing systems need good user interfaces to attract volunteers to participate and encourage them to stay. In commercial systems, users would also need good interfaces for submitting jobs and receiving results.

3 The Bayanihan Framework

To address these issues, we have built a software framework using Java and HORB [12], a distributed object package similar to Sun's RMI, but more widely compatible. By using HORB to access remote objects transparently without worrying about communication details, we are able to utilize object-oriented techniques to enable programmers to experiment with different approaches to research issues by "mixing-and-matching" objects in various ways.

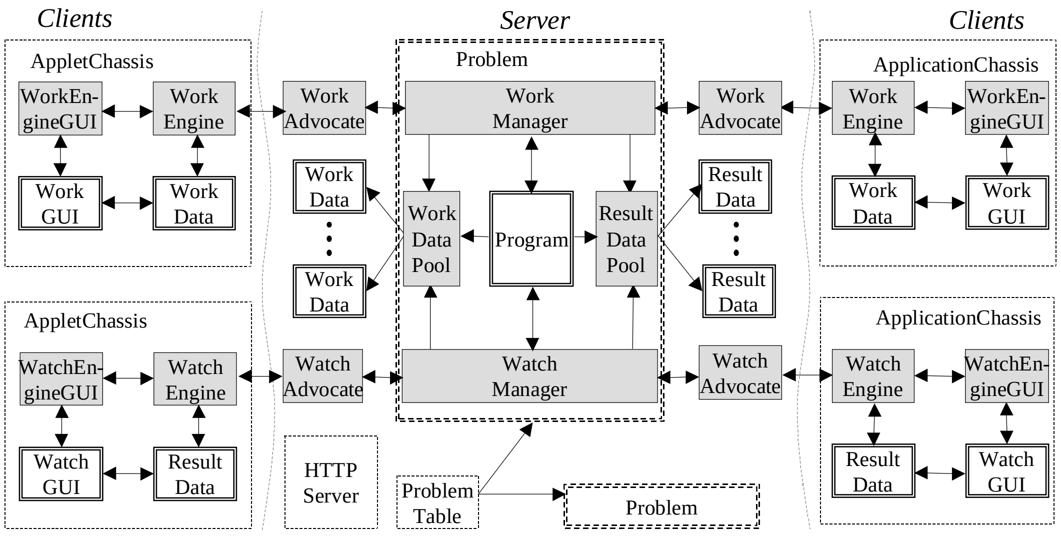


Fig. 1. A Bayanihan system with worker and watcher clients.

3.1 System Design

The Bayanihan framework defines a set of interacting components that can be extended and composed to build Bayanihan systems like that shown in Fig. 1. A Bayanihan system consists of many *clients* connected to one or more *servers*. A *client* can either be a Java *applet* started from a web browser, or a Java *application* started from the command line. There can be different kinds of clients, such as worker clients for performing computation, and watcher clients for viewing results and statistics. Each client has a *chassis*, which contains an active *engine* object that communicates with a *manager* on the server by exchanging *data* objects. A work engine, for example, may get work data objects from the work manager, execute them, and return result data objects when they are done. Data objects are generally *polymorphic*, and know how to process themselves. Work data objects, for example, may implement a `doWork()` method, which the work engine can call. Both the engine and the data objects have associated *GUI* objects that provide a user interface.

A *server* typically contains a commodity HTTP server for serving out Java class files, and a command-line Java application that creates one or more *problem* objects representing different ongoing computations. Each problem has a *program* object which creates and controls *manager* and *data pool* objects of different kinds. Figure 1, for example, shows a program object controlling a work manager which takes care of distributing work and collecting results from worker clients, and a watch manager which distributes results to watcher clients. Each client connected to the server is represented by an *advocate* object, which forwards the client's remote calls to the appropriate manager.

3.2 Using the Framework

Writing an application using the Bayanihan framework typically involves first selecting and using existing *generic* library components (shown as shaded boxes in Fig. 1), and then defining new *application-specific* components (shown as double-bordered boxes) by extending existing base classes. In this way, application programmers can write a wide variety of applications that share a common 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 Bayanihan framework, however, also allows programmers and researchers to change the generic objects themselves, making it easy for them to implement and experiment with new *generic* functionality and mechanisms. For example, researchers 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 (i.e., subclassing) 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.

By providing extensibility on these two levels – i.e., the applications and generic mechanisms levels – the Bayanihan framework makes it easy for programmers and researchers not only to build a variety of applications, but also to study the different technical issues in volunteer computing and develop possible approaches to them. In the remainder of this paper, we show how we are using this two-level flexibility, and discuss our current results.

4 Building Applications: Programming and User Interface

4.1 Programming Interface

Although the Bayanihan framework can support other programming models, all our applications currently use the master-worker model shown in Fig. 2. In this model, the server's program object creates a set of managers and data pools, and fills the work pool with work data objects in its `createWork()` method. Volunteer *worker clients* connected to the server run in a loop, repeatedly making remote calls to the `getWork()` method of their respective work advocates. Each advocate passes these calls to the work manager, adding its process ID (pid) for identification, if desired. In response to a call to `getWork()`, the work manager returns the next available uncompleted work data object in

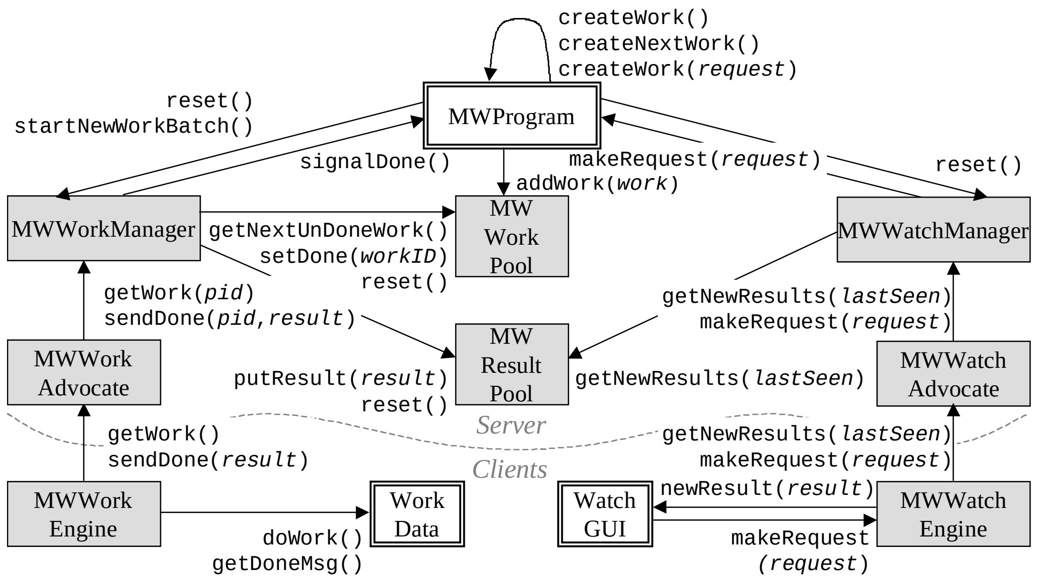
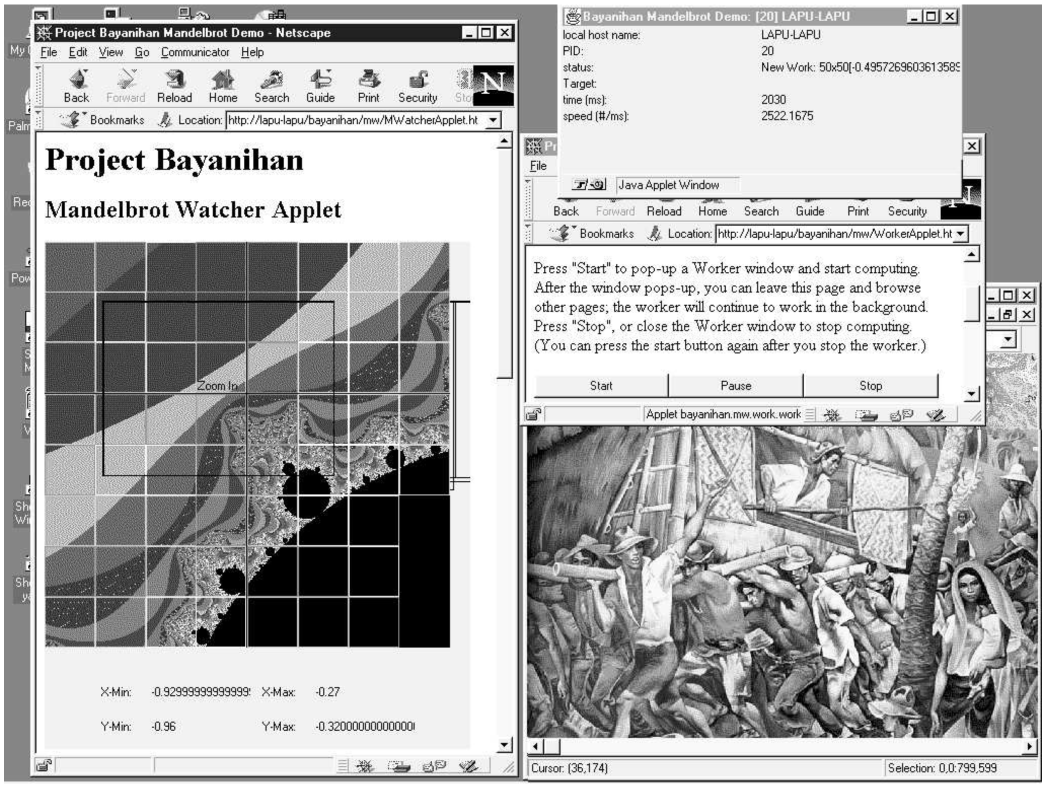


Fig. 2. Master-worker model components and methods.

the work pool. The work engine runs this work by calling its `doWork()` method, and gets the result by calling `getDoneMsg()`. Finally, it sends the result back to the work manager by calling the advocate's `sendDone()` method.

As the work manager receives results from workers, it places the results in the result pool and marks the corresponding work in the work pool done. When all the work in the pool have been done, the work manager calls its parent program object's `signalDone()` method. By default, this method calls `createNextWork()`, which sets up the next stage of the computation by refilling the work pool and calling `startNewWorkBatch()`. In applications, programmers can override `createNextWork()` or `signalDone()` to separate blocks of parallel computation such that one block is guaranteed to be computed completely before the other is started. This provides a simple form of *barrier synchronization*. In our factoring application, for example, the program object has a list of target numbers to be factored, and its `createNextWork()` method is used to move to the next target while making sure that all the results from the previous one have been received.

Users can view results and statistics and control the computation through *watcher clients* that communicate with the watch manager on the server. A watcher client's engine runs in a loop, periodically requesting the watch manager for a list of new results, and passing these results to the watch GUI, which displays them accordingly. The watcher client also allows the user to send *request* objects to the watch manager via the `makeRequest()` method. The watch manager forwards such requests to the program object, which then reacts in an application-specific way. In our Mandelbrot rendering demo, for example, the watch GUI (shown in Fig. 3) allows users to select a portion of the screen to zoom in or out to, and sends corresponding request objects to the


Project Bayanihan Mandelbrot Demo - Netscape
File Edit View Go Communicator Help
Back Forward Reload Home Search Guide Print Security
Bookmarks Location: http://lapu-lapu/bayanihan/mw/MWatcherApplet.ht
Zoom In
Bayanihan Mandelbrot Demo: [20] LAPU-LAPU
Java Applet Window
Location: http://lapu-lapu/bayanihan/mw/WorkerApplet.ht
Start
Pause
Stop
Applet bayanihan.mw.work.work
Cursor: (36,174)
Selection: 0,0:799,599

Project Bayanihan

Mandelbrot Watcher Applet

X-Min: -0.929999999999999 X-Max: -0.27

Y-Min: -0.96 Y-Max: -0.320000000000000

local host name: LAPU-LAPU
PID: 20
status: New Work: 50x50[-0.4957269603613589
Target:
time (ms): 2030
speed (#/ms): 2522.1675

Press "Start" to pop-up a Worker window and start computing.
After the window pops-up, you can leave this page and browse
other pages; the worker will continue to work in the background.
Press "Stop", or close the Worker window to stop computing.
(You can press the start button again after you stop the worker.)

Fig. 3. A screen shot of the Mandelbrot worker and watcher applets.

server. When the program object receives the request, it responds by calling `reset()` on the work and watch managers (causing the work and result pools to be cleared), and calling its own `createWork(`*request*`)` method.

To write an application for this model, programmers need only to override appropriate methods in the *application-specific* classes `MWProgram`, `WorkData`, `WorkGUI`, `ResultData`, and `WatchGUI`. In this manner, we have written a variety of applications, including factoring [13], distributed web-crawling [14], RC5-64 decryption, and Mandelbrot rendering. Although seemingly simple and limited to "embarrassingly parallel" applications, the master-worker model is actually quite versatile and practical. Our Mandelbrot demo, for example, represents *parallel rendering* applications used not only in the scientific community but in the media industry as well. Other potential applications include some forms of *data mining*, Monte Carlo simulations, and any computations in general where one wants to run the same *sequential* computation with a large number of varying input combinations. Interestingly, because most programmers today are still accustomed to sequential programming, a lot of applications in the real world may actually fall into this category.

By extending the appropriate classes, we can also use the master-worker model to support more complex programming models. For example, we have built a sub-framework for parallel *genetic algorithms* that defines a generic work data class, `GAWork`, whose `doWork()` method calls abstract evaluation, selection, reproduction, and mutation methods on a set of genes, and a generic program class, `GAProgram`, whose `createNextWork()` method redistributes the resulting genes into a new generation of work objects. By writing application-specific subclasses of `GAWork` and `GAProgram`, we have successfully applied the sub-framework to problems such as multivariable function optimization. Currently, we are looking into implementing BSP [15], a growingly popular programming model which makes programming more natural by providing remote memory access and message-passing functions, but at the same time makes implementation easier by specifying that these communication operations only take effect at the next global barrier synchronization. It should be possible to implement BSP on top of the master-work model by defining a work engine that allows work objects to make communication requests as they run, and a work manager that collects these requests and performs them during in `signalDone()` or `createNextWork()`. In the future, we may also implement a coarse-grain dataflow programming model by extending the master-worker components to support dependencies between work objects.

4.2 User Interface Design

The modular design of the Bayanihan framework allows application writers to create and use different GUIs as desired, as long as these conform to the interfaces required by the objects that will use them. Programmers can create *application-specific GUIs* to view the same object in different ways, as well as *generic GUIs* to view different objects in the same way. Figure 3 shows some examples from our Mandelbrot application. On the left is a watcher applet with an application-specific `MandelWatchGUI` object that displays `MandelResult` blocks. Colored borders corresponding to different workers give users a sense for the parallelism in the computation. The GUI also allows users to zoom in and out of selected areas, as described in Sect. 4.1. On the right is a worker applet with a generic work engine GUI that provides simple controls for starting, pausing, and stopping the work engine. The window above it is a generic work GUI that displays status and timing information.

5 Developing Generic Mechanisms

In addition to making it easy to write applications, the Bayanihan framework also makes it easy to develop generic mechanisms. In this section, we demonstrate this flexibility by showing how we have used our framework to initiate explorations into the issues of adaptive parallelism, fault-tolerance, computational security, performance, and scalability.

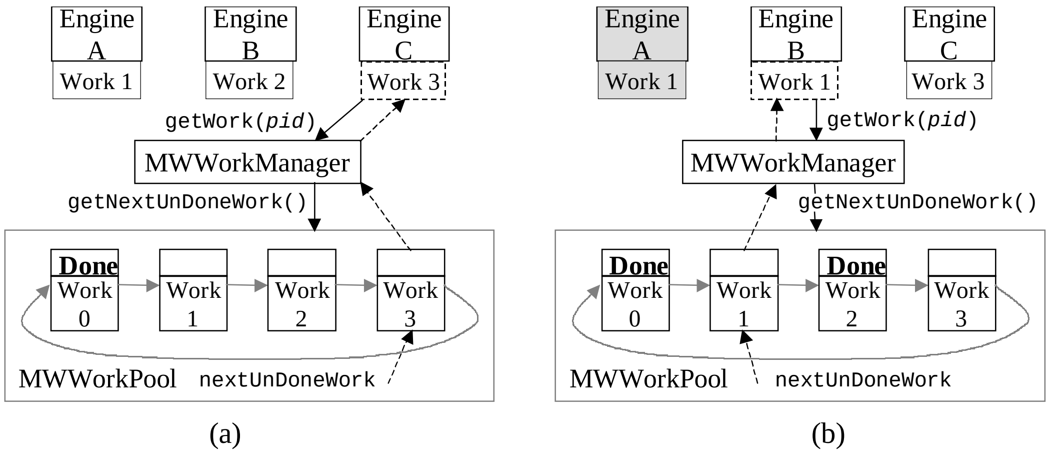


Fig. 4. Simple eager scheduling. (a) C calls `getWork()` while A and B are doing works 1 and 2. (b) B finishes work 2 before A finishes work 1; B gets work 1 too.

5.1 Adaptive Parallelism

The current master-worker programming model implementation employs a simple form of adaptive parallelism sometimes called *eager scheduling* [4]. As shown in Fig. 4, each work object has a *done* flag which is set when a worker returns the result for that object. The work objects are stored in a circular list, with a pointer keeping track of the next available uncompleted work. In Figure 4(a), for example, the `nextUnDoneWork` pointer is pointing to work 3 after works 1 and 2 have been assigned to engines A and B respectively. Thus, when engine C calls `getWork()`, it receives work 3. Since workers call `getWork()` as soon as they finish their current work, faster workers will tend to call `getWork()` more often, and will thus get a bigger share of the total work. In this way, we get a simple form of dynamic load balancing.

Moreover, since the list is circular, `nextUnDoneWork` can eventually wrap around and point to previously assigned but uncompleted work, allowing a piece of work to be reassigned to other workers. This "eager" behavior guarantees that slow workers do not cause bottlenecks – fast workers with nothing left to do will simply bypass slow ones, redoing work themselves if necessary. It also provides a basic form of crash-tolerance. In Fig. 4(b), for example, we see that when B finishes work 2 and calls `getWork()`, it receives work 1, which has not been marked done because A has crashed (or is simply slow). In this way, computation can go on as long as at least one processor is still alive. In fact, even if all the processors crash, the computation can continue as soon as a new processor becomes available.

Currently, we are also examining other forms of adaptive parallelism that can be used with the master-worker programming model. For example, we have written `MultiWorkEngine` and `MultiWorkManager` subclasses that implement

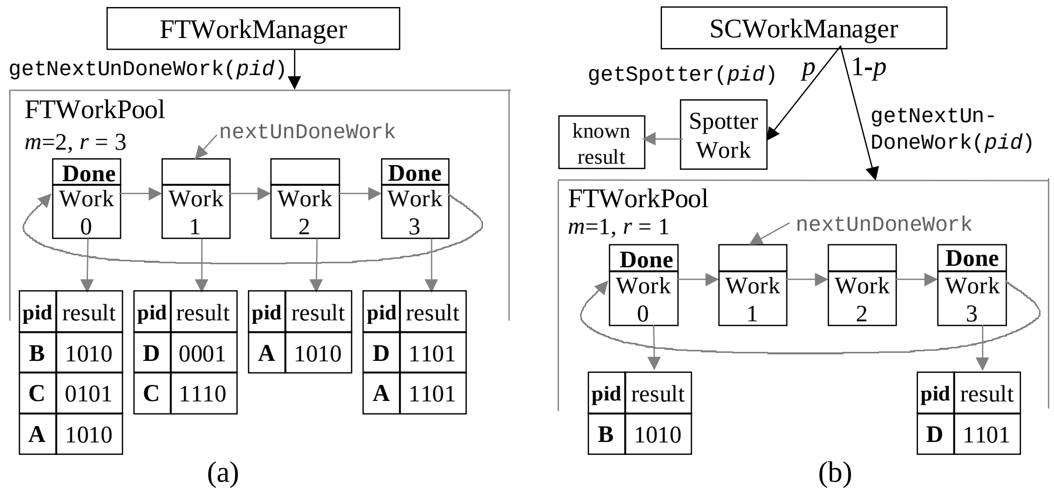


Fig. 5. Two approaches to fault-tolerance. (a) Majority Voting. (b) Spot-checking

a `getMultiWork()` function for prefetching multiple packets of work, and have used them in our distributed web-crawler application to improve performance by hiding communication latency [14]. We are also using them to study the effects of changing work sizes depending on worker speeds.

5.2 Fault-Tolerance and Computational Security

By extending the eager scheduling work manager and work pool objects as shown in Fig. 5, we have implemented two approaches to protecting against faults and sabotage: *majority voting* and *spot-checking.*

Majority voting works by requiring that at least a majority m of up to r results from different workers for the same work object have the same value. We implement it by using a new subclass of `MWWorkPool`, `FTWorkPool`, where the done flag of a work object remains unset as long as a majority agreement has not been reached and less than r results from different PIDs have been received. If r results are received without reaching a majority agreement, all the r results are invalidated, and the work object is redone. In Fig. 5(a), for example, works 0 and 3 have reached a majority agreement and are marked done, while works 1 and 2 are still considered undone.

Spot-checking, shown in Fig. 5(b), works as follows: at the beginning of each new batch of work, the `SCWorkManager` (a subclass of `FTWorkManager`) randomly selects a work object from the work pool and precomputes its result. Then, whenever a work engine calls `getWork()`, the work manager returns this *spotter work* with probability p. In this way, the work manager can check the trustworthiness of a worker by comparing the result it returns with the known result. If the results do not match, the offending worker is *blacklisted* as untrustable, and the work manager *backtracks* through all the current results,

Table 1
Preliminary results from fault-tolerance experiments.

ideal time 20.8 s		Voting (m=2, r=3)			Spot-checking & backtracking, no blacklisting							
					p=10%				p=20%			
# bad (plus 5 good)	f (% bad)	ave time (s)	eff (%)	ave err (%)	ave time (s)	eff (%)	ave err (%)	ave # caught	ave time (s)	eff (%)	ave err (%)	ave # caught
0	0	40.4	52	0	23.1	90	0	0	26.2	79	0	0
1	16.7	40.6	51	0	22.7	89	3.0	4.8	26.1	79	1.4	8.4
2	28.6	36.1	48	17	22.8	86	6.0	8.8	27.2	75	2.2	18
3	37.5	32.2	45	30	21.2	86	12	11	28.5	71	3.3	28
4	44.4	28.6	43	41	22.4	84	10	15	29.4	68	3.8	38
5	50.0	26.3	40	50	21.5	80	17	19	30.1	66	4.0	48
7	58.3	21.9	36	62	21.5	77	20	28	31.0	63	5.9	64

invalidating any results dependent on results from the offending worker.

Table 1 shows results from using these two fault-tolerance mechanisms. In this experiment, we created a saboteur work engine that always corrupts results in a fixed way. We then ran the Mandelbrot application on the spiral range (see Sect. 5.3.1), with 5 good workers and a varying number of saboteurs. To keep the pool of saboteurs from being completely eliminated in spot-checking, we disabled blacklisting, and allowed caught nodes to reconnect after a 1 second delay. For each configuration, we ran 10 rounds, measuring the average running time and the average error rate (err). From these, we computed the *efficiency* of each configuration by first multiplying the ideal time (measured with 5 good workers, no saboteurs and no fault-tolerance) by the fraction of correct answers ($1 - \text{err}$), and then dividing the result by the actual running time. This estimates how efficiently the *good* workers are being utilized towards producing *correct* final answers.

Majority voting performed as expected, having an error rate close to the theoretical expected value of $f^2(3 - 2f)$ for 2-out-of-3 voting (where f is the fraction of workers who are saboteurs, and where we assume that saboteurs agree on their answers). Although this error rate is large for the values of f shown, it should improve significantly in more realistic situations where f is small, since it is proportional to f^2. The bigger problem with voting is its efficiency, which, as shown, is at best only about 50% since all the work has to be done at least twice even when there are no saboteurs. In this respect, spot-checking performed more promisingly. As shown, its efficiency loss was only about p when the number of saboteurs was small, and, thanks to backtracking, its error rates remained relatively low – even when there were *more* saboteurs than good workers. In a real system where blacklisting is enabled, we can expect even lower error rates. As shown, saboteurs were being caught at a high rate – as many as 64 saboteurs every 31 s in one case. This means that unless saboteurs can somehow assume a very large number of false identities (e.g., by faking IP addresses or digital certificates) and switch between them dynamically and quickly – a highly unlikely scenario – they would quickly get eliminated, and the error rate would decrease rapidly in time.

Although very encouraging, these results are clearly just the beginning of much theoretical, experimental, and developmental research that can be done in this area. Some research questions include how well spot-checking would work in cases where saboteurs do not always give bad answers, and how we can avoid unnecessarily blacklisting "innocent" nodes that just suffer temporary glitches. The most important challenge, however, is that of applying these mechanisms in real applications. Spot-checking may be sufficient for some classes of applications where a small percentage of bad answers may be acceptable, or can be screened out. These include *image rendering*, where a few scattered bad pixels would be acceptable, some *statistical computations*, where outliers can be detected and either ignored or double-checked, and *genetic algorithms*, where bad results are naturally screened out by the system. Most scientific applications, however, assume 100% reliability. For these, it maybe useful to combine voting, spot-checking, backtracking, and blacklisting to shrink the error rate as much as possible. In the future, we also plan to explore the use of traditional and novel security mechanisms such as *checksums*, *digital signatures*, *encrypted computation*, and *dynamic obfuscation* [13] to reduce the error rate further by making it difficult for saboteurs to falsify results in the first place.

5.3 Performance and Scalability

5.3.1 Relative Speedup and Absolute Performance

Figure 6 shows results from running the Mandelbrot application on 17 200MHz Pentium Pro machines (1 server and 16 workers) connected by 10Mbit Ethernet, running Windows NT 4.0, Netscape 4.03 on the clients, and Sun's JDK 1.1.6 JIT compiler on the server. In this experiment, the target work was an 800x800 pixel array, divided into 256 square chunks. To represent different computation granularities, we tried four different target ranges with different average depths (iterations per pixel). For comparison, ideal speedup was computed using the sequential computation speed on a single unpartitioned 800x800 array with maximum depth. As shown, we get good speedup for large granularities – achieving 91% and 85% efficiency with 16 workers at depths 2048 and 1037, respectively. As the granularity decreases, however, communication and other overheads begin to dominate, limiting efficiency to only 42% and 1% with 16 workers at depths 198 and 3, respectively. Possible approaches to this problem (which is not unique to Java-based volunteer computing) include reducing overhead, and adaptively changing the problem granularity.

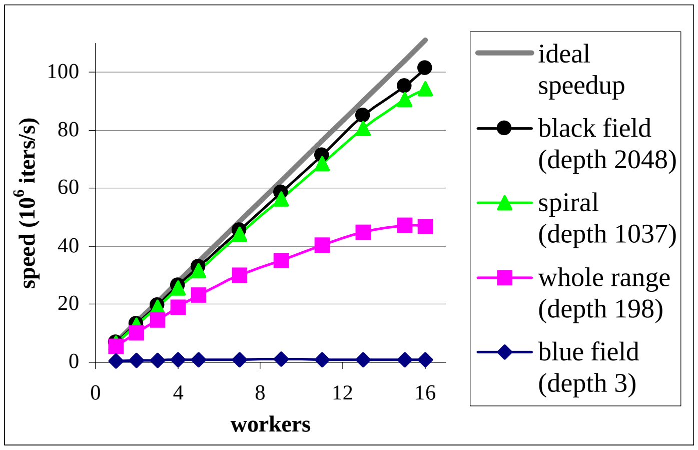


Fig. 6. Speedup measurements for the Mandelbrot application.

Table 2
Absolute Java and C speeds for the Mandelbrot demo on a 200MHz Pentium Pro.

	speed (10^6 iters/s)	relative speed
Netscape 4 JIT on NT4.0	6.94	1.00
gcc	1.40	0.20
gcc -O*best*	8.86	1.28

Table 2 shows a comparison of *absolute* speeds from sequential Java and native C executions of the same Mandelbrot code. Here, we see that with *just-in-time* (JIT) compilation, Java was actually faster than unoptimized C code, while only slightly slower than optimized C code (compiled with `djgpp`'s `gcc` [8] using the -O option that produced the best result). In another test, our RC5 code was about 8 times slower than `distributed.net`'s version. While not as impressive, this is still notable, considering that `distributed.net`'s code was hand-optimized using processor-specific assembly code, while Java does not even directly support some necessary operations such as bitwise rotates.

5.3.2 Server Scalability

Due to security restrictions, Java applets can only communicate with the web server from which they were downloaded. This forces browser-based volunteer computing networks into star topologies which have high congestion and limited scalability. To address this problem, we have developed a simple *volunteer server* system that volunteers with HTTP servers on their own machines can download and run as a Java application. This application creates a generic `VSProblem` object which, like other problem objects, contains data pools and managers for serving worker and watcher clients. Unlike other problem objects, however, `VSProblem`'s `createWork()` method does not create new work on its own, but instead gets groups of work data from the main server. Correspondingly, its `signalDone()` method sends the results back to the main server, and its `createNextWork()` method requests more work.

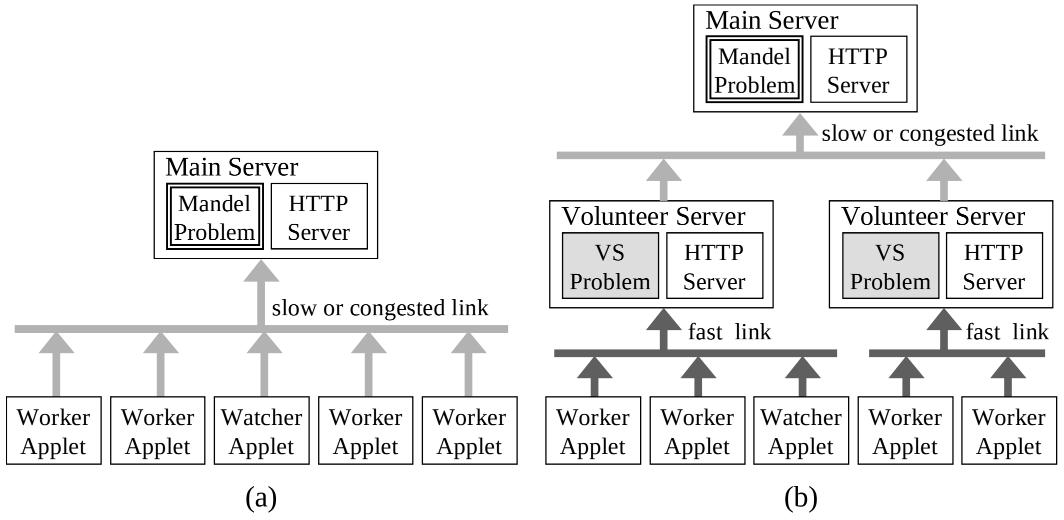


Fig. 7. Using volunteer servers. (a) Slow links result in unnecessary delays and idling. (b) Volunteer servers help by exploiting communication parallelism and locality.

Table 3
Running the Mandelbrot application on a volunteer server.

time (s) with 5 workers	main server on fast link	main server on slow link	volunteer server on fast link, main server on slow link
without watcher	5	198	125
with watcher	5	233	124

Figure 7 shows how volunteer servers can help improve a system's performance and scalability. Consider the scenario shown in Fig. 7(a), where some volunteers are slowed down by delays due to congestion or some other constraint of the server link (e.g., the server may be in a different country). In such a situation, we can improve overall running time by having the workers connect indirectly through volunteer servers with faster links (e.g., uncongested servers, or servers in their own countries), as shown in Fig. 7(b). Table 3 shows results from an experiment simulating these scenarios using a 28.8Kbps modem link for the slow link, and 10Mbit Ethernet for the fast link. Note that although the computation took only 5 s with the main server on a fast link, it took almost 200 s when the main server was placed on a slow link, as workers were forced to wait idly while the server received their results and sent them new work through the slow link. To address this problem, we used one volunteer server to act as a "cache" between the main server and the workers, using the fast link as shown in Fig. 7(b) to allow them to work at full speed without idling. As shown, this reduced the total running time to 125 s, of which the first 5 s were spent by the workers to do the computation, and the remaining 120 s by the volunteer server to send all the results back to the server through the slow link. The volunteer server also allowed watchers to be added without further congesting the slow link and slowing down the computation.

In general, volunteer servers enable us to overcome congestion and long latencies by exploiting locality and parallelism in communications. Other potential applications of volunteer servers include forming *server pools* for handling large numbers of clients, and building networks with non-star topologies.

6 Conclusion: Related, Present, and Future Work

Project Bayanihan joins a growing number of projects enabling people to use Java for web-based parallel computing. Most early projects such as ATLAS [3], and JPVM [9], and many newer projects such as IceT [11], Java// [7], Ninflet [16], and others [10,1,2], use Java *applications*. These are less restricted 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 maximize accessibility seem to be fewer, but have also been growing in number. Early systems include simple ones such as DAMPP [17], and more complex general-purpose ones such as Charlotte [4], and Javelin [6].

Project Bayanihan's approach is to maximize flexibility by taking full advantage of object-oriented techniques. To this end, we have developed a general-purpose framework that allows programmers and researchers to build systems by simply "mixing-and-matching" interacting objects. We have demonstrated the effectivity of this framework by successfully using it to build a variety of master-worker styles applications, and to initiate explorations into several interesting research areas. The results from our explorations give a very positive outlook on the feasibility and practicality of volunteer computing systems, and encourage us to do further research in the field by writing more applications, implementing more programming models, and developing more generic mechanisms for supporting volunteer computing.

Acknowledgement

Thanks to Eamon Walsh, who wrote the genetic algorithm sub-framework; Lydia Sandon, who wrote the RC5 application; Alex Yip, who helped write the Mandelbrot GUI; and the Charlotte group, who let us use some math and graphics code from their own Mandelbrot demo. Thanks to Profs. Steve Ward and Charles Leiserson of MIT, and all the reviewers of this paper, for their helpful comments and suggestions. The work presented here was performed at MIT and ETL, and supported in part by MIT, ETL, and DARPA. In addition, Luis Sarmenta was supported in part by the Philippine Department of Science and Technology, the MIT Japan Program, and Ateneo de Manila University. The image in Fig. 3 is from *Bayanihan*, by Carlos "Botong" Francisco, used with permission from its owner, Unilab Philippines, Inc.

References

[1] *Proc. ACM 1997 Workshop on Java for Science and Engineering Computation* (Las Vegas, 1997). http://www.npac.syr.edu/users/gcf/03/javaforcse/acmspecissue/latestpapers.html

[2] *Proc. ACM 1998 Workshop on Java for High-Performance Network Computing* (Palo Alto, 1998). http://www.cs.ucsb.edu/conferences/java98/

[3] J.E. Baldeschwieler, R.D. Blumofe, and E.A. Brewer, ATLAS: An Infrastructure for Global Computing, in: *Proc. 7th ACM SIGOPS European Workshop: Systems Support for Worldwide Applications* (1996).

[4] A. Baratloo, M. Karaul, Z. Kedem, and P. Wyckoff, Charlotte: Metacomputing on the Web, in: Proc. 9th Intl. Conf. on Parallel and Distributed Computing Systems (1996). http://cs.nyu.edu/milan/charlotte/

[5] A.L. Beberg, J. Lawson, and D. McNett, http://www.distributed.net/

[6] P. Cappello, B.O. Christiansen, M.F. Ionescu, M.O. Neary, K.E. Schauser, and D. Wu, Javelin: Internet-Based Parallel Computing Using Java, in: *Proc. ACM Workshop on Java for Science and Engineering Computation* (Las Vegas, 1997).

[7] D. Caromel, J. Vayssière, A Java Framework for Seamless Sequential, Multi-threaded, and Distributed Programming, in: *Proc. ACM 1998 Workshop on Java for High-Performance Network Computing* (Palo Alto, 1998).

[8] D. Delorie. djgpp Home Page. http://www.delorie.com/djgpp

[9] A. Ferrari. JPVM: Network Parallel Computing in Java, in: *Proc. ACM 1998 Workshop on Java for High-Performance Network Computing* (Palo Alto, 1998).

[10] G.C. Fox, ed., Special Issue on Java for Computational Science and Engineering – Simulation and Modeling, *Concurrency: Practice and Experience.* **9**(6) (1997).

[11] P.A. Gray, V.S. Sunderam, IceT: Distributed Computing and Java, in: *Proc. ACM Workshop on Java for Science and Engineering Computation* (1997).

[12] S. Hirano, HORB: Distributed Execution of Java Programs, in: *Proc. WWCA'97, Lecture Notes in Computer Science*, Vol. 1274 (Springer, Berlin, 1997) 29-42. http://www.horb.org/

[13] L.F.G. Sarmenta, Bayanihan: Web-Based Volunteer Computing Using Java, in: *Proc. WWCA'98, LNCS*, Vol. 1368 (Springer, Berlin, 1998) 444-461. http://www.cag.lcs.mit.edu/bayanihan/

[14] L.F.G. Sarmenta, S. Hirano, S.A. Ward, Towards Bayanihan: Building an Extensible Framework for Volunteer Computing Using Java, in: *Proc. ACM Workshop on Java for High-Performance Network Computing* (Palo Alto, 1998).

[15] D. Skillicorn, J.M.D. Hill, W.F. McColl, Questions and answers about BSP, *Scientific Programming*, **6**(3) (1997) 249-274. http://www.bsp-worldwide.org/

[16] H. Takagi, S. Matsuoka, H. Nakada, S. Sekiguchi, M. Satoh, U. Nagashima, Ninflet: a Migratable Parallel Objects Framework using Java, in: *Proc. ACM Workshop on Java for High-Performance Network Computing* (Palo Alto, 1998).

[17] L. Vanhelsuwe, Create your own supercomputer with Java, *JavaWorld*, (Jan. 1997). `http://www.javaworld.com/jw-01-1997/jw-01-dampp.ibd.html`