Bayanihan: Web-Based Volunteer Computing Using Java

Luis F. G. Sarmenta

MIT Laboratory for Computer Science
Computer Architecture Group
545 Technology Square
Cambridge, MA 02139
lfgs@cag.lcs.mit.edu
http://www.cag.lcs.mit.edu/bayanihan

DRAFT: Nov. 16, 1997 *

Abstract. This paper presents and discusses the idea of Web-based *volunteer computing*, wherein people cooperate in solving a large parallel problem by using standard Web browsers to volunteer their computers' processing power. Because volunteering requires no prior human contact and very little technical knowledge, it becomes very easy to build very large volunteer computing networks. At its full potential, volunteer computing can make it possible to build world-wide massively parallel computing networks more powerful than any supercomputer. Even on a smaller, more practical scale, volunteer computing can be used within companies or institutions to provide supercomputer-like facilities by harnessing the computing power of existing workstations. Many interesting variations are possible such as *networks of information appliances* (NOIAs), *paid* volunteer systems, and barter trade of computing resources.

In this paper, we discuss these intriguing possibilities, and identify several issues that will need to be addressed in order to successfully implement them. We also present an overview of the current work being done in the Bayanihan volunteer computing project.

1 Introduction

The introduction of Java by Sun Microsystems has made it possible to place platform-independent executable programs called *applets* on the Web, so that Internet users can execute them on their own machines without needing anything more than a Java-capable browser. Applets are being used by an increasingly large number of people and organizations to enhance their web pages with such features as animation, interactive demos, user-friendly forms, etc. So far, however, most of these uses have only concentrated on providing local usefulness

* An earlier draft was submitted to the *2nd International Conference on World-Wide Computing and its Applications* (WWCA'98), Tsukuba, Japan, March 4-5, 1998.

– that is, additional functionality or ease-of-use for the client user, but few additional benefits for anyone else. The potential for applets to be used for doing cooperative or collaborative computation has largely been unused

In this paper, we present and discuss one way to exploit this underutilized potential of Java – *volunteer computing*. The idea behind volunteer computing is to allow users from anywhere on the Internet to join in the solving of a big parallel problem by simply using a Java-capable browser and visiting a web site. Because it requires no prior human contact and very little technical knowledge on the part of the client user, volunteer computing makes it possible to build very large parallel computing networks very easily. Potentially, such a network can involve thousands, even millions of computers distributed around the world, and make it possible to achieve performance levels far beyond that of any current supercomputer. And, since volunteer computing does not require new hardware to be purchased, it can provide affordable supercomputing capabilities even to financially-constrained organizations such as small universities and universities and companies in developing countries. In fact, since volunteer computing makes it easy for organizations to pool their resources, it opens new possibilities for cooperative and collaborative research between institutions around the world.

This paper discusses these potential benefits of volunteer computing, as well as the issues and hurdles that must be addressed when implementing volunteer computing systems. It also presents a brief overview of the current work being done in the Bayanihan volunteer computing project. Sections 2 and 3 describe the motivations for implementing volunteer computing, including the disadvantages of current systems and the potential advantages of volunteer computing. Section 4 describes the current state of Project Bayanihan, presenting key research issues, and a framework designed to address them. Finally, Sect. 5 concludes with a brief summary.

2 Motivations

Volunteer computing is actually just a new variation on the old idea of using a network of workstations (NOW) to solve a parallel problem. NOWs first became popular because they allowed people to take advantage of existing (and mostly idle) workstations, enabling them to do parallel processing without having to purchase an expensive supercomputer. Global-scale NOWs, employing computers geographically distributed around the world and communicating through the Internet, have been used with great success to solve large parallel problems as far back as the early 90's [1, 2], and until today [3, 4].

Unfortunately, most of these earlier projects have used *ad hoc* software systems. Typically, a subsystem for providing communication and coordination between machines in the NOW had to be developed mostly from scratch as part of each project. Furthermore, in many cases, the systems were not even fully automatic. Participants had to *manually* request jobs, load them into their computers, execute them, and again manually submit the results. Thus, while these software systems were successfully used to build NOWs containing several thousands of workstations, doing so required a large amount of human effort in terms of setting up, coordinating, and administering the system.

2.1 PVM and MPI

In the past five years, this situation has been notably improved by the development of general-purpose and cross-platform parallel processing systems such as Parallel Virtual Machine (PVM) [5] and Message Passing Interface (MPI) [6].In such systems, the amount of manual work required is reduced significantly by the runtime system, which takes care of such things as automatically executing the appropriate code on each of the processors involved, keeping track of existing processors, routing and delivering messages, etc. At the same time, programming is made much easier by a general-purpose applications programming interface (API) which hides most of the details of the runtime system, and allows the user to write parallel programs for NOWs using a relatively simple high-level message-passing model. All this allows programmers to concentrate on writing applications instead of worrying about the low-level details.

Although systems like PVM and MPI make programming and setting-up NOWs significantly easier than in earlier ad hoc systems, setup requirements still impose practical limits on the size of NOWs that can be used with these systems. To perform a parallel computation using PVM, for example, one must:

1. Install the PVM daemon on all machines to be used in the computation.
2. Compile the application binaries for each target architecture.
3. Distribute these binaries to all the machines (either by explicitly copying them, or by using a shared filesystem).
4. Provide the owner of the computation with remote shell access (i.e., the ability to execute shell functions and programs remotely) on all machines, to allow remote execution of the PVM daemon and the application binaries.

Clearly, this process not only requires a lot of effort on the part of the system administrator, it also requires a lot of trust between the involved machines, and consequently, a lot of prior human communication between the administrators of different machines. For this reason, the use of PVM and MPI has mostly been restricted to internal use within research institutions, where such close coordination is possible.

2.2 Non-Java-Based Volunteer Computing

On October 19, 1997, `distributed.net`, a world-wide computing network composed of thousands of volunteer machines successfully cracked the RSA Systems RC5-56 code challenge [4]. This network achieved an average aggregate computing power equivalent to about 26,000 commodity PCs, and involved as many as 500,000 different computers (as distinguished by their IP addresses) over a span of 250 days [7].

A large part of this success can be attributed to the relative ease with which users are able to join `distributed.net`. To volunteer, a user only needs to:

1. Go to the `distributed.net` web site and download a compiled binary for his or her own machine architecture. Versions of the application software are coded, compiled, and tested by the `distributed.net` team for all major architectures.
2. Unarchive and install the software on the volunteer machines.
3. Run the software on the volunteer machines.

At this point, the software takes care of automatically communicating with the `distributed.net` server, requesting work, and sending back results. All the user has to do is leave the software running in the background on these machines. Unlike PVM, `distributed.net` does not require administrators to personally install daemons on the volunteer machines, or contact the volunteer users to acquire remote shell access. Thus, from the volunteers' point-of-view, `distributed.net` is significantly easier to use and understand than PVM.

While this approach to volunteer computing has clearly proven its worth, it still has its limitations. First, volunteers still need some technical knowledge – at least enough to download, install, and run the software. Furthermore, a non-trivial amount of programmer effort is still required to write, compile, and test code for all the possible target architectures. In fact, this step may even be harder with `distributed.net` than with PVM because there might not exist a standardized *cross-platform* library for doing low-level operations, such as communications, and synchronization, that are already provided by PVM,

Perhaps the biggest problem with systems like `distributed.net`, however, is security. Although `distributed.net`, unlike PVM, does not require volunteers to give explicit remote shell access to anyone, it does require them to run programs which implicitly provides the same kind of remote-shell-like access anyway. The code which a volunteer user downloads is executed with user permissions, which means it can access any of its user's files and data, including sensitive data such as credit card numbers and passwords. Clearly, such a serious security risk would (and should) make people wary of volunteering, and thus poses a major obstacle in achieving volunteer computing networks of larger sizes.

2.3 Java-Based Volunteer Computing

Java-based volunteer computing provides an alternative that addresses the problems we have seen so far, improving on both PVM and `distributed.net` in three main areas:

1. **Ease-of-use.** In a Java-based system, volunteering takes only one step: use a Java-capable browser to visit the server web site. There is no technical knowledge required beyond that of using a web browser. The user doesn't even have to be aware of the file system, or to understand what downloading an applet really means. Thus, *anyone*, even minimally-computer-literate users of commodity Internet services, can join in such computations.
2. **Platform-independence.** Java-based systems are easy for the programmers and administrators to use as well. Since the Java language and libraries

are platform-independent, programmers can "write once, run everywhere" – they can just do their development on any platform, compile the final code into one set of binaries (i.e., Java bytecode), and then post it on the appropriate web sites. Once this is done, any machine that has a Java runtime environment can download these applets and run them. Since free Java browsers and runtime environments have already been implemented on most major platforms today, practically anybody on the Internet can easily volunteer their computers. In fact, this would include even users of network computers (NCs) [10], whose operating systems may not allow them to download and execute non-Java binary programs such as those used by PVM and `distributed.net`.
3. **Security.** Like `distributed.net`, a Java-based system does not require giving away accounts or remote-shell access on the volunteer machines. Unlike `distributed.net`, however, the Java applets that volunteers download are executed in a *sandbox* that prevents them from stealing or doing damage to the volunteer's data.

In short, a Java-based volunteer computing system requires minimal setup and distribution effort on the part of the programmers and administrators, minimal technical knowledge and inconvenience on the part of volunteers, and minimal security risk. This characteristic ease with which a volunteer computing system can be set up is what makes it attractive as the next step in bringing supercomputing closer to the reach of the common user.

3 Potentials and Possibilities

3.1 True (or Altruistic) Volunteer Computing

Networks like `distributed.net` can be called *true* volunteer networks – their participants are volunteers in the true sense of the word. Specifically, they are:

1. *autonomous* – they join (and leave) of their own free will.
2. *altruistic* – they generally do not expect to be compensated for volunteering.
3. *anonymous* – they are unknown to the adminstrators, and therefore *untrustable.*

The appeal of true volunteer computing is in its potential to pool together the computing power of the 30 million computers on the Internet to form a worldwide network more powerful than any single supercomputer by several orders of magnitude. Though the idea of millions of people volunteering their computers may seem far-fetched, the success of projects like `distributed.net`, which managed to involve half a million computers in cracking the RC5-56 challenge, argues strongly to the contrary. The goal of Java-based volunteer computing is to go even further by making volunteering so easy and painless that literally anyone *and* everyone on the Internet can do it.

True volunteer computing has two major problems: One is that of security and correctness of the computation. While the volunteers are reasonably secure against malicious attacks (thanks to Java), the computation itself is not. Since volunteers are anonymous, there is no guarantee that a volunteer is not actually a saboteur who will not do the work assigned to it, but instead return fabricated results. There may be ways to use cryptography and redundancy to alleviate this problem, but their effectivity is still unclear (see Sect. 4).

The other problem is motivation – *why* would people want to volunteer their computers to do work for someone else? Some people would say, "Why not?" The typical PC or workstation is idle most of the time anyway, so why not put it to good use? Most people, however, would like to know what these "good uses" are before they volunteer. Thus, the choice of appropriate target problems plays a large part in attracting and motivating volunteers.

One class of target problems that have successfully attracted people are "cool" challenges like the RSA cryptographic challenges [8], Mersenne prime verification, and distributed chess [9]. A more interesting – and more useful – class of target problems, however, is what we might call *worthy causes*. These are realistic and practical problems which can be seen as serving "the common good" – either globally (e.g., protein folding computations for vaccine research), or locally (e.g., simulation studies for a traffic-congested mega-city). A recent and interesting example of a possible worthy cause problem,[2] is the SETI@home project, which aims to use volunteers' home computers to analyze radio telescope data for unusual patterns that may indicate extraterrestrial life [11].

3.2 Forced Volunteer Computing

Interestingly, *forced* volunteer systems are possible as well. These are systems where the participants are *not autonomous* (they do not have a choice about joining the computation), and *not anonymous* (they are known to the computation's administrators) An example of a forced volunteer system would be one where administrators run a background process on all company computers to allow them to be used for parallel computation whenever their users are idle.

Forced volunteer systems cannot match true volunteer systems in potential size and power, but they are more down-to-earth and feasible. For example, forced volunteer networks are not as prone to sabotage as true volunteer ones are. The use of encryption and firewalls can prevent attacks from *external* sources. *Internal* saboteurs may still be a problem, but these can be traced and controlled more easily (and punished more severely) than in true volunteer systems. In general forced volunteer systems are more manageable and more *secure* than true volunteer systems.

Applications. Forced volunteer computing can be used wherever there is a need or opportunity to pool existing (and idle) resources to attain supercomputing power that would otherwise be unaffordable. For example, companies can use computers on their intranet for solving CPU-intensive problems such as

[2] Or maybe just a cool challenge, depending on who one talks to.

market and process simulations, and data-mining [12]. Similarly, research labs and universities can use volunteer computing to turn their existing networks of workstations into virtual supercomputers. In financially-constrained institutions such as universities in developing countries, volunteer networks can also be used to *teach* supercomputing techniques, without requiring actual (expensive) supercomputers.

By making it very easy for institutions around the world to share their computing resources, volunteer computing opens up exciting new possibilities in world-wide cooperative and collaborative research efforts. For example, research institutions on opposite sides of the globe could barter-trade for each other's computing power (barter trade can also be a form of *paid* volunteer computing, see Sect. 3.3) – a university in the United States can allow a lab in Japan use of its CPUs at night, when it is day in Japan, in exchange for being able to use the Japanese lab's CPUs during the day, when it is night in Japan.[3]

Forced volunteer computing is already being done by many institutions using tools such as PVM or their own ad hoc tools ([13] has some links to PVM-based projects). Java-based forced volunteer computing improves on these by being much easier to use for everyone – users, programmers, and administrators. If a company decides to use its machines to do parallel computation, for example, the administrators would not need to spend time manually installing the computational software to be run on all the company machines – they could simply tell their employees to point their browsers to a certain web page on the company intranet, and leave the browser running while they work, or when they go home.[4] In this way, setting-up a parallel computation, a process that would normally take weeks for software installation and user education, can be done literally overnight.

3.3 Paid (or Commercial) Volunteer Computing

So far, we have assumed that volunteers are *altruistic*. That is, the participants work for the benefit of the common good (of the world, or the company), not expecting to be repaid. Non-altruistic, *paid* volunteer systems are also possible.

One way to increase participation in a volunteer computing system is to provide *individual compensation* as an incentive for volunteers. This can take the form of electronic credits proportional to the amount of work that they do (some ideas about this are presented in [14]), or it can be in the form of a *lottery*, such the one used in `distributed.net`, where part of the $10,000 RC5 challenge prize money from RSA Systems was offered to the volunteer team whose computer actually found the key[4]. Such lottery schemes not only require less money from the sponsor, but also encourage participation (i.e., the more one works, the bigger one's chances of winning are), and discourages cheating (i.e., pretending to do the work does not give one any chance of winning).

---

[3] I have so far been unable to find the original reference for this example.

[4] It is possible and easy to write an applet that continues to run in the background, even if the user moves to a different web page.

If we extend the idea of individual compensation to that of group trading of computing resources as commodities, we get a *market system*. Not only can users (or groups of users, such as companies) sell their computing resources, they can also buy the resources from others, or barter trade (like in the example in Sect. 3.2). If reliable mechanisms for electronic currency, accounting, and brokering are developed, market systems promise a practical and productive use for volunteer computing technology. The SuperWeb group at UCSB is already actively pursuing this goal [14].

Forced and paid volunteer computing can be combined in *contract-based* systems: clients can be required by contract to volunteer their computers as (part of) their payment for the goods or services received. For example, information providers such as search engines, news sites, and shareware sites, might require their users to run a Java applet in the background while they sit idle reading through an article, or downloading a file.[5] Such terms can be represented as a two-sided contract that both sides should find acceptable. While it is actually possible to hide forced volunteer Java applets inside web pages, for example, most users will consider this ethically unacceptable. Sites that require users to volunteer must say so clearly, and allow the user to back-out if they do not agree to the terms. Some of these ideas are discussed in [15, 16].

Some problems in anonymous paid volunteer systems are *cheating* (pretending to do work and getting paid for it), and *spying*. If Company A purchases computing power from Company B, a spy in B may read data from A's computation while he is doing it. For some classes of computations, *encrypted computation* techniques can be used to alleviate this problem[14].

3.4 Networks of Information Appliances (NOIA)

Many experts, both in industry and in the academic community, predict that in the near future, *information appliances* – devices for retrieving information from the Internet which are as easy to use as everyday appliances such as TVs and VCRs – will become commonplace[17, 18]. In the United States today, companies such as WebTV[19] are starting to develop and sell information appliances, and cable companies are starting to include support for cable modems, using the same cable that carries the TV signals[20]. It is not hard to imagine that within the next five or ten years, the information appliance will be as commonplace as the VCR, and high-speed Internet access as commonplace as cable TV.

This brings up an interesting possibility: why not use volunteer computing to harness the power of all these information appliances? Even if an information appliance is not as powerful as a desktop PC, the sheer number of them, potentially in the tens of millions (i.e., the number of people with cable TV), can make up for it.[6] Aside from their great size, these *networks-of-information-appliances*,

[5] To ensure that the user does not turn Java off, the server can check if it has been periodically receiving data from the applet – if it has not, then the server stops providing service to the user.

[6] Given the reasonably appropriate applications.

or NOIAs,[7] have many interesting features.

NOIAs can be *contract-based.* Cable or ISP companies can agree on a contract with their clients that would require clients to leave their information appliance boxes on and connected to the network 24 hours a day, running Java applets in the background when the user is idle. In exchange, the clients would receive compensation in the form of discounts, or premium services. In addition to the option of not participating, clients may also be given a choice of different kinds of computations they can participate in (e.g., charitable or worthy causes, commercial computations, etc.).

NOIAs also have the advantage of being easier to program and administer than other kinds of volunteer networks. Hardware-based cryptographic devices can make NOIAs *secure*, by preventing malicious volunteers from forging messages or modifying the applet code that they are given to execute. Also since users leave their information appliances on all the time, NOIAs are *stable* – the chance of a particular node leaving a NOIA is smaller than that in other kinds of volunteer networks. NOIAs composed purely of information appliances using the same type of processor are also *homogenous.* All these properties lessen the need for adaptive parallelism and fault-tolerance (see Sect. 4.1), and allows greater efficiency and more flexibility in the range of problems that can be solved on a NOIA.

While actual NOIAs may not be feasible right now, it is useful to keep them in consideration. Techniques developed for the other forms of volunteer computing (especially forced volunteer computing) are likely to be appliable to NOIAs when their time comes.

4 Project Bayanihan

The goal of Project Bayanihan,[8] is to identify and investigate the issues and problems involved in all forms of volunteer computing, and to develop useful technology and theory that can be used to towards turning these concepts to reality. We intend to pursue this goal in manageable steps, starting with the more feasible ideas, and moving on to more complex ones by extension and expansion. Specifically, we intend to start with forced volunteer computing, and move on to true volunteer computing where security and scalability issues become more complex. It is our hope that the results of research in these areas will become applicable to paid volunteer systems and NOIAs when the mechanisms and infrastructure necessary for their implementation (i.e., electronic currency and accounting for paid systems, and hardware infrastructure for NOIAs) become available.

[7] Interestingly, the word *noia*, Greek for "mind", conjures-up images of a brain-like massively parallel network of tens of millions of small processors around the world.

[8] Pronounced like **buy**-uh-*nee*-hun, *bayanihan* is a Filipino word meaning communal unity and cooperation, and epitomized by the old tradition of neighbors helping a relocating family by physically carrying their house, and moving it to its new location.

At present, we are developing a flexible software framework that would make it easy to program and set-up forced or true volunteer computing networks for various useful applications. Our goal is to develop a Java-based system that would be for volunteer computing what PVM was for NOW-based computing – an easy-to-understand interface that would allow programmers to write parallel programs for volunteer computing without worrying about low-level details such as network communication, scheduling, etc. In this section, we discuss some issues we face, and propose possible approaches to them. Then we give an overview of our current prototype framework, and present preliminary results.

4.1 Research Issues: Problems and Approaches

Adaptive Parallelism. By their nature, volunteer networks are *heterogenous* and *dynamic*. Volunteer nodes can have different kinds of CPUs, and can join and leave a computation at any time. Even nodes with the same type of CPU cannot cannot be assumed to have equal or constant computing capacities, since each can be loaded differently by external tasks (especially in systems which try to exploit users' idle times). For these reasons, models for volunteer computing systems must be *adaptively parallel* [23]. That is, unlike many traditional parallel programming models, they must *not* assume the existence of a fixed number of nodes, or or depend on any static timing information about the system.

Various strategies for implementing adaptive parallelism have already been proposed and studied. In *eager scheduling* [21], packets of work to be done are kept in a pool, from which worker nodes get any undone work whenever they run out of work to do. In this way, faster workers get more work according to their capability. And, if any work is left undone by a slow node, or a node that "dies", it eventually gets reassigned to another worker. Systems that implement eager schedule include Charlotte [21] and the factoring demo used in Sect. 4.3.

The *Linda* model [22] provides an associative *tuple-space* that can be used to store both data and tasks to be done. Since this tuple-space is global and optionally blocking, it can be used both for communication and synchronization between parallel tasks. It can also serve as a work pool, which like in eager scheduling, can allow undone tasks to be redone. Linda was originally used in the Piranha [23] system, and more recently implemented in Java by WWWinda [24], Jada [25], and Javelin [26].

In *Cilk* [27], a task running on a node, A, can spawn a child task, which the node then executes. If another node, B, runs out of work while Node A is still running the child task, it can *steal* the parent task from Node A and continue to execute it. This *work-stealing* algorithm has been shown to be *provably* efficient and fault-tolerant. Cilk has been implemented for NOWs [28], and has been implemented using Java applications in ATLAS [29].

Project Bayanihan aims to develop a framework that does not limit the programmer to using only one form of adaptive parallelism, but instead makes it easy to implement any of these forms and even develop new ones.

Fault-Tolerance. Faults in distributed systems can generally be classified into *stopping faults* and *Byzantine faults* [31]. Stopping faults cover cases of processors crashing or leaving, and are automatically handled by adaptively parallel systems. Byzantine faults cover all other kinds of faults, including *unintentional* random faults such as data loss or corruption due to faulty network links, or faulty processors, as well as *intentional* malicious attacks. In general, however, we can think of Byzantine faults simply as faults that result in the generation of incorrect result packets.

Byzantine faults can be handled using *replication* techniques such as *majority voting* or the more sophisticated and reliable algorithms in [31]. (To prevent sabotage by groups, replicated nodes should preferrably belong to different Internet domains.) However, this has the disadvantage of being very inefficient since a replication factor r means a factor r drop in aggregate computational speed.

We can improve efficiency by *spot-checking*. For each work packet that a node receives, there would be some probability p that the server already knows the correct answer, and just wants to check if the node is faulty. Although faulty nodes may be able to slip through for a while, the probability of *not* getting caught approaches zero exponentially in time, so faulty nodes get caught *eventually*.

Once a node gets caught, the server *backtracks* through the results, recomputing any results depending on the offending nodes results, and then *blacklists* the offending node, never allowing it to join the computation again. (More flexible versions of blacklisting can also be used in situations where unintentional and transient occasional errors are expected even from non-faulty nodes, e.g., due to power fluctuations, and we do not want to blacklist a node immediately and forever just because of these temporary failures.)

Another way to achieve fault-tolerance is by simply choosing more fault-tolerant problems. Such problems include those that do not require 100% accuracy in the first place, such as sound or graphics processing where a little static or a few scattered erroneous pixels would be unnoticeable or can be averaged out to be make them unnoticeable. Other problems include those that have easily-verifiable results, such as search problems (e.g., travelling salesman) with *rare* results that can easily be checked by the server without adding significant extra load, and rendering, where a human user can visually recognize any unacceptable errors and ask the system to recalculate (and blacklist the node responsible for the problematic areas).

Security. Malicious attacks or *sabotage* can take many forms. In general, it involves a malicious node returning erroneous data. This node can either be an internal saboteur who is actually a participating volunteer, or an external *spoofer* – a node that has *not* volunteered, but sends messages forged to look like they came from one of the volunteers.

Spoofing can be prevented with *digital signatures* [32]. These enable the server to verify that a result packet indeed comes from a volunteer. They can also be used in the other direction, to assure a volunteer that an applet really comes from the server, and not from a spoofer. Digital signatures are most useful in protecting non-anonymous volunteer systems, such as forced volunteer networks or NOIAs, from external attack. Unfortunately, however, digital signatures are ineffective against internal saboteurs, and thus useless in true volunteer systems where anyone, even saboteurs, are allowed to volunteer. This is because digital signatures can only authenticate the *source* of a data packet, not its *content*.

One way to authenticate the content of a data packet is to include a *checksum* computation in the code. This way, if the node does not run the code, or runs it incorrectly, the checksum will not match, and the server can be alerted. In most cases, checksums will catch both unintentional errors, and simple malicious attacks such as returning random packets. We can also use checksums to authenticate sources (and prevent spoofing) by transmitting a *different* checksum key with each work packet. This way, an external spoofer would not know the correct key to use, and cannot forge bogus result packets.

Digital signatures and checksums are only useful against malicious attacks if volunteers are forced to compute them and cannot compute them independently of the work. This is true for NOIAs, where the node hardware and firmware can prevent users from disassembling or modifying the bytecodes they receive from the server. In general, however, it is possible for sophisticated malicious volunteers to disassemble the bytecode they receive, identify the signature and checksum generating code, and use these to forge a result packet containing arbitrary data. One way to guarantee that a node cannot fake a checksum computation is to prevent it from disassembling the code. This is not an easy task, but in some cases, it may be possible to apply *encrypted computation* techniques, such as those that [14] proposes to use against spying.

If cryptographically preventing disassembly is not possible, we can resort to *dynamic obfuscation* techniques to make understanding the disassembled code and isolating the checksum code as difficult as possible. Dynamic obfuscation extends the idea of *static* obfuscation (such as done in Borland's JBuilder [33]) by continously obfuscating code *dynamically* in time. For example, we can randomly vary the checksum formula and its location within the bytecode from work packet to work packet. This would prevent hackers from manually disassembling and modifying the bytecode once and for all. We can also insert *dummy code* at varying locations. These schemes are inspired by *polymorphic computer viruses*, which use them to hide themselves from anti-virus programs [34].

Although very difficult, de-obfuscating polymorphic viruses is not impossible because viruses, being self-reproductive (by definition), contain the obfuscating code. Thus, once one version has been cracked and disassembled, it is possible to reverse engineer this code and develop an "antidote" which can disassemble other instances of the viruses. Dynamic obfuscation in volunteer computing, however, has the advantage of using the server to do the obfuscation. This should make it possible to constantly and arbitrarily replace the code, and make it impossible for hackers to catch up. Provided that the volunteer computing system allows the work code to change from packet to packet (as the current Bayanihan framework does), dynamic obfuscation may easier to implement and more generally applicable than encrypted computation.

Scalability and Congestion. One of the major problems in Java-based volunteer computing is that currently, security restrictions dictate that applets running in users' browsers can only communicate with the Web server from which they were downloaded. This forces Java-based volunteer networks into *star topologies*, which have the disadvantage of having high congestion, no parallelism in communications, and not being scalable.

To solve this problem, we may allow volunteers who are willing to exert extra effort to download Java *applications*, and become *volunteer servers*. Volunteer server applications need to be run outside a browser, but do not have security restrictions. This lets them connect with each other in arbitrary topologies, as well as act as star hubs (centers) for volunteers running applets. Newer browsers such as Netscape and Microsoft Internet Explorer are starting to support *signed applets*, which will also be free of restrictions. These applets can be used as volunteer servers as well.

4.2 System Design

Our current goal in Project Bayanihan is to develop a flexible software framework that will allow us to explore these problems by making it possible to implement various solutions to them. At present we have a basic prototype using HORB [35], a distributed object library that provides remote objects capabilities in Java similar to those of Sun's RMI [36], but not requiring JDK 1.1. In this section, we provide a brief overview of this prototype framework. The details will be discussed in more depth in a separate paper [37].

Architecture. Figure 1 shows a high-level block diagram of the current Bayanihan system.

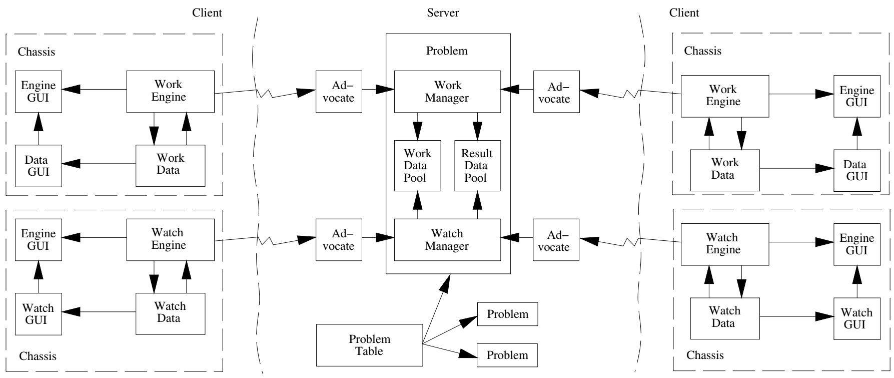

Fig. 1. A Bayanihan system

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's done. Data objects are generally polymorphic and know how to process themselves. To execute work, a work engine needs only to pass the work data a pointer to itself, and call the work data's `run()` method. The data will compute itself, using the work engine to communicate with the work manager, if necessary. Both the engine and data have associated *GUI* types 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*[9] object serves as the engine's representative and forwards the engine's calls to the manager. The manager has access to several *data pools*. These pools 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*. There may be many independent problems existing in a server at the same time. A *problem table* keeps track of these, and can be used by volunteers to choose the problem they want to participate in.

Flexibility. The Bayanihan framework heavily exploits the object-oriented features of the Java language to provide flexibility in the kinds of systems that can be implemented using it. Implementations of Bayanihan can be varied at three main levels:

1. *Applications.* At the highest level, application programmers can write their own data objects and GUIs, and create their own problem objects by putting together appropriate engine, manager, and data pool components chosen from a component library. Given a component library for eager-scheduling, for example, one can write applications for rendering, RC5, and many other applications simply by varying the work data and GUI classes. Within a single application, it is also possible write different user interfaces. For example, different data GUIs may be written to display data in different ways.
2. *Components.* Programmers can also write their own engines, managers, and data pools to implement different computation models. This would allow us, for example, to experiment with different forms of adaptive parallelism, or different fault-tolerance mechanisms.
3. *Infrastructure.* It is also possible to change other support objects such as the chassis, and the problem table. For example, there may be an application chassis and an applet chassis. Also, different chassis objects may provide different ways to let the user select problems from the problem table. Making changes at this level may allow us to implement different security mechanisms, or address scalability issues.

[9] A better name may be *proxy*. However, *proxy* means something else in HORB.

4.3 Preliminary Results

Currently, we have used the Bayanihan framework to write a simple application that factors a long integer N by dividing the search space $\{1, ..., \sqrt{N}\}$ into fixed-sized work packets which are executed by the work engines. This problem, inspired by [38], although somewhat unrealistic, was chosen because its simplicity and predictability make it easy to measure performance and analyze results.

Figure 2 shows some results for N =1234567890123456789, with 112 work packets of size 100000000 each. We used five identical 200MHz dual-Pentium machines (one server, and four clients) connected to a 10Base-T Ethernet hub, and running Windows NT 4.0. The server application was run using Sun's JDK 1.1.4, and the worker applets were run using Netscape 4.02. The speeds in the figure represent the average number of numbers checked per millisecond, over the course of searching the whole target space. The number of workers represents the number of worker applets run on the client machines. Each client machine was used to run up to two worker applets.[10] The ideal speed is computed as the pure computation speed (measured at the client side) multiplied by the number of workers. With eight workers, we get the figures in Table 1.

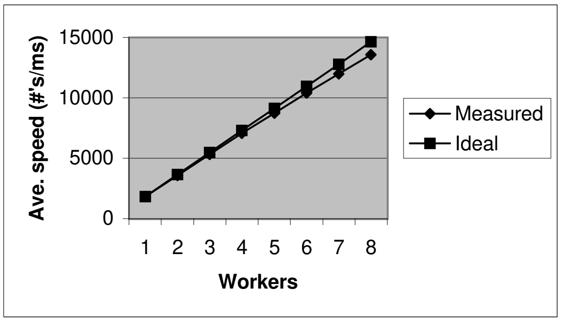

Fig. 2. Timing measurements

Table 1. Speedup: 8 workers

measurement	num / ms
pure computation (1 worker)	1828
ave. speed (8 workers)	13571
speedup	7.42
efficiency	92.8%

[10] Since the worker applet has only one (sequential) computation thread, running only one instance of it does not result in parallelism, even on a dual-Pentium machine.

To evaluate the performance of Java, relative to C, we wrote a simple C program, compiled it with `djgpp` (the `gcc` implementation for MS-DOS). Comparing its performance with the one-worker pure computation performance, we get the results shown in Table 2. These results show that at least for simple tight-loop computations such as our factoring algorithm, the performance of the Java just-in-time-compiler in Netscape is comparable to native C code.

Table 2. Comparing C and Java

version	num / ms
C	1879
Java	1828

While these results are admittedly preliminary and based on a toy problem, we believe that they give us hope that Java-based implementations of volunteer computing will be feasible and practical.

5 Conclusion

In this paper, we have shown how volunteer computing brings potential benefits, but also involves many challenging problems. There is plenty of space in this field for exploration, and indeed many have already started [14, 16, 21, 26, 29, 30, 39]. Project Bayanihan joins these efforts with a different approach: since the perfect approach is not clear, we do not settle on a single one, but instead take advantage of remote object technology to build a flexible framework that we hope would allow us to try *any* approach. Work on Project Bayanihan is ongoing, but preliminary results give us a positive outlook and hope for success.

6 Acknowledgements

Thanks to Dr. Satoshi Hirano of ETL for his invaluable moral support and help in the writing of the HORB version of Bayanihan; to my colleagues and friends who reviewed and edited this paper: Danilo Almeida, Mark Herschberg, Steven Johnson, and Victor Luchangco; to Lydia Sandon and Mark Herschberg for helping out with the demo for SC97; to my thesis supervisor Prof. Steve Ward; and to everyone else who has given me their feedback and support. While doing this work, I was been financially supported in part by the Department of Science and Technology of the Philippines, MIT teaching assistantships, the MIT Japan Program, Ateneo de Manila University, and the Defense Advanced Research Projects Agency.

References

1. Volker Strumpen. Coupling Hundreds of Workstations for Parallel Molecular Sequence Analysis. *Software - Practice and Experience.* 25(3) 1995. pp. 291-304.
2. Steven Levy. Wisecrackers. *Wired*, Issue 4.03, March 1996. `http://www.hotwired.com/wired/4.03/features/crackers.html`
3. W. Gibbs. CyberView. *Scientific American*, May 1997.
4. Adam L. Beberg, Jeff Lawson, and David McNett. `distributed.net`. `http://www.distributed.net`
5. Al Geist et al. *PVM: Parallel Virtual Machine: A User's Guide and Tutorial for Networked Parallelism.* MIT Press, 1994. `http://www.netlib.org/pvm3/book/pvm-book.html`
6. William Gropp, Ewing Lusk, Anthony Skjellum. *Using MPI.* MIT Press, 1994. `http://www.epm.ornl.gov/ walker/mpi/index.html`
7. `distributed.net` press release, Oct. 22, 1997. `http://www.distributed.net/pressroom/56-announce.html`
8. RSA Data Security. RSA Factoring Challenge. HTML document. URL: `http://www.rsa.com/factor/chalenge.htm`
9. Remy de Russcher. Possible Projects. `http://www.distributed.net/projects.html`
10. NCWorld web site. `http://www.ncworld.com/`
11. SETI@home home page. `http://www.bigscience.com/setiathome.html`
12. *Communications of the ACM*, Nov. 1996.
13. PVM home page. `http://www.epm.ornl.gov/pvm/`
14. A. D. Alexandrov, M. Ibel, K. E. Schauser, and C. J. Scheiman. SuperWeb: Towards a Global Web-Based Parallel Computing Infrastructure *11th International Parallel Processing Symposium.* Geneva, April 1997.
15. DigiCrime Computational Services via Java. HTML Document, June, 1996. `http://www.digicrime.com/java.html`
16. Laurence Vanhelsuwe. Create your own supercomputer with Java. *JavaWorld*, Jan. 1997. `http://www.javaworld.com/jw-01-1997/jw-01-dampp.ibd.html`
17. Bill Gates. *The Road Ahead.* Viking, a division of Penguin Books USA Inc., 1995.
18. Nicholas Negroponte. *Being Digital.* Vintage Books, a division of Random House, Inc., 1995.
19. WebTV home page. `http://www.webtv.com/ns/index.html`
20. Media One home page. `http://www.mediaone.com/`
21. Arash Baratloo, Mehmet Karaul, Zvi Kedem, and Peter 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/`
22. Nicholas Carriero and David Gelernter. Linda in context. *Communications of the ACM*, 32(4):444-458, April 1989.
23. David Gelernter and David Kaminsky. Supercomputing out of recycled garbage: Preliminary experience with Piranha. In *Proceedings of the 1992 ACM International Conference of Supercomputing*, pages 417-427, Washington, D.C., July 1992.
24. Y. S. Gutfreund. The WWWinda Orchestrator. `http://info.gte.com/ftp/circus/Orchestrator`
25. Davide Rossi. Jada home page. `http://www.cs.unibo.it/ rossi/jada/`

26. 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.
27. Blumofe, et al. Cilk: An Efficient Multithreaded Runtime System, in *Proceedings of the 5th ACM SIGPLAN Symposium on Principles of Parallel Programming* (PPOPP '95), July 19-21,1995, Santa Barbara California, pp. 207-216. `http://theory.lcs.mit.edu/~cilk/`
28. Robert D. Blumofe, and Philip A. Lisiecki. Adaptive and Reliable Parallel Computing on Networks of Workstations. In *Proceedings of the USENIX 1997 Annual Technical Symposium*, Anaheim, California, January 6-10, 1997.
29. J. Eric Baldeschwieler, Robert D. Blumofe, and Eric A. Brewer. ATLAS: An Infrastructure for Global Computing, in *Proceedings of the Seventh ACM SIGOPS European Workshop: Systems Support for Worldwide Applications*, Sept. 1996.
30. Tim Brecht, et al. ParaWeb: Towards World-Wide Supercomputing, in *Proceedings of the Seventh ACM SIGOPS European Workshop: Systems Support for Worldwide Applications*, Sept. 1996.
31. Nancy A. Lynch. *Distributed Algorithms*. Morgan Kauffman Publishers, Inc., 1996.
32. B. Schneier. *Applied Cryptography*. 2nd ed. Jon Wiley, 1996.
33. Borland, Inc. JBuilder. `http://www.borland.com/jbuilder/`
34. McAfee Associates. Virus Information Library: Polymorphism. `http://www.mcafee.com/support/techdocs/vinfo/t0022.asp`
35. HIRANO Satoshi. HORB: Extended execution of Java Programs. in *Proceedings of the First International Conference on World-Wide Computing and its Applications (WWCA97)* `http://ring.etl.go.jp/openlab/horb/`
36. Sun Microsystems, Inc. Remote Method Invocation. `http://chatsubo.javasoft.com/current/rmi/index.html`
37. Luis F. G. Sarmenta, Satoshi Hirano, et al. Bayanihan: an Object-Oriented Framework for Volunteer Computing Using Java. To be submitted to the *ACM 1998 Workshop on Java for High-Performance Network Computing*. Palo Alto, California, Feb. 1998.
38. Geoff Voelker and Dylan McNamee. The Java Factoring Project. HTML document with Java Alpha version applets, Sept. 1995. `http://www.cs.washington.edu/homes/dylan/ContestEntry.html`
39. *ACM 1997 Workshop on Java for Science and Engineering Computation*. Las Vegas, June 1997.