

Volunteer Computing

by

Luis F. G. Sarmenta

B.S. Physics and B.S. Computer Engineering,
Ateneo de Manila University (1992,1993)
S.M. Electrical Engineering and Computer Science,
Massachusetts Institute of Technology (1995)

Submitted to the Department of Electrical Engineering and Computer Science
in partial fulfillment of the requirements for the degree of

Doctor of Philosophy

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

June 2001

© Massachusetts Institute of Technology 2001. All rights reserved.

Author
Department of Electrical Engineering and Computer Science
March 8, 2001

Certified by
Stephen A. Ward
Professor of Electrical Engineering and Computer Science
Thesis Supervisor

Accepted by
Arthur C. Smith
Chairman, Department Committee on Graduate Students

Volunteer Computing
by
Luis F. G. Sarmenta

Submitted to the Department of Electrical Engineering and Computer Science
on March 8, 2001, in partial fulfillment of the
requirements for the degree of
Doctor of Philosophy

Abstract

This thesis presents the idea of *volunteer computing*, which allows high-performance parallel computing networks to be formed easily, quickly, and inexpensively by enabling ordinary Internet users to share their computers' idle processing power without needing expert help. In recent years, projects such as SETI@home have demonstrated the great potential power of volunteer computing. In this thesis, we identify volunteer computing's further potentials, and show how these can be achieved.

We present the Bayanihan system for *web-based* volunteer computing. Using Java applets, Bayanihan enables users to volunteer their computers by simply visiting a web page. This makes it possible to set up parallel computing networks in a matter of minutes compared to the hours, days, or weeks required by traditional NOW and metacomputing systems. At the same time, Bayanihan provides a flexible object-oriented software framework that makes it easy for programmers to write various applications, and for researchers to address issues such as adaptive parallelism, fault-tolerance, and scalability.

Using Bayanihan, we develop a *general-purpose* runtime system and APIs, and show how volunteer computing's usefulness extends beyond solving esoteric mathematical problems to other, more practical, *master-worker* applications such as image rendering, distributed web-crawling, genetic algorithms, parametric analysis, and Monte Carlo simulations. By presenting a new API using the *bulk synchronous parallel* (BSP) model, we further show that contrary to popular belief and practice, volunteer computing need not be limited to master-worker applications, but can be used for coarse-grain *message-passing* programs as well.

Finally, we address the new problem of maintaining reliability in the presence of malicious volunteers. We present and analyze traditional techniques such as *voting*, and new ones such as *spot-checking*, *encrypted computation*, and *periodic obfuscation*. Then, we show how these can be integrated in a new idea called *credibility-based fault-tolerance*, which uses probability estimates to limit and direct the use of redundancy. We validate this new idea with parallel Monte Carlo simulations, and show how it can achieve error rates several orders-of-magnitude smaller than traditional voting for the same slowdown.

Thesis Supervisor: Stephen A. Ward
Title: Professor of Electrical Engineering and Computer Science

To my dearest Michelle,

To my parents, Rey and Baby Sarmenta

*and to the memory of my grandmother,
"Nanay" Sabina Sarmenta,
who always wanted me to be a "doctor".*

Ad majorem Dei gloriam.

Acknowledgments

Forever I will sing the goodness of the Lord.
– Psalm 89:1

Rejoice in the Lord always. I shall say it again: rejoice!
– Philippians 4:4

As I finish this thesis, I indeed cannot help but rejoice in the Lord – not only because I am finally done (which is a miracle in itself!), but even more because as I look back on these past years, I recall all the ways God has blessed me, and it just fills me with gratitude and joy. In this short section, I would like to acknowledge the biggest of these many blessings – the people in my life who, in one way or another, whether they know it or not, have been instruments of God’s grace and love for me throughout this long journey.

First, of course, I would like to thank my thesis committee, who made this thesis possible. Thanks to my advisor, Steve Ward, for his kind support and help throughout my 8 years at MIT, and especially for his time and help this past year as I struggled to finish. Thanks to Steve too for the funding that he has helped me get at times when I really needed it. Thanks also to my readers, Charles Leiserson and Lynn Stein, for all the feedback, ideas, and writing tips they gave me, and for their time and ever-cheerful support and encouragement.

I would also like to give my very special thanks to Satoshi Hirano, my mentor at ETL in Japan, without whom I could not have made it very far in my project. Since my first stay at ETL, when I developed the Bayanihan framework using his HORB, Hirano-san has always given me not only much technical help and advice, but much friendship and cheer as well. I will always be thankful for having him as a mentor and friend.

Thanks to the other people who contributed to this work in one way or another. Thanks to the students who worked with me: Lydia Sandon (RC5), Eamon Walsh (GAs), and Alex Yip (Mandelbrot graphics) at MIT, and Mark Bucayan (ICQ), Jerome Punzalan (shared whiteboard), and Eric Vidal (multi-user chat) at Ateneo. Thanks to Mark Herschberg for forming the other half of our two-man research exhibit booth at SC’97. I still can’t believe we managed to pull that off. Special thanks to Victor Luchangco and Danilo Almeida, who have spent many hours discussing my work with me and giving me much feedback, help, ideas, and encouragement. Victor, in particular, helped me a lot with the fault-tolerance math in Chap. 6. Danilo and Victor also happen to be two of my best friends at MIT, and they have been with me and helped me in many many other ways. Thanks to Chris Terman for his kind support and encouragement, and for helping me get funding last summer.

*This work and my studies at MIT have been supported in part, directly or indirectly, by an Engineering and Science Education Project (ESEP) scholarship from the Department of Science and Technology of the Philippines (DOST), MIT, Ateneo de Manila University, and the Electrotechnical Laboratory of Japan (ETL).

Thanks too to Marilyn Pierce and Cornelia Colyer for helping me get this document in. Finally, thanks to Arvind and Michael Rappa, who have not only been nice to me, but who have also unknowingly played a key part in this thesis. It was in Spring '96, while teaching parallel processing as a TA for Arvind's architecture class, and discussing models for collaboration in Michael's research management class that everything fell in place and I had my "eureka moment".

Thanks also to my officemates at MIT LCS for their company and friendship, especially to Russ Tessier, with whom I enjoyed sharing the highs and lows of graduate student life, as well as occasional episodes of Seinfeld and The Simpsons on our ancient TV set.

Thanks to the people I've had the pleasure of working with at ETL: to Yasu-san and Takagi-san, with whom I worked on HORB, and to Ichisugi-san, Suzaki-san, and Yves Roudier, for the good times I've had at ETL and for their friendship. Thanks too to Higanosan for her help in getting me settled in Japan, for all her help at work, and for her friendship. And of course, thanks also to the people of the MIT Japan Program who made my trip there possible in the first place.

Thanks also to all the other people with whom I have had good discussions with at workshops and conferences throughout these years. I have especially enjoyed knowing the Javelin people, particularly Klaus Schauer (whom I first met and exchanged concept papers with at ASPLOS '96), Peter Cappello, and Michael Neary, who have all been very friendly, and with whom I have had good and helpful discussions. Thanks too to Arash Baratloo and Mehmet Karaul of the Charlotte group at NYU. The first version of my Mandelbrot demo actually borrowed some of the graphics code from the Charlotte demo.

Finally, special thanks to the people back home without whom I could not even have gotten to MIT. Thanks to the people at DOST/ESEP, especially to Dr. Estrella Alabastro, who was in charge of ESEP during most of my scholarship, for her kindness, and to Teody Dayaon, not only for making sure I got my monthly sustenance, but also for her kind support with many other details. Special thanks to Fr. Ben Nebres, S.J., and Arnie del Rosario of Ateneo for helping me getting the ESEP scholarship which made all this possible. Fr. Ben's humility and simplicity never cease to amaze and inspire me. I never imagined that a university president can be so approachable, friendly, and truly caring as Fr. Ben has been. Arnie was my first mentor and sponsor in the academic world of computer science. I am very thankful to him for the opportunities he opened up to me. Special thanks too to John Paul Vergara who has not only given me feedback and support for my thesis, but has also been kind enough to actually take on practically double his teaching load of classes for a semester and a half, just so I can finish my thesis. Of course, thanks also to Marco Changho who did the same thing for a semester. I think I owe them a *very* big dinner. Finally, thanks to all my other colleagues at Ateneo for their support, and to Nanette and Mark Bautista and the other staff for their help with various everyday stuff at work.

Of course, life is not just about research and work. In fact, if there is one lesson that I have learned in all the years of my Ph.D. education at MIT, it is that life is *a whole lot more* than just research and work. Discovering new ideas, writing programs and papers, and going to conferences are all great, but in the end, these are all nought compared to the more important things in life – how we love others, experience their love, and grow in God together. In this light, I would also like to thank all the people who have not necessarily contributed directly to this thesis, but who have, by their love and friendship, nevertheless made all this, and more, possible.

Before all, I would like to thank my parents, who have filled my life with love, care, and support from the very beginning, and who have taught me what love means. They

have raised me up well, and have taught me the important things in life. Without them, I would not be where I am, and who I am, right now. This thesis is every bit theirs as it is mine. I can only hope that as I now reach the summit of my academic education, I can continue in the rest of my life's education by learning to be as loving as they are.

Thanks also to my siblings, Cocoy, Clarissa, and John, for their love, prayers, and support, and for making me enjoy and look forward to going home. I could not have done my thesis without them. Thanks too to my uncle and godfather, "Ninong" Rainer Sarmenta, for introducing me to computers when I was 9. Thanks also to him, and to all my uncles, aunts, and cousins for their love, friendship, and support since I was little. And, of course, thanks to my grandparents, who have all passed on, but to whom I am ever grateful.

Thanks to all my friends from back home, who have supported me with their prayers and friendship even from a distance. Special thanks to HOUSTON – Cara and Baba Fernandez, Ina and Ed Villano, and JJ Tagorda – for their support all these years, and for the good times we always have whenever I go home. Thanks also to the "Miradorians" – Fr. Kit Bautista, Rofel Brion, Doris Montecastro, Charlotte Natividad, Tere Navarro, and Papo Ong Ante – for sharing such a wonderful retreat with me, and for their prayers, friendship, and support. Thanks as well to all my brothers and sisters in Ligaya and CYA for their prayers.

Thanks to all my friends from FSA and TCC at MIT, especially to Alex Diaz, for helping me when I was new in Boston and for being a good friend throughout, and to Marc Ibañez, my old roommate and good friend.

Very special thanks to my friends from Catholic Fellowship. CF has truly been a miracle. Since it started in 1995, it has changed my life at MIT. I remember walking to campus one day and catching myself feeling the same happy feeling I feel when going to church. Strange thought, but true. Because of CF, I now see MIT as a place of prayer – sometimes even more than a place of study. Thanks to all my friends, especially to Danilo Almeida, Br. Nick Austriaco, Lawrence Chang, Sunil Konath, Desmond and Beata Lim, Marissa Long, Mike Lopez, Victor Luchangco, Terri and Nick Matsakis, Robert Palazzolo, Br. Darren Pierre, George Toolan, and Johan van Biesen, for all the times they have shared my joys and sorrows, and shared theirs with me as well. They have been instruments of God's peace and love to me, and I have grown a lot because of them.

Aside from my friends from CF, God has also sent me good shepherds to strengthen, guide, and encourage me along my spiritual journey. First, I would like to thank Fr. Tom Holahan, who gave me much support and guidance while he was at MIT. Second, and most especially, I would like to thank Fr. John Ullrich, OFM, my spiritual director and friend, who has supported me, guided me, and been with me through the best and worst times of my spiritual life these past three years. He has taught me the wonders and mysteries of God's love and kindness, and has done so, not only through his kind and wise words, but even more through the example of his life. In Fr. John, I can catch a glimpse of God's own kindness and love, and for this I am truly grateful.

Finally, I would like to thank my dearest friend, and God's most wonderful gift to me, Michelle Charles. Michelle has been with me through the hardest times of my stay at MIT, and has made these times also the happiest. My life has truly been so much more joyful since I have met her, and I feel so blessed and thankful to God whenever I think of her. Being with her has made me grow and has brought me closer to God, and through her, I have really felt God's love. I love her very, very much, and I can't wait to start sharing the rest of our lives together, learning, loving, and growing together in God's love.

As I end here, I would like to thank God again, who has given all these blessings, and

without whose grace none of this would have been possible. It always amazes me how God's plans for us are so much better than we can ever imagine ourselves. So now, I look forward with much joy and anticipation to what He has planned next for my life. Thank you so much, Lord!

Contents

1	Introduction	19
1.1	Toward Electronic Bayanihan	19
1.2	Thesis Goals	21
1.3	Thesis Overview	22
1.4	Contributions	27
2	Potentials and Challenges	29
2.1	What is Volunteer Computing?	29
2.1.1	Metacomputing	29
2.1.2	Volunteer Computing	31
2.2	Potential Forms of Volunteer Computing	32
2.2.1	True Volunteer Computing	32
2.2.2	Private Volunteer Computing	34
2.2.3	Collaborative Volunteer Computing	35
2.2.4	Commercial Volunteer Computing	36
2.2.5	NOIAs	36
2.3	Research Issues and Challenges	38
2.3.1	Accessibility	38
2.3.2	Applicability	39
2.3.3	Reliability	41
2.4	A Taxonomy of Volunteer Computing Systems	43
2.5	Context and Conclusion	44
3	The Bayanihan Web-based Volunteer Computing System	45
3.1	Web-based and Java-based Volunteer Computing	45
3.1.1	Limitations of Application-based Systems	45
3.1.2	Web-based and Java-based Systems	46
3.2	The Bayanihan Web-based Volunteer Computing System	48
3.2.1	Design Goals	48
3.2.2	System Design	48
3.2.3	Multi-level Flexibility	50
3.2.4	Using the Framework	51
3.3	Ease-of-Use and Accessibility	51
3.3.1	Volunteering	51
3.3.2	Watcher Applets and Interactive Parallel Computing	53
3.3.3	Practical Results	54
3.4	Programmability and Applicability: Writing Applications	55

3.4.1	The Generic Master-Worker Runtime System	55
3.4.2	Writing Applications: The Basic APIs	57
3.4.3	Application Frameworks and Other APIs	59
3.5	Developing Generic Mechanisms	59
3.5.1	Adaptive Parallelism	60
3.5.2	Reliability	61
3.5.3	Scalability	64
3.6	Conclusion	66
3.6.1	Related Work	66
3.6.2	Future Work	67
4	Master-Worker Style Applications	69
4.1	Introduction	69
4.2	Brute-Force Search	71
4.2.1	Example: Brute-Force Factoring	71
4.2.2	Example: RC5 Decryption	73
4.3	Image Rendering and Processing	73
4.3.1	Example: Interactive Mandelbrot Set Renderer	73
4.4	Genetic Algorithms	75
4.4.1	Framework Design	76
4.4.2	Example: Function Optimization	77
4.5	Communicationally Parallel Applications	77
4.5.1	Example: Distributed Web Crawling	78
4.5.2	Other Possible Applications	79
4.6	Parametric Analysis and Monte Carlo Simulations	79
4.6.1	Writing Parametric Analysis Applications	80
4.6.2	Writing Monte Carlo Applications	80
4.6.3	Example: Computing π	81
4.6.4	Example: Fault-Tolerance Simulator	82
4.7	Conclusion	85
5	BSP as a Programming Model for Volunteer Computing	87
5.1	Introduction	87
5.2	The BSP Programming Model	87
5.3	Implementing the BSP Model	89
5.3.1	The Bayanihan Master-Worker Based Implementation	89
5.3.2	Adaptive Parallelism and Fault-Tolerance	90
5.4	Writing Bayanihan BSP programs	91
5.4.1	The Bayanihan BSP Methods	91
5.4.2	Sample Code	97
5.5	Implementation Results	97
5.6	Improving Bayanihan BSP	99
5.6.1	Caching	100
5.6.2	Other Mechanisms Supporting Caching	101
5.6.3	Other Possible Improvements	102
5.7	Conclusion	103
5.7.1	Related Work	103
5.7.2	Summary	104

6	Reliability in the Presence of Malicious Volunteers	105
6.1	Introduction	105
6.2	Overview of Approaches	106
6.2.1	Application Choice	106
6.2.2	Authentication, Encryption, and Obfuscation	109
6.2.3	Redundancy and Randomization	110
6.3	Voting and Spot-checking	111
6.3.1	Models and Assumptions	111
6.3.2	Majority Voting	113
6.3.3	Spot-Checking with Blacklisting	115
6.3.4	Spot-checking without Blacklisting	118
6.3.5	Combining Spot-Checking and Voting	120
6.4	Credibility-based Fault-Tolerance	121
6.4.1	Overview: Credibility and the Credibility Threshold Principle	121
6.4.2	Calculating Credibility	123
6.4.3	Using Credibility	126
6.5	Simulation Results	130
6.5.1	The Simulator	130
6.5.2	Results	132
6.6	Conclusion	141
7	Conclusion	145
7.1	Summary	145
7.2	Future Work	146
7.3	Final Words	147
A	Sample Code	151
A.1	Brute-Force Factoring	151
A.1.1	Work Object	151
A.1.2	Result and Related Objects	153
A.1.3	Config and Request Object	155
A.1.4	Watcher GUI Object	156
A.1.5	Program Object (Passive Style)	160
A.1.6	Program Object (Active Style)	163
A.2	Computing Pi through Monte Carlo Methods	167
A.2.1	Work Object	167
A.2.2	Result Object	169
A.2.3	Statistics Collector Object	169
A.2.4	Program Object	171
A.3	Fault-tolerance Simulator	172
A.3.1	Sample Program Object	172
B	Encrypted Computation and Obfuscation	175
B.1	Motivation	175
B.2	Overview	176
B.2.1	The Problem	176
B.2.2	Applications	176
B.2.3	Possible Solutions	177

B.3	Encrypted Computation	177
B.3.1	Basic Examples	178
B.3.2	Sander’s and Tschudin’s work	179
B.3.3	Critique	181
B.4	Obfuscation	182
B.4.1	Obfuscation Transformations	183
B.4.2	Critique	187
B.5	Time-limited Blackbox Security	190
B.5.1	Overview	190
B.5.2	Critique	192
B.6	Suggestions for Future Research	194
B.6.1	Redundancy and Randomization	195
B.6.2	Combining Encrypted Computation and Obfuscation	196
B.7	Conclusions	200

List of Figures

1-1	Bayanihan in art: <i>Bayanihan</i> . Mural by Carlos Francisco.	20
1-2	Bayanihan in real life: <i>Moving a House</i> . Photograph by Nik Wheeler.	20
2-1	A Taxonomy of Volunteer Computing Systems.	43
3-1	A Bayanihan system with worker and watcher clients.	49
3-2	Volunteering as a worker using Bayanihan.	52
3-3	Watching results and controlling the computation using a watcher applet.	54
3-4	Master-worker runtime layer components and interactions.	56
3-5	Eager scheduling in Bayanihan.	60
3-6	Two approaches to fault-tolerance: (a) Majority Voting, (b) Spot-checking.	61
3-7	A screenshot of the Mandelbrot application with spot-checking enabled.	64
3-8	Using volunteer servers.	65
4-1	The master-worker model of computation.	70
4-2	Brute-force factoring: Speedup plot.	72
4-3	A screenshot of the Mandelbrot application in action.	74
4-4	Mandelbrot application: Speedup plot.	75
4-5	Pseudo-code for a basic genetic algorithm.	76
4-6	The distributed web-crawler experiment.	78
4-7	Fault-tolerance simulator: Speedup relative to the fastest machine.	83
4-8	Fault-tolerance simulator: Efficiency.	84
4-9	Fault-tolerance simulator: Normalized speedup.	84
5-1	A BSP superstep.	88
5-2	BSP pseudo-code for a two-step cyclic shift.	88
5-3	Implementing BSP on top of the Bayanihan master-worker framework.	89
5-4	BSP pseudo-code without code-skipping.	94
5-5	Transformed BSP pseudo-code with code skipping.	94
5-6	Code for <code>bsp_sync()</code> , <code>bsp_step()</code> , <code>bsp_cont()</code>	95
5-7	BSP pseudo-code with code skipping and <code>#define</code> macros.	96
5-8	Bayanihan BSP code for matrix multiplication.	98
5-9	Bayanihan BSP applications: Speedup relative to sequential version.	99
6-1	Eager scheduling work pool with m -first majority voting.	113
6-2	Majority Voting: theoretical error rate for various values of φ and m	114
6-3	Majority voting: theoretical slowdown required to achieve target error rates, given various fault rates.	115
6-4	Spot-checking with blacklisting: theoretical average error rate (ϵ_{sobl}).	117

6-5	Spot-checking without blacklisting: theoretical error rate with long-staying saboteurs (err_{scnb}).	119
6-6	A credibility-enhanced eager scheduling work pool.	122
6-7	m -ahead margin-based voting: theoretical error rate for various values of φ and m .	127
6-8	Majority voting: slowdown vs. maximum final error rate at various values of f and m .	132
6-9	Spot-checking with batch-limited blacklisting: simulated and theoretical error rates vs. s .	133
6-10	Spot-checking without blacklisting: simulated error rates and theoretical upper bound (f/ql) vs. length-of-stay l .	134
6-11	Combining voting and spot-checking with blacklisting: simulated and theoretical error rate vs. s at $q = 0.1$, $N = 10000$, $P = 200$ for $f = 0.5, 0.2, 0.1$ and (a) $m = 2$, (b) $m = 3$. The solid curves represent theoretically predicted error rates using the actual observed values of n , while the dashed curves represent a purely theoretical predictions assuming $n = m(N/P)/(1 - q)$.	135
6-12	Combining voting and spot-checking without blacklisting: maximum simulated error rate vs. l .	135
6-13	Credibility-based voting with spot-checking and batch-limited blacklisting: error rate vs. s at $f = \{0.2, 0.1, 0.05\}$.	136
6-14	Credibility-based voting with spot-checking and batch-limited blacklisting: average slowdown vs. maximum final error rate at various values of f .	137
6-15	Credibility-based voting with spot-checking and batch-limited blacklisting: slowdown vs. maximum final error rate for first and tenth batches.	137
6-16	Credibility-based voting with spot-checking, no blacklisting: error rate vs. length-of-stay l averaged over 10 batches.	138
6-17	Credibility-based voting with spot-checking, no blacklisting: error rate vs. length-of-stay l for: first and tenth batches.	138
6-18	Credibility-based voting with spot-checking, no blacklisting: error rate vs. s at $f = \{0.2, 0.1, 0.05\}$.	139
6-19	Credibility-based voting with spot-checking, no blacklisting: average slowdown vs. maximum final error rate at various values of ϑ and f .	140
6-20	Credibility-based voting with spot-checking, no blacklisting: slowdown vs. maximum final error rate for first batch and tenth batches.	140
6-21	Using credibility-based voting for spot-checking, no blacklisting: error rate vs. s at $f = 0.2, 0.1, 0.05$ for various thresholds ϑ .	141
6-22	Using credibility-based voting for spot-checking, no blacklisting: average slowdown vs. maximum final error rate at various values of ϑ and f .	142
6-23	Using credibility-based voting for spot-checking, no blacklisting: slowdown vs. maximum final error rate for first and tenth batches.	142
B-1	Programs can be attacked in hostile environments.	176
B-2	Computing with encrypted functions (CEF).	178
B-3	Computing with encrypted data (CED).	179
B-4	Examples of trivial opaque constructs.	186
B-5	CED using obfuscation as an encryption scheme.	197
B-6	Computing $a=b+c$; $b=b+1$; $c=a*b$; using obfuscated data.	198

List of Tables

3.1	Results from preliminary fault-tolerance experiments.	63
3.2	Running the Mandelbrot application with a volunteer server.	65
4.1	Brute-force factoring: Speedup with 8 workers.	72
4.2	Brute-force factoring: comparing C and Java speeds.	72
4.3	Mandelbrot application: comparing C and Java.	75
4.4	Genetic Algorithm application: Running times (in ms) and speedups.	77
4.5	Distributed web-crawler: Timing measurements	79
5.1	Basic Bayesian BSP methods	92
5.2	Basic BSPMainWork methods	96

Chapter 1

Introduction

1.1 Toward Electronic Bayanihan

In the countryside villages of old in the Philippines, when a family moved to a new place, they literally moved their whole house with them. First, they would place long bamboo poles under their house (which was usually elevated by stilts). Once this framework was in place, their friends and neighbors would then gather in numbers under it, and carry the house on their shoulders to its new location. This tradition, called *bayanihan* (pronounced “buy-uh-nee-hun”), has been a favorite subject of artists (see Fig. 1-1) and is still practiced today in some rural areas (see Fig. 1-2). More than a way to move houses, *bayanihan* has come to be a dramatic symbol of the power of cooperation. Like its counterparts in other cultures, including *gotong royong* in Indonesia and Malaysia [60], *harambee* in Africa [103], and *barn raising* in the United States [161], *bayanihan* reminds us that seemingly impossible feats can become possible through the concerted effort of many people with a common goal and a sense of unity.

This thesis seeks to bring the *bayanihan* spirit into the world of computing by presenting and developing the idea of *volunteer computing*, a newly emerging form of computing that makes it easy for people on the Internet to pool their existing computer resources, and work together to solve large computational problems that no one can solve alone.

Volunteer computing works by enabling people to share their computers’ idle processing power through easy-to-use and highly accessible software that does not require expert help or technical knowledge. Like its predecessors, *networks-of-workstations* (NOWs) [7] and *metacomputing* [102], volunteer computing is a form *parallel computing*. It allows a large computational problem to be solved much faster than possible on a single computer by making use of many networked computers “in parallel” working at the same time on different parts of the same problem. Unlike other forms of parallel computing, however, volunteer computing does not require system administrators to install complex software or create special accounts on users’ machines. A user wanting to volunteer his or her machine would be able to do so by simply downloading and installing a screensaver or, even more easily, by visiting a web page.

By making it easy for people to volunteer their computers by themselves, volunteer computing creates many new possibilities. Because it allows even casual users with no



Figure 1-1: Bayanihan in art: *Bayanihan*. Mural by Carlos Francisco, 1962, courtesy of Unilab Philippines, Inc.



Figure 1-2: Bayanihan in real life: *Moving a House*. Photograph by Nik Wheeler, courtesy of corbis.com. Another photograph of real-life bayanihan can be seen in National Geographic magazine [101].

technical knowledge to volunteer their machines, volunteer computing makes it possible to build parallel computing networks much larger and much more quickly than possible before. Potentially, such networks can involve thousands, even millions of computers distributed around the world, and can achieve performance levels far beyond that of any current supercomputer. Furthermore, since volunteer computing does not require new hardware to be purchased, it can also provide affordable supercomputing capabilities to financially constrained organizations such as small universities, as well as companies, institutions, and universities in developing countries. In fact, since volunteer computing makes it easy for organizations to pool their resources, it opens new possibilities for collaborative research between institutions around the world.

Recently, highly successful volunteer computing projects have given us a glimpse of the potentials of volunteer computing. The first of these to gain widespread publicity was distributed.net [40], a cooperative group of programmers and thousands of volunteers around the world who, in October, 1997, solved the \$10,000 RSA RC-56 decryption challenge after trying 34 quadrillion different keys [40]. At the time the search ended, distributed.net had over 4,000 active volunteer teams (having anywhere from one to hundreds of computers each), and was searching through keys at a rate of over 7 billion per second – equivalent to the combined power of about 26,000 high-end PCs at that time. Over the span of the 250 days it took to crack the code, they received results from as many as 500,000 different computers (unique network addresses) around the world [41]. Even more impressive is the SETI@home project for searching massive amounts of radio telescope data for signs of extraterrestrial intelligence [137]. SETI@home was started in May 1999, and today has had 2.7 million users registered so far, with currently over 500,000 active users (users who have submitted a result in the last 2 weeks) providing a total of over 15 TeraFLOPs of computing power [85]. It is currently faster than the fastest single supercomputer in the world, IBM's ASCI White, which runs at around 12 TeraFLOPs, and is considerably less expensive, having cost SETI only \$500,000 so far, compared to ASCI White's \$110 million price tag [138].

1.2 Thesis Goals

Although already very impressive, these examples are just the tip of the iceberg. Volunteer computing has a lot more to offer if we can go beyond *application-specific* systems such as these, and develop *general-purpose* technology to implement a wider range of applications and address the many new technical challenges that volunteer computing brings. This thesis aims to paint a bigger picture of volunteer computing's potentials, and show how these can be achieved by developing ways of addressing key concerns and technical challenges in volunteer computing.

Specifically, in this thesis, we will demonstrate how these potentials extend beyond what existing systems have shown by showing how volunteer computing:

1. is a whole new form of computing, not limited to public and unpaid *true* volunteer systems such as SETI@home and distributed.net, but including private as well as commercial networks as well,

2. can be made easier for volunteers and programmers alike, thus encouraging more volunteers to join, as well as enabling programmers to write more useful applications,
3. need not be limited to esoteric scientific and mathematical problems but can be applied to a wide range of useful applications,
4. need not be limited to “embarrassingly parallel” or master-worker style applications, as current belief and practice may suggest, but can be used for other coarse-grain applications that may use more complex communication patterns, and
5. can be made reliable even in the presence of malicious volunteers without overly sacrificing performance.

In doing so, we do not aim to compete with other volunteer computing systems such as SETI@home and distributed.net, but to *complement* them by encouraging their further use and by presenting new ideas, techniques, and tools that developers of these systems can apply to their own systems to help achieve their full potentials.

1.3 Thesis Overview

The rest of this thesis is roughly divided into chapters according to these major goals. In this section, we give an overview of each chapter.

Potentials and Challenges

Chapter 2 presents the motivations for the rest of this thesis. In this chapter, we identify the further potentials of volunteer computing, as well as the new research challenges involved in realizing these potentials.

In the first half of this chapter, we present the background and motivation for volunteer computing, and define it as a new form of parallel computing that focuses on maximizing the ease with which people can volunteer their machines. We then show how volunteer computing need not be limited to “true volunteer” networks like SETI@home and distributed.net, wherein the volunteers come from the general public and are unpaid, but can take other forms as well, including private and collaborative “forced volunteer” networks, barter trade systems, commercial market-based systems, and NOIAs (networks of information appliances). All these are made possible by the key idea of making it easy for people to share their computers’ idle time.

In the second half of this chapter, we identify and discuss the important new research issues that need to be addressed in building volunteer computing systems in general. We begin by discussing the general technical issues of *accessibility*, *applicability*, and *reliability*. Accessibility is concerned with making volunteer computing as easy, open, and inviting to as many volunteers as possible, and involves issues in ease-of-use, platform-independence, volunteer-side security, and user-interface design. Applicability is concerned about making volunteer computing useful in the “real world”, and involves issues

in programmability, adaptive parallelism, performance, and scalability. Reliability is concerned about making volunteer computing work in the presence of faults and malicious volunteers, and involves finding ways to protect against attacks from these volunteers such as sabotage, cheating, and espionage. We also discuss *economic issues* which we do not focus on in this thesis, but will play an important part in volunteer computing's future in commercial systems.

Tying together the two halves of this chapter, we present a taxonomy that classifies these forms according to *autonomy* (whether or not volunteers can join and leave of their own free will), *anonymity* (whether or not the volunteers are known and trusted by the administrators), and *altruism* (whether or not volunteers expect to be compensated for volunteering). This taxonomy allows us to identify the key research issues that call for special attention in each form of volunteer computing.

While many others have also recognized the potentials and challenges of volunteer-based parallel computing systems albeit under different names (e.g., [6, 12, 46, 56, 111] and others), our own approach in this chapter is to present volunteer computing as a distinct new form of computing, and to present a high-level map of the domain of this new form of computing, as well as of the research terrain ahead for those who wish to explore it.

The Bayanihan Web-based Volunteer Computing System

In Chap. 3, we present the *Bayanihan* volunteer computing system, which simultaneously addresses all these issues by making it easier not only for users to volunteer their machines, but also for programmers to write volunteer computing software and for researchers to develop, implement, and experiment with different ways to address the various technical challenges in volunteer computing.

Bayanihan uses Java applets [59] to implement the idea of *web-based* volunteer computing, which enables users from anywhere on the Internet to volunteer their computers by simply visiting a web page. By allowing users to volunteer without even having to install software on their machines, and by taking advantage of Java's platform-independence and security, Bayanihan not only makes the process of volunteering easier and faster than in other systems, but also allows us to accommodate a wider range of volunteers, including casual users, users wary of security risks, and users of machines with operating systems that do not allow them to install software. We demonstrate the benefits of web-based volunteer computing by describing how Bayanihan has allowed us to set up parallel computing networks in a matter of seconds or minutes (compared to the hours, days, or weeks that it would take to install traditional NOW and metacomputing systems), and to use machines on which application-based volunteer computing software would have been difficult or impossible to install.

Bayanihan also uses Java to make it easier for programmers to write software. By taking advantage of Java's object-oriented features, Bayanihan provides a flexible general-purpose software framework with multiple levels of extensibility. Through this framework, Bayanihan not only allows programmers to write a variety of applications, but also allows researchers to implement and experiment with different mechanisms for addressing technical issues in volunteer computing. We demonstrate the use of Bayanihan for applications by describing a general-purpose runtime system and application programming

interfaces (APIs) for *master-worker* applications, which we use to implement a number of applications presented in Chap. 4. Finally, we show how Bayanihan's flexibility allows programmers to do research on the principles of volunteer computing itself by describing how we have used Bayanihan's highly modular architecture to develop and experiment with mechanisms for achieving adaptive parallelism, reliability, and scalability for master-worker applications in general.

Master-Worker Style Applications

In Chap. 4, we present a number of *master-worker* style applications. These applications are ideal for volunteer computing systems because they are *coarse-grain* and *embarrassingly parallel*, and thus work well even with slow Internet links, and can easily be made adaptively parallel. Here, we show that in addition to the esoteric mathematical and scientific problems that volunteer computing systems are most well-known for solving today, many other more practical applications fall under this category and can easily be implemented on volunteer computing systems as well.

Using Bayanihan's master-worker runtime system and APIs, we explore a wide range of master-worker style applications, including brute-force search, image rendering, genetic algorithms, communicationally parallel applications, parametric analysis, and Monte Carlo simulations. For each of these classes of applications, we present general-purpose mechanisms, APIs, and application frameworks that we have developed for implementing applications in that class, and present specific example applications that demonstrate how we have achieved useful speedups for these applications.

Among the most promising of these applications at present are those involving parametric analysis and Monte Carlo simulations, wherein the goal is to empirically observe the behavior of hard-to-analyze systems by running a sequential simulation many times under different, independent, combinations of parameters. Parallelizing these applications by running the different parameter combinations on different machines at the same time offers programmers a straightforward way to save time and thus experience the benefits of volunteer computing. As a first-hand example, we show how we ourselves have used Bayanihan to implement a parallel Monte Carlo simulator for studying the fault-tolerance mechanisms we propose in Chap. 6, and show how the speedups we got from running this simulator on up to 50 volunteer machines in parallel made our research possible.

BSP as a Programming Model for Volunteer Computing

In Chap. 5, we take applicability even further by showing how volunteer computing can be used for an even wider range of applications than current popular belief and practice would suggest. In particular, we show that volunteer computing need not be limited to master-worker style and embarrassingly parallel applications as it so far has been, but can also be used for *message-passing* style applications as long as they are coarse-grain enough.

We do this by presenting a new programming interface based on the BSP (*bulk synchronous parallel*) model. Like the traditional and popular message-passing model, and unlike the master-worker model, BSP allows worker processes to exchange data with each other, and thus supports a much wider range of applications. Unlike the traditional

message-passing model, however, the BSP model does not let messages move immediately but only moves them “in bulk” at specific synchronization points in time. This makes BSP easier to implement on a variety of parallel computing architectures – including master-worker style ones like Bayanihan and most other volunteer computing systems today.

In this chapter, we implement a Java-based API for writing BSP-style programs that can run on Bayanihan’s master-worker runtime layer. This API allows us to write more communicationally complex applications while still taking advantage of the adaptive parallelism, reliability, and scalability mechanisms we have developed for master-worker applications. It also makes writing master-worker style applications easier by giving programmers an SPMD (single-program multiple-data) style interface similar to that of most message-passing systems today.

We demonstrate the use of this API by implementing a number of example applications. These show how the Bayanihan BSP API now allows programmers to write communicationally complex applications such as Jacobi iteration, which were previously difficult or impossible to write for volunteer computing systems. At present, the performance and scalability of our implementation is limited for some applications due to limitations in the underlying runtime layer. However, these are expected to improve with future work on mechanisms for periodic checkpointing and smart rollback, lazy migration and itinerant processes, dataflow-style BSP, and possibly peer-to-peer communication. Meanwhile, the BSP programming interface we have developed represents a significant improvement in programmability and flexibility over current programming interfaces for volunteer-based systems, and enables programmers to start porting and writing an even wider variety of applications for volunteer computing systems than before.

Reliability in the Presence of Malicious Volunteers

In Chap. 6, we address the problem of assuring the correctness and security of a computation in the presence of *malicious* volunteers. This is a new problem unique to volunteer computing, and not yet very well-studied. Thus, in this chapter, we present some of our most unique and novel results.

This chapter is divided into three main parts. In the first part, we begin with an overview of general approaches to this problem. First, we show how reliability can easily be achieved by simply choosing to do applications that are either *naturally fault-tolerant* or *verifiable*. Then, we give a high-level discussion on how *authentication*, *encryption*, and *obfuscation* techniques can be used (whether or not our applications are naturally fault-tolerant or verifiable) to reduce error rates by making it difficult for saboteurs to produce bad results to begin with. Finally, we consider the use of *redundancy* and *randomization* in cases where saboteurs are able to produce bad results despite these mechanisms.

In the second part, we focus on techniques based on redundancy and randomization. We begin by analyzing the traditional technique of *voting*, and show how it reduces the original error rate exponentially with redundancy (i.e., the number of times we repeat each work object), and is thus useful in systems with low original error rates. For systems with high original error rates, we propose the use of new mechanisms called *spot-checking* and *blacklisting*, which we show reduces the error rate by a factor that is linear with the amount of work to be done, while only increasing the total running time by a relatively small

fraction of the original time, instead of by several multiples as in voting. By combining voting and spot-checking, we show how we can exponentially shrink the already linearly reduced error rate, and thus achieve error rates that are orders-of-magnitude smaller than those offered by voting alone.

Finally, in the last part of this chapter, we generalize our results by presenting a new technique called *credibility-based fault-tolerance*. In this technique, we estimate the *credibility* of results and workers as the probability of their being correct given the results of using voting, spot-checking, and other techniques. By then using these estimates to determine whether a piece of work needs to be repeated or is credible enough to be accepted, we are not only able to attain mathematically guaranteeable levels of correctness, but are also able to do so with much smaller slowdown than possible with traditional voting techniques.

Using a parallel Monte Carlo simulator running on Bayanihan itself, we demonstrate the effectivity of these techniques, and present surprising and encouraging results. In one case, where we assume that saboteurs are Bernoulli processes and that each volunteer does about 50 pieces of work in a batch, we show that even with as many as 20% of the volunteers being saboteurs, credibility-based fault-tolerance can limit the average error rate to less than 1 in 10^6 with only a 2.5 times slowdown compared to a system with no fault-tolerance. This is almost 10^5 times smaller than the error rate achievable with traditional majority voting for the same slowdown. Furthermore, we can expect even smaller error rates for larger jobs where workers do more pieces of work in a batch and therefore have a higher chance of getting caught by spot-checking.

Related Work, Future Work, and Conclusion

Throughout this thesis, we cite related work as we go along. In addition, at the end of some chapters, we present a section placing the results of the section in the context of related work, and discussing possible future work. Finally, in Chap. 7, we conclude this thesis by summarizing our results, and suggesting possible directions for future research.

Appendices

In addition to the main text, we have two appendices. Appendix A contains sample source code for some applications described in Chap. 3 and Chap. 4. Appendix B is a survey and critique paper, previously written separately [134], that studies the topic of protecting mobile agents from malicious hosts. Although this paper is mostly concerned about applications in the field of mobile agents in general, such as those used in e-commerce systems, the ideas presented here are directly applicable to volunteer computing systems where the code run by the volunteers can be seen as mobile agents that can be attacked by a malicious volunteer. In this paper, we survey techniques involving *encrypted computation* as well as *obfuscation* (i.e., instruction scrambling). We also look into the idea of *time-limited blackbox security* [71], which is very similar to our idea of *periodic obfuscation*. In the end, we propose the use of some form of time-limited blackbox security or periodic obfuscation using *parameterizable obfuscation* techniques, and suggest some ways that this can be implemented.

1.4 Contributions

In summary, the major contributions of this thesis include:

- **General-purpose volunteer computing as a new form of computing.** We were among the first to identify the potential for a new form of computing distinguished from traditional NOWs and metacomputing specifically by its emphasis on maximizing ease-of-use and accessibility for volunteers. We coined the term “volunteer computing” in 1996 [131] to refer to it, and presented one of the first papers mapping out the numerous possibilities that it creates [131, 132]. We were also among the first to identify and promote the idea of *web-based* volunteer computing as a way of achieving even more accessibility and ease-of-use than the more common application-based systems.
- **The Bayanihan web-based volunteer computing system.** Bayanihan was one of the first *general-purpose* web-based volunteer computing systems using Java. It was also among the first to use a remote object system (i.e., HORB) to simplify programming, and the first to focus on maximizing flexibility and extensibility through object-oriented design. This flexibility has allowed us not only to write a wide variety of applications, but also to explore a wide range of technical issues and develop several new ideas and mechanisms for addressing them.
- **The Bayanihan BSP programming interface.** We were the first to identify the applicability of the BSP programming interface to volunteer computing systems in particular, and the first to implement it in Java. In addition, since our BSP implementation requires only an underlying master-worker runtime system, it should be implementable in other volunteer computing systems as well.
- **Approaches to the problem of malicious volunteers.** This thesis provides the most comprehensive treatment to date of the new problem of dealing with malicious volunteers. Here, we cover mechanisms ranging from encrypted computation and obfuscation, to naturally fault-tolerant and verifiable applications, to voting and spot-checking.
- **Credibility-based fault-tolerance.** This idea, one of our most unique and novel ideas, provides system designers with a simple, but highly generalizable framework for achieving mathematically guaranteeable error levels with minimal redundancy and slowdown. It is not limited by the specific assumptions, mechanisms, and credibility metrics that we present in Chap. 6, but can be used under other assumptions and with other mechanisms as long as the appropriate credibility estimates can be derived.

In addition to these major contributions, some notable minor contributions include:

- **NOIAs.** We were the first to propose the idea of the NOIAs (networks of information appliances), and coined the term NOIA.

- **Interactive parallel computing.** Bayanihan was one of the first volunteer computing systems to provide watcher applets that not only allow users to view the results of a computation, but also allow them to control the computation even before a parallel batch is finished.
- **Communicationally parallel applications.** We were one of the first to explore the use of volunteer computing for communicationally parallel applications through our distributed web-crawler application [133].
- **Domain-specific application frameworks.** In addition to providing general-purpose APIs for master-worker applications, we also provide more specific frameworks for particular classes of applications. In particular, we provide a framework that makes it easy to write parallel Monte Carlo simulations and parametric analysis applications (see Sect. 4.6) by providing transparently parallel methods for doing multiple Monte Carlo runs in parallel, and by providing efficient and easy-to-use classes for gathering statistics.
- **Pre-compiler free portable checkpointing.** As part of the Bayanihan BSP API, we developed a way to do checkpointing without the need for precompilers or any changes to the Java virtual machine. The general programming constructs we have developed can be used not only for writing BSP work objects for volunteer computing systems, but also for writing other Java code that may require checkpointing. They may even be used in the future in non-Java code as well to provide portable checkpointing.

In the end, all these just begin to explore the vast new field of volunteer computing research. By presenting them here, it is our hope that they will serve as a starting point and roadmap for further research into volunteer computing.

Chapter 2

Potentials and Challenges

In this chapter, we identify the potential benefits and applications of volunteer computing, as well as the various problems and challenges that need to be overcome in turning these potentials into reality. We begin by comparing volunteer computing with more traditional forms of parallel computing, and proceed to identify the various potential forms that volunteer computing systems can take and their potential benefits. We then identify the various research issues involved in implementing volunteer computing systems, as well as suggest possible ways to approach these problems.

2.1 What is Volunteer Computing?

2.1.1 Metacomputing

Volunteer computing is a variation on the idea of *metacomputing* – using a network of many separate computers as if they were one large parallel machine, or *metacomputer*.

Metacomputing systems today primarily take the form of *networks of workstations*, or NOWs [7], that allow people to pool together existing and mostly idle workstations in their own local or institution-wide networks and use them to do parallel processing without having to purchase an expensive supercomputer [48]. Global-scale NOWs, employing computers geographically distributed around the world and communicating through the Internet, have also been used with great success to solve large parallel problems as far back as the early 1990's [143, 91], and until today [40, 56, 158, 137, 117].

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 some cases the systems were not even fully automatic. Participants had to manually request jobs by email or `ftp`, load them into their computers, execute them, and again manually submit the results [91].

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.

PVM and MPI

Fortunately, this situation has been notably improved since the development of general-purpose and cross-platform parallel processing systems such as Parallel Virtual Machine (PVM) [54] and Message Passing Interface (MPI) [63] in the early 90's. 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 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 file system).
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, but also requires a lot of trust between the involved machines, and consequently, a lot of prior human communication between the administrators of different machines. All this means that it can take days, weeks, or even months to set up a large NOW – that is, if it can be set up at all. 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.

Other Metacomputing Systems and Grid Systems

More recent metacomputing projects have started going beyond the bounds of individual institutions, and have started addressing the problems of security, reliability, and scalability in nation-wide and global networks. These include Legion [62], Globus [50], NetSolve [27], and Ninf [135]. One of the biggest projects so far is the Partnerships for Advanced Computational Infrastructure (PACI) program, which in 1998 received a five-year grant of \$340 million from the National Science Foundation to develop a “National Technology Grid” for the United States [141]. Since then, a lot of metacomputing research has been focused on creating *grid systems*, which aim provide seamless access to computing resources around the country and the world such that users who need processing power can just

“plug-in” to the grid, just as one can plug-in to a wall socket to get electric power from the national power grid [51].

2.1.2 Volunteer Computing

Volunteer computing is a form of metacomputing that focuses on maximizing the ease with which people can have their machines be part of a metacomputer.

While other forms of metacomputing such as grid computing seek to make *using* compute servers on the network easy, *setting up* these compute servers is currently not as easy. Currently, most of these projects are focusing on making supercomputers and computers in research labs available for use by others. Since these machines are already in the hands of experts, making them easy to install is not a top priority. Thus, like PVM and MPI, metacomputing systems still have complex setup requirements that prevent ordinary users from participating in these systems.

The idea behind volunteer computing is to eliminate these setup requirements, and make it so easy to join a computing network that anyone, even a casual computer user without any technical knowledge, can do it. By allowing casual users to contribute their computers' idle processing power in this way, volunteer computing makes it possible to harness the power of thousands, or even millions, of computers in very little time. At the same time, it also makes metacomputing and all its benefits more accessible to the common Internet user.

Application-based Volunteer Computing Systems

As mentioned in Sect. 1.1, volunteer computing systems, such as SETI@home, distributed.net, and a rapidly increasing number of other projects (see [117] for a good listing of different volunteer computing-type projects), are already demonstrating this great potential of volunteer computing. A large part of this success can be attributed to the relative ease with which users are able to join their systems. To volunteer, a user only needs to:

1. Go to the project's web site and download an executable file for his or her own machine architecture and operating system. Typically, versions of the application software are provided for major architectures and operating systems such as Windows, Macintosh, and UNIX.
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 project server, requesting work, and sending back results. All the user has to do is leave the software running in the background on these machines. (Most of these systems also provide a screen-saver mode that would let the worker program run when the user is idle.) Unlike PVM, these systems do not require administrators to personally install daemons on the volunteer machines, or contact the volunteer users to acquire remote shell access. In fact, for popular operating systems such as Windows, it can take as little as 10 minutes or less to install the the volunteer worker program on one's machine and have it start computing already.

Web-based and Java-based Volunteer Computing

Recent advances in World Wide Web technology offer to make volunteer computing even easier. Key among these is Sun Microsystems' Java technology [59], which allows people to write platform-independent programs called *applets*, that can be placed on a web page together with ordinary text and images. When a user views a web page, any applets on the page are automatically downloaded by the user's browser and executed on the user's machine, where they can provide dynamic and interactive features to the web page not possible with ordinary text and images. Java applets can be used to enhance web pages with such features as animations, interactive demos, and user-friendly input forms. Java's built-in networking features also allow applets to be used for interactive multi-user applications that allow people to interact with each other and share ideas through the Web, such as games, "chat rooms", and "shared whiteboards". Java's versatility has made it one of the most popular and fastest-growing software technologies today. All major web browsers now support Java, and thousands of individuals, companies, and schools are now using Java for a wide variety of personal, commercial, and educational purposes. (The JARS Java repository web site [76] contains a good collection of these applets.)

Java's applet technology, coupled with its popularity and ubiquity, offers us an opportunity to make volunteer computing both easier and more accessible than in systems like distributed.net. With Java, programmers who want to set up a metacomputing network can write their software as Java applets and post them on a web page. Then, people who want to volunteer need only view the web page with their browsers, and wait. The applets are downloaded automatically and run without any human intervention. The volunteers do not even have to install any software themselves. In this way, adding a computer to a metacomputing network – something that used to require expert administrators and take days of setup time in conventional NOWs and metacomputers – can be done by the average user with literally a single mouse click. As we shall see in Chap. 3, this brings with it a lot of advantages and creates many new possibilities.

2.2 Potential Forms of Volunteer Computing

By making it easy for people to volunteer their machines, volunteer computing not only allows us to form parallel computing networks more quickly and with more processors than possible before with traditional metacomputing systems, it also makes it possible for people who have not considered parallel processing at all before – due to lack of expertise, time, or resources – to start considering it. In this way, volunteer computing, like the tradition of *bayanihan*, creates many new possibilities where there were none before.

In this section, we identify some of these possibilities, and show that volunteer computing is not limited to *true volunteer computing networks* like SETI@home and distributed.net, but can take many other forms as well.

2.2.1 True Volunteer Computing

Systems such as SETI@home and distributed.net may be called *true volunteer computing systems*. That is, their participants are volunteers in the true sense of the word in that they:

(1) come and leave of their own free will, (2) are unknown to the administrators before they volunteer – i.e., they can come from anywhere, not just from the administrators’ domains, and (3) do not expect substantial compensation.

By allowing anyone on the Internet to join as they please, true volunteer computing systems have the potential for pooling together the computing power of the millions of computers on the Internet. Today, it is estimated that there are about 300 million computers on the Internet [89]. If we can somehow get even a small fraction of these to work together, we can have a world-wide supercomputer more powerful than any single supercomputer. The experience of distributed.net and SETI@home already attest to this fact.

One question in true volunteer computing today is what kinds of applications can attract a large number of volunteers. To date, the most popular and successful applications have been what we might call “cool” challenges – challenging computational problems that people join just for fun and the pride of being part of a global effort. So far, these have mostly involved mathematical problems such as searching for prime numbers (e.g., GIMPS), and cryptographic challenges (e.g., the 1994 factoring effort, and distributed.net). Other possibly more interesting cool challenges for the future include: chess (imagine a “Kasparov vs. the World match”, where Kasparov plays against a world-wide volunteer computing network¹), international computer olympics (countries form their own volunteer computing networks, which then play against each other in competitions, such as chess, where the winner is determined by a combination of algorithm design and the number of loyal citizens willing to participate), and distributed graphics applications (“Join in the making of Toy Story III!”).

Scientific problems in areas that appeal to the public can also attract volunteers. The naturally interesting topic of SETI@home project – i.e., finding signs of extra-terrestrial life – accounts for SETI@home’s great popularity. More recently, other volunteer computing projects have started with similarly interesting and popular topics such as finding cures for AIDS [49] and cancer [114, 149], fighting influenza [119], and others.

Ultimately, true volunteer computing is most useful for *worthy causes* – practical, useful, and relevant problems that serve the common good. Worthy causes can be local or global. Local worthy causes are those relevant to a particular city, country, or geographic area. These can include traffic simulation studies for congested mega-cities, and local weather and seismological simulation research. Global worthy causes have a wider scope, and include those that can help humankind as a whole, such as the medical applications mentioned above, as well as others such as simulating the effects of environmental policies on global warming and other environmental effects.

With true volunteer computing, an organization with a worthy cause can set up a volunteer computing system, and invite concerned Internet users to donate their idle cycles by downloading a screensaver or visiting a web page, and leaving these programs running in the background while they work. Volunteers contribute processing power towards the worthy cause willingly, and according to their own capacity. In this way, the end result of true volunteer computing is electronic bayanihan – people working together for the good of their community and even the world.

¹Interestingly, a “Kasparov vs. the World” match has already been played on the web with *human* opponents [95]. In this game, anyone on the Internet could vote for a move through web page. The move with the highest number of votes was then chosen.

2.2.2 Private Volunteer Computing

Although true volunteer computing may be the most visible and most noble form of volunteer computing, it is not the only one. Volunteer computing can take other, less lofty but more practical, forms as well.

At the lowest level, volunteer computing principles can be used in *private volunteer computing networks* within organizations such as companies, universities, and laboratories to provide inexpensive supercomputing capabilities. Many companies and universities today have internal networks (intranets) with many PCs and workstations that remain idle most of the time – not only during off-hours, but also during office hours when they are mostly used for non-computational tasks such as word processing. Volunteer computing can be used to pool together the computing power of these existing and under-utilized resources to attain supercomputing power that would otherwise be unaffordable. This not only makes it possible for research organizations to satisfy their computational needs inexpensively, but also creates an opportunity for other organizations to consider the use of computational solutions and tools where they have not done so before.

For example, companies with heavy computational needs can use volunteer computing systems as an inexpensive alternative or supplement to supercomputers. Some potential applications include: physical simulations and modeling (e.g., airflow simulation in aircraft companies, crash simulation in car companies, structural analysis in construction firms, chemical modeling in chemical, biomedical, and pharmaceutical labs, etc.), intensive data analysis and data mining (e.g., for biomedical labs involved in genome research, for financial analysts and consulting firms, etc.), and high-quality 3D graphics and animations (e.g., for media and advertising companies).

At the same time, companies that have hitherto deemed computational solutions or tools too expensive can start benefiting from them. For example, manufacturing companies can benefit from computational analysis and process simulations of their complex pipelines. These can be used to expose bottlenecks and weak points in processes, predict the effects of changes, and lead to finding ways to make them more efficient and cost-effective. Marketing departments of companies can do data mining on sales and consumer data, finding patterns that can help them formulate strategies in production and advertising. In general, by providing an affordable way to do computation-intensive analysis, volunteer computing can let companies enhance the intuitive analysis and ad hoc methods currently used in planning and decision making, and make them more informed and systematic.

Like companies, universities and research labs can use volunteer computing to turn their existing networks of workstations into virtual supercomputers that they can use for their research. This is especially useful in financially constrained institutions, such as universities in developing countries that cannot afford to buy supercomputers and have thus far been unable to even consider doing computation-intensive research. Even in non-research-oriented universities that do not have a true need for supercomputing, virtual supercomputers built through volunteer computing can be used to teach supercomputing techniques, giving students skills they can use in graduate school or industry.

Most of the applications mentioned here can be implemented with conventional NOW or metacomputing software, and are in fact already being implemented in a growing num-

ber of organizations today. The advantage of volunteer computing over these is that: (1) it makes implementing applications dramatically easier for users and administrators alike, and (2) by doing so, it makes the technology accessible to many more organizations, including those that do not have the time and expertise to use conventional NOWs. For example, if a company decides to use its machines for parallel processing, system administrators do not need to spend time manually installing software on all the company machines anymore. With a web-based system, they can simply post the appropriate Java applets on the company web site, and then tell their employees to point their browsers to a certain web page. The employees do so, and leave their machines running before they go home at night. In this way, a company-wide volunteer computing network can be set up literally overnight, instead of taking days or weeks for installing software and educating users as it would in conventional systems.

2.2.3 Collaborative Volunteer Computing

The same mechanisms that make volunteer computing *within* individual organizations work can be used to make volunteer computing *between* organizations as well. By volunteering their computing resources to each other, or to a common pool, organizations around the world can share their computing resources, making new forms of world-wide collaborative research possible.

Much research today is going into developing technologies for *collaboratories* – world-wide virtual laboratories where researchers in different labs around the world can interact freely with each other and share data and ideas, as if they were all in one big laboratory [124, 159]. Technologies currently under development include videoconferencing, shared whiteboards, remote instrument control, and shared databases. Volunteer computing can enhance collaboratories by allowing researchers to share not only data and ideas, but computing power as well.

Even organizations that do not collaborate very closely can use volunteer computing mechanisms to *barter trade* their resources, depending on their need. This has interesting geographical possibilities. For example, a university in the United States can allow a lab in Japan to use 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). In this way, we get a win-win situation where both labs get twice the processing power during the hours they need it.²

Taken to an extreme, pooling and barter trade of computing resources between organizations can lead to the formation of *cycle pools* – massive pools of processing power to which people can contribute idle cycles, and from which people can tap needed processing cycles. This can be used for organizations' individual gains, as in the example above, but can also be a venue for larger organizations to help smaller ones and contribute toward a better social balance of resources. For example, a large university with more idle computer power than it needs can donate its idle cycles to the pool, and allow smaller universities and schools to make use of them for their own teaching and research needs.

Note that these possibilities are not unlike what grid systems seek to realize. Thus, in a way, volunteer computing can be seen as an enabling technology for grid-based systems

²A similar idea is proposed in [46].

as well.

2.2.4 Commercial Volunteer Computing

By giving people the ability to trade computing resources depending on their needs, volunteer computing effectively turns processing power into a commodity. With appropriate and reliable mechanisms for electronic currency, accounting, and brokering, *market systems* become possible, allowing people and groups to buy, sell, and trade computing power. Companies needing extra processing power for simulations, for example, can contact a broker machine to purchase processing power. The broker would then get the desired processing power from a cycle pool formed by companies selling their extra idle cycles. Even individuals may be allowed to buy and sell computing power.

Commercial applications of volunteer computing also include *contract-based systems*, where computing power is used as payment for goods and services received by users. 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. Such terms can be represented as a two-sided contract that both sides should find acceptable. For example, while it is actually possible in today's browsers to hide Java applets inside web pages so that they run without users knowing about them, most users will consider this ethically unacceptable, and possibly even a form of theft (of computing power). Sites that require users to volunteer must say so clearly, and allow users to back-out if they do not agree to the terms.

If appropriate economic models and mechanisms are developed, commercial volunteer computing systems can allow people not only to save money as they can in organization-based systems, but to make money as well. Such an incentive can attract a lot of attention, participation, and support from industry for volunteer computing. With commercial volunteer computing systems, what happened to the Internet may happen to volunteer computing as well – it will start out as something primarily used in academia, and then when it becomes mature, it will be adopted by industry, which will profit immensely from it.

In fact, in the past year, this has already started to happen. Several new startup companies, including Entropia [45], Parabon [113], Popular Power [118], Process Tree [121], and United Devices [148], have taken the technology and ideas from SETI@home and distributed.net and are setting up market-based systems, hoping to make profits by acting as brokers between people that need computing power, and people that are willing to share their idle computing power for pay.

2.2.5 NOIAs

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 [53, 107]. In the United States today, companies such as WebTV [156] are starting to develop and sell information appliances in the form of “set-top boxes”, while many cable companies already support high-speed cable modems that use the same cable that carries the TV signals to connect users to the Internet up to 50 times faster than a telephone mo-

dem can. 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 24-hour Internet access as commonplace as cable TV.

This brings up an interesting idea: what if we use volunteer computing to allow users of these information appliances to volunteer their appliances' idle cycles? These nodes can perform computation, for example, when the user is not using the information appliance, or while the user is reading a web page. Such networks-of-information-appliances, or NOIAs, as we can call them (appropriately, the acronym also means "mind" in Greek [*νοια*], and evokes an image of a brain-like massively parallel network of millions of small processors around the world), have the potential for being the most powerful supercomputers in the world since: (1) they can have tens of millions of processors (i.e., potentially as many as the number of people with cable TV), and (2) although they are only active when the user is idle, we can expect the user to be idle most of the time.

NOIAs can be contract-based systems. Cable companies or Internet service providers (ISPs) can sign 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. This may be acceptable to many clients since most people today are used to leaving their VCR on 24 hours a day anyway. In some cases, however, clients may object to having someone else use their appliances when they are not using it, so a reasonable compromise may be to always give clients the option of not participating, but give benefits such as discounts or premium services to clients who do. In addition, volunteering clients may be allowed to indicate which kinds of computations they do and do not want to be involved in. For example, a client may allow her appliance to be used for biomedical research directed at lung cancer, but not for tobacco companies doing data mining to improve their advertising strategy.

From programmers' and administrators' points-of-view, NOIAs also have several advantages over other kinds of volunteer networks. Hardware-based cryptographic devices can make NOIAs secure against malicious volunteers attempting to perform *sabotage* (see Sect. 2.3.3) by preventing such saboteurs from forging messages or modifying the applet code that they are given to execute. NOIAs are also significantly more stable than other forms of volunteer computing. That is, since users are likely to leave their information appliances on all the time, then the chance of a particular node leaving a NOIA is smaller than that in other kinds of volunteer networks. Finally, NOIAs composed purely of information appliances would also be homogeneous, since each participating information appliance would have the same type of processor. All these lessen the need for *adaptive parallelism* (see Sect. 2.3.2), and allows greater efficiency and more flexibility in the range of problems that a NOIA can solve.

It may take some time before information appliances and high-speed Internet access become widely available enough to make NOIAs possible. However, it is useful to keep them in consideration since techniques developed for the other forms of volunteer computing are likely to be applicable to NOIAs when their time comes.

2.3 Research Issues and Challenges

While volunteer computing offers all these promising potentials, realizing these potentials and implementing real volunteer computing systems involves many interesting and challenging technical questions and problems. These include technical issues that need to be addressed in making volunteer computing possible and effective. These technical issues can be classified broadly into *accessibility* (making volunteer computing as easy, open, and inviting to volunteers as possible), *applicability* (making volunteer computing useful in real life), and *reliability* (making volunteer computing work in the presence of faults and malicious volunteers). In addition, there are *economic* issues as well, which are especially relevant in implementing commercial systems. In this section, we present all these issues and discuss the challenges that they bring to researchers and developers of volunteer computing systems.

2.3.1 Accessibility

The key to volunteer computing's advantages over other forms of metacomputing is its *accessibility*. It is by making it as easy as possible for as many volunteers as possible to join, volunteer computing can do things that other forms of metacomputing cannot.

Achieving accessibility involves addressing several issues, including:

- **Ease-of-use and platform-independence.** In order to maximize the potential worker pool size and minimize setup time, volunteer computing systems must be usable and accessible to as many people as possible. Thus, they must be *easy to use* and *platform-independent*. Volunteering must require as little technical knowledge from volunteers as possible. Even a seemingly simple setup procedure such as downloading and installing a program may be too complex, since most computer users today generally only know how to use applications, not how to install them. At the same time, users should be able to participate regardless of what type of machine and operating system they use, and preferably without having to identify their platform type.
- **Volunteer-side security.** Volunteer computing systems must also be *secure* in order not to discourage people from volunteering. Since programs will be executed on the volunteers' machines, volunteers must be given the assurance that these programs will not do harm to their machines.
- **User-interface design.** Finally, volunteer computing systems should have a *good user interface design* to encourage volunteers to stay and participate. In most traditional parallel systems, user interfaces for the processing nodes are unnecessary because these nodes are usually hidden inside a large supercomputer and cannot be accessed independently anyway. In volunteer computing systems, however, volunteers need an interface for doing such things as starting and stopping applets, or setting their priority. They also need some sort of progress indicator to assure them that their machines are actually doing useful work. User interfaces for viewing results and statistics, or for submitting jobs or specifying parameters, are also important. Whereas these are traditionally available only to the server's administrators, we may want

to make them available to users as well. In commercial systems, for example, users would like a good interface for submitting problems, and receiving results.

2.3.2 Applicability

Of course, volunteer computing would not be interesting if it were not useful. Thus, the *applicability* of volunteer computing is of prime concern. This involves issues in *programmability*, *adaptive parallelism*, *performance*, and *scalability*.

Programmability

The first aspect of applicability is *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.

One of the key benefits that PVM and MPI brought to the parallel computing and meta-computing worlds, for example, is to provide an easy-to-use, general-purpose programming interface that programmers can quickly learn, and use for a wide variety of applications. This enabled programmers to start using the idea of parallel computing for their own applications. Similarly, in order for people to start benefitting from volunteer computing systems, it would be good to provide them with an easy-to-use and general-purpose APIs and frameworks that would enable them to implement their own applications on them.

The Java programming language provides a good starting point in this respect. Aside from being platform-independent and secure, Java is also *object-oriented*, encouraging (often even *forcing*) programmers to write code in a modular and reusable manner. One challenge in using Java for volunteer computing, therefore, is to develop programming models and interfaces that allow users to take advantage of object-oriented programming while making it easy to write parallel programs and to port existing programs, usually written in C or Fortran, to Java.

Adaptive parallelism

One thing that makes writing programs for volunteer computing systems challenging is the *heterogenous* and *dynamic* nature of volunteer computing systems. 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 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* [55]. That is, unlike many traditional parallel programming models, they must *not* assume the existence of a fixed number of nodes, or depend on any static timing information about the system.

Traditional *message-passing*-based parallel systems, such as those using PVM or MPI, are generally not adaptively parallel. In these systems, it is not uncommon to write programs that say something like, "At step 10, processor P_1 sends data A to processor P_2 ." This may not work or may be inefficient in a volunteer computing system because P_2 may be a slow machine, and may not be ready to receive the data at the time P_1 reaches step

10. Worse, P_2 may simply choose to leave the system, in which case P_1 would get stuck with no one to send the data to.

Various strategies for implementing adaptive parallelism have already been proposed and studied. In *eager scheduling* [34], 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. Volunteer computing systems that implement eager scheduling include Charlotte [12], and Javelin++ [104].

The Linda model [26] 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 [55] system, and more recently implemented in Java by WWWinda [64], Jada [125], and the older version of Javelin, SuperWeb [6]. Sun itself is currently using a Linda-like tuple-space as a basis for their Jini and JavaSpaces technologies [145].

In Cilk [17], 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 in C for NOWs in Cilk-NOW [18], and a proof-of-concept system using Java command-line applications (not browser-based applets) was implemented in ATLAS [11].

Performance and scalability

Finally, for volunteer computing to be truly useful, it must ultimately provide its users with speedups better than, or at least comparable to, other available metacomputing technologies. This implies that volunteer computing systems must have good *raw performance* and *high scalability*.

These issues, while important in all forms of volunteer computing, are of particular concern in Java-based volunteer computing systems because of Java’s historically slow execution speed and restricted communication abilities. One of the major problems in Java-based volunteer computing, for example, 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 for volunteers running applets. Alternatively, we can use *signed applets*, which will also be free of restrictions. These applets can be used as volunteer servers as well.

Another approach would be to design *peer-to-peer* networks which apply the basic ideas

used in popular file-sharing networks such as Napster [100] and Gnutella [58] for providing computational resources instead of files. Some of the more recent commercial metacomputing companies, including those part of the Intel Peer-to-Peer Working Group [74] claim to be taking this approach. There are many challenges in taking this approach, however, as it exacerbates the problems of adaptive parallelism, security, and reliability.

2.3.3 Reliability

Because of their size and their open nature, volunteer computing systems are more prone to faults than other forms of parallel computing. Not counting the problem of volunteers crashing or leaving (which is already covered by adaptive parallelism), faults can include not only unintentional random faults, such as data loss or corruption due to faulty network links or faulty processors, but also *intentional* faults caused by *malicious* nodes submitting erroneous data.

One type of malicious attack is *sabotage*, where volunteers (and possibly also non-volunteers) intentionally submit erroneous results. An unscrupulous user in a commercial system, for example, may try to get paid without doing any real work by not doing its work but simply returning random numbers instead. Such *cheating* volunteers can cause financial loss for the commercial system, not only because they cheat the system of money, but even more because they can generate errors that can propagate and render other computations invalid – even those from honest volunteers.

Another type of attack, particularly relevant in commercial volunteer computing networks, is *espionage*, where volunteers steal sensitive information from the data they are given. Suppose, for example, that a company *A* purchases computing power from a commercial cycle pool in order to process its sales data. Without some form of protection, a competitor *B* can then spy on *A*'s data by joining the cycle pool and volunteering to do work.

Guarding against malicious attacks is a very challenging problem. It is also one that has not been studied very well since, so far, people have been able to trust the parts in their computer not to intentionally sabotage their computations. Possible ways of addressing this problem include using cryptographic techniques such as *digital signatures* and *checksums* to protect against volunteers sending back random results, *encrypted computation* to protect against espionage and attempts to forge digital signatures and checksums, and *periodic obfuscation* (i.e., instruction scrambling) techniques to simulate encrypted computation when it is not possible.

In cases where these mechanisms may not work, however, one must resort to some form of redundancy. For example, we may give the same piece of work to three different processors, and have them *vote* on the correct answer. Unfortunately, however, such techniques have a high computational cost, since repeating work *r* times generally means taking *r* times longer to solve the whole problem. Thus, an interesting research problem is to develop effective but efficient fault-tolerance techniques. As we shall show in Sect. 6.3.3, one possible approach is *spot-checking* with *blacklisting*, where results are only double-checked occasionally, but faulty nodes that are caught are not used again. This may be less reliable than replication, but potentially much more efficient.

In some cases, reliability problems can be addressed 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 suitable problems may include problems like 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), and *self-correcting* algorithms such as *genetic algorithms* where bad results are naturally screened out.

Economic Issues

Issues in paid and commercial systems. In implementing commercial volunteer computing systems, we need to have models for the value of processing power. How does supply and demand affect the value of processing power? How about quality of service? How does one arrive at guidelines for fair barter trade? We also need mechanisms to implement these models. These include electronic currency mechanisms to allow money to be electronically exchanged in a safe and reliable manner, an accurate system for accounting of processing cycles, and efficient brokering mechanisms for managing the trading of processing cycles. Because these mechanisms deal with real money, it is important that they are accurate and robust. Loopholes that permit crimes such as electronic forgery of currency, and “cycle theft” (using someone’s computer without paying them) can lead not only to loss of computational power and data, but also to direct financial losses.

Many groups are already studying these issues [6, 24, 88, 22, 97], and several companies have already started brokering volunteer computing resources for pay as noted in Sect. 2.2.4. Meanwhile, it may also be worthwhile to look into implementing commercial systems with more relaxed mechanisms, such as lotteries and barter trade, that may be less accurate, but at least cannot result in large direct financial losses.

Hidden costs. One of the early arguments for using NOWs instead of supercomputers was that of cost-effectiveness: whereas a single supercomputer costs millions of dollars, a NOW uses existing workstations and thus costs next to nothing. In this way, NOWs can bring parallel processing capabilities to people who cannot afford to buy supercomputers. When considering *massive* NOWs such as volunteer computing networks or NOIAs, however, the validity of this argument may not be so clear anymore. Several issues arise and require further study.

For one, the cumulative cost of supplying electrical power to a massive NOW with thousands or millions of nodes may eventually offset the cost advantage gained from using existing hardware. Because of overhead, fault-tolerance redundancy requirements, and other factors, we expect that a massive NOW with the same processing power as a supercomputer would need several times as many nodes. Furthermore, each of these nodes would be an entire PC or workstation instead of just a single processor. Thus, it is easily conceivable that a NOW can use much more total power than a supercomputer.

We might argue that this power is already being spent by computers sitting idle, and that we are just putting it to better use. This may be true, but it deserves more study. It may be that with power-saving mechanisms, PCs and information appliances would spend less energy when idle. As a very simple example, when a user is not using his

	Unpaid		Paid	
Forced	“cycle thief” hidden applet	traditional NOW; forced private or collaborative network	forced NOIA	contract-based (e.g., trade cycles for info.)
Voluntary	true volunteer network	voluntary private or collaborative network	voluntary NOIA	market-based system
	Untrusted	Trusted	Trusted	Untrusted

Figure 2-1: A Taxonomy of Volunteer Computing Systems.

information appliance, he can turn it off. If we required the user to allow his information appliance to be used in a NOIA as part of his contract, then he cannot turn it off and it will be spending energy which he would normally not be spending otherwise.

Another source of hidden costs is the network. The high-speed links needed to alleviate congestion and improve performance in massive cycle pools may end up costing more than a supercomputer. These links are often leased from a telephone company, and thus can be quite expensive. Furthermore, if a volunteer computation uses a public network such as the Internet, then moving data between computers can cause congestion in some areas, which can then affect unrelated Internet traffic, and cause equivalent financial losses in places uninvolved with the project. Note that these problems are not necessarily enough to make volunteer computing completely untenable. However, in the long run, they are real concerns and should be studied further.

2.4 A Taxonomy of Volunteer Computing Systems

Figure 2-1 shows a taxonomy (drawn as a Karnaugh map) of various forms of volunteer computing systems discussed in Sect. 2.2 according to the following “three A’s” criteria:

1. *autonomy* (truly voluntary vs. forced) – whether volunteers can join and leave of their own free will at any time or are “forced” into volunteering,
2. *anonymity* (known and trusted vs. untrusted) – whether the volunteers are known and trusted by the administrators, or are unknown or untrustable, and
3. *altruism* (unpaid vs. paid) – whether volunteers expect to be compensated for volunteering or not.

This taxonomy is useful in identifying the various issues that need to be addressed in implementing these systems, as discussed in Sect. 2.3.

In general, autonomy is related to the need for adaptive parallelism. That is, forced (non-autonomous) volunteer networks would in general be easier to implement and would potentially have higher performance because they can rely on processing nodes to stay in the system longer. In such networks we may be able to consider running longer jobs than in voluntary systems, where we need to make jobs short enough so they can be done before a volunteer quits. Also, we may be able to do peer-to-peer communication, which, as noted earlier, cannot be done in systems where machines can leave the system at arbitrary times.

Anonymity is related to the security and reliability of the system. Generally, trusted (non-anonymous) networks would be easier to implement than untrusted networks because they do not need to be secure against cheating, sabotage, and espionage as described in Sect. 2.3.3.

Altruism is related to the need for economic mechanisms, for security and reliability, and for performance. Altruistic systems are generally easier to implement since they do not need precise and accurate accounting and payment mechanisms. Also, they are not as prone to cheating, sabotage, and espionage because there is no economic incentive for volunteers to do so. Furthermore, they can also afford to be less efficient in terms of performance since they are not financially accountable to paying clients who might be concerned about getting their money's worth.

In summary, forced, trusted, and unpaid systems are easier to implement because of fewer issues need to be addressed in them. (In fact, traditional NOWs and metacomputing software are built to run on such systems.) On the other hand, voluntary, trusted, and paid systems have the greatest potential for attracting the largest number of volunteers and thus offer the promise of very high performance if their associated problems can be overcome.

2.5 Context and Conclusion

In this chapter, we have presented the many potential forms and benefits of volunteer computing, as well as the many new research issues and challenges it brings. Although they use different names such as *web-based metacomputing*, *global computing*, and *Internet computing*, many others have also recognized the potentials and challenges of volunteer-based parallel computing systems, and have presented their ideas (e.g., [6, 12, 46, 56, 111] and others). Our own approach in this chapter has been to present volunteer computing as a distinct new form of computing, specifically distinguished from other network-based parallel computing systems such as NOWs and metacomputing by its emphasis on ease-of-use and accessibility for volunteers. We have also aimed to present a high-level map of the domain of this new form of computing, as well as of the research terrain ahead, for us and others who wish to explore it. In the rest of this thesis, we present the results of our own explorations in this new space, presenting ways to address the many new research issues, and in the process, showing how volunteer computing's potentials can be achieved.

Chapter 3

The Bayanihan Web-based Volunteer Computing System

In this chapter, we present the idea of *web-based volunteer computing*, and present a Java-based implementation called Bayanihan. We begin by discussing the advantages of web-based volunteer computing over existing application-based systems such as SETI@home and distributed.net. We then present the design of the Bayanihan system, and demonstrate how it allows us to: (1) provide better ease-of-use and accessibility than application-based systems, (2) achieve wider applicability by allowing programmers to write a wider variety of applications than these *ad hoc* systems, and (3) achieve adaptive parallelism, reliability, and scalability. In the process, we not only demonstrate the effectivity and applicability of Bayanihan itself, but also present new ideas, techniques, and tools that can be applied to other systems as well.

3.1 Web-based and Java-based Volunteer Computing

3.1.1 Limitations of Application-based Systems

As shown in Sect. 2.1.2, the success of volunteer computing systems such as distributed.net and SETI@home has mainly been due to their great accessibility. By allowing people to volunteer by simply downloading and installing a screensaver program, these systems have been able to attract many volunteers and achieve much higher levels of performance than traditional NOW and metacomputing systems.

These systems, however, still have limitations. For one, they still require some technical knowledge from volunteers – i.e., at least enough to know their machine architecture, choose the appropriate binary files, download the files, install them, and run the software. Although this may seem like nothing to experienced computer users, it may be intimidating enough to stop a casual user who only knows how to use applications but not how to install them. Furthermore, even advanced users of non-Windows or non-Macintosh operating systems may find installing software difficult (if supported at all). While installing SETI@home on a Windows machine is quite easy, for example, (it takes less than 10 minutes and only a few mouse clicks to install SETI@home on a Windows PC, including

finding and downloading the file and browsing through the documentation during installation), installing it on a UNIX machine requires choosing from 31 different versions, and setting up scripts to run the program from the command line.

Even if users know how to install software on their machines, however, some people may not want to. One problem is *security*. Although volunteer computing systems such as distributed.net do not require giving explicit remote shell access to project administrators like traditional NOW systems such as PVM do, they do require running programs that effectively have the same kind of remote-shell-like access anyway. The screensaver programs for these systems are usually run in user mode, which means that these programs have access to the user's data, potentially including sensitive data such as credit card numbers and passwords. Users may (and should) be wary of volunteering for fear of downloading a Trojan horse or virus that would steal or destroy their data. In recognition of this problem, both SETI@home and distributed.net have issued stern warnings against downloading their client programs from other web sites. They also provide a checksum of their files to allow users to verify their integrity. Unfortunately, however, most users would probably not know how to verify these checksums.

Finally, in some cases, users may not even be capable of installing software on their systems. This may be because there is no version of the software for their operating system, (Entropia [45], for example, only runs on Windows machines), or because their operating systems do not support installing software. The latter case is true in *network computers*, *set-top boxes*, and other forms of *information appliances* that may supply users with web-browsing and Java capabilities, but may not allow them to install other software. It may also be true in cases where a user does not have the access rights to install software as might be the case of a user in an Internet café, or a user using a school-owned machine.¹

The problem of multiple platforms also limits the applicability of non-web-based systems, and in particular, their programmability. Because of the variations between machine architectures and operating systems, programmers of these systems need to exert a non-trivial amount of effort to write, compile, and test code different versions for all the possible target architectures. In fact, programming these systems may even be harder than programming traditional non-volunteer metacomputing systems. This is because in many cases, these volunteer computing systems are *ad hoc* and cannot take advantage of standardized cross-platform libraries for doing low-level operations such as communication and synchronization, like those provided by systems like PVM.

3.1.2 Web-based and Java-based Systems

Sun Microsystem's widely popular and ubiquitous Java technology [59] allows us to address these problems by making it possible to write the volunteers' code as platform-independent *applets*, which are automatically downloaded and executed on users' ma-

¹We do not mean to encourage users to volunteer machines that are not their own, especially if it is against the policy of the owner, as is the case in some schools. However, in some cases, such as in Internet cafés where a user is paying for the use of the computer, then the user arguably has the right to volunteer the machine if he wants to – provided that he only does so during the time that he pays for, and that he kills and uninstalls the program after he leaves. Application-based systems would not allow users to do this, but web-based systems using Java would.

chines when they visit a web page with a Java-capable browser. Using Java to build *web-based* volunteer computing systems in this way improves accessibility in three ways:

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 does not even have to be aware of the file system, or understand what downloading an applet really means. Thus, *anyone*, even minimally computer-literate users of commodity Internet services, can join in such computations. Volunteering one's computer can literally be as easy as a single mouse click.
2. **Platform-independence.** One of the strengths of Java is its platform-independence. This means that programmers need only write one version of their code and post it on the Web as an applet, and it will run on any machine that has a Java-capable browser – which is practically any machine today. Even users of information appliances and Internet café machines, whose operating systems may not allow users to download and execute non-Java programs can participate.
3. **Security.** Unlike the screensaver programs used by distributed.net and SETI@home, the Java applets that volunteers download are executed in a *sandbox* that prevent them from accessing the user's data. This protects volunteers from viruses and Trojan horses that might try to steal or destroy their data. Thus, potential volunteers who are wary of security problems can be reassured and encouraged to participate.²

In addition to improving *accessibility* in this way, using Java also improves the *applicability* of volunteer computing systems through better programmability. Java's platform-independence, as well as its built-in support for graphical user interfaces and network communications, allows programmers to spend less time worrying about low-level details and more time focusing on writing code. Writing code itself is made easier by Java's relatively clean and easy-to-use language features, which allow programmers to take advantage of object-oriented design techniques without the complexity of C++. These make it easier to develop *general-purpose* and *extensible* frameworks and APIs that can help programmers write a wider variety of applications than current *ad hoc* systems currently do.

Of course, web-based and Java-based systems have their own limitations. One of these is the historically slow execution speed of the Java virtual machine (JVM) that executes the platform-independent bytecode. Another problem comes from the security restrictions imposed on Java applets that prevent applets from accessing local storage space or communicating with machines other than the host from which they came. Together, these two problems may limit the performance and scalability of Java-based systems.

Most of these problems, however, can be addressed by using *signed applets*, that come with a digital signature that identifies their author. A volunteer using such a signed applet can decide whether he or she trusts the author or not, and tell the browser to lift the security restrictions on it, allowing it to access local storage, communicate with other machines, and, in some operating systems, even run native code.

²Of course, there is always a risk of having bugs that would allow people have been able to exploit loopholes in implementations of Java [36, 86]. However, Sun, as well as implementors of Java browsers, are constantly on the watch for such loopholes and can plug them reasonably quickly once they are discovered.

Furthermore, even without the use of signed applets, Java-based systems can still be useful. We show in Sects. 4.2.1 and 4.3.1 that although older JVMs performed poorly, new JVMs with *just-in-time compilation* can achieve performance levels comparable to native code in tight-loop computations. Furthermore, we also show in this chapter and in Chap. 4 that volunteer computing can provide good performance and scalability for a useful variety of applications, even with the security restrictions imposed on ordinary applets.

3.2 The Bayanihan Web-based Volunteer Computing System

3.2.1 Design Goals

To demonstrate the benefits of web-based volunteer computing, and to explore the different issues of volunteer computing in general at the same time, we have developed the Bayanihan web-based volunteer computing systems using Java.

In designing this system, we had three main goals:

1. to maximize accessibility and ease-of-use for volunteers by allowing them to volunteer by simply visiting a web page,
2. to provide application programmers with an easy-to-use and flexible programming interface for building volunteer computing systems for various applications, and
3. to allow researchers to experiment with various ways of building volunteer computing systems and approaching the many research issues in volunteer computing.

To achieve these goals, we have built an object-oriented software framework that provides programmers with basic ready-made components, as well as an infrastructure for composing, interconnecting, and extending these objects. In this section, we examine the design of this framework, and discuss the approach we take in achieving our goals.

3.2.2 System Design

To maximize flexibility and programmability, the Bayanihan framework employs a highly object-oriented design. In addition to using Java's built-in object-oriented features, we use HORB [68], a distributed object package similar to Sun's RMI [144], that hides the details of network communications from programmers, and allows them to use objects on separate computers as if they were all on the same computer. HORB was chosen over RMI because it does not require JDK 1.1 (which was not as ubiquitous as JDK 1.0 for several years), is easier to use [65], and is faster [69, 162]. 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.

The Bayanihan framework defines a set of interacting components that can be extended and composed to build Bayanihan systems such as the one shown in Fig. 3-1. Bayanihan systems are composed of many **clients** connected to one or more **servers**.

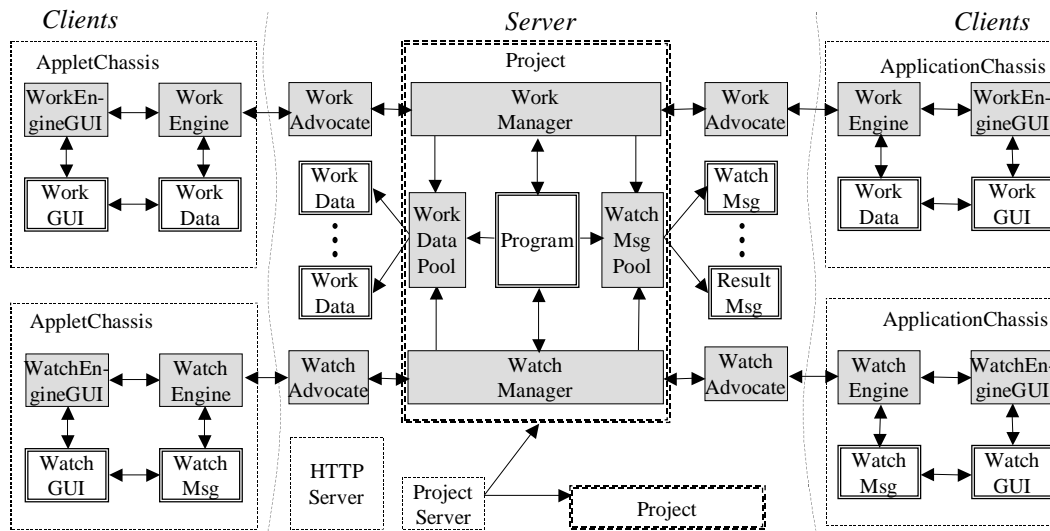


Figure 3-1: A Bayesian system with worker and watcher clients.

In Bayesian, a **client**³ 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 an **engine** that takes care of receiving and processing **data** objects from its corresponding **manager** on the server through remote method calls to its **advocate** on the server side. Both the engine and the data have associated **GUI** objects which can be used for user interface. Data objects are generally polymorphic and know how to process themselves. The engine and the GUIs are all contained in a **chassis** object, which can be an applet or application. A *worker* client, for example, could be an applet chassis containing a work engine which takes care of getting work data from the work manager, processing it, and requesting more work when it is done. To process the work, the work engine calls the work data's `doWork()` method, passing it pointers to itself and the work GUI. The work data then computes itself, using the work engine to communicate with the work manager, and the work GUI to report its progress status to the the user.

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 **project** objects, each containing various objects belonging to separate ongoing computations. To join a computation, a volunteer uses a standard Java-enabled web browser to visit a web page that contains a chassis applet. The chassis applet connects to the server's **project table**, which contains a list of available projects. After selecting a project and an **engine type** (e.g., worker, watcher, etc.) the chassis retrieves the appropriate engine and GUI classes and starts executing the

³In some other systems such as Javelin [23], the term *client* is reserved for the user who wants a computation to be done and who will receive the results, while the term *hosts* is used to refer to volunteers who do the work. In Bayesian, we will use the term *client* in the more traditional sense of clients in a client-server systems. This terminology is also used by SETI@home and other systems. In Bayesian, it is actually even more appropriate because a client does not have to be a volunteer. A client can refer not only to *workers*, but also to *watchers*, as well as to clients in the Javelin sense.

engine on the client side. The engine then connects and registers with the server, which creates a server-side **advocate**⁴ object to represent the client. This advocate goes between the engine and other server-side objects such as managers, and forwards the engine's remote calls to the server-side objects and vice-versa. It also contains client-specific information that the server-side objects may want to use, such as its server-assigned processor ID (PID), its hostname, processing speed, etc. In general, a **manager** object takes care of distributing to and collecting data from its client engines. It typically has access to several **data pools**, which may be shared by other managers serving other purposes. Different managers and work pools can serve different purposes. For example, a *work* manager would take care of things such as distributing unfinished work to a worker client, while a *watch* manager would take care of serving results. Finally, each project has a **program** object, which creates and controls managers, data pools, and data. The program may be *active*, i.e., running in its own thread, or *passive*, i.e., only reacting to callbacks from other objects, such as when the work manager runs out of data and asks for more.

Figure 3-1 shows an example of a Bayanihan system with one server and four clients (two workers and two watchers). The work engines request work data from the work managers, compute them, and then send result data back to the work manager. The work manager then gives these results back to the program object, which can process them and report status information to watchers by placing watch message data objects in the watch pool. The work and watch 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.

3.2.3 Multi-level Flexibility

The highly object-oriented design of the Bayanihan framework enables it to provide multi-level flexibility and extensibility not provided by many simpler volunteer computing frameworks. By allowing programmers to vary objects in the framework independently of others, Bayanihan not only allows users to write a wide variety of *applications*, but also allows them to implement the underlying *generic components* in different ways. This allows researchers to develop and experiment with different ways of addressing issues such as fault-tolerance, adaptive parallelism, and scalability, and makes the Bayanihan framework a good tool for volunteer computing researchers.

Consider the Bayanihan system shown in Fig. 3-1, for example. To write an application for such a system, a programmer need only define new subclasses or implementations of the *application-specific* components shown as double-bordered boxes (i.e., the program object, the work and result data objects, and the GUIs). In this way, application programmers can write a wide variety of applications that assume the particular programming model implemented by the existing engine, manager, and data pool objects. Chapter 4, for example, shows how we can write a variety of applications all using the same master-worker programming interface and implementation.

⁴A more standard name for this object might be *proxy*, after the Proxy design pattern [52]. However, we choose to use another name for two reasons: (1) the term *proxy* already refers to something else in Java and HORB, and (2) an advocate is not limited to forwarding method calls but can keep its own information and act on its own, unlike the simple proxies described in the original Proxy pattern.

Bayanihan's flexibility goes beyond just allowing programmers to write a variety of applications, however. By writing new subclasses or implementations of the *generic* components shown in Fig. 3-1 as shaded boxes (i.e., the managers, data pools, advocates, and engines), programmers can 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, as will be shown in Sect. 3.5.2, we can implement the different fault-tolerance mechanisms discussed in Chap. 6, such as voting and spot-checking, by simply changing or extending the manager, data pool, and advocate objects. Programmers can even implement entirely new parallel programming models by creating new *sets* of engines, managers, and data pools. As an extreme example, in addition to writing volunteer computing systems, we have also used the Bayanihan framework to write multi-user messaging applications, including multi-user chat, shared whiteboard, and ICQ-style messaging applications.⁵

3.2.4 Using the Framework

By taking advantage of this multi-level flexibility, we have been able to use Bayanihan not only to demonstrate the benefits of web-based volunteer computing and to write useful applications, but also to develop new mechanisms for addressing a number of technical issues. In the rest of this chapter, we discuss how we have done so, and present our results.

3.3 Ease-of-Use and Accessibility

As a web-based and Java-based system, Bayanihan greatly increases accessibility and ease-of-use for volunteers, and allows us to do things previously impossible with traditional metacomputing systems and application-based volunteer computing systems. In this section, we show how Bayanihan implements the idea of web-based volunteer computing and discuss our experiences in using it and benefitting from it.

3.3.1 Volunteering

Volunteering to join a Bayanihan system as a worker is very easy, and requires no technical knowledge beyond that of using a web browser. To volunteer, all a user has to do is:

1. Go to the web site running the application he or she wants to join, e.g., <http://bayanihan.lcs.mit.edu/>.
2. Select the link for the desired application, choosing the worker applet.
3. Press the "Start" button on the chassis applet on the web page, once it appears.

At this point, an applet window with the work GUI appears, and the work engine itself connects to the server and starts doing work. Figure 3-2 shows an example where a user has started a work engine for the brute-force factoring application described in Sect. 4.2.1. As it runs, the work GUI displays progress information and local timing statistics. Once

⁵These were joint work with Mark Bucayan, Jerome Punzalan, and Eric Vidal [21].

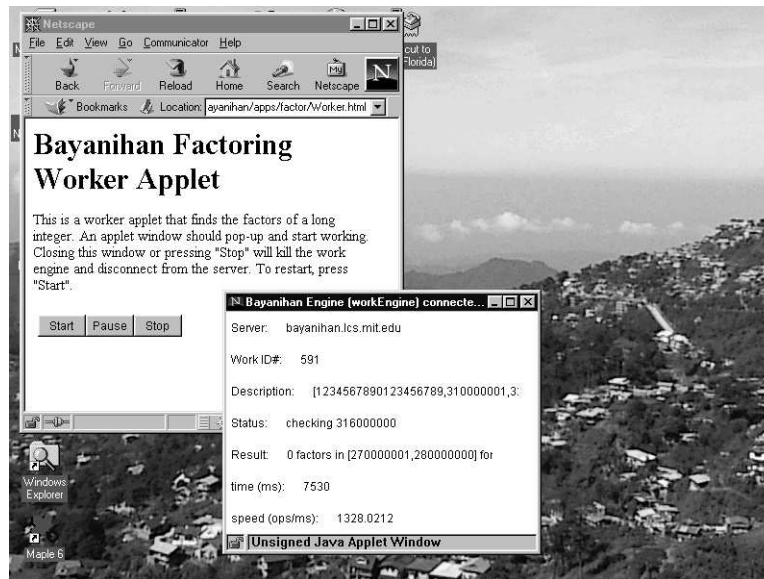


Figure 3-2: Volunteering as a worker using Bayanihan.

the work engine has started running, the user can then minimize the applet window and let it run in the background. The user can even leave the original web page and visit other web sites, and the work engine would continue to run in the background as long as the browser itself is still running. To stop the work engine, the user can simply close the applet window. Alternatively, the user can press the “Pause” or “Stop” buttons on the chassis applet.

Bayanihan also has an *auto-start* option, which allows the work engine to be started automatically without requiring the user to press “Start”. This makes volunteering even easier and faster, and also makes redirection possible as described below. To give the user control over the applet after it starts, the normal “Start”, “Pause” and “Stop” buttons still allow the user to stop and restart the engine. Potentially, however, auto-start applets can also be used for “cycle thief” work engines that do not have GUI components and can remain invisible to the user. These applets, inserted in a web page, will start automatically and keep working even after the user leaves the web page, as long as the user’s browser is still running. Although we do not encourage the use of such applets, they do represent the ultimate in ease-of-use, since a user does not even have to be conscious of volunteering. They may also have legitimate applications in *contract-based systems* such as described in Sect. 2.2.4.

By default, the work engine runs in the lowest priority setting available. In most operating systems, this is *idle priority*, which means that the work engine only runs at times when there are no other programs needing the processor. This is the same priority setting used by distributed.net and other systems, and generally works very well. At this setting, the work engine does not noticeably slow down other applications. At the same time, however, it keeps the CPU fully utilized while the user is idle and can achieve almost full speed even while the user is using programs such as word processors, and other non-

computationally intensive programs. If necessary, Bayanihan also supports changing the priority level of the work engine. When running the fault-tolerance simulator on the MIT Athena workstations (as described in Sect. 4.6.4), for example, we used normal priority so that the work engine gets priority over the screen saver, which ran at low (but non-idle) priority when the machines were unattended.

If a Bayanihan work engine gets disconnected while it is running – e.g., because of network failure or server failure – it would automatically try to reconnect to the server periodically until it succeeds. When the network connection or the server comes online again, the work engine then continues doing work in a new session. This means that a user need only start the work engine once, and it will keep running despite network or server failures as long as the user does not exit the browser application. (Note that in the current implementation, when a worker gets disconnected, it loses the current work that it is doing, and starts working on an entirely new object when it reconnects. Bayanihan does not yet support disconnected operation as SETI@home does.)

Through this reconnection mechanism, Bayanihan also supports having the work engine switch applications automatically without requiring the user's intervention. Currently, we can do this by stopping the server, and restarting it with a new application. Doing this will cause the workers to disconnect when the server stops, and then reconnect and start working on the new application when the server is restarted. In the future, we may also allow a worker to switch applications without having to disconnect and reconnect. This can be done by simply giving it work objects for different applications. We do not yet support this feature since the advocate and manager objects in the current implementation can only be used with one work pool (and thus one application) at a time.

Finally, Bayanihan also supports automatic *redirection* of a work engine to another application on another server. We can do this despite the applet security restrictions against establishing TCP/IP socket connections with other servers by making use of the `showDocument()` function in Java applets, which allows an applet to ask its parent browser to display a web page given a URL. To have a work engine reconnect from server *A* to server *B*, for example, we send the engine a special `RedirWork` object, which calls `showDocument()` to have the browser show server *B*'s web page. Server *B*'s web page would have an auto-start applet that would then be run automatically as server *B*'s page is displayed. Since this applet is now coming from server *B*, it can safely connect to server *B* even with security restrictions in place. Meanwhile, `RedirWork` causes the original applet from server *A* to quit and stop working in the background. This completes the changeover from server *A* to server *B*.

3.3.2 Watcher Applets and Interactive Parallel Computing

In addition to providing worker applets that allow users to volunteer their computing power, Bayanihan also provides *watcher* applets that make *interactive parallel computing* possible by not only letting users view the results of a computation as they are received, but also letting them control the computation even as it happens.

Figure 3-3, for example, shows the corresponding watcher applet for the brute-force factoring application. Here, the watcher applet displays the list of factor pairs found so far, and also displays timing information. At the same time, the watcher applet also allows

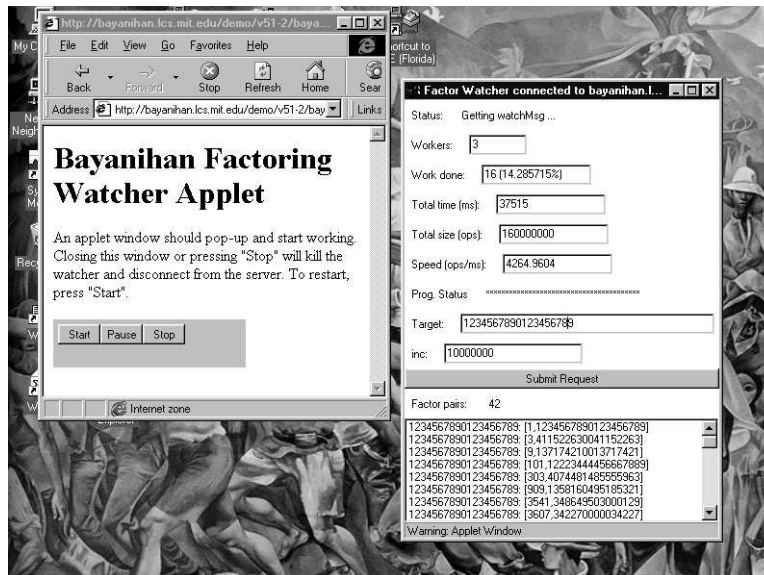


Figure 3-3: Watching results and controlling the computation using a watcher applet.

the user to change the target number by entering a number in the box and pressing “Submit Request”. In response to such a request, the server would immediately stop working on the current target number, and start working on the new one. A more complex example involving image rendering is also shown in Sect. 4.3.1, where the user interfaces allows the user to use the mouse to interactively zoom in and zoom out of the image being drawn.

As far as we know, Bayanihan is the first, if not the only, volunteer computing system to allow real-time control and to support interactive parallel computing in this way. Other volunteer computing systems display results as they get them, but do not have easy support for making interactive requests as Bayanihan does.

3.3.3 Practical Results

Bayanihan’s ease-of-use has made it possible for us to set up parallel computing networks much more quickly and easily than possible with traditional metacomputing systems and application-based volunteer computing systems such as SETI@home.

For example, we have done demos at several conferences and seminars where we set up a volunteer computing network on the spot and have it start running the Mandelbrot renderer application (described in Sect. 4.3.1) using whatever workstations are available in the seminar room. In these demos, we install a web server and the Bayanihan server on a laptop beforehand. Then, at the demo site, we connect the laptop to the local network using DHCP (which automatically configures the laptop to connect to the network). Once the laptop is connected, starting a parallel computation is then simply a matter of starting a web browser on each machine and pointing it to the web site on the laptop. (If the audience has the computers in front of them, we ask the audience themselves to volunteer the machines by themselves.) The whole process takes only a few minutes, and the resulting parallel network achieves near linear speedups given enough granularity in the

computation.

Beyond demos, Bayanihan has proven itself in real applications as well. For example, it allowed us to get much needed speedups for running the fault-tolerance simulator (see Sect. 4.6.4), by allowing us to use up to 50 machines on MIT's campus network (run at night when no one was using them). A significant advantage in this case, was not only the fact that we did not need to install software on the machines, but even more so that we did not need to *uninstall* software afterwards. This not only relieved us from having to clean up after experiments, but also made it easier for us to ask friends and colleagues to volunteer their machines by reassuring them that it would only be temporary and that they would not have to clean up afterwards.

The reconnection and redirection features of Bayanihan also proved useful when running the fault-tolerance simulations. These allowed us to keep machines running for weeks at a time without requiring volunteers to restart their work engine (i.e., they started their browsers once and left them running in the background at idle priority). During these weeks, we were able to restart the server several times and run it with different parameters and even with new computational code (by defining new work object subclasses). Occasionally, we even had the workers switch to the Mandelbrot application and back to the fault-tolerance simulator using redirection.

3.4 Programmability and Applicability: Writing Applications

In addition to making it easier for users to volunteer their machines, Bayanihan also makes it easier for programmers to write a variety of applications. In this section, we describe the implementation of a runtime system for Bayanihan that provides the underlying mechanisms for running interactive master-worker style applications, as well as a number of programming interfaces that allow programmers to write different applications on top of this runtime system. In Chap. 4, we present several applications that we have actually implemented using these APIs.

3.4.1 The Generic Master-Worker Runtime System

Basic Components

Figure 3-4 shows the components of the Bayanihan runtime system and the interactions between them.

As shown, the generic `MWProj` project object on the server side contains a set of generic managers and data pools for implementing the master-worker model. The project also creates and starts an application-specific program object (specified through the project's configuration file) by calling its `startProgram()` method, which then fills the work pool with work data objects in its `createNewBatch()` method.

Each volunteer machine runs a **work engine**, which runs in a loop, repeatedly making remote calls to the `getWork()` method of its corresponding **work advocate** on the server. Each work advocate passes these calls to the **work manager**, which takes care of distributing work contained in the work pool to the workers.⁶ As it calls the manager, each

⁶In older versions of Bayanihan, the work manager was actually a single object. In the current version,

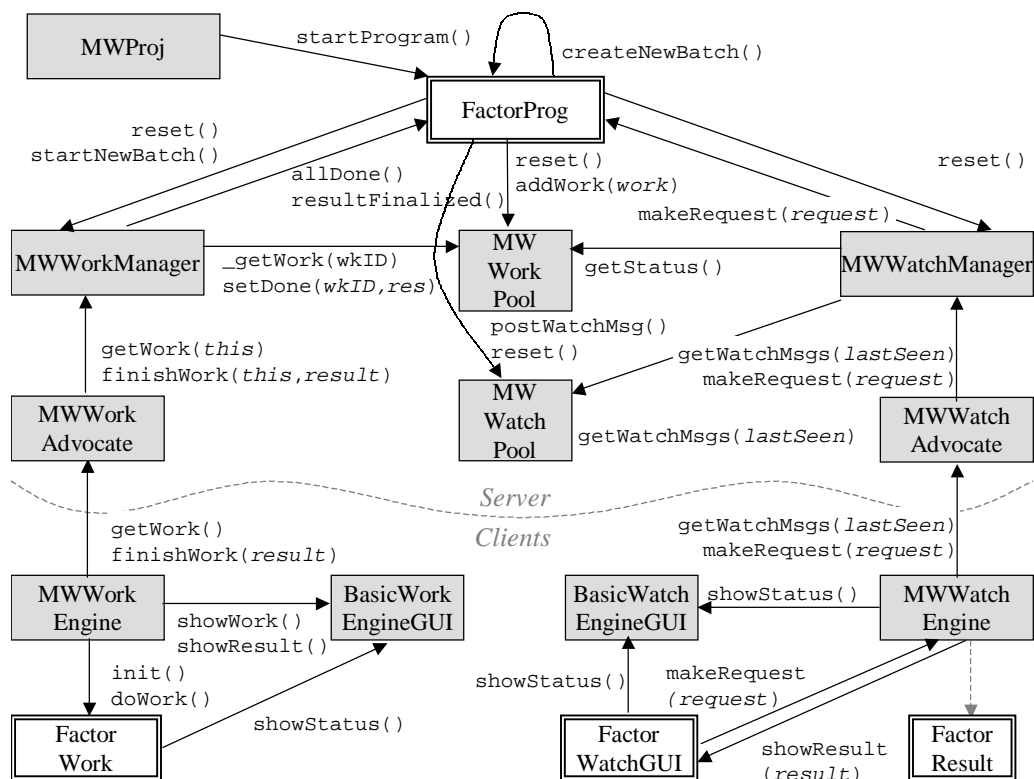


Figure 3-4: Master-worker runtime layer components and interactions.

advocate also passes a pointer to itself to allow the manager to identify it and call it back if desired.

In response to a call to `getWork()`, the work manager returns the next available un-completed work data object in the work pool. The work engine runs this work object by calling its `doWork()` method, which returns a result object. Finally, it sends the result back to the work manager by calling the advocate's `finishWork()` method.

As the work manager obtains a final result (i.e., a result after voting, if voting is enabled) for each work entry, it associates each result with its corresponding work entry in the work pool and marks the work entry done. At the same time, it calls the `result-Finalized()` method to notify the program of the new result, and to give it a chance to process the result and post a corresponding **watch message** onto the watch pool, if desired. When all the work in the pool have been done, the work manager calls its program object's `allDone()` method. In this method, the program can process the results, and start the next stage of the computation, if any, by refilling the work pool with new work and calling `startNewBatch()`. (By default, `allDone()` calls `createNewBatch()`.) This implements a simple form of barrier synchronization, where a new batch is not started until the previous one is completely finished.

Support for Interactive Parallel Computing

Users can view results and statistics as well as control the computation, as described in Sect. 3.3.2, 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 **watch messages** which may have been posted by the program onto the watch pool. Any watch messages it receives (which typically contain result objects) are then passed 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. When the program object receives the request, it can, if desired, immediately call `reset()` on the work and watch managers (causing the work and result pools to be cleared), and then create and start a new batch according to the request object.

3.4.2 Writing Applications: The Basic APIs

To write an application, programmers need to create classes that define appropriate methods in the *application-specific* objects shown in double-bordered boxes in Fig. 3-4. Appendix A contains an example with the source code for the brute-force factoring application described in Sect. 4.2.1.

Of the classes that a programmer needs to define, the two key classes are the *work* and the *program* classes.

the responsibility of the work manager has been divided among several objects for greater modularity and flexibility. For simplicity, we will refer to these objects together as if they were still just one work manager object.

Work objects. The *work* object class contains the computational code that runs on each volunteer worker's machine. Each instance of a work object contains the code and data for an independent subpart of a parallel step – e.g., a subrange to be searched in a brute-force search (see Sect. 4.2), or a particular parameter configuration in a parametric analysis (see Sect. 4.6). The data are stored in the fields of the object, while the code is defined in the `doWork()` method. Thus, the `doWork()` method should be written to take the values of the appropriate fields, perform the desired (sequential) computation according to these initial values, and then return a *result* object, which can be of any type. In the factoring application, for example, a `FactorWork` object (as shown in Sect. A.1.1 searches for factors of the target number `target`, within a subrange, `[first ... last]`. It therefore contains the `long` fields, `target`, `first`, and `last`, and its `doWork()` method contains a loop searching for factors within that range.

Passive-style programs. The *program* object is responsible for creating *batches* of work objects, and interpreting the results they return. Typically, a program would create one batch of work by adding work to the work pool, start the batch by calling `startNewBatch()`, and then wait for the batch to be completed. Then, it processes these results and creates a new batch of work, possibly depending on the results of the previous batch.

The most basic way to write a program object is in **passive style**, wherein the program does not have a thread of its own but only reacts to method calls from other objects, as described in Sect. 3.4.1 – e.g., calls to `startProgram()` from the project object, `allDone()` from the work manager, and `handleRequest()` from the watch manager. In this case, a programmer must define `allDone()` to respond to the end of the batch and create a new batch.

Passive style is easy to use if the program does exactly the same thing in each batch, except possibly for a change in parameter or configuration values. In the factoring example shown in Sect. A.1.5, for example, the code for filling in a batch is always the same and can be encoded in `createWork()`, which is called in `allDone()` (via `createNewBatch()`). In applications where the batches are different, however, such as the fault-tolerance simulator, then passive style is difficult to use, since it would require that `allDone()` have some sort of `if` or `switch` statement to determine which batch to start next.

Active-style programs. To handle such cases, Bayanihan also provides a slightly higher-level API layer for **active-style** programs. In active-style programs, the programmer defines a `run()` method which is run in its own thread when `startProgram()` is called. In the `run()` method the programmer creates batches by adding work to the work pool, and then starts them by calling `startNewBatch()`. At this point, unlike in passive-style programs, the program continues to run in its own thread. To wait for the batch to complete, the program calls `waitForAllDone()`, which blocks until the batch is completed. At this point, the program can then process the results, create a new batch, and start it again.

The advantage of active-style programs is that they allow the code in `run()` to look like sequential code. With active-style programs, programmers can hide the calls to `startNewBatch()` and `waitForAllDone()` in convenience methods, and thus provide “parallel methods” that look like ordinary method calls but actually perform computation in

parallel. Section A.1.6, for example, shows an active-style version of the factoring program, wherein we have a `doParallelFactor()` parallel method. The `doBatch()` method of the Monte Carlo API, as discussed in Sect. 4.6.2 and shown in Sects. A.2.4 and A.3.1, are other examples of parallel methods.

In active-style programs, programmers can process results at the end of the batch (i.e., after `waitForAllDone()` but before the next `startNewBatch()`), or on-the-fly through the use of a `ResultListener`-type object. Both of these types of result processing are shown in the example in Sect. A.1.6. Requests can also be handled at the end of the batch through a request queue (not shown in the examples), or on-the-fly by having the program call `waitForRequest()` instead of `waitForAllDone()`, as shown in Sect. A.1.6. If no request is made, then `waitForRequest()` blocks until the batch is done (just like `waitForAllDone()` does), and returns `null`. If a request is made before the batch is done, however, then `waitForRequest()` immediately returns with the request object. The program can then handle the request and decide whether to continue the current batch, or to stop it and start a new one. In this way, we can achieve interactive parallel computing.

3.4.3 Application Frameworks and Other APIs

In addition to using the passive-style and active-style APIs directly, the object-oriented design of the Bayanihan framework and APIs make it easy to develop more specific *application frameworks* that provide support for writing *families* of similar applications. For example, extending the Mandelbrot application, we have written an application framework for rendering in general, which we can use in the future for depicting other fractal objects (e.g. the Julia set), as well as for rendering 3D images. In this framework, the programmer can reuse common objects such as the GUIs and would simply have to write the code for computing the color of each pixel. Other examples of application frameworks we have written include ones for Monte Carlo simulations and genetic algorithms, as described in Chap. 4.

Furthermore, we can also write other APIs that are more general-purpose than application frameworks, but still use the same underlying runtime system. Examples of these are the Bayanihan BSP programming interface presented in Chap. 5, which is actually built on top of the active-style API but provides the user with an even easier-to-use are more powerful programming interface.

3.5 Developing Generic Mechanisms

Finally, Bayanihan's highly flexible design not only makes it easy to write specific applications, but also makes it easy to develop *generic* mechanisms that apply to volunteer computing applications and systems in general. In this section, we show how we have used Bayanihan's flexibility to address the issues of adaptive parallelism, reliability, and scalability. In the rest of this thesis, we present other new techniques and technology that we have developed with the help of the Bayanihan framework.

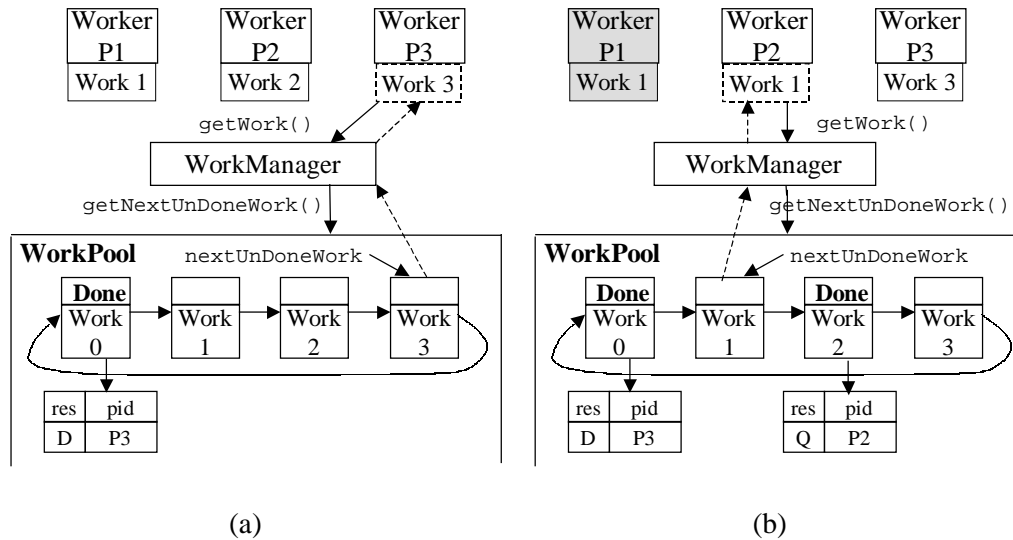


Figure 3-5: Eager scheduling in Bayanihan. (a) P3 calls `getWork()` after works 1 and 2 are assigned to P1 and P2; P3 gets work 3. (b) P1 crashes or is slow; work 1 gets reassigned to P2 after P2 finishes work 2.

3.5.1 Adaptive Parallelism

Eager Scheduling

Generally, adaptive parallelism can be implemented by writing work managers that follow appropriate scheduling strategies. In our master-worker runtime system we employ a simple form of adaptive parallelism sometimes called *eager scheduling* [34] (also discussed earlier in Sect. 2.3.2). In our implementation of this scheme, as shown in Fig. 3-5, 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 3-5(a), for example, the `nextUnDoneWork` pointer is pointing to work 3 after works 1 and 2 have been assigned to work engines P1 and P2 respectively. Thus, when work engine P3 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. 3-5(b), for example, we see that when P2 finishes work 2 and calls `getWork()`, it receives work 1, which has not been marked done because P1 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

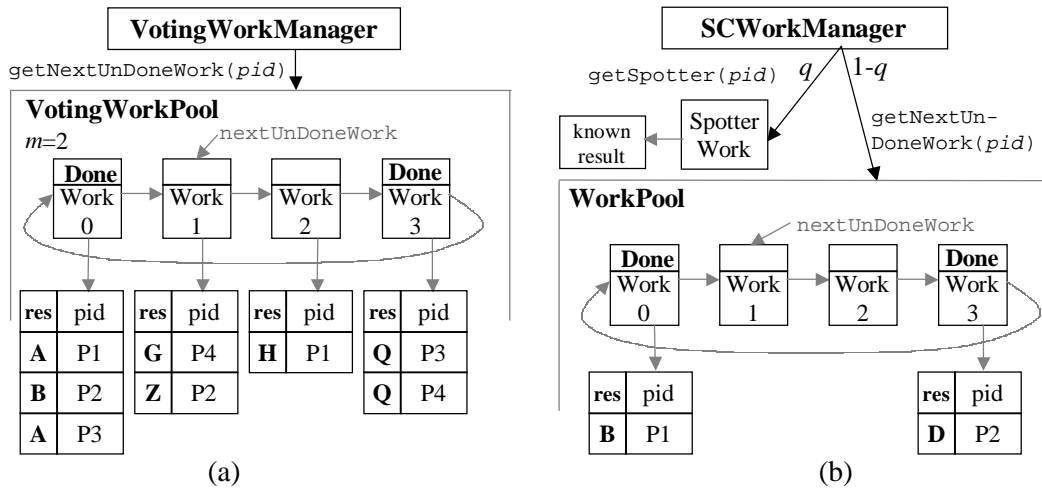


Figure 3-6: Two approaches to fault-tolerance: (a) Majority Voting, (b) Spot-checking.

becomes available.

Other Possibilities

Other forms of adaptive parallelism can be implemented with the master-worker runtime system as well. For example, we have written `MultiWorkEngine` and `MultiWorkManager` subclasses that implement a `getMultiWork()` function for prefetching multiple packets of work. As described in Sect. 4.5.1, these have proven useful in our distributed web-crawler application for improving performance by hiding communication latency. They have also been useful in implementing volunteer servers, as described in Sect. 3.5.3.

Potentially, Bayesian can be used for implementing other forms of adaptive parallelism as well. For example, one can conceivably implement PVM-like functionality [54] by writing a work engine that functions like a PVM daemon, and writing PVM-like programs or tasks as work data objects. The work data objects would depend on the work engine to provide message-passing and task-spawning capabilities. The work engine may implement message-passing directly (if it has no network restrictions), or it can forward the responsibility to the manager (if restricted applets are used). The manager can then route messages to the appropriate work engines and work data on other machines. Of course, PVM is a bad example since it is not adaptive. However, we use it here as an extreme case – i.e., if you can implement PVM, then implementing other programming models (e.g., Linda [26], Cilk [17]) is at least as easy, since PVM (and message-passing in general) can implement these.

3.5.2 Reliability

By extending the eager scheduling work manager and work pool objects as shown in Fig. 3-6, we have written implementations of the *majority voting* and *spot-checking* mechanisms described in more detail in Chap. 6.

We implement majority voting by using a new subclass of `MWorkPool`, `FTWorkPool`, where the `done` flag of a work object remains unset as long as a majority agreement with m matching results has not been reached. In Fig. 3-6(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. (To allow results to be compared to each other, programmers must implement the `hasSameValue()` method in their result objects.)

Our spot-checking implementation, shown in Fig. 3-6(b), works as follows: at the beginning of each new batch of work, the `SCWorkManager` subclass 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 q called the *spot-check rate*. 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, invalidating any results dependent on results from the offending worker.

Table 3.1 shows results from an experiment using these two fault-tolerance mechanisms. In this experiment, we created a saboteur work engine that always corrupts results in a fixed way⁷. We then ran the Mandelbrot application (on the “spiral” range with depth 1037 from Sect. 4.3.1), with 5 good workers and a varying number of saboteurs. To keep the pool of saboteurs from being completely eliminated by spot-checking, we did not enforce blacklisting, and allowed caught nodes to reconnect after a 1 second delay under a new identity. We did, however, enforce backtracking, such that all results submitted by a saboteur under the same identity are invalidated once a saboteur is caught. 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.

As shown, majority voting performed as expected, having an error rate close to the theoretical expected value of $f^2(3 - 2f)$ for $m = 2$ majority 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 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 q 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 enforced, 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 –

⁷That is, the sabotage rate s (defined in Sect. 6.3.1) is 1, and bad results for the same work object match such that saboteurs can vote together

Table 3.1: Results from preliminary 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

a highly unlikely scenario – they would quickly get eliminated, and the error rate would decrease rapidly in time.

Figure 3-7 shows a screenshot of the Mandelbrot application with spot-checking enabled (together with backtracking but without blacklisting as described above). In this case, we have two workers and one saboteur.⁸ Here, as in Sect. 4.3.1, the color of the border of each block represents the worker which worked on that block. The blocks with the X-like pattern are blocks from the saboteur, who is trying to sabotage the computation by submitting erroneous images. The blocks with circles represent results that have been invalidated by backtracking because the worker that submitted them has gotten caught. As shown, in this case, the saboteur has currently gotten away with submitting five erroneous blocks under its current identity without getting caught yet. Note, however, that it has been caught a number of times under its previous identities, as evidenced by the invalidated results. If it gets caught in the future, then the five erroneous blocks shown here would also be invalidated.

To study the effectivity of these fault-tolerance mechanisms even better, we have also developed a parallel fault-tolerance simulator running on Bayanihan, as described in Sects. 4.6.4 and 6.5.1. This simulator uses adapted versions of the same work pool implementations used in this experiment, but uses simulated workers instead of real machines doing real work. This allows us to test the fault-tolerance mechanisms on a much larger number of workers, saboteurs, and work objects than possible in a real experiment. In this way, we have been able to extend the results of the experiment shown here, and test the effects of blacklisting, as well as that of varying the sabotage rate and many other factors, as shown in Sect. 6.5.1. The results we got from these simulations in turn led us to the new and highly generalizable idea of *credibility-based fault-tolerance*, described in Chap. 6.

⁸The machines used here were different and slower than the ones used in Sect. 4.3.1, so the timing results are not comparable.

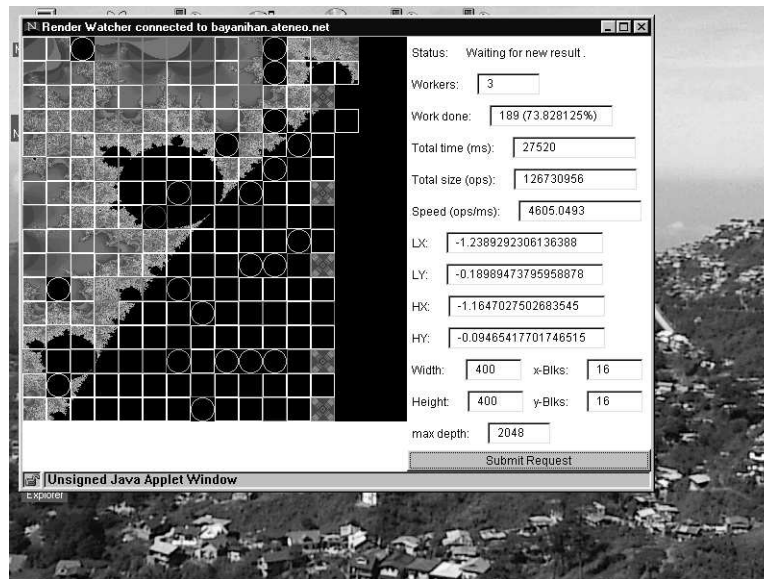


Figure 3-7: A screenshot of the Mandelbrot application with spot-checking enabled.

3.5.3 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* with the server in the middle, and can potentially lead to high congestion and limited scalability.⁹ To address this problem, we have developed a simple *volunteer server* system that volunteers can download and run as a Java application (together with an HTTP server for serving out the applets and web pages to the clients). This application creates a generic `VSPProgram` object whose `createNewBatch()` method does not create new work on its own, but instead calls `getMultiWork()` (as described in Sect. 3.5.1) to requests *groups* of work objects from the main server, and then places these work objects in its own work pool to serve its clients. Correspondingly, its `allDone()` method sends the results back to the main server and calls `createNewBatch()` to get more work.

Figure 3-8 shows how volunteer servers can help improve a system's performance and scalability. Consider the scenario shown in Fig. 3-8(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. 3-8(b).

Table 3.2 shows results from an experiment simulating these scenarios using a 28.8 Kbps modem link for the slow link, and 10Mbit Ethernet for the fast link. Note that

⁹We say "potentially" because it is possible to have a star topology and still support a large number of users. This has been demonstrated by SETI@home, which employs a star topology with one or a few central servers at Berkeley, but has nevertheless been able to successfully handle hundreds of thousands of users. However, SETI@home has also experienced congestion problems at times, which shows that having a more scalable topology would be ideal and preferable even in these cases.

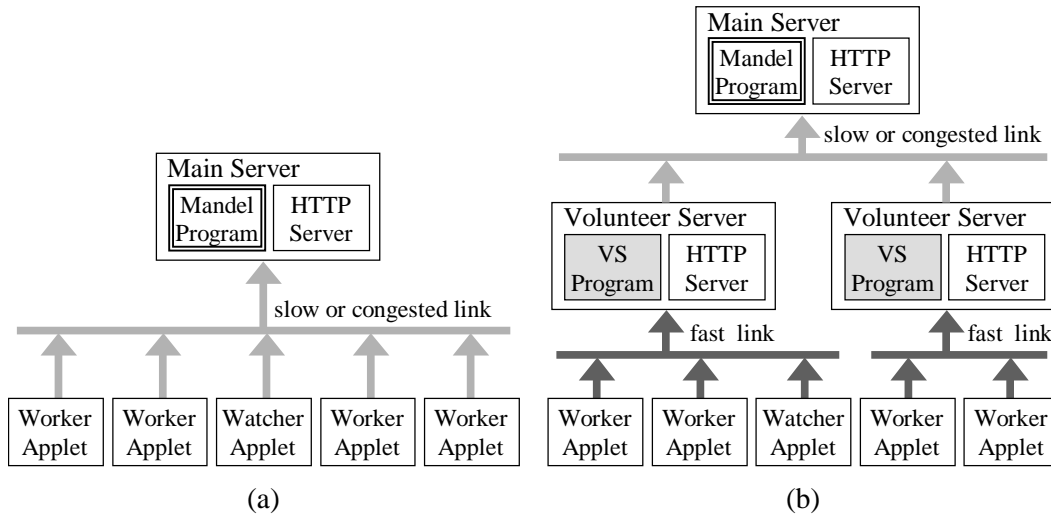


Figure 3-8: Using volunteer servers. (a) Slow links result in unnecessary delays and idling. (b) Volunteer servers help by exploiting communication parallelism and locality.

Table 3.2: Running the Mandelbrot application with 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

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. 3-8(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. These results demonstrate how volunteer servers can enable us to overcome congestion and long latencies by exploiting locality and parallelism in communications.

Currently, we do not have a way to automatically direct clients from the main server to an appropriate volunteer server. If we can develop mechanisms for load balancing, however, then we can use the redirection technique mentioned in Sect. 3.3.1. This is future work that remains to be done.

3.6 Conclusion

In this chapter, we have presented the Bayanihan web-based volunteer computing system, and have shown how it has made volunteer computing easier not only for users but for programmers and researchers as well. We conclude this chapter by looking at our results in the context of related work and suggesting possible directions for future research.

3.6.1 Related Work

Java-based parallel computing. Bayanihan is actually just one among a growing number of Java-based parallel computing systems that have independently been developed since the original release of Java by Sun in 1995. Since 1997, for example, many papers on using Java for parallel computing have been presented at various conferences [3, 4, 5].

Most of the earliest projects, such as ATLAS [11], Paraweb [19], and JPVM [47], and many of those that came after, such as IceT [61], Java// [25], Ninplet [146], DOGMA [77], Manta [109], Parabon [113], and others, use Java applications that are run from the command line. Being application-based systems, these do not enjoy the same level of ease-of-use and accessibility as applet-based systems. In fact, most of these are not actually volunteer computing systems, but are more like Java-based versions of traditional NOW or metacomputing software like PVM or MPI. Their main advantage over traditional systems, is the portability and programmability that Java brings.

Projects that implement web-based volunteer computing by using applets seem to be fewer. Early systems include simple *ad hoc* ones such as the University of Washington factoring applet [154] (which inspired our factoring application), DAMPP [151], the RC5 cracking project [57], and more complex general-purpose ones such as Charlotte [12, 80, 14], the original versions of SuperWeb/Javelin [6, 23] (newer versions of Javelin [104, 106] now use applications, not applets), and JET [111]. More recent ones include SWC [8] and the gamma flux project [75] that is now part of Process Tree [121]. One reason that applet-based systems are fewer than application-based systems is that Java's security restrictions make peer-to-peer communication – and thus traditional MPI-like systems – impossible to implement. Also, applet-based systems need to support *autonomous* volunteers who can leave and join the system at any time. Thus, developing *general-purpose* applet-based systems is harder in general because it requires developing and using new programming models. On the other hand, applet-based systems are much more accessible and easier to use than application-based systems, and thus have the potential for allowing us to build larger systems in a shorter amount of time, as demonstrated in Sect. 3.3.3.

Applicability, Programmability, and Flexibility. Although some of these systems are *ad hoc*, such as [154, 57, 75], many of them are general-purpose, and allow programmers to write a variety of applications, as Bayanihan does. Most of these systems, however, provide only one way of writing programs, while Bayanihan provides several. In addition to allowing programmers to use the passive-style, active-style, and BSP-style APIs, Bayanihan also provides application frameworks such as the rendering, genetic algorithm, and Monte Carlo frameworks. Moreover, Bayanihan also makes it easy for programmers to

create their own APIs and application frameworks by extending and building on top of existing APIs and application frameworks.

Bayanihan is also among the first systems to use a distributed object framework for communications. Most earlier systems performed communications by handling TCP/IP sockets directly. By using HORB, Bayanihan enables programmers (including ourselves) to focus on the object-oriented design of the system instead of the low-level communication details, and not only saves them a lot of development time and effort, but also makes more complex interactions between the server-side and client-side objects possible. Others have since also independently recognized the advantages of using distributed objects, and have designed (or redesigned) their systems using RMI (e.g., see [104, 13, 77, 109]), or CORBA (e.g., see [81, 120]).

Perhaps the most distinguishing feature of the Bayanihan framework, however, is the flexibility that it provides researchers and implementors of volunteer computing systems. Although other systems may have their own way of addressing the issues of programmability, adaptive parallelism, fault-tolerance and scalability, they usually only have a single set of mechanisms which programmers are required to use. In contrast, Bayanihan has been intentionally designed from the beginning to allow programmers to extend the provided mechanisms and even to create their own. As shown in Sect. 3.5, and in the rest of this thesis, this flexibility has allowed us to explore the field of volunteer computing more broadly than possible in other systems. In addition to allowing us to implement and further develop well-known ideas such as eager scheduling and volunteer servers, it has also allowed us to discover and develop unique new ideas such as the BSP programming interface and the fault-tolerance mechanisms discussed in Sect. 3.5.2 and Chap. 6.

3.6.2 Future Work

The results shown in this chapter and in the rest of this thesis represent only the beginning of much more possible research. Possible future work on the Bayanihan system and framework include the following:

- **Better volunteer and server management infrastructure.** Although our current infrastructure has already allowed us to run campus-wide experiments as described in Sect. 3.3.3, it still lacks some features needed for deployment to the general public. Currently, we do not keep track of user identity and contact information, or provide access control to the client applets. It would also be good to provide server administrators with an interface allowing them to monitor the availability of worker machines and to stop, restart, or redirect these workers from the server's side, if desired. This can be implemented as a watcher applet which is only available to server administrators.
- **Better fault-tolerance.** Our current implementation of spot-checking is good for proof-of-concept demonstrations and experiments, but needs to be improved for practical use. In particular, the server currently uses the same spotter work object repeatedly in one batch. Thus, a smart saboteur can identify the spotter work by keeping track of the work objects it has received and watching for duplicates. A better implementation could be to use several spotter works instead of just one, or, even

more securely, we can decide to spot-check results on-the-fly as they are received (with probability q) instead of pre-selecting the spotter works beforehand. We could also implement spot-checking by voting as described in Sect. 6.4.3.

Alternatively, instead of selecting whole spotter works from the work pool, the server can randomly select a small subpart of each work object in the pool and check that part when it receives the corresponding result from a worker. In this case, the server spot-checks workers all the time (i.e., $q = 1$), but each spot-check has only a probability $h < 1$ of detecting an error. Thus, this scheme should be similar to the old scheme with q replaced by h , but has the advantage of not having a single spotter work that smart saboteurs can distinguish from the rest. Note however, that as is, this new scheme is mainly applicable in applications such as image rendering, where the work objects themselves are partitionable into smaller independent parts. In other applications, we may be able to use the techniques presented in [99] to allow us to check subparts of a sequential computation.

- **Better scalability.** Like our spot-checking scheme, our volunteer server implementation is currently just a proof-of-concept implementation for demonstration purposes. In order for volunteer servers to be used in real applications, we still need to implement some mechanisms such as a way to direct volunteers to the nearest volunteer server, as well as a way to load balance between volunteer servers. It may also be useful to look into other systems that also support the use of volunteer servers, such as Javelin 2 [104, 106], which uses a combination of eager scheduling and Cilk-style work stealing, and distributed.net, which allows volunteers who administer their own LANs or intranets to install a “personal proxy” [42] – i.e., a local server acting as a sort of cache between distributed.net and their internal machines (much like Bayanihan’s volunteer servers do).

Chapter 4

Master-Worker Style Applications

4.1 Introduction

The ideal application for volunteer computing systems is one that is *coarse-grain* and *embarassingly parallel* (i.e., easily partitionable into completely independent parts). Coarse-grain applications are ideal because most volunteer computing systems employ commodity Internet network links which have limited bandwidth and high latencies. Thus, the higher the computation-to-communication ratio (i.e., the *granularity*), the better the performance. Embarassingly parallel applications are ideal not only because they tend to be coarse grain as well (or can easily be made coarse grain by partitioning the work into large blocks), but also because the independence of their subparts makes adaptive parallelism and fault-tolerance easier to achieve than in other applications. In particular, since work objects are independent of each other, the master can easily reassign work left undone by workers who have left or crashed, or undo work done by bad workers. Also, since a worker's code does not need to be aware of information such as how many other workers there are, when other workers join or leave the system, or which workers are faulty, it is easier to migrate a worker process from one physical processor to another for load balancing, or to run the same worker process on multiple processors for fault-tolerance.

A programming model that is especially appropriate for coarse-grain and embarassingly parallel applications is the *master-worker* model, depicted in Fig. 4-1. In this model, a *computation* is divided into a sequence of *batches*, as shown. Within each batch, the *master* node divides the work to be done into many mutually independent *work objects*, and distributes these to available *worker* nodes. The worker nodes then execute the work objects in parallel, and return their *results* to the master. When the master has collected the results for all the work objects in the batch, it processes these results, and generates the next batch of work objects (possibly using data from the results of the current batch). Then the whole process is repeated for the new batch.

Because of its simplicity and appropriateness to volunteer computing, many wide-area metacomputing systems today, as well as most, if not all, *autonomous* volunteer computing systems (i.e., ones wherein volunteers have complete freedom to join or leave the computation at any time) assume, or at least highly encourage, using a master-worker model of computation. These include Condor [48] (one of the first metacomputing systems employ-

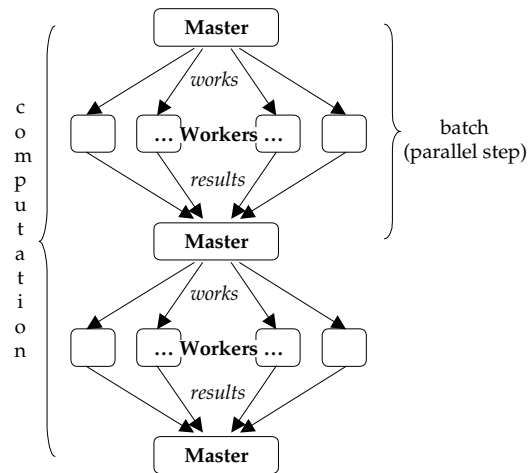


Figure 4-1: The master-worker model of computation.

ing idle workstations), as well as the most successful true volunteer networks today such as distributed.net [40], GIMPS [158], and SETI@home [137].

While some of these systems, such as Condor, provide a general-purpose programming interface, many of these, especially the most popular true volunteer computing systems, are *ad hoc* systems that are written for specific problems and do not provide an easy way for programmers to use their system for other problems. Moreover, the problems that these systems have been tackling so far have mostly been limited to esoteric problems that do not have direct practical use in real life. For example, although the authors of distributed.net have successfully applied their technology beyond cracking RC5, their other projects are similarly obscure mathematical problems such as finding Optimal Golomb Rulers [43]. Thus, someone hearing the term “volunteer computing” today would probably associate it with projects that crack codes, compute π , or search for prime numbers, and would not think that it can be used for practical applications.

In this chapter, we show that this is not the case. Using Bayanihan’s runtime system and APIs for master-worker applications (described in Sect. 3.4.1), we show that there are a wide variety of useful applications that can be implemented under the master-worker model, including brute-force search, image rendering, genetic algorithms, communicationally parallel applications, parametric analysis, and Monte Carlo simulation. Since most, if not all, volunteer computing systems today already have mechanisms for running master-worker style applications (even though they may just use these mechanisms for only one specific application), then these applications represent what these systems can do if their developers provide a general-purpose API for them like Bayanihan’s. Thus, in presenting these applications, we not only demonstrate the applicability of Bayanihan itself, but that of volunteer computing systems in general as well.

In each of the following sections, we present a *class* of master-worker applications, and a few example applications in that class. Although the example applications may not be practical by themselves in some cases, they are meant to represent other, more practical, applications in the same class that can be implemented with the same coding patterns (or

in some cases, using an application framework we have already developed for that class of applications). Similarly, although we present performance results to show that we actually achieve useful speedups in these applications, the focus is not on the performance but on the breadth of applicability. In many cases, performance can probably be improved if these applications are implemented on other systems, especially native-code based ones. The more important point is the fact that these other systems can be used for these applications as well.

4.2 Brute-Force Search

One of the simplest classes of master-worker applications are *brute-force search* problems. These are applications where there is a large search space of possible solutions, and we simply examine all the possible solutions in parallel until we find a solution.

The RC5-64 cracking application run by distributed.net is an example [44]. There, they divide the range of 2^{64} possible keys into subranges and give a different subrange to each worker. Each worker tries to decrypt the target message with each key within the subranges it is given until it finds a key that works (RSA has given the first few characters of the decrypted message, so that people can tell if a decryption is successful or not). The OGR-25 project, also by distributed.net, is another brute-force search application [43]. Many other applications fall under this category.

Applications in this category, have several advantages. For one, they are simple to implement because for most applications, the master does not need to do anything except partition the search space and wait for a worker to find the desired result. This makes it possible to implement them even in systems that do not support performing arbitrary computations between batches on the master node. Many of these applications are also efficient because they require very little data transferred from master to worker and back for a large amount of computation. For example, in the RC5-64 application, the master need only transmit the beginning and end of the range, and receive a single boolean result (i.e., “key found” or “key not found”).

Another advantage of these applications is that *positive* results are easily verifiable. In the RC5 application, for example, if a worker claims to have found the right key, the master can easily check this claim by using the key to decrypt the message. (The RC5 challenge organizers give out the first few characters in the message to allow crackers to verify that they have the right key.) This makes it possible to implement a simple form of fault-tolerance based on eager scheduling, as described in more detail in Sect. 6.2.1.

In the rest of this section, we present two examples: brute-force factoring and RC5. Since this class of applications is already well-represented in existing volunteer computing systems, our goal in presenting these applications here is not so much to show new applications, but to present performance results that demonstrate the effectivity of Bayesian and Java for these applications.

4.2.1 Example: Brute-Force Factoring

As a toy example, we have used the Bayesian framework to write a simple application that factors a Java long integer N by dividing the search space $\{1, \dots, \sqrt{N}\}$ into fixed-

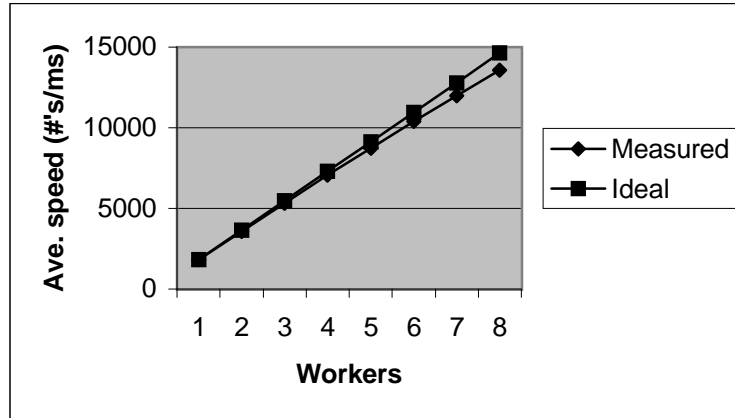


Figure 4-2: Brute-force factoring: Speedup plot.

Table 4.1: Brute-force factoring: Speedup with 8 workers.

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

sized work packets which are executed by the work engines. This example, inspired by [154] (one of the first applet-based volunteer computing applications), although somewhat unrealistic, was chosen because its simplicity and predictability make it easy to measure performance and analyze results.

Figure 4-2 shows some results for $N = 1234567890123456789$, with 112 work packets of size 100000000 each. We used five identical 200 MHz 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 rate (i.e., numbers checked per millisecond) at which work packets were processed, 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 to make use of the two processors in each machine. 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 4.1.

Table 4.2: Brute-force factoring: comparing C and Java speeds.

version	num / ms
C	1879
Java	1828

To evaluate the performance of Java, relative to C, we wrote a simple C program, compiled it with `djgpp` [37] (a `gcc` implementation for MS-DOS). Comparing its performance with the one-worker pure computation performance, we get the results shown in Table 4.2. 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.

4.2.2 Example: RC5 Decryption¹

As a more realistic example, we implemented our own version of an application for solving the RSA RC5-64 on Bayanihan. The efficiency of this applications was similar to that of factoring, achieving over 95% efficiency with 10 processors. The performance relative to native code, however, was not as good as in the case of brute-force factoring, being 8 times slower than `distributed.net`'s version. This is still notable, though, 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.

4.3 Image Rendering and Processing

Another useful class of master-worker applications are image rendering and processing applications, wherein the final value or color of a pixel is determined through a long independent computation. Examples of these include ray-tracing of scenes for 3D animation. It also includes image processing applications, as described in [157]. These applications have great potential for actual commercial use, particularly in the multimedia industry. They can also be useful for scientific use, in particular to aid in the visualization of models and data, as well as in the processing of image data.

Another useful property of applications in this class that make them particularly appropriate for volunteer computing is that they are *naturally fault-tolerant* (see Sect. 6.2.1). That is, even if a relatively large percentage (e.g., a few percent) of the pixels end up with erroneous values, these would be unnoticeable to the user if they are scattered throughout the frame and averaged out. Furthermore, if they become noticeable, then the user can identify the noticeable errors and ask to have them be redone.

4.3.1 Example: Interactive Mandelbrot Set Renderer

As an example, we implemented an application for rendering images of the Mandelbrot set [38]. Like raytracing, this involves performing a long computation independently for each pixel. A unique feature not found in other volunteer computing systems that also demonstrate raytracing and Mandelbrot set rendering, such as [23, 12, 113], is an *interactive* user-interface that allows users not only to view results as they are received from other workers, but also to control the computation *in real time* through a watcher applet. Figure 4-3, for example, shows how computed blocks are displayed in the watcher applet as they are received, without waiting for the whole picture to be done. Borders around the blocks,

¹This is joint work with Lydia Sandon [130].

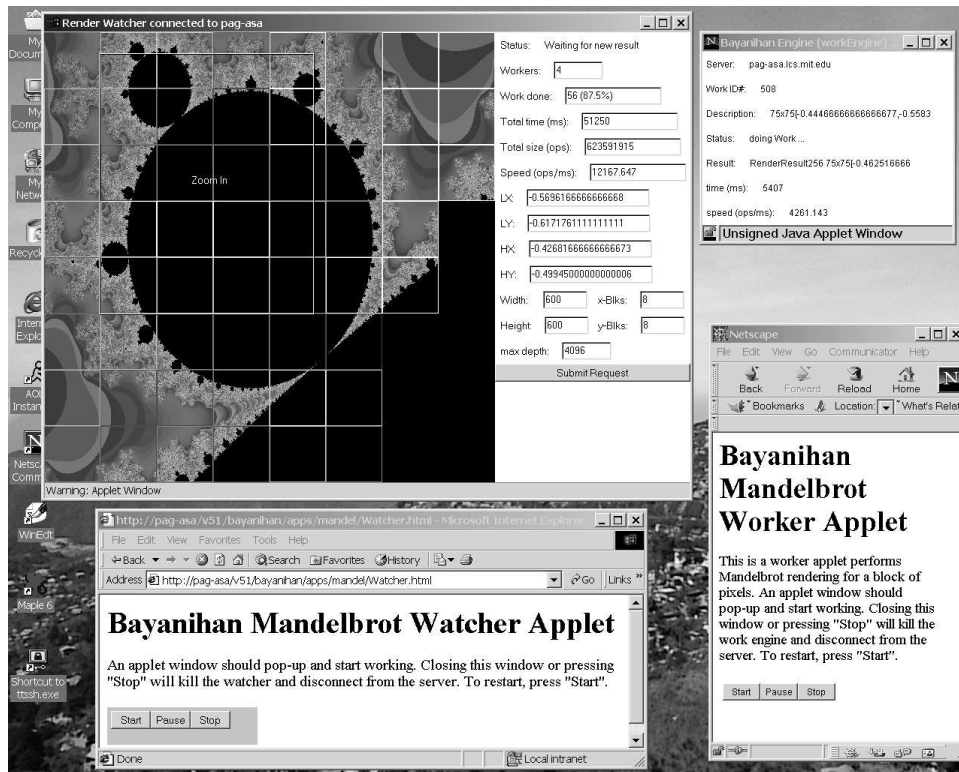


Figure 4-3: A screenshot of the Mandelbrot application in action.

with colors corresponding to different workers, give users a sense for the parallelism in the computation. Furthermore, using the request mechanisms described in Sect. 3.4.1, the GUI also allows users to zoom in and out of selected areas or recompute the problem with different parameters, even if the current picture is not yet completely plotted. In other applications, this feature can be used to provide live interactive control for visualization applications. For example, it could allow scientists to manipulate 3D images of molecular models or MRI data in real-time.

Performance Results. Figure 4-4 shows performance results from running the Mandelbrot application on 17 200 MHz Pentium Pro worker machines (1 server and 16 workers)² 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, commu-

²These are different and faster per processor than the 200 Mhz dual-Pentium (not Pro) machines used in Sect. 4.2.1 so the absolute speeds here are not comparable.

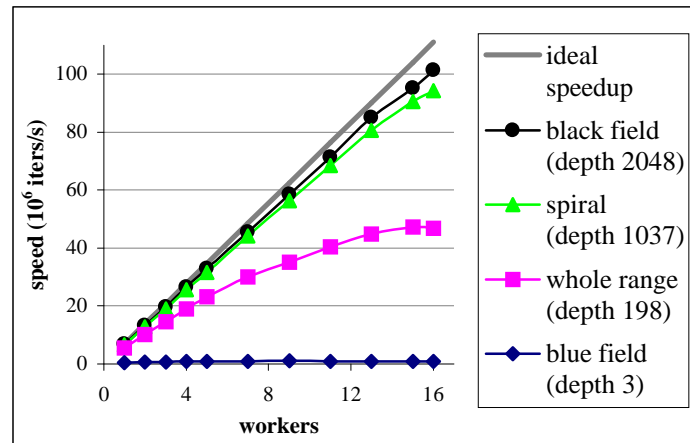


Figure 4-4: Mandelbrot application: Speedup plot.

Table 4.3: Mandelbrot application: comparing C and Java.

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

nication 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.

Table 4.3 shows a comparison of *absolute* speeds from sequential Java and native C executions of the same Mandelbrot code. Again, as in Sect. 4.2.1, 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` [37] using the `-O` option that produced the best result).

4.4 Genetic Algorithms³

Genetic algorithms [72, 98] are algorithms that solve search and optimization problems through the biological principle of natural selection. These algorithms function by maintaining a population of chromosomes and performing operations on that population, simulating the biological evolutionary process. Each chromosome encodes a possible solution to the problem being solved. By manipulating the chromosomes much like strands of DNA – through combination, crossover, and mutation – the genetic algorithm produces new chromosomes from the old ones. By selecting the chromosomes with the best solutions

³This section is joint work with Eamon Walsh [155].

```

Initialize a (random) population.
test for termination criteria (time, fitness, etc.)
while not done do
  give each chromosome an evaluation (fitness)
  select parent chromosomes for reproduction
  produce children through genetic recombination
  perform other genetic operations (mutate, etc.)
  evaluate the children
  select survivors to advance to the next generation
end loop

```

Figure 4-5: Pseudo-code for a basic genetic algorithm.

from successive generations, the genetic algorithm eventually reaches a final, optimized result.

Genetic algorithms have many real-world applications. Their high degree of adaptability and versatility makes them ideal for problems with sudden, unexpected, or complex conditions. They have many possible applications such as constructing schedules and timetables, managing network routing, finding maxima and minima of complex functions, playing complex games, and modeling complex systems in general including biological, ecological, economic, and social systems [79, 98]. And, because they perform several operations on thousands of independent chromosomes in a cycle, they can easily benefit from parallelization (see [157] for some examples).

In addition, genetic algorithm applications are also of particular interest to our research because their *self-correcting* nature can give them a certain degree of immunity from sabotage. That is, since any non-desirable solutions returned by faulty processors or saboteurs will be screened-out through natural selection, then at worst, saboteurs can only make the algorithm take longer by introducing undesirable solutions into the system. At best, they may even inadvertently introduce benevolent mutations that will lead to the desired solutions more quickly.

Using Bayanihan, we have developed an application framework for parallelizing genetic algorithms. We describe it here, and describe the results of using it for a toy example involving finding the maximum point of complicated 2D functions.

4.4.1 Framework Design

Figure 4.4.1 shows pseudo-code describing the work cycle of a basic genetic algorithm [79]. To implement a genetic algorithm, using the framework, a programmer needs to provide the following information by extending existing classes in the framework:

- Format for encoding information in the chromosomes
- Population size
- Number of generations before stopping
- Evaluation function for scoring the chromosomes

	1 worker	4 workers	Speedup
Function 1	2253	662	3.40
Function 2	2257	683	3.30
Function 3	2270	711	3.19

Table 4.4: Genetic Algorithm application: Running times (in ms) and speedups.

- Reproduction operators (mutation, crossover, etc.)
- Survival function for performing natural selection

Once this information is defined, the framework executes the genetic algorithm in parallel. As the algorithm executes, the workers execute the code within the loop while the master takes care of partitioning the chromosome population into work objects, collecting the new chromosomes as results, and repartitioning them again for the next cycle.

4.4.2 Example: Function Optimization

We used this framework to write a sample genetic algorithm and run it on the Athena student network at MIT. The genetic algorithm finds the absolute maximum value of a function $f(x, y)$ by maintaining a population of chromosomes which contain encoded x and y values. The genetic material is a string of 44 bits, which is divided into two 22-bit numbers. The numbers are scaled from the range $[0, 22^2 - 1]$ to the range $[-100, 100]$ and used as inputs to the function. The score of a chromosome is equal to its function value. Thus, since the highest-scoring chromosomes come to dominate the population, the population eventually evolves to the point where a maximum is found.

Two trial runs were performed. In the first trial, the algorithm was run with a single worker client on one machine. In the second trial, four worker clients on separate machines were used. In each trial, the algorithm was given three functions to maximize. After five generations, the highest-scoring chromosome in the population was returned and its score recorded. The algorithm succeeded in finding the maximum value of the three different functions with an error of less than 0.2% in all cases. In the first trial, one generation was finished approximately every 2260 ms. In the second trial, one generation was finished approximately every 685 ms. Although network overhead caused a loss in performance during the second trial, the four machines working in parallel were still over three times faster than the single machine.

4.5 Communicationally Parallel Applications

If restricted network access (as in the case of applets) is not a concern, then volunteer computing can also be used not only for multiplying computational power, but *communication* power as well. This opens up the possibility for many interesting new applications.

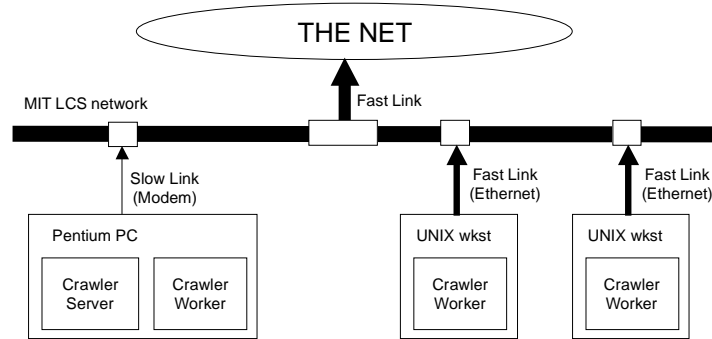


Figure 4-6: The distributed web-crawler experiment.

4.5.1 Example: Distributed Web Crawling

Consider, for example, 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*. That is, 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 Bayanihan, and used it in the setup shown in Fig. 4-6. 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 28.8 Kbps modem.⁴ 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 Sun’s `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 than on the notebook) but faster network links (i.e., 10 Mbps Ethernet).

Table 4.5 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 suppressed 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 the work manager objects, and created `MultiWork` versions as described in Sect. 3.5.1 that allow the worker to prefetch several packets of work at the same time so that it can do new work while simultaneously reporting the results of previously done work. As Table 4.5 shows, this resulted

⁴This experiment was done in 1997, when a 166 Mhz Pentium was not so slow, and UNIX versions of the Java virtual machine did not use just-in-time compilation.

Table 4.5: Distributed web-crawler: Timing measurements

measurement	PC-mdm	UNIX-Ethernet					
		Basic			MultiWork		
		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

in up to 80% more speedup over the basic version. Furthermore, the new `MultiWork` 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.

4.5.2 Other Possible Applications

Another communicationally parallel application that can be implemented through volunteer computing is *distributed web benchmarking*. In web benchmarking, the goal is to measure the average delays involved in accessing a particular web site by trying to access it from various points on the Internet and gather timing information. Some companies such as Keynote Systems [83] provide such a benchmarking service. Volunteer computing may be able to improve on such services by providing more widespread coverage (since volunteers can come from all over the Internet). We have not implemented distributed web benchmarking on Bayanihan, but SolidSpeed Probestor [142] is starting to do so using Entropia's volunteer network [45].

4.6 Parametric Analysis and Monte Carlo Simulations

One of the most promising classes of applications for volunteer computing today is *parametric analysis*. Parametric analysis applications are ones wherein we run a large number of independent computationally intensive *sequential* computations, each with a slightly different set of input parameters. This allows us to study the effects of the different parameters on the output value of the computation, allowing us to find combinations of parameter values that would lead to optimum output values, or to derive general relationships that let us analytically predict the outcome given arbitrary input parameters.

Parametric analysis applications probably have the greatest potential for benefitting from volunteer computing among all applications today. This is so not only because these applications are typically coarse-grain and easily parallelizable and thus would run efficiently on volunteer computing systems, but even more so because they form a natural extension of the simulators that most programmers write today. Since most programmers today do not have access to parallel machines, many simulators today are still written as sequential programs. Programmers use these by feeding them different parameter combi-

nations and taking note of the results for each combination. Parallel parametric analysis offers these programmers a straightforward way to save time by enabling them to run their programs on many machines with many different parameter combinations at the same time. Since no change in the sequential code is needed, a programmer can start benefiting from volunteer computing right away without having to learn how to parallelize their code. In this way, volunteer computing can be used for a wide variety of useful real-life applications.

A particularly promising class of such applications are *Monte Carlo simulations*. These simulations compute certain properties of a hard-to-analyze system by simulating its behavior given many different *random* sequences of events, and analyzing the results. Monte Carlo simulations were used in the Manhattan Project for developing nuclear theory [94], and enjoy many applications today, from physics and medicine (e.g., clinical radiation dosimetry [16]) to finance (e.g., valuation of financial derivatives [116]), and many other applications (see [66] for a good list of current projects). Note that Monte Carlo simulations are simply parametric analysis computations where the parameter that is varied is the random number generator seed that generates the random events within the sequential simulator. Thus, they can be run on any system that supports parametric analysis, and are particularly appropriate for volunteer computing systems.

4.6.1 Writing Parametric Analysis Applications

Bayanihan provides a simple framework for writing parametric analysis applications. To write an application, one needs to define four main types of objects: the work, result, configuration, and program objects. The *work* object contains the *sequential* code for doing one run of the computation with a specific set of parameters. It takes a *configuration* object representing the parameters, and returns a *result* object representing the output of the computation for that particular configuration. The *program* object is responsible for creating a list of parameter configurations that the programmer wants to try, and then adding work objects with each of these configurations to the work pool so that they can be processed in parallel by the volunteer workers. The program is also responsible for processing the results after they have been collected.

In this way, we can parallelize the computation by letting different parameter configurations be run on a different machines at the same time. In a sense, the other applications described in this chapter can also be considered simple forms of parametric analysis. For example, in brute-force factoring and RC5, the parameters that are varied are the range boundaries of the subspaces to be searched. The Mandelbrot set application can also be seen as an interactive parametric analysis computation where the parameters consist of the block boundaries. In general, however, parametric analysis applications are not limited to these, but can include simulators such as the fault-tolerance simulator to be presented shortly.

4.6.2 Writing Monte Carlo Applications

On top of this basic framework for parametric analysis, we also provide an API for writing Monte Carlo simulations. This API provides the following features:

- **Random number generator classes.** These classes, adapted from the Colt project [73], provide random number generators that have better statistical properties and are more well-suited to Monte Carlo simulations than the built-in Java `Random` class. One of the classes taken from Colt is a random seed generator, that can be used to generate sequences of non-correlated random seeds to be given as initial parameters to each work object. This is crucial in *parallel* Monte Carlo simulators since these simulators require not only that the random number generators within each work object have good pseudo-random properties, but also that the random numbers generated in each work object are independent from those in other objects.
- **Statistics summary class.** This class, `DStat`, has an `addSample()` method that takes samples in the form of a `double` or another `DStat` object, and efficiently computes running values of statistics such as the mean, standard deviation, and maximum and minimum values as each sample is added. This makes the task of collecting results from each work object simple. The standard deviation statistic that this computes can also be used as an indicator of the precision of the results so far, and can be used to determine whether the results are good enough or whether more samples need to be taken.
- **Parallel methods.** By extending the `ActiveMCPProg` object, Monte Carlo programs can make use of methods that automatically create the work objects and collect statistics given the desired work object and initial parameters. This makes the Monte Carlo program code very simple and readable. In particular, the API provides a `doBatch()` method, which takes a work class object and a configuration object, and allows us to run many different instances of the work class in parallel using the same parameter configuration but with different random number seeds, and collect the results through a *collector* object that extracts information from result objects generated by work classes and places them in corresponding `DStat` objects.

4.6.3 Example: Computing π

Section A.2.4 in App. A presents the code for the classic Monte Carlo toy example used in parallel computing literature [63, 157] – computing the value of π . In this application, we compute the value of π by randomly generating points within a 2×2 square, and counting how many of those fall within the unit circle (i.e., with radius 1) inscribed within the square. Since the area of the unit circle is $\pi r^2 = \pi$ while the area of the square is 4, then the fraction of points that fall within the square gives us $\pi/4$, which, multiplied by 4, gives us π .

Although this is not a practical algorithm for computing π (the amount of work required to get the next decimal digit of precision grows exponentially), it is presented here as an example of how Monte Carlo applications work and how they can be coded in Bayesian.

4.6.4 Example: Fault-Tolerance Simulator

Overview

A more practical example is `ftsim`, the fault-tolerance simulator that we use in Chap. 6. This application, described in more detail in Sect. 6.5.1, is both a parametric and Monte Carlo simulator. As a Monte Carlo simulator, it computes the average expected error rate of a volunteer computing system for specific values of parameters such as the faulty fraction (f), sabotage rate (s), and credibility threshold (ϑ), by running multiple instances of the simulator with the same parameters but different random number seeds. At the same time, it allows us to do parametric studies by varying the different parameters and observing the effect on the error rate.

In the plots in Sect. 6.5.2, for example, each point represents the result of 100 Monte Carlo simulations run in parallel using the same combination of parameters, and gives us the expected performance (i.e., error rate and slowdown) of the system for a particular combination of f , s , and ϑ , averaged over these simulations. At the same time, the plots themselves represent the result of running *many* such *sets* of Monte parallel Carlo simulations, and allow us to see how the error rate and slowdown vary with these different parameters. In this way, the fault-tolerance simulator has enabled us to verify our theories and make valuable new discoveries.

In Sect. A.3.1, we show sample code for a fault-tolerance experiment. Here, each call to `doBatch()` (within `doSScan()`) runs a parallel Monte Carlo simulation using a particular work class and configuration. After the call to `doBatch()` the collector object contains statistics such as the error rate and slowdown, which we use to plot one point in the plots in Sect. 6.5.2. To get the whole plot, we run many such parallel Monte Carlo simulations with various parameter combinations, and record the statistics in a file.

Performance

Since each point in these plots requires 100 simulations of 10 batches of 10,000 work objects each, these simulations can take hours to plot a single point running on a single machine. Thus, parallelizing the application was really necessary in this case.

In generating the results in Sect. 6.5.2, we used a variety of machines within the MIT campus network running on different operating systems. These included Windows PCs (a 166 MHz Pentium Pro, 180 MHz and 200 MHz dual-Pentium machines, 350 MHz Pentium machines, and others), Linux PCs (800 MHz Pentium III machines), and Sun workstations (Sun Sparcstation 5 and Ultra 5 workstations). Because the machines had different speeds, speedup in this case was not clearly defined. Figure 4-7 shows the speedup of the system *relative to the fastest machine*. To measure this speedup, we measured the speed of each individual machine by having it report, for each work object it does, the amount of local computation time required (not counting communication time) and the number of operations done during that time. (In this case, the latter was measured as the number of simulated calls to `getWork()` in the simulator.) At the end of the batch, the server then goes through all the results submitted by the worker and computes the total number of operations and the total time. Dividing these gives us the average speed of each machine for that batch. Finding the fastest machine and comparing its speed to the actual throughput

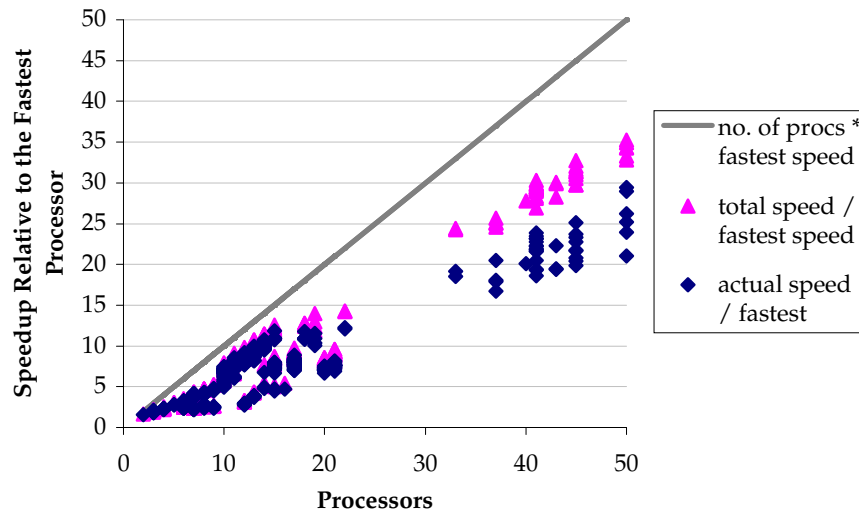


Figure 4-7: Fault-tolerance simulator: Speedup relative to the fastest machine.

at the server (measured as the total number of operations for the batch over “wall clock” time at the server), we get the speedups indicated by diamonds in Fig. 4-7.

As shown, we get more speed as we add more processors. Apart from that, however, it is difficult to draw any conclusions about the performance of the system. Since some of the machines are slower, we cannot expect to get P speedup for P processors (as we normally do in homogeneous systems) – even under ideal conditions. Thus, to get a better idea of how much speedup we can expect, we took the total of the individual speeds of the machines at the end of the batch. As shown by the triangles in Fig. 4-7, this was less than the number of processors, as we expected, but more than the actual net speed measured at the server. Given this total speed, we can get a better measure of the performance of the system by calculating its *efficiency* as the actual speed over the total speed. Figure 4-8 shows a scatterplot of the efficiencies we observed across different batches and runs of the simulator with a varying combination of machines, while Fig. 4-9 shows an equivalent plot with the *normalized speedup*, computed by multiplying the efficiency by the number of processors.

As shown, we get reasonably good efficiency, even at 50 processors. Interestingly, the losses in efficiency in this case are *not* due to communication and server-side overhead, as we might at first suspect. The simulations shown here were fairly coarse-grain computations – i.e., the average time taken by the fastest computer for each work object (i.e., sequential Monte Carlo run) was between 2 and 200 seconds with a median of around 20 seconds (the times varied because some parameter combinations resulted in longer or shorter simulations). We suspect that most of the inefficiencies are instead due to a limitation of eager scheduling that applies when the work batches are not big enough. In this case, we had only 100 work objects distributed among up to 50 workers. This caused fast workers to quickly run out of unassigned work to do, and start doing work already assigned to slower workers – possibly finishing the work before they do. While this is a good feature that makes sure that slow processors do not slow down the faster processors

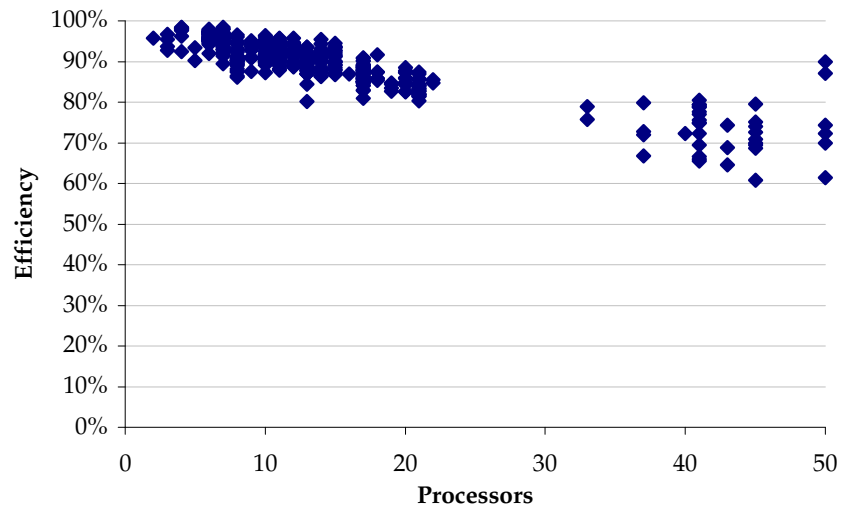


Figure 4-8: Fault-tolerance simulator: Efficiency.

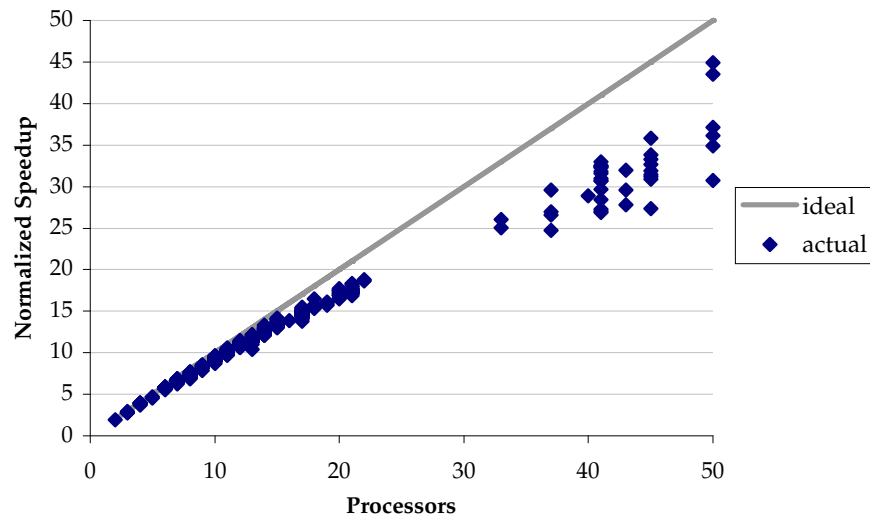


Figure 4-9: Fault-tolerance simulator: Normalized speedup.

(i.e., in the worst case, all the work will be done in the time it would take the faster processors to do the job by themselves, as if the slower processors were not there), it does have the drawback of wasting the processor power of the slower processors. Thus, since this processing power is accounted for in the total speed, the fact that it was used for work which was redone by a faster processor shows up as a loss in efficiency.⁵

In general, the way to get around this problem and get more efficiency is to have longer batches. In this case, one solution might be to provide a `doMultiBatch()` method in the Monte Carlo API that would take multiple pairs of configuration and collector objects, and allow Monte Carlo simulations with more than one set of parameter values to run in parallel in the same batch. Suppose for example, that we want to see the effects of changing s from 0.1 to 0.2 to 0.3. Instead of having to run a sequence of three batches of 100 work objects each as we do now, we may run a single batch with 300 work objects instead, and thus improve efficiency.

In any case, however, even as it is, the fault-tolerance simulator has already proven itself as a real example of volunteer computing's great potential for use in real-life applications. Even with its slightly suboptimal efficiency, it has served as an invaluable *enabling tool*, that demonstrates the ability of volunteer computing to empower people to conduct previously impossible research.

4.7 Conclusion

In this chapter, we have shown how volunteer computing is not limited to esoteric mathematical problems but can easily be applied to a wide variety of useful applications if we can provide even simple general-purpose APIs for writing master-worker applications like Bayanihan does. Recently, other volunteer computing systems have also started providing such general-purpose APIs and using them for a wide range of practical applications. One of the most notable of these is Parabon [113], whose Java application-based Frontier programming platform is currently being applied to several classes of applications including most of those we have discussed in this chapter (i.e., brute-force search, rendering, parametric analysis, and Monte Carlo simulation), and others, such as DNA sequence analysis, and exhaustive regression [115].

Other applications that we have not discussed in this chapter but also have potential for benefitting from volunteer computing include *branch-and-bound* computations and *data mining* applications.

Branch-and-bound computations are like brute-force search computations but are more efficient in that they do not necessarily search the entire space but cut down the search space as they go along by identifying subtrees of the search that cannot possibly produce results better than those already found. These have applications in optimization problems,

⁵In our current implementation, if a slow worker is unable to finish even *one* work packet before the batch ends, then its individual speed might not be included in the total speed for that batch. This may result in efficiencies that seem higher than they actually are. Although this problem needs to be fixed in future versions, we believe that it does not affect the results shown here that much. The fact that a worker was so slow that it could not do one work object in the time it took the whole system to do 100 work objects means that its contribution to the total speed would have been relatively small anyway (i.e., at most around 1%), and thus would not have affected the efficiency computation by much.

as well as in playing strategy games such as chess. Javelin++ demonstrates the implementation of branch-and-bound on a Java application-based volunteer computing system with an example program that solves the traveling salesperson problem [105].

Data mining applications are those wherein we process a large amount of data in order to find or deduce useful patterns in the data. SETI@home and Paragon's DNA sequencing applications are examples of data mining applications done using volunteer computing. Other examples of data mining not yet implemented with volunteer computing include finding patterns in customers' buying records that may help guide marketing strategies (e.g., customers who buy product *A* are likely to buy product *B* [110]), and many others [122, 140]. These applications have the potential of bringing the benefits of volunteer computing beyond scientific applications to commercial applications as well.

For future work, it would be useful not only to implement more examples of applications under the classes already described here, but to find other classes of applications that can benefit from volunteer computing systems as well.

Chapter 5

BSP as a Programming Model for Volunteer Computing

5.1 Introduction

While the master-worker model covers a large number of real-world applications, these represent only a fraction of the variety of parallel applications that users may want to run on volunteer computing systems. In particular, most parallel applications today, such as those written with PVM and MPI, assume a message-passing model. Unfortunately, however, the need for adaptive parallelism and crash-tolerance makes implementing these applications on volunteer computing systems difficult. The MPI model for example, assumes that the number of physical processors is known and fixed. MPI provides little or no support for situations where nodes leave or join the system at unexpected times. PVM provides ways to detect nodes that leave or crash, but requires programmers to deal with these explicitly and implement their own way of reassigning work to other workers.

In this chapter, we present Bayesian BSP, a new programming interface based on the *bulk synchronous parallel* (BSP) model [150, 139], that makes it possible to write message-passing style parallel programs while at the same time allowing us to take advantage of the adaptive parallelism and fault-tolerance mechanisms we have developed for master-worker style systems.

5.2 The BSP Programming Model

In BSP, a parallel program runs across a set of virtual processors – called **processes** to distinguish them from physical processors) – and executes as a sequence of parallel *supersteps* separated by barrier synchronizations. Each superstep is composed of three ordered phases, as shown in Fig. 5-1: (1) a *local computation* phase, where each process can perform computation using local data values and issue communication requests such as remote memory reads and writes; (2) a *global communication* phase, where data is exchanged between processes according to the requests made during the local computation phase; and (3) a *barrier synchronization*, which waits for all data transfers to complete and makes the transferred data available to the processes for use in the next superstep.

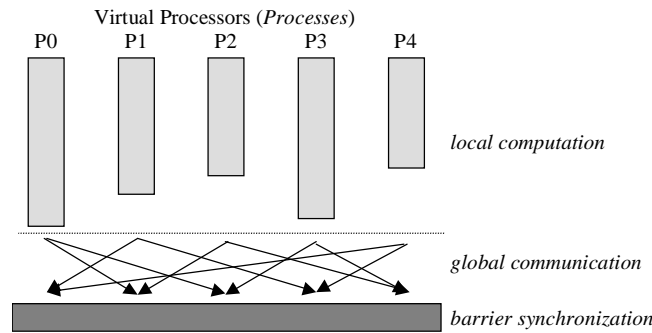


Figure 5-1: A BSP superstep.

```

1 void shift2()
2 { right = ( bsp_pid()+1 ) % bsp_nprocs();
3   s = 0; d = bsp_pid();
4   bsp_register( "d",&d ); //register variable "d"
5   // begin superstep 0
6   bsp_put( right,"d",d ); // put my pid to right's d *at next sync*
7   s = d; // s <- current d (i.e., mypid )
8   bsp_sync(); // synchronize
9   // begin superstep 1; d is now left's pid
10  bsp_send( right, d ); // send d to right * at next sync *
11  s += bsp_rcv(); // error! nothing to receive yet!
12  bsp_sync(); // synchronize
13  // begin superstep 2; msg is now in rcvq.
14  d = bsp_rcv(); // receive d sent in line 7
15  s += d; // s <- s + 2nd-left's pid
16 }

```

Figure 5-2: BSP pseudo-code for a two-step cyclic shift.

Figure 5-2 shows pseudo-code for a simple program where each process passes data to its right neighbor and performs a little local computation. As shown, BSP is very similar to conventional SPMD/MPMD (*single/multiple program, multiple data*)-based programming models such as MPI, and is at least as flexible, having both remote memory (e.g., `bsp_put()`) and message-passing (e.g., `bsp_send()` and `bsp_rcv()`) capabilities. The *timing* of communication operations, however, is different – since the global communication phase does not happen until all processes finish the local computation phase, the effects of BSP communication operations are not felt until the *next* superstep. In line 6 of Fig. 5-2, for example, `s` gets the *unchanged* local value of `d`, even though a `bsp_put()` operation is called in line 5. Similarly, in line 9, the call to `bsp_rcv()` returns null or generates an exception.

This postponing of communications to the end of a superstep is the key idea in BSP. It removes the need to support *non-barrier* synchronizations between processes, and guarantee that processes within a superstep are *mutually independent*. This makes BSP easier to implement on different architectures than traditional models, and makes BSP programs easier to write and to analyze mathematically. For example, since the timing of BSP com-

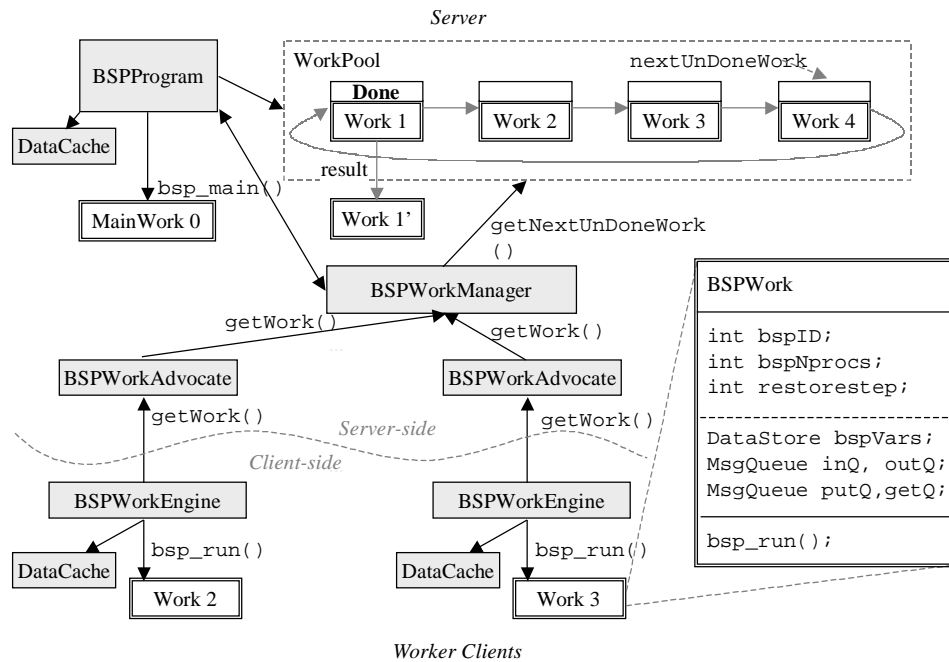


Figure 5-3: Implementing BSP on top of the Bayanihan master-worker framework.

munications make circular data dependencies between BSP processes impossible, there is no risk of deadlock or livelock in a BSP program. Also, the separation of the computation, communication, and synchronization phases allows one to compute time bounds and predict performance using relatively simple mathematical equations [139].

Because of these advantages, a growing number of people are now applying BSP to a wide variety of realistic parallel applications, ranging from small general-purpose numeric libraries (e.g., for matrix operations), to large computational research applications used in both academia and industry (e.g., computational fluid dynamics, plasma simulation, etc.) [20]. By implementing BSP in Java, we hope to enable programmers to port these applications to Java, and make it easier to do realistic research using volunteer computing systems.

5.3 Implementing the BSP Model

5.3.1 The Bayanihan Master-Worker Based Implementation

BSP's structure makes BSP relatively easy to implement in volunteer computing systems. Since processes are independent within supersteps, we can package the work to be done in a superstep as separate work objects and farm them out to volunteer worker nodes using a master-worker system. Figure 5-3 shows how we have implemented BSP using the existing Bayanihan master-worker runtime system described in Sect. 3.4.2. As shown, BSP processes are implemented as `BSPWork` objects and placed in a work pool on the server.

When the work engine receives a new work object, it calls the work object's `bsp_run()` method, which starts the local computation phase of the superstep indicated by the `restorestep` field. As `bsp_run()` executes, communication requests are logged in `MsgQueue` fields, and the values of shared variables are saved in the `bspVars` field, a hash table that stores objects under `String`-type names. Execution continues until `bsp_sync()` is called, at which point `restorestep` is incremented and control returns to the work engine. The work engine then sends the whole work object – including `bspVars`, the `MsgQueues`, and `restorestep` – back to the server, which in turn marks the original work object “done”, and stores the returned work object as its result own result.

When all work objects are done, the work manager notifies the `BSPProgram` object, and the global communication and barrier synchronization phases begin. The `BSPProgram` object goes through the result work objects, and performs all the global communication operations by moving data from one work object to another. The remote memory access operations (`bsp_get()` and `bsp_put()`) are done by reading from and writing to the work objects' `bspVars` fields. Similarly, the message-passing style `bsp_send()` operation is performed by enqueueing the message data into the destination node's `inQ` field, so that in the local computation phase of the next superstep, they can be retrieved by calling `bsp_recv()`. After all the communication operations have been performed, each work object in the work pool is replaced by its updated version, and control returns to the work manager. The next superstep begins as the work manager starts giving workers new work objects wherein `restorestep` indicates the new superstep, `bspVars` contains updated variable values, and `inQ` contains new incoming messages.

In addition to the work objects in the work pool, the server also has a special *main work* object, which is like other work objects, except that it is not farmed out to workers, but is run on the server itself. In place of `bsp_run()`, the server calls the `bsp_main()` method. As described in Sect. 5.4.1, the main work object can be used for such tasks as spawning new work objects, coordinating other work objects, collecting and displaying results, and interacting with users.

5.3.2 Adaptive Parallelism and Fault-Tolerance

Since no communications occur while these work packets are running, we can rerun work packets or run several copies of the same work object at a time, resolving any redundancies safely at the end of the superstep. This makes it possible to implement adaptive parallelism and fault-tolerance mechanisms as described in Sect. 3.5.2 and Chap. 6.

In addition, implementing the BSP processes as work objects makes *checkpointing* easy. Previous research in C-based BSP systems has shown that the call to `bsp_sync()` at the end of the superstep provides a convenient place to checkpoint the program state and migrate processes to achieve adaptive parallelism and crash-tolerance [108, 67]. In our implementation, checkpointing happens automatically as each work object returns itself to the server at the end of a superstep. Since the returned work object includes all necessary process state in `bspVars`, the `MsgQueue`'s, and other non-transient fields, this state can easily be saved and restored later or migrated to another machine.

Note that this is in strong contrast to traditional message-passing APIs such MPI, where adaptive parallelism, fault-tolerance, and checkpointing are hard to implement. In such

systems, it is difficult or impossible to rerun processes because messages that a process has already sent out to other processes cannot be taken back. Redundant process also need to be explicitly coordinated somehow. Finally, checkpointing is difficult because different processes can be at different stages in the computation at the same time, since there are no convenient synchronization points as in BSP.

5.4 Writing Bayanihan BSP programs

To write a Bayanihan BSP program, one simply extends the `BSPWork` base class and defines an appropriate `bsp_run()` method containing the BSP program code. No changes to the Java language or Java virtual machine (JVM) are required.

5.4.1 The Bayanihan BSP Methods

The `BSPWork` class provides several methods which subclasses inherit and can call within `bsp_run()` in the same way one would call library functions in C. These are based on `BSPLib` [20], a programming library for C and Fortran currently being used in most BSP applications, but have been modified to account for language and style differences between Java and C. These methods are shown in Table 5.1, subdivided by function into the following categories (in order): *inquiry*, *message passing*, *direct remote memory access*, and *synchronization*.

Message-Passing

The message-passing methods are similar in functionality to those found in systems such as MPI [63] or PVM [54], except that, as noted earlier, messages sent during a superstep can only be received by the recipient in the *next* superstep. In the present implementation, messages are queued at the recipient in arbitrary order, but can be identified by tag using the `bsp_peekTag()` method, or by calling the `getBSPTag()` method of the `BSPMessage` object returned by `bsp_rcv()`. The data itself can be retrieved as an object by calling the `getData()` method of the `BSPMessage`. Any data object that implements the `java.io.Serializable` interface or has a HORB proxy (generated by the HORB compiler) can be passed in a message. Note that Java's object-oriented features and platform-independence make sending and receiving messages much easier than in C-based systems such as MPI or PVM, where one must explicitly decompose complex messages into its primitive components.

Direct Remote Memory Access

The direct remote memory access (DRMA) methods, `bsp_put()` and `bsp_get()`, allow processes to write to and read from remote variables by specifying the owner's process ID and the `String`-type name of the desired variable.

The `bsp_save()` method registers a variable into the `bspVars` table. This allows it to be saved at a checkpoint, and to be read remotely using `bsp_get()`. A variable remains registered until it is removed with `bsp_unregister()`, and its contents can be modified

Table 5.1: Basic Bayanihan BSP methods

method	meaning
<code>int bsp_pid()</code>	my BSP process identifier
<code>int bsp_nprocs()</code>	number of virtual processes
<code>double bsp_time()</code>	local time in seconds
<code>void bsp_send(BSPMessage msg)</code>	send prepared message
<code>void bsp_send(int dest,int tag, Object data)</code>	create tagged message, and send to <i>dest</i>
<code>BSPMessage bsp_recv()</code>	receive message
<code>int bsp_qsize()</code>	number of incoming messages
<code>BSPMessage bsp_peek()</code>	preview next message
<code>int bsp_peekTag()</code>	preview next message tag
<code>void bsp_save(String name,Object data)</code>	save data under <i>name</i>
<code>Object bsp_restore(String name)</code>	restore data under <i>name</i>
<code>Object bsp_unregister(String name)</code>	remove data under <i>name</i>
<code>void bsp_put(int dest,String destName, Object data)</code>	write data to <i>destName</i> variable of <i>dest</i>
<code>Object bsp_get(int src,String srcName, String destName)</code>	read <i>srcName</i> variable of <i>src</i> to local <i>destName</i>
<code>boolean bsp_sync()</code>	sync; possibly checkpoint and migrate
<code>boolean bsp_step()</code>	mark beginning of superstep
<code>boolean bsp_cont()</code>	mark continuation of superstep

freely by its owner using accessor methods. If the object itself is replaced, however, then `bsp_save()` must be called again to update the reference stored in `bspVars`. A variable can also be registered by processes other than its owner by using the `bsp_put()` method. This technique can be used by `BSPMainWork` objects to initialize worker variables.

The `bsp_restore()` method restores the value of a variable so it can be used within a superstep. Variables that are used in all or most supersteps can be restored in the `initVars()` method, which is always called before `bsp_run()`. This guarantees that the variables are always restored and available at the beginning of a superstep. If an attempt is made to restore an unregistered name, `bsp_restore()` returns `null`. Typically, variables restored with `bsp_restore()` are declared as `transient` fields since their values are already stored in `bspVars` and do not need to be serialized independently.

Like the message-passing methods, the DRMA methods do not take effect until the next superstep. Thus, for example, a `bsp_get()` called during a superstep will only update the destination variable in the next superstep. The value received by `bsp_get()` is the value of the source variable at the end of the local computation phase, before any puts or gets are done. Similarly, the value sent by `bsp_put()` is the value of the data object at the end of the local computation phase. This may be different from the value of the data object at the time `bsp_put()` was called, unless the data is first *cloned* before `bsp_put()` is called.

Synchronization and Checkpointing

Calls to `bsp_sync()` not only indicate barrier synchronizations, but also mark potential checkpoint locations where a virtual process may be paused, saved, and moved to a different physical processor. While pausing and moving processes is easily done as described in Sect. 5.3.1, resuming execution is harder. In C, resuming execution can be done relatively easily by either using system-specific low-level mechanisms (as done in [67]), or using a pre-compiler to annotate the source code with a jump table using `switch` and `goto` statements such as done in Porch [123]. Unfortunately, these techniques are not applicable in standard Java, where it is not possible to do a dynamic jump to a variable code location (i.e., one that is determined only at runtime not at compile-time).¹

Thus, instead of using direct jumps, we use `if` statements to *skip* over already-executed code. Figures 5-4 and 5-5 show an example. In general, the code for a superstep should be enclosed in an `if (bsp_step())` block and ended with a call to `bsp_sync()`, as shown in step 0. As a syntactic shortcut, an `if (bsp_sync())` statement can be used to end a superstep and start the next one at the same time, as shown at the end of step 1. In cases where function calls, `for`, `while`, `if-else`, or other special constructs prevent a superstep's code from being enclosed in one block, one can use an `if (bsp_cont())` statement to continue the superstep on the other side of the offending statement as shown.

¹Although it may be possible to do Porch-like `switch` statements in Java, we expect that this would not present a performance improvement since going through the `switch` would require a similar number of steps as going through a chain of `ifs`. What would be useful to have in Java would be a `goto x` instruction, where `x` is a variable, not a constant. If we had this functionality, we could save the current program counter in a variable, and then jump to it upon restoring. This would be similar to the system-specific low-level mechanisms used by other C-based systems. However, we cannot do this in Java.

```

void bsp_run()
{ step 0;
  bsp_sync();
  step 1;
  bsp_sync();
  step 2, part a;
  for ( int i=0; i<n; i++ )
  { step 2+i, part b;
    bsp_sync();
    step 2+i+1, part a;
  }
  // code continued in next column
}

step 2+n, part b;
foo(); // func with sync
step 3+n, part b;
bsp_sync();
} // end bsp_run()

void foo()
{ // cont. current step
  step 2+n, part c;
  bsp_sync();
  step 3+n, part a;
} // end foo();

```

Figure 5-4: BSP pseudo-code without code-skipping.

```

void bsp_run()
  throws BSPSyncException
{ if ( bsp_step() )
  { step 0
  }
  bsp_sync();
  if ( bsp_step() ) // explicit
  { step 1
  }
  if ( bsp_sync() ) // shortcut
  { step 2, part a
  }
  for ( int i=0; i<n; i++ )
  { if ( bsp_cont() )
    { step 2+i, part b
    }
    if ( bsp_sync() )
    { step 2+i+1, part a
    }
  }
  // code continued in next column
}

if ( bsp_cont() )
{ step 2+n, part b
}
foo(); // func with sync
if ( bsp_cont() )
{ step 3+n, part b
}
bsp_sync();
} // end bsp_run()

void foo() throws
  BSPSyncException
{ if ( bsp_cont() )
  { // cont. current step
    step 2+n, part c
  }
  if ( bsp_sync() )
  { step 3+n, part a
  }
} // end foo()

```

Figure 5-5: Transformed BSP pseudo-code with code skipping.

```

protected boolean bsp_sync()                protected boolean bsp_step()
    throws BSPSyncException                { return (curstep>=restorestep );
{ if ( curstep++ >= restorestep )          }
  { restorestep = curstep;
    throw new BSPSyncException();
  }
return bsp_step();
}

```

Figure 5-6: Code for `bsp_sync()`, `bsp_step()`, `bsp_cont()`.

Figure 5-6 shows how these three methods are implemented. When restarting a checkpointed `BSPWork` object, `bsp_run()` is called with `curstep` initialized to 0, and `restorestep` set to the step to be resumed. The `bsp_step()` and `bsp_cont()` functions, called at the beginning of each code block, return `false` while `curstep` has not yet reached `restorestep`, allowing blocks that have already been executed to be skipped. At the end of each skipped superstep, the call to `bsp_sync()` increments `curstep` so that `curstep` eventually reaches `restorestep`, and the desired superstep's block is allowed to execute. The next call to `bsp_sync()` then ends the superstep by updating `restorestep` and throwing an exception which causes `bsp_run()` to return control to the work engine.

As demonstrated in Fig. 5-5, these methods can be used even when `bsp_sync()` calls occur in loops, `if-else` blocks, and function bodies. Moreover, the code transformation rules are simple enough to be applied manually without using pre-compilers. In many cases, all a programmer needs to do is enclose the superstep code in braces, and add an `if` in front of `bsp_sync()`. As shown here, and more clearly in the realistic sample code in Sect. 5.4.2, the resulting code is still reasonably readable. In fact, the indented blocks may even help emphasize the divisions between supersteps. This simplicity offers a significant advantage in convenience over other programming interfaces that require language changes and sophisticated pre-compilers, since it does not require programmers to generate and keep track of extra files, and allows them to more easily use off-the-shelf integrated development and build environments. Furthermore, if more readability is desired, one can implement a parallel language with nothing more than a C-style preprocessor by using `#define` macros to replace `if (bsp_step())` with `parbegin`, `if (bsp_sync())` with `parentbegin`, `bsp_sync()` with `parent`, and `if (bsp_cont())` with `parcont`, as shown in Fig. 5-7.

As far as we know, the only significant disadvantage of this code-skipping technique is that resuming execution inside or after an unbounded or very long loop may take unnecessary extra time because the skipping has to go through all previous iterations of the loop before reaching the current superstep. We expect, however, that it is possible to solve this problem by using a variant of `bsp_cont()` that updates loop variables and `curstep` without having to iterate through the loop.

```

void bsp_run()
    throws BSPSyncException
{ parbegin
  { step 0
  }
  parend;
  parbegin          // explicit
  { step 1
  }
  parendbegin      // shortcut
  { step 2, part a
  }
  for ( int i=0; i<n; i++ )
  { parcont
    { step 2+i, part b
    }
    parendbegin
    { step 2+i+1, part a
    }
  }
}
// code continued in next column

parcont
{ step 2+n, part b
}
foo(); // func with sync
parcont
{ step 3+n, part b
}
parend;
} // end bsp_run()

void foo() throws
    BSPSyncException
{ parcont
  { // cont. current step
    step 2+n, part c
  }
  parendbegin
  { step 3+n, part a
  }
} // end foo()

```

Figure 5-7: BSP pseudo-code with code skipping and #define macros.

Table 5.2: Basic BSPMainWork methods

method	meaning
<code>int bsp_spawn(String class, int n)</code>	spawn <i>n</i> BSPWork's of class <i>class</i>
<code>int bsp_spawn(BSPWork w, int n)</code>	spawn <i>n</i> BSPWork's of same class as <i>w</i>
<code>void bsp_postResult(Result res)</code>	post result for watchers
<code>Object bsp_getRequest()</code>	get user input from watchers
<code>boolean isMain()</code>	am I the main work?

BSPMainWork

A Bayesian BSP program is started by giving the name of a subclass of BSPMainWork to the BSPProgram object. BSPMainWork is a subclass of BSPWork that provides extra methods for server-side control of a computation, as shown in Table 5.2. The BSPProgram object assigns the BSPMainWork the reserved ID of 0, and calls its `bsp_main()` method, which is then expected to spawn work objects of the desired class. As the program runs, the main work object runs together with the other processes, and can communicate with them using BSP methods. In addition to spawning new work, the main work object can be used for coordinating other work objects, for collecting and sending results to interested watcher clients, and for receiving and reacting to user requests. This is similar to the common programming practice in SPMD/MPMD systems like PVM, MPI, and BSPLib of having a “master” or “console” process that handles data distribution and coordinates the operation of all the other processes.

To write an application, one can write separate BSPMainWork and BSPWork classes, or a single BSPMainWork class with separate `bsp_main()` and `bsp_run()` methods. The

latter is useful when the main work object has the same fields as the other work objects, and has the added advantage of keeping all the code in one file. Since `bsp_main()` defaults to calling `bsp_run()`, one can also write a single `BSPMainWork` class with a single `bsp_run()` method containing code for *both* the main work and the other processes. This allows SPMD-style programming, and is useful in cases where the main work code is very similar to the worker code, or where it is useful to see the master and worker codes for each superstep in the same place. The `isMain()` function can be used within `bsp_run()` to let the main work process differentiate itself.

5.4.2 Sample Code

Figure 5-8 shows a matrix multiplication example demonstrating the use of the Bayanihan BSP programming interface. Here, we use a single `BSPMainWork` subclass with separate `bsp_main()` and `bsp_run()` methods. The `initVars()` method is called in both the worker and main work objects before running `bsp_run()` and `bsp_main()`. The algorithm used is based on one of the MPI example programs in [63], where each process i from 1 to n is given a copy of the matrix B and row $i - 1$ of A , and computes row $i - 1$ of the product C .

5.5 Implementation Results

By hiding the previously exposed implementation details of barrier synchronization and process communication from the programmer, the new BSP programming interface makes it much easier to write Bayanihan applications than before. At present, we have successfully used it not only to rewrite existing applications in a more readable and maintainable SPMD style, but also to implement new programs and algorithms such as *Jacobi iteration*, whose more complex communication patterns were difficult to express in the old model.

Unfortunately, while it is now possible to *write* a much wider variety of parallel programs with Bayanihan, such programs may not necessarily yield useful speedups yet since the underlying implementation is still based on the master-worker model. For example, even though one can now use peer-to-peer style communication patterns in one's code, these communications are still sequentialized at the implementation level since they are all performed by the central server. Also, since all work objects are currently checkpointed at the end of each superstep (i.e., workers send back their *entire* process state to the server at each `bsp_sync()`), algorithms that keep large amounts of local state between supersteps will perform poorly as this state would be unnecessarily transmitted back and forth between the server and the worker machines.

Figure 5-9 shows speedup results from some tests using 11 200 MHz Pentium PCs running Windows NT 4.0 on a 10 Mbit Ethernet network, with Sun's JDK 1.1.7 JVM on the server and Netscape 4.03 on the 10 clients. The Mandelbrot test is a BSP version of the original Bayanihan Mandelbrot demo running on a 400x400 pixel array divided into 256 square blocks, with an average depth of 2048 iterations/pixel. The matrix multiplication test is a variation of the sample code in Sect. 5.4.2 modified such that the worker processes get 10 rows of A at a time (to improve granularity) and do not get C back from the main work. It was run using 700x700 `float` matrices. The Jacobi iteration test solves Poisson's

```

public class BSPMatMultMain extends BSPMainWork {
    transient Matrix A, B, C;
    // ... some code omitted ...
    /** called before bsp_main() and bsp_run() */
    public void initVars() // initialize B and C
    { B = (Matrix)bsp_restore( "B" );
      C = (Matrix)bsp_restore( "C" );
    }
    /** run by main work on server */
    public void bsp_main() throws BSPSyncException
    { if ( bsp_step() ) // *** superstep 0 ***
      { // create n-by-n matrices, spawn, send A and B
        A = createSampleMatrix( n ); B = createSampleMatrix( n );
        bsp_spawn( this, n ); // spawn n workers of this type
        for ( int i=1; i <= n; i++ )
          { bsp_put( i, "B", B ); // write B
            bsp_send( i, i-1, A.getRow(i-1) ); // send row of A
          }
      }
      if ( bsp_sync() ) // *** superstep 1 ***
      { // workers compute C; main does nothing
      }
      if ( bsp_sync() ) // *** superstep 2 ***
      { // collect and print results on server
        BSPMessage msg;
        C = new Matrix( n );
        while( (msg = bsp_rcv()) != null ) // get result rows
          { C.setRow( msg.getBSPTag(), (Row)msg.getData() );
          }
        trace( "C = \n" + C ); // print C
        bsp_save( "C", C ); // allow workers to get C
      }
      if ( bsp_sync() ) // *** superstep 3 ***
      { // workers print C
      }
    }
    /** run by worker clients */
    public void bsp_run() throws BSPSyncException
    { if ( bsp_step() ) // *** superstep 1 ***
      { // restored local var B contains matrix B put
        BSPMessage msg = bsp_rcv(); // get row
        if ( msg != null )
          { int i = msg.getBSPTag(); // get row #
            Row a = (Row)msg.getData(); // get row data
            Row c = Matrix.rowMatMult( a, B ); // compute
            bsp_send( MAIN, i, c ); // send row back
          }
        bsp_unregister( "B" ); // we don't need B anymore
      }
      if ( bsp_sync() ) // *** superstep 2 ***
      { // main collects results, get it
        bsp_get( MAIN, "C", "C" ); // get C for next step
      }
      if ( bsp_sync() ) // *** superstep 3 ***
      { // local var C now contains result
        trace( "C = \n" + C ) // print C
      }
    }
  }
}

```

Figure 5-8: Bayesian BSP code for matrix multiplication.

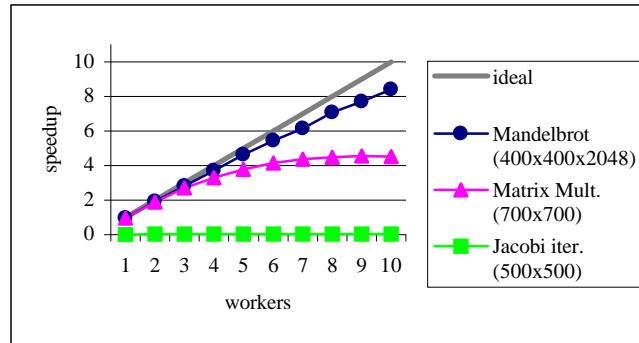


Figure 5-9: Bayanihan BSP applications: Speedup relative to sequential version.

equation on a 500x500 double array decomposed into a 10x10 2D grid, with an exchange of border cells and a barrier synchronization done at the end of each iteration.

As shown, the embarrassingly parallel Mandelbrot demo performed and scaled reasonably well. In contrast, the much finer-grain Jacobi iteration test performed very poorly, being at best about 40 times slower than a pure sequential version. This is not so surprising, though, given that it takes less time for the server to compute an element than to send it to another processor. Although speedups are theoretically still possible if the array is very large and workers are allowed to keep “dirty” local state for a long time, we cannot expect to get any speedup in this case since work objects send all their elements back to the server at the end of each iteration. On a more positive note, the matrix multiplication test, which is more coarse-grain (and happens to employ a master-worker-like communication pattern, although it is written as a message-passing program), performed much better than the Jacobi iteration test, though not as well as the Mandelbrot test. The limiting factor in this case was the time needed to send the matrix B to the worker clients. Since we currently do not support multicast protocols, the server has to send B to workers individually. Thus, the broadcast time grows linearly with the number of worker clients, and eventually limits speedup. In the future, it may be possible to improve performance by using a more communication-efficient algorithm. However, most such algorithms keep local state between supersteps, and will thus probably be inefficient in the current implementation.

5.6 Improving Bayanihan BSP

Our experimental results show that at present, Bayanihan BSP works well mainly for coarse-grain applications that do not require keeping a lot of dirty local state. This limitation, however, is not due to the BSP programming model or interface but is due to our current runtime system implementation. By implementing additional mechanisms at the runtime system level, we should be able to make Bayanihan BSP perform and scale well even for other applications.

5.6.1 Caching

One mechanism that can make a significant difference is *caching*. In the matrix multiplication program in Sect. 5.4.2, for example, the matrix B is sent to all *virtual* processes. Without caching, this would cause B to be sent over the communication network potentially thousands of times. Adding a cache to each work engine as shown in Fig. 5-3 in Sect. 5.3.1, we can save communication time by just giving the virtual processes *references* to B instead of the actual data. Then, when a process gets a reference to B , it first checks the cache for a copy. If it finds one, then it simply uses that copy. Otherwise, it would request the actual data from the server, and then put it in the local cache so it can be used by other virtual processes running on the same machine. In this way, we can send the actual data of B only once for each *physical* processor, regardless of how many virtual processes that processor may run. Using such a scheme, we have achieved the results for the matrix multiplication program presented in Sect. 5.5. (The original version without caching performed much more poorly.)

This same caching scheme can also be extended to allow us to keep dirty local data on the worker machines between supersteps. The idea is to use references in the reverse direction as well. That is, instead of sending the actual state data of a virtual process when we send the result object back to the server, we only send references to this data. The global communication phase (which is executed by the server) is performed by exchanging these references between the virtual processes. As these virtual processes (i.e., the work objects) get farmed out again to the physical machines in the next batch, they bring the references with them. If a virtual process is assigned to a physical machine that has the actual data in its local cache, then the reference is resolved by simply taking the data from the local cache (without communications). Otherwise, the work engine requests the data from the server. If the server has the data in its own cache, it returns it to the requesting worker, just as in our previous scheme. However, if the server does not have the data in its own cache, then it requests it from the worker machine who has it, and then stores it in its cache for future use.

This extended scheme not only improves performance by allowing virtual processes to keep their state in local memory between supersteps, but also allows us to exploit locality by allowing different virtual processes within the same physical machine to communicate with each other between supersteps without transmitting actual data over the network.

Implementing this scheme in Bayanihan, we have been able to achieve significantly faster turnaround times by eliminating the massive (and mostly useless) transfer of local state between the server and each worker at the end and beginning of each superstep. Running the Jacobi iteration application on a 10 Mbit Ethernet network with one client worker (to test maximum efficiency), for example, the new caching scheme saved us 5 to 6 seconds per superstep, corresponding to the time it takes to send all the data (i.e., a 500x500 double array) to server and back to the client.² Unfortunately, we still have not gotten any useful speedups with the Jacobi iteration application due to its very fine granularity. This situation should improve, however, in applications where the computation time per

²Theoretically, such an array, which consists of $500 \times 500 \times 8 = 2 \times 10^6$ bytes, has a minimum transfer time of $2 \times 10^6 \text{ bytes} / (10^7 \text{ bits per s} / 8 \text{ bits per byte}) = 1.6 \text{ s}$ each way or 3.6s total per superstep. The extra delay of around 2 s in this case is probably mostly due to processing and serialization overhead.

element becomes large enough, as may be the case in other problems such as those in computational fluid dynamics.

5.6.2 Other Mechanisms Supporting Caching

Although this new caching system successfully reduces communication time in controlled lab experiments, it is not yet complete. In particular, several mechanisms still need to be implemented to make this caching system practical for use in real systems. These include the following:

Checkpoint-on-quit. If a volunteer stops a work engine – either explicitly or by closing his or her browser – the work engine applet should send all its dirty data back to the server. This prevents data from being lost when volunteers leave.

Periodic checkpointing and smart rollback. Since a volunteer’s machine can crash without having a chance to send all its dirty data back to the server, we cannot depend on checkpoint-on-quit alone. We need to require the work engines on volunteers’ machines to periodically (but not too often) send their dirty data back to the server, and then have the server store these so it can perform a *rollback* from these checkpoints in the future if necessary.

Interestingly, it may be possible to do rollbacks faster in Bayanihan BSP than in other parallel systems (and even other BSP systems). Since the server is responsible for transferring data between virtual processes at the end of each superstep, it can keep track of dependencies between processes, and even store the actual message data transferred between them. Thus, to recover the lost state of a process for a particular step, for example, the server need only retrieve the *input* messages of that process for that step, and rerun the work object for that step using these messages.³ There is no need to rollback *all* the other processes.

Lazy Migration and Itinerant Processes. Finally, for caching to be beneficial, the virtual processes (i.e., work objects) must be assigned as much as possible to physical machines that already have the data needed by the processes in their local cache. Thus, for processes that rely on a large amount of dirty local state, we should apply a *lazy migration* strategy that keeps them on the same physical machines as long as possible. On the other hand, processes that do not have much local state but need a lot of data from another process can actually be migrated to where the data is currently located instead of having the data to sent it. We may call these *itinerant processes*.

Our current implementation of Bayanihan BSP does not try to direct migration in any way. We simply send all work objects back to the server and then farm them out again from the server using the same eager scheduler we use for ordinary master-worker applications.

³Note that the server need not have the actual message data with it. These data can actually be stored in the local caches of other machines as long as they have not crashed.

With a large number of heterogenous machines, this basically results in a random migration pattern. A better scheme would keep track of where a work object “belongs” and try to assign work objects to each worker accordingly when the worker calls `getWork()`.

One question open to further research here, however, is how to apply the idea of eager scheduling, which achieves adaptive parallelism by allowing work objects to be reassigned to more than one processor as long as it is not yet done. Reassigning work objects too eagerly may result in too much data transfer – which would be unnecessary if the original worker is about to produce the result shortly anyway. On the other hand, waiting too long to reassign work could let slow or crashed workers become bottlenecks in the system. Thus, we need to develop sound rules for determining how long we should wait before trying to reassign a work object to another worker.

5.6.3 Other Possible Improvements

In addition to these mechanisms, which we need to implement in order to allow caching and keeping of dirty local state, there are also other mechanisms that are not really necessary, but may be able to improve performance and scalability as well. These include:

Sabotage-tolerance. Once the checkpointing and rollback mechanisms are in place, it should be possible to implement sabotage-tolerance mechanisms such as those described in Sect. 3.5.2 and Chap. 6. In particular, backtracking should still work, although it may now be complicated by the occasional need to rollback processes that have been done by a caught saboteur.

Programming interface improvements. Aside from performance improvements, we can also look into implementing some programming interface improvements such as the loop version of `bsp_cont()` mentioned in Sect. 5.4.1, tag-based retrieval of incoming messages, and saving and restoration of BSP variables in recursive functions.

Peer-to-peer communication. Performing the global communication operations on the server makes it easy for worker applets to communicate with each other despite the Java applet security restrictions that limit an applet to communicating only with its source host. Unfortunately, it can also be an unnecessary sequential bottleneck in cases where such restrictions can be removed, such as within a trusted intranet. For these situations, we can implement peer-to-peer communication by modifying the work engine to send and accept communication requests to and from other work engines directly. We can still distribute the resulting worker code as an applet by either *signing* it, or asking volunteers to manually turn off their browser’s applet network restrictions (Microsoft Internet Explorer 4.0, Sun’s Hot Java browser, and Sun’s Java plug-in [59], which works with Netscape, allow this). Implementing peer-to-peer communications should greatly improve the performance of parallel algorithms that depend on parallelizing communications as well as computation (such as many arrays written for systolic arrays and such). Note, however, that making use of peer-to-peer communication also makes it more difficult to achieve adaptive parallelism and fault-tolerance. Thus, it should only be done in systems where these are not necessary.

Dataflow-style BSP. Because a BSP process can only receive data sent by other processes in the *previous* superstep, it is possible to determine in advance which data a process is going to need in a superstep before the superstep starts. This allows us to determine the data dependencies between the different processes at each superstep, and allows us to migrate processes to take advantage of locality as noted above. Furthermore, however, it also allows us to implement “soft barriers” that would allow processes to go through a barrier as soon as the incoming data that they need are already available. These soft barriers in turn, free processes from having to wait for a *global* barrier synchronization and can improve not only running time but overall scalability as well.

Note that although this goes against the original idea behind BSP, and breaks the theoretical BSP model [150, 139], it may be worth considering in volunteer computing systems, where adaptive parallelism, reliability, performance, and scalability are much more important goals than theoretical analyzability.

5.7 Conclusion

5.7.1 Related Work

Several Java-based parallel computing systems have been developed with interfaces based on PVM (such as JPVM [47]), and MPI, (such as mpiJava [9], jmpj [39], DOGMA [77], and others [10]). Most of these systems, however, use command-line Java *applications*, and thus still require some time and user expertise to set up. Making them use Java *applets* that can be run in a browser would be difficult or impossible not only because they require peer-to-peer communication not normally available to applets, but also because the semantics of the conventional message-passing models they use make it difficult to implement the adaptive load-balancing and fault-tolerance mechanisms required in the dynamic environment of applet-based volunteer systems.

Applet-based systems, such as Charlotte [12], Javelin [23], DAMPP [151], JET [111], and SWC [8], are much easier to use than application-based systems since they allow a volunteer to join a computation by simply visiting a web page. However, applet security restrictions and the need for adaptive parallelism and crash-tolerance seem to have so far limited most of these applet-based systems to using master-worker-based programming models. Among applet-based systems, the most general and programmable one so far seems to be Charlotte [12]. Charlotte has a clean programming interface that provides transparent cache-coherent distributed shared memory between browser-based worker applets, while supporting adaptive parallelism in the form of eager scheduling. Its execution model is very similar to BSP in that computation is also divided by barrier synchronizations into superstep-like parallel blocks, and changes to distributed memory are not felt by other processors until after the next barrier. In addition, Charlotte has a distributed caching mechanism that may be more fully developed than Bayanihan's. Charlotte, however, requires writing a separate Java class for each superstep, and does not have message-passing functions. Thus, Charlotte is less intuitive and somewhat harder to use for long multi-step programs than Bayanihan BSP, which provides programmers with a more traditional SPMD-style interface.

As far as we know, Bayanihan BSP is the first and only explicitly BSP-style parallel programming interface implemented in pure Java. The only other Java-based BSP implementation we are aware of is NestStep [82]. NestStep, however, does not use the Java language itself. It extends Java with its own new language constructs, and requires programmers to use a compiler that generates bytecodes than can be run by an application-based (not web-based) Java virtual machine (JVM). Thus, NestStep is more difficult to use than Bayanihan BSP, not only for users but for programmers as well. It does, however, provide mechanisms for distributed shared memory and for *nesting* supersteps in hierarchical fashion, that may be worth looking into.

5.7.2 Summary

In this chapter, we have shown how the BSP programming model can be implemented and used in Java-based volunteer computing systems. Our implementation, which uses Bayanihan's master-worker runtime system can take advantage of existing mechanisms for adaptive parallelism and fault-tolerance, such as eager scheduling, majority voting, and spot-checking, while providing a new programming interface with familiar and powerful methods for remote memory and message passing operations. At present, the performance and scalability of our implementation is limited in some applications, but is expected to improve with future work.

Meanwhile, the BSP programming interface we have developed represents a significant improvement in programmability and flexibility over current programming interfaces for volunteer-based systems. It shows that contrary to popular belief and practice, volunteer computing need not be limited to master-worker and embarrassingly parallel applications, but can be used for message-passing style applications as well. Thus, the only major limitation left for volunteer computing applications is that they must be *coarse-grain* enough to be efficient on commodity Internet lines – which is the same limitation faced by NOWs, metacomputers, and wide-area parallel computing systems in general. Furthermore, since our implementation here only requires an underlying master-worker runtime system, it should be implementable in other volunteer computing systems with similar runtime systems. Thus, our results here represent not only what can be done with Bayanihan but what can potentially be done in other volunteer computing systems as well.

Chapter 6

Reliability in the Presence of Malicious Volunteers

6.1 Introduction

The key advantage of volunteer computing over other forms of parallel computing and metacomputing is its ease-of-use and accessibility to the general public. It is by making it easy for anyone on the Internet to join in a parallel computation that volunteer computing makes it possible to build very large parallel computing networks very quickly. This same advantage, however, also creates a new problem: if we allow *anyone* to join a computation, how do we prevent *malicious volunteers* from *sabotaging* the computation by submitting bad results?

This problem is relatively new and unstudied. To date, most research in “fault-tolerance” in the context of parallel computing has been focused on what we may call “failure-tolerance” or “crash-tolerance”, where all faults are assumed to be in the form of *stopping faults* – faults where one or more of the processing elements, or the communication network links between them, simply stops generating or transmitting data, either temporarily or permanently. Little research has been done on protecting against faults where the processors do not stop producing data, but instead produce bad data. Even less research has been done on cases where these bad data are generated *intentionally and maliciously* by hostile parties.¹

This is actually not very surprising because until recently, most parallel computing has been done within single-machine supercomputers, where the processors and the network connecting them are all physically located in the same place and under the control of the owner of the computation. In such systems, the primary, if not the only, source of faults would be the hardware itself. Since hardware faults are relatively rare and generally tend to cause either stopping faults or random data corruption, it has mostly been possible to either simply ignore the possibility of errors, or use parity and checksum schemes to detect

¹Actually, in the field of *distributed systems*, much research *is* being done on this problem in the form of *Byzantine agreement*. See, for example, [93]. However, the emphases and goals in these works tend to be different from those of researchers who use parallel computing for *high-performance computation*, and thus their results tend to be impractical, inefficient, or both, when used in such contexts.

data corruption errors and treat uncorrectable errors as stopping faults (i.e., invalidate the entire answer, making it equivalent to no answer at all).

Today, however, as more and more parallel computing is being done on network-based systems where the processing elements are not only physically distributed but also owned by different people, these traditional fault-tolerance mechanisms are becoming insufficient. While parity and checksum schemes work well against random hardware errors, they are not effective against *intentional* attacks by malicious volunteers – or *saboteurs* – who can disassemble the code, figure out the checksum-generating part, and be able to produce valid checksums for bad data. Thus, there is a need for new *sabotage-tolerance* mechanisms that work in the presence of malicious saboteurs without depending on checksums or cryptographic techniques.

In this chapter, we present such techniques. We begin with an overview of general approaches to this problem, including choosing appropriate applications, using mechanisms for authentication, encrypted computation, and obfuscation, and using techniques based on redundancy and randomization. We then proceed to focus on the latter and present a theoretical analysis of the traditional technique of majority voting as well as the new technique of spot-checking, first presented in Sect. 3.5.2. Then, in the second half of this chapter, we integrate these mechanisms by presenting the new idea of *credibility-based fault-tolerance*, which uses probability estimates to efficiently limit and direct the use of redundancy. We demonstrate the effectivity of credibility-based fault-tolerance through parallel Monte Carlo simulations, and show how credibility-based fault-tolerance allows us to achieve error rates that are orders-of-magnitude smaller than that of traditional voting for the same slowdown. Finally, we discuss how credibility-based fault tolerance can also be used with other mechanisms and in other applications.

6.2 Overview of Approaches

6.2.1 Application Choice

Reliability is easier to achieve in some applications than in others. Some such applications include *naturally fault-tolerant* applications, as well as those involving *verifiable* computations.

Naturally Fault-Tolerant Applications

Naturally fault-tolerant computations are ones that do not require 100% accuracy, but can still produce acceptable results even with error rates as large as a few percent. As noted in Chap. 4, these include image rendering applications where a few scattered erroneous pixels would be unnoticeable or can be averaged out to make them unnoticeable, as well as *self-correcting* algorithms such as *genetic algorithms* where bad results are naturally screened out by the fitness function used for natural selection.

It may also be possible to make some Monte Carlo simulations naturally fault-tolerant. If the goal of the simulation is simply to predict the expected behavior of a system, and we are not concerned with outliers and behavior at the tails of the statistical curve, then we can tolerate faults by guarding against the effect of outliers. We can do this either by

heuristically detecting and rejecting outliers based on specific knowledge about the system, or by using *median*-based statistics instead of the more common *mean*-based statistics. Median-based statistics may cost more in terms of computation and memory space (since they require some kind of sorting-based algorithm), but are generally much more robust against outliers than mean-based statistics and do not require heuristics or prior knowledge about the system.

Note that while these techniques address the problem of faults in the form of outliers, however, they may not work against saboteurs who submit answers that are incorrect but are intentionally made close enough to the mean or median so as not to get rejected by our algorithms. This problem remains to be studied more carefully, but intuitively, we suspect that in this case, we can still get acceptably accurate values for the mean or median, but other statistics such as the standard deviation and the shape of the curve may be affected.

Verifiable Computations

Verifiable computations are those where it is possible to check the validity of a given result in much less time than it takes to generate the result itself. These computations allow us to detect errors and have them be recomputed as necessary. Thus, although it may take a little extra time to get the final result, we can at least guarantee its correctness. Note that the verification itself can be done either automatically by machine or, in some cases, by a human user. In some applications of image rendering, for example, a human user can look at the final results and visually recognize unacceptable errors and have them be recomputed. Another example is SETI@home, which uses many algorithms to verify the significance and validity of possible signals of alien life, but has a team of scientists personally study and verify any positive results that withstand these rigorous tests.

Many verifiable computations take the form of *search problems* such as SETI@home and those discussed in Sect. 4.2. In general, these search problems involve looking for solutions or input sets (e.g., a key, a set of factors, or a set of radio signals) that satisfy particular criteria (e.g., they decode an encrypted message, they factor an integer, or they indicate the existence of alien life). Thus, verification can be done simply by checking solutions for the required criteria.

The best such search problems with respect to verifiability and reliability are those where: (1) checking for the criteria can be done quickly, and (2) true positive solutions are known to be rare, if they exist at all. An example of this is the RC5 code cracking application, where there is known to be only one correct key, and the server can easily check claims by workers of finding the right key by using the key to decrypt the message. The rarity of true positive solutions in such problems means that we do not expect to receive positive results very often, and can thus allow more time for the verification process. It also means that denial-of-service attacks, launched by saboteurs attempting to overload the server by giving it a lot of false positive results to verify, will not work. That is, a saboteur cannot keep a server busy verifying results because to do so, it would have to find many true positive results – which we know is impossible since such results are rare. If a saboteur tries to give a false positive result instead, however, it would get caught and can be *blacklisted* (as discussed in Sects. 3.5.2 and 6.3.3) and thus prevented from submitting more results.

Unfortunately, although false positives are not a problem in such applications, false negatives are. If a worker claims that a certain subrange does *not* contain the right key, for example, the only way to verify that this claim is true is to search the subrange again. Thus, this means that volunteers can *cheat* – i.e., get paid for doing work without actually doing work. At the same time, they can also prevent the system from finding the right answer by intentionally *not* reporting the presence of the correct answer even if it is there. Moreover, in this case, we cannot rely on spot-checking (discussed in Sects. 3.5.2 and 6.3.3). That is, since the correct result for a vast majority of subranges is negative (i.e., “not found”), it is impossible to tell if a saboteur is cheating unless we give it a spotter work whose correct answer is known to be ‘positive (i.e., ‘found’). Unfortunately, since such results are rare, we probably do not know of any, or know only a few – which are useless since smart saboteurs can then learn to recognize them and avoid them by watching for the same work being given again.

Fortunately, however, if our only goal is to find the answer and make sure the answer is correct, then despite these problems, we can still achieve our goal relatively easily. In this case, we can employ a scheme similar to eager scheduling (see Sect. 3.5.1) – i.e., if the master receives results for all the work but does not find a positive result, then it goes back and starts reassigning work. Eventually, assuming that there are more good workers than bad, a good worker will find the right solution. More specifically, if we assume that the probability of finding a positive result is $1 - f$ (e.g., if the fraction of workers that are bad is f and all workers run at the same speed), then we can expect the average number of times we have to go through the whole batch before finding the right answer to be $1/(1 - f)$ times more than before. Interestingly, this means that if f is small, there is practically no extra cost required to protect against cheating workers. And, even if f is as large as $1/2$, the average slowdown is only around 2. Moreover, if we somehow know the *credibility* of each worker, as described in Chap. 6,² then we can improve efficiency even further by employing some form of credibility-based fault tolerance wherein we give more priority to redoing negative results done by less credible sources, than to those done by known-good workers.

In paid systems, a harder problem is to avoid losing money by paying cheating volunteers who do not actually do any work. One way to address this problem is to only pay for positive results. This is done in distributed.net and other projects where the prize money goes to the volunteer who finds the key. In this case, there is no economic incentive for a worker to cheat since if it does not do any work, it cannot get paid. Addressing this problem in cases where we have to pay workers even for negative results (e.g., in a commercial market-based environment) is a much harder problem. One way is suggested in [99], and involves setting trace points in the worker’s code when the worker stores current state information (i.e., the *trace* at that point), and requiring workers to submit these traces together with their results to prove that they did the work. In this scheme, the master can verify that a worker did the work (with high probability) by selecting a random subset of these traces and comparing them to their correct values.

²In this case, since we do not have spot-checking, a worker’s credibility would be a static credibility, based on its domain and other information about it, as described in Sect. 6.4.2.

6.2.2 Authentication, Encryption, and Obfuscation

Regardless of whether the applications we run are naturally fault-tolerant, verifiable, or otherwise, it would still be useful to try to reduce errors as much as possible. One way to do so, is to deal with the source of the errors – the saboteurs themselves.

Malicious attacks can either come from an internal *saboteur* who is actually a participating volunteer, or an external *spoofers* – a node that has *not* volunteered, but sends messages forged to look like they came from one of the volunteers. Spoofing can be prevented by using *digital signatures* [136]. 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 (see Chap. 2), from external attack. Unfortunately, however, digital signatures are ineffective against internal saboteurs, and thus useless in anonymous 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 bytecode 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, as described in App. B.

If cryptographically preventing disassembly is not possible, we can resort to *periodic obfuscation* to make understanding the disassembled code and isolating the checksum code as difficult as possible.³ Periodic obfuscation extends the idea of *static* obfuscation [30] by periodically obfuscating code 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 and scramble the code in many other ways (as discussed in App. B).

This idea of dynamically and periodically changing how the code looks like is similar to

³In earlier versions of our work [131, 132], we called this idea *dynamic obfuscation*. We decided to rename this term since “dynamic” obfuscation may sound like obfuscation that happens as the program runs (e.g., as in “dynamic compilation”), which is not the case here.

techniques used by *polymorphic computer viruses*, which use them to hide themselves from anti-virus programs [28]. Although very difficult, deobfuscating 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. Periodic 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. (As noted in App. B, this idea is very similar to the idea of *time-limited blackbox security* as applied to mobile agents [71].) Provided that the volunteer computing system allows the work code to change from work object to work object (as Bayanihan does, for example), periodic obfuscation should be easier to implement and more generally applicable than encrypted computation.

Although we have not done any concrete implementations involving encrypted computation or periodic obfuscation, we have started surveying various techniques from other fields. Appendix B contains a critique paper covering some of these works, and suggesting future work. It also discusses the application of encrypted computation and obfuscation not only to the problem of *sabotage* but to the problem of *espionage* as well.

6.2.3 Redundancy and Randomization

In cases where one cannot rely on authentication, encrypted computation, and obfuscation, there are generally two effective ways of dealing with intelligent saboteurs: *redundancy* and *randomization*. Redundancy allows us to check each worker’s results against those of others doing the same work. Assuming that there are sufficiently more good workers than bad in the volunteer population, this allows us to identify bad answers and reduce our chances of accepting them as good. Randomization protects us against specific intentional attacks by saboteurs. If the work server always operates in the same predictable manner, then it may be possible for an intelligent saboteur to develop a specialized plan of attack against it. Randomizing the behavior of the server, however, greatly limits what a saboteur can do, since, even if he can develop specific attacks, he can never be sure if those attacks are applicable at any given moment.

Techniques that use redundancy and randomization include voting and spot-checking, as described in Sect. 3.5.2. In voting, redundancy takes the form of having each work object be done a number of times by different workers. Randomization is used in selecting the workers, and makes it hard for saboteurs to gather votes. In spot-checking, redundancy takes the form of the spotter works, and is less than that in voting since we generally do not have to do all work objects several times. Randomization is used in selecting when to spot-check a worker. As we shall see, this lowers the error rate by forcing the worker to submit bad results less often in order to avoid being caught.

Since techniques that use redundancy and randomization are more general and robust (i.e., they assume less about the application, or about what workers can or cannot do with the code they receive), we will focus on them more than on other approaches in the rest of this chapter. Note, however, that we can always combine these techniques with the other approaches to achieve even better reliability. Thus, although we will show that voting and

spot-checking, used together through credibility-based fault-tolerance, can achieve very low error rates even without encryption and obfuscation, it would still be useful to implement the latter techniques if possible. Doing so would reduce the original error rates even further, and result in even lower error rates (especially since voting's error rate shrinks exponentially with the original error rate as shown in Sect. 6.3.2). Similarly, although these techniques may be able to make error rates low enough to be useful in ordinary applications, we can get even lower error rates if we use these techniques for running naturally fault-tolerant or verifiable applications.

6.3 Voting and Spot-checking

In this section, we present the traditional technique of *majority voting*, and the new technique of *spot-checking*. We show how they can be implemented, and analyze their theoretical performance. Empirical results from simulations are shown in Sect. 6.5.2.

6.3.1 Models and Assumptions

In this section, we describe the models and assumptions that we make in our analyses in the following sections. Although we have made simplifying assumptions to make the problems more tractable in some cases, we believe that these assumptions are not too unrealistic to be practical. In any case, Sect. 6.6 discusses how our results can be extended to cases with even more realistic assumptions.

Computational Model. We assume a *work-pool-based master-worker model* of computation, which we described in Sect. 4.1, and which is presently used not only in Bayanihan but in most, if not all, volunteer computing systems, as well as in many grid systems, meta-computing systems, and other wide-area network-based parallel computing systems in general.

To recall, in this model, a **computation** is divided into a sequence of **batches**, each of which consists of many mutually independent **work objects**. At the start of each batch, these work objects are placed in a **work pool** by the **master** node, and are then distributed to different **worker** nodes who execute them in parallel and return their **results** to the master. When the master has collected the results for all the work objects, it generates the next batch of work objects (possibly using data from the results of the current batch), puts them in the work pool, and repeats this whole process.

Error rate. We define the **error rate**, ε , as the ratio of **bad results** or **errors** among the **final results** accepted at the end of the batch. Alternatively, we can also define it as the probability of each individual final result being bad, such that on average, the expected number of errors in a batch of N results would be given by εN .

For simplicity, we assume for the most part that batches are independent of each other, such that errors in one batch do not affect the next. Moreover, in designing our mechanisms, we assume that there is a non-zero **acceptable error rate**, ε_{acc} , and aim to make the

error rate lower than it. Thus, these mechanisms would work best with naturally fault-tolerant applications, which can tolerate having a relatively large fraction (1% or more) of the individual final results being bad, and thus have high acceptable error rates. In other applications which do not tolerate any errors at all, ε_{acc} must be set to make the probability of getting any errors at all correspondingly small. For example, suppose that a computation has 10 batches of 100 work objects each, and the batches *are* dependent on each other such that a single error in any of the 10 batches will cause the whole computation to fail. In this case, to make the probability that the *whole* computation would fail, $P(\text{fail})$, less than 1%, the error rate ε , which is the probability of an *individual* result failing, must be at most $\varepsilon_{\text{acc}} = P(\text{fail})/(10 \times 100) = 0.001\% = 1 \times 10^{-5}$. Fortunately, although this error rate may seem small, we show in Sect. 6.4 that achieving such low error rates does not necessarily require a lot of redundancy.

Saboteurs. We assume that a **faulty fraction**, f , of the worker population, P , are **saboteurs**. The master node (which we assume is trustworthy) may not know the actual value of f , but is allowed to assume an upper bound on it, such that no guarantees need be made if the real faulty fraction is greater than the assumed bound.

For simplicity, we assume that all workers, including saboteurs, run at roughly the same speed, such that the work objects are uniformly and randomly distributed among the workers and saboteurs. Thus, since each result we receive is equally likely to come from any of the workers, the **original error rate** – i.e., the expected error rate without fault-tolerance – would simply be f .

Sabotage Rate and Collusion. For simplicity, we model each saboteur as a Bernoulli process with a probability s of giving a bad result, and assume that s , called the **sabotage rate**, is constant in time and the same for all saboteurs.

Note that this assumes that workers and saboteurs are independent of each other and do not communicate. In particular, in cases where saboteurs do not *always* give bad results, we assume that the saboteurs do not agree on *when* to give a bad result, but decide to do so independently. However, if two or more saboteurs happen to decide to give a bad result for a particular work entry, we will assume, unless otherwise stated, that their bad answers would match. This allows saboteurs to “vote” together, and is a conservative assumption since intuitively, we can expect final error rates to be lower if saboteurs cannot vote together.

Redundancy and Slowdown. To measure the efficiency of fault-tolerance mechanisms, we consider **redundancy** and **slowdown**. Redundancy is defined as the ratio of the average total number of work objects actually assigned to the workers in a batch when using the mechanism, N_{total} , vs. the original number of work entries, N . Slowdown is defined as a similar ratio between the running times of the computation with and without the use of the mechanism. In general, redundancy leads to an equivalent slowdown, since we assume that there are many times more work objects than workers, so that there are no idle workers most of the time. For example, a mechanism that on average does all the work twice will generally take twice as long. Note however, that in some cases – especially

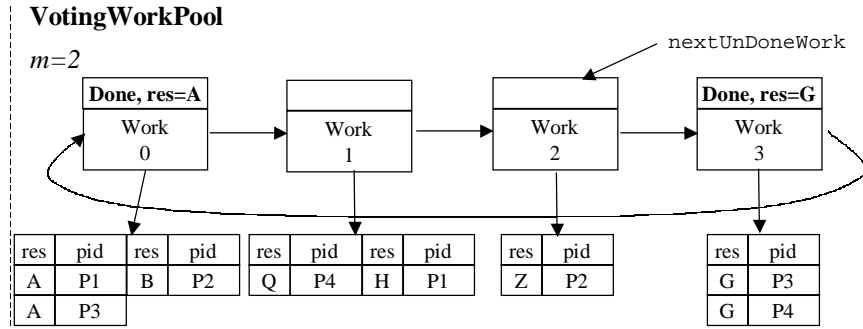


Figure 6-1: Eager scheduling work pool with m -first majority voting.

when workers can join, leave, or get blacklisted in the middle of a batch – slowdown may be different from redundancy. If workers leave or are blacklisted, for example, then the remaining workers must take over their work. This increases the slowdown, even though the total amount of work, and thus the redundancy is the same.

In general, fault-tolerance mechanisms should aim to (in order of priority): (1) minimize the final error rate as much as possible, or at least reduce it to an acceptable level, (2) minimize redundancy, and (3) minimize slowdown.

6.3.2 Majority Voting

Using an eager scheduling work pool as described in Sect. 3.5.1, we can easily implement the traditional scheme of *majority voting* by leaving the done flag unset until we collect $2m - 1$ results for each work entry from different workers,⁴ and then choosing the result which has at least m or more matching instances (i.e., a *majority*) among the $2m - 1$ copies. More efficiently, we can employ an *m-first voting scheme*, as shown in Fig. 6-1, where we stop collecting results after we get m matching instances of a result. Assuming that bad answers match, this scheme achieves the same error rate as simple majority voting, but uses only half as much redundancy, averaging at around $m/(1 - \varphi)$ compared to $2m - 1$. Given a **fault rate**, φ , representing the probability of an individual final result being bad (generally equal to fs , or simply f , if $s = 1$), we can compute the final error rate due to majority voting, $\varepsilon_{\text{majv}}$, as:

$$\varepsilon_{\text{majv}}(\varphi, m) = \sum_{j=m}^{2m-1} \binom{2m-1}{j} \varphi^j (1 - \varphi)^{(2m-1-j)} \tag{6.1}$$

which is the probability that the bad results form the majority. As shown in Fig. 6-2, this is roughly exponential in m . More precisely, it can be shown to be bounded by the following

⁴In practice, we can prevent double-voting, and achieve lower error rates, by not giving a work object to the same worker more than once. Our analysis, however, does not assume this.

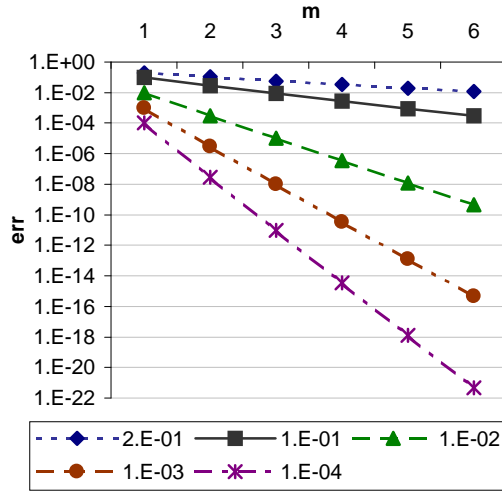


Figure 6-2: Majority Voting: theoretical error rate for various values of φ and m .

approximation [163]:⁵

$$\begin{aligned} \varepsilon_{\text{majv}}(\varphi, m) &< \binom{2m-1}{m-1} \frac{(\varphi(1-\varphi))^m}{1-2\varphi} \\ &< \approx \frac{(4\varphi(1-\varphi))^m}{2(1-2\varphi)\sqrt{\pi(m-1)}} \end{aligned} \tag{6.2}$$

Or, more simply, as φ decreases and m increases, we have:

$$\varepsilon_{\text{majv}}(\varphi, m) \approx (c\varphi)^m \tag{6.3}$$

where c is a roughly constant function of φ and m , starting at $\sqrt{3-2\varphi}$ (i.e., around 1.7 for small φ) at $m = 2$ and approaching 4 at $m = \infty$.

This exponentially shrinking error rate has two notable implications: (1) voting works very well in systems with small φ , and (2) it gets increasingly better as φ decreases. Thus, in systems with very low fault rates to begin with, such as hardware systems, it does not take much redundancy to shrink the error to extremely low levels.

Unfortunately, however, voting also suffers from drawbacks that limit its applicability to volunteer computing systems. First, it is inefficient when φ is not so small. Figure 6-2 shows, for example, at $\varphi = 20\%$, doing all the work 6 times still leaves an error rate larger than 1%. More generally, Fig. 6-3 shows how the slowdown required by voting to achieve low error rates dramatically increases as φ increases. Second, and in many cases, more importantly, it has a minimum redundancy of two. That is, *any* desired reduction of the original error rate, no matter how small, requires doing *all* the work at least twice. This is true regardless of φ , and even if we use the *m-ahead voting* scheme to be described in

⁵Actually, numerical computations seem to show that replacing $m - 1$ with m in the denominator would work as well and provides a slightly tighter bound. However, a literal adaptation of Yuev's derivation in [163] (which expresses the formula in terms of $k = m - 1$ instead of m) yields the form in Eq. 6.2.

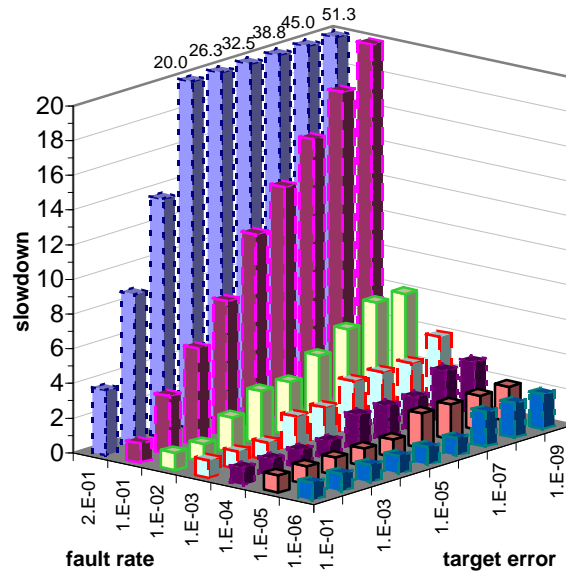


Figure 6-3: Majority voting: theoretical slowdown required to achieve target error rates, given various fault rates. These values are derived by solving Eq.6.2 for m , and then multiplying by $1/(1 - \varphi)$.

Sect. 6.4.3, which performs better at larger values of φ . For these reasons, voting is only practical in cases where: (1) the fault rate φ is small (i.e., $\varphi < \approx 1\%$), and (2) either (a) we have enough idle or spare nodes to take on the extra load without causing additional slowdown (as in the case of hardware-redundant triple modular redundancy systems), or (b) a slowdown of at least two (or in general m) is deemed to be an acceptable price to pay for the desired error reduction.

6.3.3 Spot-Checking with Blacklisting

In cases where either the fault rate φ is large, or our maximum acceptable error rate is not too small, we can use a novel alternative we call *spot-checking*, which can achieve better error reductions than voting with redundancies less than two. In spot-checking, the master node does not redo *all* the work objects two or more times, but instead randomly gives a worker a *spotter* work object whose correct result is already known or will be known by checking it in some manner afterwards.⁶ Then, if a worker is caught giving a bad result, the master *backtracks* through all the results received from that worker so far, and invalidates all of them. The master may also *blacklist* the caught saboteur so that it is prevented from submitting any more bad results in the future.

Because spot-checking does not involve replicating all the work objects, it has a much lower redundancy than voting. If we assume that the master spot-checks each worker with a Bernoulli probability q , called the *spot-check rate*, then the redundancy, on average, will

⁶We are intentionally vague here about how spotter works are chosen and how they are checked. This is to allow for different ways of implementing spot-checking. Some possible implementations are discussed in Sect. 3.5.2.

be just $1/(1 - q)$. For example, if $q = 10\%$, then 10% of the work the master gives would be spotter works. This means that on average, the master gives out $(1/(1 - 0.1)) = 1.11N$ work objects during the course of a batch with N real work objects.

Even with this low redundancy, however, spot-checking can still achieve very low error rates. The key here is that even though we do not double-check all the results given by a worker, it is alright as long as the worker gets caught *eventually* (i.e., by the end of the batch⁷), since in that case, backtracking will invalidate any unchecked bad results anyway. Since the probability of a saboteur not getting caught shrinks exponentially with time (i.e., with the number of work objects received by a saboteur), the error rate decreases as well, especially in large computations with many work objects to be done.

To see this, consider the case where caught saboteurs are *blacklisted* and never allowed to return or do any more work (at least not within the current batch). In this case, errors can only come from saboteurs that survive until the end of the batch. Assuming that a saboteur is given a total of n work objects during a batch (i.e., n is the saboteur's share in the total work, i.e., N/P , plus the $1/(1 - q)$ redundancy due to spot-checking, plus the extra load that the remaining workers have to take when a worker gets blacklisted), then the *average final error rate with spot-checking and blacklisting*, $\varepsilon_{\text{sabl}}$, can be computed as:

$$\begin{aligned}
\varepsilon_{\text{sabl}}(q, n, f, s) &= P(\text{result taken at random is bad}) \\
&= s \cdot P(\text{result is from a saboteur}) \\
&= s \cdot P(\text{worker is a saboteur} \mid \text{worker survived}) \\
&= s \cdot \frac{P(\text{worker is a saboteur and worker survived})}{P(\text{worker survived})} \\
&= \frac{sf(1 - qs)^n}{(1 - f) + f(1 - qs)^n} \tag{6.4}
\end{aligned}$$

where s is the sabotage rate of a saboteur, f is the fraction of the original population that were saboteurs, $(1 - qs)^n$ is the probability of a saboteur surviving through n turns, and the denominator represents the fraction of the original worker population that survive to the end of the batch, including both good and bad workers.

Figure 6-4 shows this error rate at various values of n and s . As shown, a maximum point occurs at some s between 0 and 1. This makes intuitive sense. If a saboteur gave bad results all the time (i.e., $s = 1$), it would tend to get caught too quickly. On the other hand, although reducing s increases a saboteur's chances of surviving, it also reduces the total number of bad results that the saboteur tries to give. Thus, reducing it too much eventually reduces the net error rate as well.

Unfortunately, there is no closed form for the exact maximum point of Eq. 6.4. However, we can find an upper bound by maximizing the following strict upper bound instead:

$$\varepsilon_{\text{sabl}}(q, n, f, s) < \hat{\varepsilon}_{\text{sabl}}(q, n, f, s) = \frac{sf(1 - qs)^n}{1 - f} \tag{6.5}$$

⁷...or by the end of the computation, if we can backtrack through past batches. In this thesis, however, we will assume that we cannot backtrack through batches.

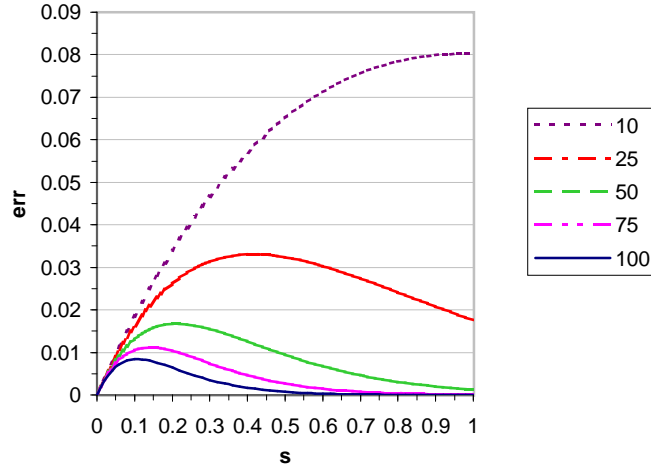


Figure 6-4: Spot-checking with blacklisting: theoretical average error rate ($\varepsilon_{\text{sdbl}}$), at $q = 0.1$ and $f = 20\%$, for various values of s and n .

This upper bound has a maximum point at

$$\hat{s}_{\text{sdbl}}^*(q, n) = \min\left(1, \frac{1}{q(n+1)}\right) \quad (6.6)$$

and a maximum value of

$$\hat{\varepsilon}_{\text{sdbl}}^*(q, n) = \begin{cases} \frac{f}{1-f} \cdot (1-q)^n & n \leq \frac{1-q}{q} \\ \frac{f}{1-f} \cdot \frac{\binom{n}{n+1}^n}{q(n+1)} & \text{otherwise} \end{cases} \quad (6.7)$$

Since the expression $\frac{\binom{n}{n+1}^n}{n+1}$ is strictly and asymptotically bounded from above by $1/ne$ (where e is the base of the natural logarithm), we get:

$$\varepsilon_{\text{sdbl}}(q, n, f, s) < \hat{\varepsilon}_{\text{sdbl}}^*(q, n) < \frac{f}{1-f} \cdot \frac{1}{qne} \quad (6.8)$$

At first glance, this seems like a counter-intuitive result. It says that instead of reducing the expected error rate exponentially with n , as we first thought, spot-checking only reduces it inversely with n . Note, however, that Eq. 6.8 represents the *worst-case* error rate – i.e., given a saboteur who somehow knows n and q and selects s accordingly *in advance*. Note that for any particular fixed value of s , $\varepsilon_{\text{sdbl}}$ still shrinks exponentially in time and with n .

Intuitively, we can interpret the worst-case s given by Eq. 6.6 as saying that a saboteur's best bet is to plan on generating a *constant* number of errors in a batch. Specifically, a saboteur should plan on generating $n\hat{s}_{\text{sdbl}}^* \approx 1/q$ errors randomly distributed over the n results. Similarly, Eq. 6.8 implies that the *total number* of errors that a master can expect from a single saboteur is bounded by a constant $1/qe$, *regardless of* n . Thus, to reduce the *fraction* of errors, i.e., the error rate, it is to the master's advantage to make the batches longer so that n is larger.

6.3.4 Spot-checking without Blacklisting

Ideally, we would like to permanently blacklist any saboteurs that we catch. Unfortunately, however, this may not always be possible. Although we can require volunteers to give their email addresses, and then blacklist saboteurs according to email address, it is not too hard for a saboteur to create a new email account and volunteer as a “new” person. (Microsoft’s `hotmail.com` free email service, for example, allows a user to create a new account in a matter of minutes without having to submit authentic personal information [96].) Blacklisting by IP address would not work either because many people use ISPs that give them a dynamic address that changes every time they dial up. Requiring more verifiable forms of identification such as home address and a telephone number can turn away saboteurs, but would probably turn away many well-meaning volunteers as well. Furthermore, in some cases, it might not be desirable to blacklist a worker permanently. Good workers, for example, may occasionally suffer transient faults due to power fluctuations, operating system glitches, etc. In these cases, we do not want to blacklist a node immediately and permanently just because of a temporary fault that it did not intend to produce. Thus, for all these reasons, it is useful to consider the effectivity of spot-checking *without* blacklisting.

Unfortunately, if blacklisting cannot be enforced, and it is easy for a saboteur to leave and immediately come back an unbounded number of times under a new identity, then a saboteur can render spot-checking practically useless by joining a computation, doing only *one* work object, submitting a bad result for it, and then leaving and repeating the same process under a new identity. Of course, other factors, such as the time it takes to forge a new identity and sign up as a new volunteer, will probably discourage a saboteur from leaving and rejoining under a new identity too often. However, even if a saboteur decides to limit its stay to at most a constant number of work objects l , then it can still do significantly more damage than it could with blacklisting, as we shall see in this section.

Long-staying saboteurs. To derive a bound on the error due to spot-checking without blacklisting, we first consider the case where $l \geq n$ – i.e., a saboteur stays at least as long as a batch, or effectively, as long as possible until it gets caught or the batch ends. We also assume, pessimistically, that a saboteur somehow knows when it is caught and is able to rejoin under a new identity immediately.⁸ This second assumption implies two things: (1) the probability of a final answer coming from a saboteur is simply f , since the number of saboteurs in the system is constant, unlike the case with blacklisting, and (2) if a saboteur gets caught near the end of the batch when the saboteur has only $i < n$ work objects left to do, then upon returning under a new identity, it would only need to survive through i turns, instead of n . Thus, the average total error rate can be computed as follows:

$$\begin{aligned} \text{err}_{\text{scnb}}(q, n, s) &= f \cdot \frac{1}{n} \cdot s \cdot \text{Average length of “last run”} \\ &= \frac{fs}{n} \left[n(1 - qs)^n + \sum_{i=0}^{n-1} i(1 - qs)^i qs \right] \end{aligned} \quad (6.9)$$

⁸In practice, the master can continue to assign work and collect (but ignore) answers from caught saboteurs so that they do not know they are caught. This would lead to lower error rates, as discussed in Sect. 6.6.

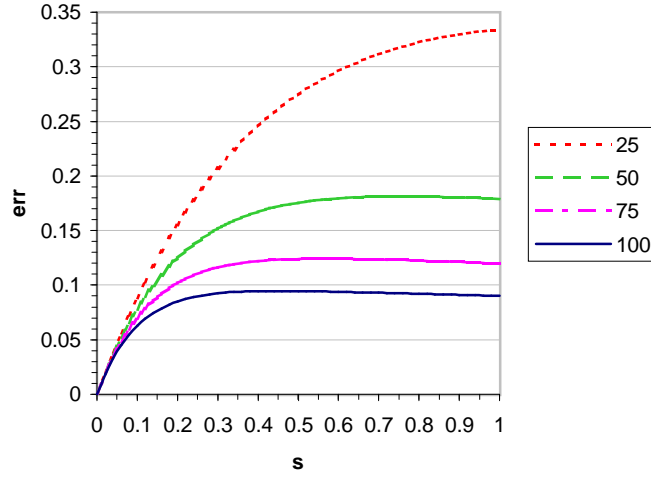


Figure 6-5: Spot-checking without blacklisting: theoretical error rate with long-staying saboteurs (err_{scnb}), at $f = 20\%$ and $q = 0.1$, for various values of s and n .

where the value in brackets represents the average number of work objects a saboteur does between the last time it is caught and the end of the batch. Errors can only come from this last run because any errors before the last time it was caught would be invalidated by backtracking.

This equation simplifies to:

$$err_{\text{scnb}}(q, n, f, s) = \frac{f(1 - qs)(1 - (1 - qs)^n)}{qn} \quad (6.10)$$

which is shown in Fig. 6-5. Interestingly, and somewhat surprisingly, although Eq. 6.10 seems less intuitive than Eq. 6.4, it still has a simple upper bound that is inversely proportional to n . Specifically, Eq. 6.10 has a maximum at

$$s_{\text{scnb}}^*(q, n) = \min\left(1, \frac{1 - (n + 1)^{-1/n}}{q}\right) \quad (6.11)$$

and has the value

$$err_{\text{scnb}}^*(q, n, f) = \begin{cases} f(1 - q)(1 - (1 - q)^n) & \frac{1 - (n+1)^{-1/n}}{qn} \geq 1 \\ \frac{f}{q(n+1)^{\frac{n+1}{n}}} & \text{otherwise} \end{cases} \quad (6.12)$$

which asymptotically approaches, and is strictly bounded from above by, the following expression:

$$err_{\text{scnb}}^*(q, n, f) < \frac{f}{qn} \quad (6.13)$$

Short-staying saboteurs. Of course, if there were no blacklisting, then saboteurs would most likely choose to stay for a smaller number of turns, $l < n$, instead of waiting until they are caught or blacklisted. Intuitively, this would lead to larger error rates because saboteurs need only to survive uncaught for l turns or less, instead of n turns.

The analysis that leads to Eq. 6.13 allows us to get an approximate upper bound on the error rate when $l < n$. To derive this upper bound, we first imagine that we can divide the n work objects that a saboteur does in a batch into contiguous groups of l work objects each. Then, we make the pessimistic assumption, as before, that a saboteur knows when it is caught and immediately rejoins under a new identity if caught. Furthermore, we assume – again pessimistically – that each saboteur needs only to survive until the end of the current l -length group of work in order to have its errors accepted, and thus be able to do damage. In this case, the error rate for each of the l -length groups can be derived using Eq. 6.9, but substituting l for n . This leads to the following approximate upper bound for the overall error rate:

$$err_{\text{scl}}^*(q, l, f) \lesssim \frac{f}{ql} \quad (6.14)$$

Note that this is *approximate* because if l does not divide n evenly, then the error rate for the remaining piece (which would have length $i < l$) could be higher. (This is especially true if we allow the saboteur to have a different sabotage rate for this shorter piece.) Note, however, that if l is small compared to n , then the size of this remaining piece would be small compared to the total size, and it would not affect the overall error rate very much. Similarly, as l approaches n , then we end up with two pieces, a large l -length piece, and a short $i = n - l$ -length piece. Although the error rate in the shorter piece may be higher, this is counterbalanced by the fact that the length of the piece is shorter as well, so the total number of errors remains the same or becomes smaller. Thus, although further work may be needed to refine this bound, Eq. 6.14 is good for now. In fact, at least for the results shown in Fig. 6-16 in Sect. 6.5.2, Eq. 6.14 seems to be good enough.

Limiting error rates. In any case, the important point to note here is that the error rate without blacklisting is significantly worse than that with blacklisting. Unlike the error rate with blacklisting, Eq. 6.8, which shrinks inversely with n , the error rate without blacklisting, Eq. 6.14, shrinks inversely with l . This means that we cannot reduce the error by making a batch longer. The best thing that we can do in this case, is to try to force saboteurs to stay longer (i.e., increase l) by making it hard for them to forge a new identity or by imposing delays.

If our batches are not too long, then we can impose a rule that new users would not be allowed to join until the *next* batch, so that a saboteur does not gain anything by leaving early. In this case, the error rates would be the same as in Sect. 6.3.3. This scheme is not practical if batches are long, however, since it would waste the potential power of good volunteers who would be forced to wait for the next batch.

6.3.5 Combining Spot-Checking and Voting

So far, we have seen that voting tends to reduce error rates exponentially in m with the fault rate φ in the base, while spot-checking reduces error rates by a factor that is linear in

the number of work objects given to a worker in a batch, n . This leads to the following idea: why not use voting on top of spot-checking to exponentially reduce the already linearly reduced error rates, and thus achieve much greater error reductions per redundancy?

Monte Carlo simulations, presented in Sect. 6.5.2, show that this idea actually works well if we have blacklisting. In this case, spot-checking effectively lowers the fault rate φ of each result from f down to $\varphi_{\text{scbl}} = \varepsilon_{\text{scbl}} < \frac{f}{1-f} \frac{1}{qne}$ as given in Eq. 6.8. This reduced fault rate, when substituted into voting's error rate as given by Eqs. 6.1 and 6.3, gives us an error rate of roughly $\varepsilon = (c \cdot \frac{f}{1-f} \frac{1}{qne})^m$, which is roughly $(qne)^m$ times better than majority voting for the same m . Thus, for example, if spot-checking can reduce the fault rate by a factor of 10, then voting would become about 100 times better at $m = 2$, 1000 times better at $m = 3$, etc. Correspondingly, this also means that given a target error rate, ε_{acc} , voting combined with spot-checking requires much less slowdown than voting alone.

Unfortunately, simulation results, as shown in Sect. 6.5.2 also show that simply combining voting and spot-checking in this way does not work well without blacklisting. Not only is the fault rate higher if saboteurs can come back under new identities every l turns, but substituting the higher fault rate into voting's error rate does not work either but gives error rate predictions that are lower than what we get from simulations.

6.4 Credibility-based Fault-Tolerance

In this section, we present a new idea called *credibility-based fault-tolerance* that not only address the shortcomings of blacklisting, but more significantly, provides a framework for combining the benefits of voting, spot-checking, and possibly other mechanisms as well.

6.4.1 Overview: Credibility and the Credibility Threshold Principle

The key idea in credibility-based fault-tolerance is the **Credibility Threshold Principle**: *if we only accept a result for a work entry when the conditional probability of that result being correct is at least some threshold ϑ , then the probability of accepting a correct result, averaged over all work entries, would be at least ϑ* . This principle implies that if we can somehow compute the conditional probability that a work entry's best result so far is correct, then we can mathematically guarantee that the error rate (on average) will be less than some desired ε_{acc} , by simply leaving the *done* flag unset until this conditional probability reaches the threshold $\vartheta = 1 - \varepsilon_{\text{acc}}$.

To implement this idea, we attach **credibility values** to different objects in the system, as shown in Fig. 6-1, where the **credibility** of some object X , written $Cr(X)$, is defined as *an estimate of the conditional probability, given the current observed state of the system, that object X is, or will give, a good result*. As shown, we have four different types of credibility: that of workers (Cr_P), results (Cr_R), result groups (Cr_G), and work entries (Cr_W). The credibility of a worker depends on its observed behavior such as the number of spot-checks it has passed, as well as other assumptions such as the upper bound on the faulty fraction, f . In general, we give less credibility to new workers who have not yet been spot-checked enough, and more credibility to those who have passed many spot-checks and are thus less likely to be saboteurs or have high sabotage rates. The credibility of a worker determines the credibility of its results, which in turn determine the credibility of the *result groups*

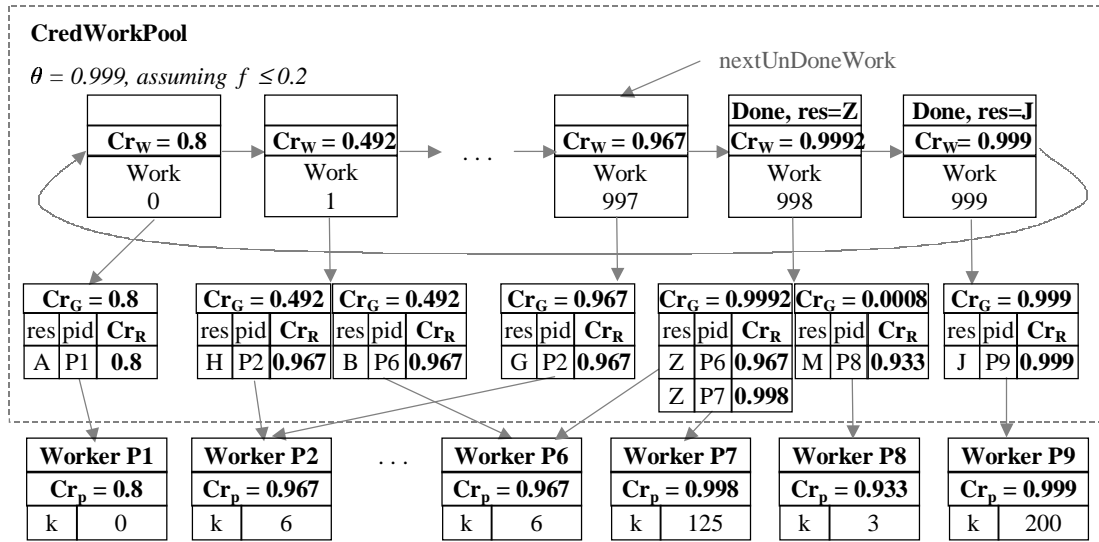


Figure 6-6: A credibility-enhanced eager scheduling work pool (using Eq.6.19 and Eq. 6.21).

in which they respectively belong. The credibility of a result group (which is composed of matching results for a work entry) is computed as the conditional probability that its results are correct, given the individual credibilities of these results, and those of other results in other result groups for the same work entry. Finally, the credibility of the *best* result group in a work entry gives us the credibility of the work entry itself, and gives us an estimate of the probability that we will get a correct result for that work entry if we accept its currently best result.

In the course of running a batch, the credibilities of the objects in the system change as either: (1) workers pass spot-checks (thereby increasing the credibilities of their results and their corresponding groups), (2) matching results are received for the same work entry (thereby forming result groups, whose credibilities increase with their size), or (3) workers get caught (thereby invalidating their results, and decreasing the credibilities of their corresponding result groups). Eventually, assuming there are enough good workers, the credibility of each work entry W reaches the threshold as the master gathers enough matching results for a work entry W , or the solvers of the results in W pass enough spot-checks to make the credibilities of their results go up, or both. When this happens, the work entry is marked done and the server stops reassigning it to workers. All this continues to happen for all undone work entries until all the work entries reach the desired threshold $\vartheta = 1 - \varepsilon_{acc}$, at which point, the batch ends. At this point, assuming that our credibilities are *good* estimates of the conditional probabilities they represent, the expected fraction of final results that will be correct should be at least ϑ , and the expected error rate would thus be at most ε_{acc} .

Note that this scheme automatically trades-off performance for correctness. It is similar to voting except that here, m is not determined in advance, but is determined dynamically, being made just as large as it needs to be for a work entry. Unlike traditional voting,

however, we do not have to redo a work entry many times (or at all) if its result was done by a worker which has been spot-checked many times and thus has a very high credibility. In this way, a work object is only repeated however many times it takes to achieve the desired correctness level, but no more. This makes credibility-based fault-tolerance very efficient, and as will be shown in Sect. 5.5, allows it achieve very low error rates with little redundancy.

6.4.2 Calculating Credibility

A key trick in this technique is computing the credibility values correctly. In general, there are many possible *credibility metrics*, corresponding to different ways of observing the current state of the system, as well as different ways of computing or estimating the conditional probability of correctness based on observations. In this section, we present particular metrics that we have found to be effective and discuss general guidelines for developing other credibility metrics.

In deriving these metrics, it is often useful to first derive the **dubiosity** of some object X , written $Db(X)$, which is defined as an estimate of the conditional probability, given the current observed state of the system, that object X is, or will give, a *bad* result. Given $Db(X)$, we can derive the credibility, $Cr(X)$, as simply $1 - Db(X)$.

Credibility of Workers and their Results

Without Spot-checking. Without spot-checking, the credibility of a worker P (and thus of its results) must be computed solely from assumptions that we are willing to make. In most cases, if we can assume a bound f on the faulty fraction of the worker population, then we simply let $Cr_P(P) = 1 - f$ for all workers, since f is the probability that a worker chosen at random would be bad. In some cases, we can assign some workers different credibilities. For example, if we know that machines from certain domains or subnets can be trusted, then we can give them high credibilities – even as high as 100% if the machines are under the personal control of the owners of the computation. On the other hand, if we notice that saboteurs tend to come from a certain domain or subnet more often than usual, then we can reduce the credibilities of any workers coming from that domain or subnet accordingly.

With Spot-checking and Blacklisting. If we have spot-checking, then we can estimate how likely a worker is to give a good result based on the number of spot-checks the worker has passed, k . Intuitively, the more spot-checks a worker passes, the more confident we can be that the worker is a good worker, or at least does not have a very high sabotage rate. (Note that we do not need to consider the credibility of workers who are spot-checked and caught, since these are removed from the system.)

In this case, it is easier to compute the credibility of a worker, P , by first considering its *dubiosity*, $Db_P(P)_{\text{sobl}}$, which is equal to the conditional probability of receiving a bad result from a worker, given that the worker has survived k spot-checks. This probability is similar to that in Eq. 6.4, and can be computed and bounded as follows:

$$Db_P(P)_{\text{sobl}} = P(\text{result from } P \text{ is bad} \mid P \text{ survived } k \text{ spot-checks})$$

$$= \frac{sf(1-s)^k}{(1-f) + f(1-s)^k} \quad (6.15)$$

$$< \frac{f}{1-f} \cdot \frac{1}{ke} \text{ (for any } s) \quad (6.16)$$

Subtracting this from 1 gives us the following credibility metric for spot-checking with blacklisting:

$$Cr_P(P)_{\text{scbl}} = 1 - \frac{f}{1-f} \cdot \frac{1}{ke} \quad (6.17)$$

which is a strict *lower bound* on the conditional probability of a worker P giving a *good* result.

Note that this equation does not apply to workers that have not yet been spot-checked, i.e., whose k is 0. In this case, we can just set $Cr_P(P) = 1 - f$. Alternatively, we can choose to just ignore results from workers that have not yet been spot-checked.

Without Blacklisting. Unfortunately, deriving a worker's credibility in the case when there is no blacklisting is not as straightforward. In general, the probability of errors is higher, so we need to assign lower credibilities to workers. Deriving an exact conditional probability like Eq. 6.17, however, is difficult, since saboteurs can leave and return under new identities, creating many different possible cases to consider. Thus, we take a different approach.

First, we note that if we assume that workers who leave or get caught rejoin immediately, then the faulty fraction of the worker population stays constant at around f . This implies that the probability of a randomly chosen worker being bad is around f , and thus the probability of a randomly chosen answer being bad is $f \cdot s$, where s is the sabotage rate of the saboteurs. Unfortunately, however, we do not know s . We can, however, derive a reasonable *estimate*, \hat{s} , based on k , and use that instead. One such estimate is $\hat{s} = 1/k$, which we can intuitively arrive at by noting that a saboteur with a sabotage rate of $1/k$ would have an average survival period of k spot-checks. Using this estimate, we get the dubiousity of a worker without blacklisting as follows:

$$Db_P(P)_{\text{scnb}} = f \cdot \hat{s} = \frac{f}{k} \quad (6.18)$$

which gives us the following credibility metric:

$$Cr_P(P)_{\text{scnb}} = 1 - \frac{f}{k} \quad (6.19)$$

As shown in Sect. 5.5, this metric proves to work well in simulations, where it always achieves the desired final error rate $1 - \vartheta$, without overly sacrificing performance.

Credibility of Results. For now, we will simply assume that the credibility of a result R , $Cr_R(R)$, is simply equal to $Cr_P(R.\text{solver})$ where $R.\text{solver}$ is the worker which produced the result. In general, however, it is possible to distinguish it from the solver's credibility. For example, one possible way to guard against saboteurs who give good results at the

beginning to earn credibility, but then start giving more bad results later when they know their credibility is already high, may be to give results received later in the batch lower credibility than those received earlier (when presumably a saboteur would give more good answers to build up its credibility).

Credibility of Result Groups and Work Entries

If a work entry W has only one result R_1 so far, then $Cr_W(W)$ is simply $Cr_R(R_1)$ of the result, which, under our assumptions, is equal to the credibility $Cr_P(R_1.solver)$. If a work entry has several results, then we group the matching results together into results groups, and compute the credibility for each group based on the conditional probability of correctness, given the current combination of results received so far.

Suppose we have g groups, each denoted as G_a , for $1 \leq a \leq g$, with m_a members respectively. In the simple case where we assume all bad answers match, then g is at most two, and we know that if we have two groups, one of them is sure to have the correct results. In this case, we can compute the credibility of G_1 , as follows:

$$\begin{aligned} Cr_{G'}(G_1) \text{ (assuming bad answers match)} & \quad (6.20) \\ &= \frac{P(G_1 \text{ good})P(G_2 \text{ bad})}{P(\text{get 2 groups where each } G_a \text{ has } m_a \text{ members})} \\ &= \frac{P(G_1 \text{ good})P(G_2 \text{ bad})}{P(G_1 \text{ good})P(G_2 \text{ bad}) + P(G_1 \text{ bad})P(G_2 \text{ good})} \end{aligned}$$

with a symmetric formula for G_2 . Here, $P(G_a \text{ good})$ is the probability of all the results in G_a being good, computed as $\prod_{i=1}^{m_a} Cr_R(R_{ai})$ for all results R_{ai} in group G_a . Correspondingly, $P(G_a \text{ bad})$ is the probability of all the results in G_a being bad, given as $\prod_{i=1}^{m_a} (1 - Cr_R(R_{ai}))$.

More generally, if bad answers are not assumed to match, then not only can we get more than two results groups, but there is also a possibility of *all* of the result groups being bad. Taking these into account leads to the more general formula:

$$\begin{aligned} Cr_G(G_a) &= \frac{P(G_a \text{ good})P(\text{all others bad})}{P(\text{get } g \text{ groups, where each } G_a \text{ has } m_a \text{ members})} \quad (6.21) \\ &= \frac{P(G_a \text{ good}) \prod_{i \neq a} P(G_i \text{ bad})}{\prod_{j=1}^g P(G_j \text{ bad}) + \sum_{j=1}^g P(G_j \text{ good}) \prod_{i \neq j} P(G_i \text{ bad})} \end{aligned}$$

Works 1 and 998 in Fig. 6-6 show some examples of how Eq. 6.21 is used.

For another example, suppose that we do not use spot-checking, so that the credibility of a result R is simply $Cr_R(R) = 1 - f$ for all results, and suppose that we currently have two groups of results, with m_1 and m_2 results, respectively. In this case, for each group G_a , for $1 \leq a \leq 2$, we have $P(G_a \text{ good}) = Cr_R(R)^{m_a} = (1 - f)^{m_a}$, and $P(G_a \text{ bad}) = (1 - Cr_R(R))^{m_a} = f^{m_a}$, which gives us:

$$Cr_{G'}(G_1) \text{ (assuming bad answers match)} \quad (6.22)$$

$$= \frac{(1-f)^{m_1} f^{m_2}}{(1-f)^{m_1} f^{m_2} + f^{m_1} (1-f)^{m_2}}$$

and

$$Cr_G(G_1) = \frac{(1-f)^{m_1} f^{m_2}}{(1-f)^{m_1} f^{m_2} + f^{m_1} (1-f)^{m_2} + f^{(m_1+m_2)}} \quad (6.23)$$

Note that in computing the credibilities of result groups, unlike in computing the error rates in majority voting, assuming that bad answers need not all match (i.e., using Eqs. 6.21 and 6.23) is more conservative. Suppose, for example, that a work entry W_1 has two different results from two different workers who both happen to have a credibility of 0.99. Also suppose that another work entry, W_2 , also has two different results from two different workers, but in this case, these workers both happen to have a credibility of 0.8. If we assume that all bad answers match, then Eq. 6.22 gives us a credibility of 0.5 for both result groups, which intuitively makes sense since, in this case, we know for certain that one of the results groups is correct, so picking one at random gives us a 50% of picking the correct one. Intuitively, however, it seems reasonable to expect that W_2 should be given lower credibility than W_1 since W_2 's results come from less credible workers. Equation 6.23 takes this into account by allowing for the possibility of all the results groups being bad, giving us credibilities of 0.497 and 0.444 for W_1 and W_2 respectively. Thus we see that Eqs. 6.21 and 6.23 not only give us more conservative credibilities than Eqs. 6.20 and 6.22, but also take into account the absolute values of the worker credibilities and not just their relative values.

6.4.3 Using Credibility

There are a variety of ways to use credibility for fault-tolerance.

Spot-checking Alone

The simplest way to use credibility is to just depend on spot-checking alone. In this case, we just wait until one worker with sufficiently high credibility submits a result for a work entry, and then mark the work entry done. Note, however, that since spot-checking only reduces the dubiousity inversely with k , and not exponentially, using credibility with spot-checking alone is not practical unless the desired error rate, $\varepsilon_{acc} = 1 - \vartheta$, is not much smaller than f . Otherwise, it would take too long before the workers get spot-checked enough times to gain enough credibility to reach the threshold.

Voting Alone (Margin-based Voting)

Without spot-checking, the credibility of each worker stays constant, but the credibility of a work entry increases as more and more workers give the same result. By continuing to collect results until the credibility threshold ϑ is reached, we effectively implement a kind of *dynamic* voting scheme where the redundancy is not determined in advance but is made as large as necessary depending on f and ϑ . More precisely, this scheme is equivalent to *margin-based voting* or *m-ahead voting*, where we wait until one of the result groups has a

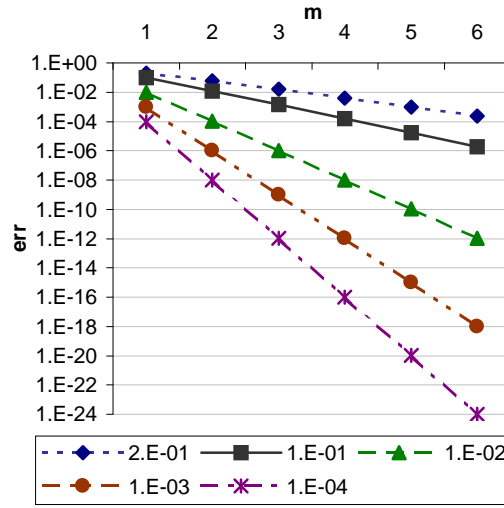


Figure 6-7: m -ahead margin-based voting: theoretical error rate for various values of φ and m .

margin of at least m more results than the other group. As we shall see, m -ahead margin-based voting is better than m -first majority voting not only because it automatically adjusts the redundancy to achieve the desired error rate, but also because the redundancy it requires is generally less than that required by majority voting to achieve the same error rate.

Assuming bad answers match. If we assume that all workers have the same credibility, $1 - f$, and if we assume that all bad answers match, then the minimum margin m by which a result group must be ahead to reach the desired threshold ϑ is constant. To see this, suppose that we have two groups, G_1 and G_2 with m_1 and m_2 results respectively, and suppose (without loss of generality) that G_1 is “ahead” of group G_2 by m results – i.e., $m_1 = m_2 + m$. Substituting $m_1 = m_2 + m$ into Eq. 6.22, we get the following credibility for group G_1 :

$$Cr_{G'}(G_1) = \frac{(1 - f)^m}{f^m + (1 - f)^m} \quad (6.24)$$

Note that this credibility does not depend on either m_1 or m_2 . Thus, we see that the probability of a result group being correct depends only on the margin m – i.e., how much ahead that result group is when we choose it.

Subtracting $Cr_{G'}(G_1)$ from 1, and making the equation more general by substituting the fault rate φ for f as we did in Sect. 6.3.2, we get the average error rate of m -ahead voting as follows:

$$\varepsilon_{m\text{-ahead}}(\varphi, m) = \frac{\varphi^m}{\varphi^m + (1 - \varphi)^m} \quad (6.25)$$

As shown in Fig. 6-7, this error rate is still exponential in form, but is now lower than that m -first majority voting (shown in Fig. 6-2) for the same m .

Given a target error rate ε_{acc} , and a corresponding credibility threshold $\vartheta = 1 - \varepsilon_{\text{acc}}$, we can solve Eq. 6.25 to get the following formula for the minimum m required to achieve the threshold:

$$m'_{\min}(\varphi, \vartheta) = \left\lceil \frac{\ln(\frac{1-\vartheta}{\vartheta})}{\ln(\frac{\varphi}{1-\varphi})} \right\rceil \quad (6.26)$$

$$\approx \lceil \log_{\varphi}(1 - \vartheta) \rceil = \lceil \log_{\varphi} \varepsilon_{\text{acc}} \rceil, \text{ if } \varphi \text{ and } \varepsilon_{\text{acc}} \text{ are not large} \quad (6.27)$$

This m corresponds to the *minimum* redundancy required by m -ahead voting. The actual redundancy, however, is unbounded since it is possible to keep on collecting results as long as no results group gets m results more than the other. Fortunately, however, in practice, this redundancy is not too large. We can derive this redundancy by considering this scheme as a form of the well-known “gambler’s ruin” problem, where in this case, the gambler starts with m dollars, makes \$1 dollar bets with a probability f of winning each bet, and plays until he either earns a net of m more dollars (equivalent to the case of getting m more bad results than good), or until he loses all his money (equivalent to the case of getting m more good results than bad). Doing so, we can show that if $\varphi < 1/2$, then the expected redundancy required can be bounded by $m/(1 - 2\varphi)$ [90], which is not much bigger than m if φ is small. Thus, for a given φ , the average redundancy required in margin-based voting to achieve an error rate less than ε_{acc} is approximately:

$$\begin{aligned} \text{Ave. redundancy of } m\text{-ahead voting} &\approx \frac{m'_{\min}(\varphi, \vartheta)}{1 - 2\varphi} \\ &\approx \frac{\lceil \log_{\varphi} \varepsilon_{\text{acc}} \rceil}{1 - 2\varphi} \\ &\approx \lceil \log_{\varphi} \varepsilon_{\text{acc}} \rceil, \text{ if } \varphi \text{ is small} \end{aligned} \quad (6.28)$$

Thus, we see that m -ahead voting effectively gets rid of the constant c in Eq. 6.3 while using only slightly more redundancy (i.e., $m/(1 - 2\varphi)$ vs. $m/(1 - \varphi)$). Comparing their performance, we find that for small values of φ and m , m -ahead voting is roughly the same as m -first voting, but as the required value of m to achieve the desired error rate gets larger, m -ahead voting gets comparatively faster. For values of φ greater than 10% and a target error rate ε_{acc} smaller than 1×10^{-6} , for example, m -ahead voting tends to take 1.5 to 2 times less redundancy than m -first voting to achieve the same error rate.

Assuming bad answers may not match. If we allow for multiple bad answers, then we can arrive at similar results by substituting $m_1 = m_2 + m$ into Eq. 6.23 instead of Eq. 6.22, and then solving for the minimum margin m required to reach the credibility threshold. In this case, the solution is not simple and is not constant, but depends on the current size of the smaller group (i.e., m_2). Its value is largest when $m_2 = 0$, and very quickly approaches Eq. 6.26 as m_2 becomes bigger. Thus at worst, the required m in this case is given by:

$$m_{\min}(\varphi, \vartheta, m_2 = 0) = \left\lceil \frac{\ln(\frac{(1-\vartheta)}{2\vartheta})}{\ln(\frac{\varphi}{1-\varphi})} \right\rceil \quad (6.29)$$

$$\approx \lceil \log_{\varphi} (\varepsilon_{\text{acc}}/2) \rceil \quad (6.30)$$

In this equation, the value before taking the ceiling is larger than that in Eq. 6.26, but only by a small amount. For example, it is only around 15% larger for $\varepsilon_{\text{acc}} = 0.01$ (for any $f < 0.5$), only around 7.5% larger for $\varepsilon_{\text{acc}} = 1 \times 10^{-4}$, and gets even closer to Eq. 6.26 as ε_{acc} becomes smaller. Added to the fact that this minimum m is even closer to Eq. 6.26 for $m_2 > 0$, this means that unless the values involved are near an integer boundary, the minimum m required would be the same as that required when we assume all bad answers match. Thus, in many cases, allowing bad answers not to match does not affect performance very much.

Voting and Spot-checking Combined

Although credibility-based fault-tolerance can be used with voting alone or spot-checking alone, it is best used to integrate voting and spot-checking together. In this case, we start with all workers effectively having a credibility of $1 - f$ and start collecting results. If the credibility threshold ϑ is low enough, and the batch is long, then by the time we go around the circular work pool, the workers may have already gained enough credibility (by passing spot-checks) to make their results acceptable. In this case, we do not need to do voting and we can reach our desired error rate with only the $1/(1 - q)$ redundancy due to the spotter works. If ϑ is high, then spot-checking would not be enough, so we start reassigning work, collecting redundant results, and voting.

Since spot-checking causes the dubiousity of surviving workers to shrink inversely with time, however, it allows us to reach the desired threshold in much less time than with voting alone. More precisely, spot-checking effectively reduces the fault rate φ in Eq. 6.26 by the the average number of times the surviving workers have been spot-checked, k . This reduces the base of the logarithm in Eq. 6.27, and thus reduces m_{min} dramatically as k increases in time. Furthermore, if we allow workers to accumulate credibility across batches, then k can become even larger, and can reduce the redundancy even further. In Sect. 5.5, for example, we simulated cases with 10 batches of $N = 10000$ work objects each, distributed across $P = 200$ workers, assuming $f = 20\%$ and $q = 10\%$, and we show that we can reach an error rate of 1×10^{-6} with an average slowdown of only around 3, compared to m -first voting's 32 and m -ahead voting's 16.

Another advantage of using credibility in combining voting and spot-checking is that it works well even if we cannot enforce blacklisting. By using the credibility metric from Eq. 6.19, we effectively neutralize the effect of saboteurs who only do a few pieces of work and then rejoin under a new identity. As shown in Sect. 6.5.2, there is now no advantage to doing so, and in fact, it seems that there is now more incentive for a saboteur to stay on for longer periods.

Using Credibility-based Voting for Spot-checking

Although using credibility to combine voting and spot-checking already works quite well, we can gain even more performance by using voting *for* spot-checking.

So far, we have assumed that a master spot-checks a worker by giving it a piece of work whose correct result is already known. Since this implies that either the master itself (as in the case in Sect. 3.5.2), or one of a few completely trusted workers, must do the work

to determine the correct result, we generally assume that the spot-check rate q needs to be small (i.e., less than 10%). Since k is roughly qn , this limits the rate at which credibilities increase and thus limits performance.

Fortunately, we can attain much better performance by using credibility-based voting as a spot-checking mechanism. That is, whenever one of a work entry's result groups reaches the threshold (such that the work entry can be considered done), we: (1) increment the k value of the solvers of the results in the winning group, and (2) treat those in the losing groups as if they had failed a spot-check (i.e., we remove them from the system and invalidate their other results).

If we assume that we have to do all the work at least twice, such that *all* results returned by a worker would have to participate in a vote, then using these votes to spot-check a worker implies that a worker will get spot-checked $k = n$ times – i.e., $1/q$ times as much as before when a worker could expect to get spot-checked only $k = nq$ times in a batch. This implies a corresponding decrease in the error rate and a corresponding increase in the credibility of good workers, which in turn allows the voting to go even faster. For example, if $q = 10\%$, this results in a worker getting spot-checked 10 times more, and a fault rate that is 10 times smaller. As noted in Sect. 6.3.5 this would improve the error rate 10^m times for a given m – in addition to the improvement we already have when using simple spot-checking. Or, viewed another way, this improves the base in Eq. 6.27, and allows us to achieve desired credibility thresholds in less time (i.e., with lower values of m). In the simulations in Sect. 6.5.2, for example, this reduces the slowdown required to achieve an error rate of 1×10^{-6} at $f = 20\%$ from around 3.5 (without blacklisting) to only around 2.5.

Note that this technique is only made possible by using credibility-based voting and ordinary spot-checking to “bootstrap” the whole process. Naïvely using traditional majority voting to spot-check workers would be dangerous because the chance of saboteurs outvoting good workers and thus getting them blacklisted would be significant, especially if f is not small. Credibility-based voting works because it guarantees that we do not vote until the probability that the vote will be right is high enough. Thus, it naturally limits the probability of good workers being outvoted to a very small value. Note, however, that we cannot start using voting for spot-checking until the result groups actually start reaching the threshold and voting. This implies that: (1) using voting for spot-checking is only beneficial when the redundancy is already at least two, and (2) we need to maintain normal spot-checking (at least for the first few batches) to allow the workers to gain enough credibility to reach the threshold early enough. Otherwise, we will not gain any improvement because we would either not do credibility-based voting at all (in the first case), or start doing it too late to make a big difference (in the second case).

6.5 Simulation Results

6.5.1 The Simulator

To verify our theoretical results, we have developed a Monte Carlo simulator that simulates the behavior of an eager scheduling work pool in the presence of saboteurs and various fault-tolerance mechanisms. This simulator uses a real eager scheduling work pool

implementation (taken from Bayanihan), but uses token work objects that do not contain any executable code. To simulate the workers and saboteurs, we create a list of P *worker entries* and randomly select fP of them to be saboteurs. We then simulate a computation done by these workers by going through the list in round-robin manner, each time simulating the action of the current worker contacting the master to return a result (for the work object it received in its *previous* turn) and to get new work. This assumes, as in Sect. 6.3.1, that all workers have exactly the same speed, so that the work is equally distributed among the workers, and each worker gets to take a turn before any other worker is allowed to take a second turn.

To simulate faults, we make each saboteur randomly decide to mark its result “bad” with probability s . At the same time, we simulate spot-checking by letting the master give out a spotter work randomly with probability q , and then check if the result returned for that work by its solver is marked bad. Then, at the end of batch (i.e., when all the work entries are marked done), we count the number of bad results among the final results, and divide that by N to get the error rate for the batch. We then create the next batch of work, and repeat this whole process using the same list of workers and saboteurs, keeping track of the error rates for each batch until the end of the simulated computation.

Since we assume that all workers have the same speed, we can estimate the *running time* of a batch by counting the number of rounds we make through the worker list before the batch ends. That is, each round represents the time it takes one worker to execute one work object (i.e., within that time, all P workers can finish 1 work object each in parallel), and the total number of rounds represents the total amount of time it takes to finish the batch in parallel. This metric works even if some workers temporarily or permanently leave the system due to blacklisting, for example. In addition, we can also compute the *slowdown* of the system by dividing the running time (measured by counting rounds) by the *ideal running time* given by N/P . This is useful for measuring the performance of the various fault-tolerance mechanisms.

Finally, to ensure statistical significance of our results, we use the Monte Carlo API described in Sect. 4.6.2 to do r runs of the simulator, each with the same parameters but with different, non-correlated, random number seeds, and then take the sample mean and the sample standard deviation, σ , of each variable of interest. Dividing σ by \sqrt{r} gives us an estimate of the expected standard deviation of the mean itself given r samples, and thus gives us a rough measure of the precision and accuracy of the mean.⁹

For our experiments, we ran $r = 100$ runs of simulated computation, each consisting of a sequence of 10 batches of $N = 10000$ work objects each, done by $P = 200$ workers. These numbers were chosen to be small enough to be simulatable in a reasonable amount of time, but large enough to provide good precision (i.e., the smallest measurable error rate is 1×10^{-7}) and to prevent blacklisting from killing all the saboteurs too early. In addition, the work-per-worker ratio, $N/P = 50$, was chosen to be large enough to show the effects of spot-checking, while still being representative of potential real applications. Also, having the computation go through 10 batches allows us to see the benefits of letting good workers gain higher credibility over time. When doing blacklisting, we only do *batch-limited*

⁹In general, the mean of r samples of a random variable with a distribution that has a mean μ and standard deviation σ , has a distribution with a mean of μ and a standard deviation of σ/\sqrt{r} [70]. That is, the mean becomes a more precise estimate of the original μ as we get more samples.

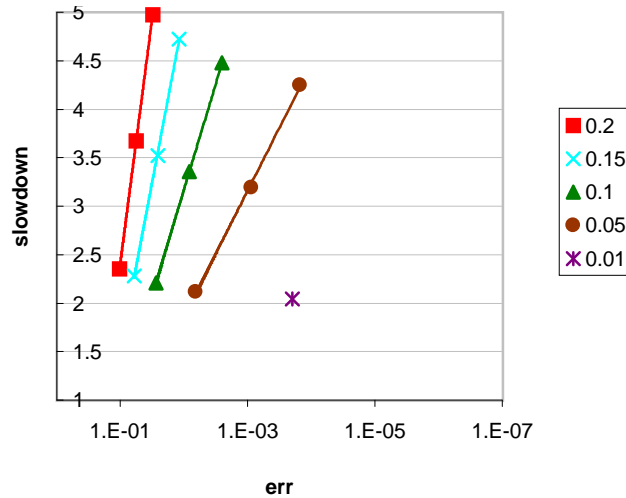


Figure 6-8: Majority voting: slowdown vs. maximum final error rate at various values of f and $m = \{2, 3, 4\}$.

blacklisting, which means that we allow blacklisted nodes to return at the start of the *next* batch. However, these return with a different worker ID and a clear record. Specifically, a returning saboteur's k is set back to zero and its credibility is correspondingly reset.

6.5.2 Results

Figures 6-8 to 6-23 show the experimental results we get from running our Monte Carlo simulator.

Majority voting. Figure 6-8 plots the resulting slowdown and error rate from majority voting given different values of the initial faulty fraction f (assuming a sabotage rate of 1). This graph is like Fig. 6-2 turned on its side, except that m is replaced by slowdown, and the values of f are different. As shown, when f is large, majority voting requires a lot of redundancy to achieve even relatively large error rates. Analytically extending the line for $f = 0.2$, we find that it would take a slowdown of more than 32 to achieve a final error rate of 1×10^{-6} . Note, however, that the slope becomes less steep as f becomes smaller. (Note that only one point for $f = 0.01$ is shown because the other points resulted in no errors in our experiments.)

Spot-checking with blacklisting. Figure 6-9 shows the error rates due to spot-checking with blacklisting, showing how the error rate varies with s as predicted in Eq. 6.4 and Fig. 6-4. As shown, in contrast to majority voting, spot-checking works well for large faulty fractions f . In fact, Fig. 6-9 shows that even when there are more saboteurs than good workers (i.e., in the first case, $f = 70\%$), we can still get relatively low error rates (i.e., in this case, $\approx 4.5\%$). This fact can be very useful for naturally fault-tolerant applications where such error rates may be acceptable already.

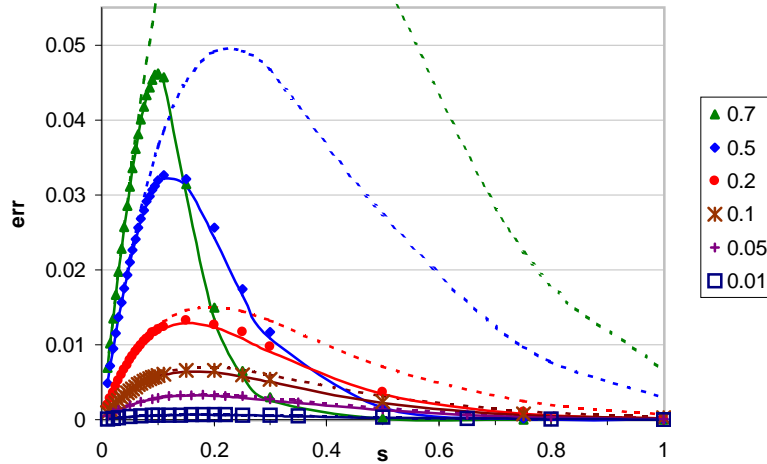


Figure 6-9: Spot-checking with batch-limited blacklisting: simulated and theoretical error rates vs. s at $q = 0.1$, $N = 10000$, $P = 200$ for $f = 0.7$ down to $f = 0.01$. The solid curves represent theoretically predicted error rates using the actual observed values of n , while the dashed curves represent a purely theoretical predictions assuming $n = (N/P)/(1 - q)$.

Note that in this case, n is not constant because remaining nodes end up doing more work as more nodes get blacklisted. This change in n causes the upper tail of the curves at $f = 0.5$ and $f = 0.7$ to go down, and the maximum point to shift slightly. As n is not straightforward to predict exactly, we compute the theoretical curves shown in Fig. 6-9 by substituting the actual n measured from the experiments into Eq. 6.4. As shown in Fig. 6-9, the experimental results match well with these theoretical results and confirm them. In practice, of course, we may not know the actual n in advance. In this case, however, it is safe to assume that $n = (N/P)/(1 - q)$ (i.e., the original worker's share plus the redundancy due to spot-checking), since we know that n would always be at least this much on average. The dashed curves in Fig. 6-9 show the predicted error rates we get by assuming $n = (N/P)/(1 - q)$, and show that these are indeed conservative upper bounds that are always greater than the actual average error rates.

Spot-checking without blacklisting. Figure 6-10 shows the error rates due to spot-checking with blacklisting, showing how the error rate varies with the length-of-stay l as discussed in Sect. 6.3.4. (Here, as in Sect. 6.3.4, we pessimistically assume that a caught saboteur knows when it has been caught and immediately returns under a new identity.) As shown, the actual error rates in this case are biggest for low l , and are bounded from above by f/ql (and, of course, f), just as predicted by Eq. 6.14 in Sect. 6.3.4.

Combining voting and spot-checking. Figures 6-11 and 6-12 shows the results of using voting and spot-checking together without credibility-based fault-tolerance, as discussed in Sect. 6.3.5.

Figure 6-11 shows the case with blacklisting enabled. As noted in Sect. 6.3.5, the resulting error rate is simply that which we would get by substituting the reduced fault rate due

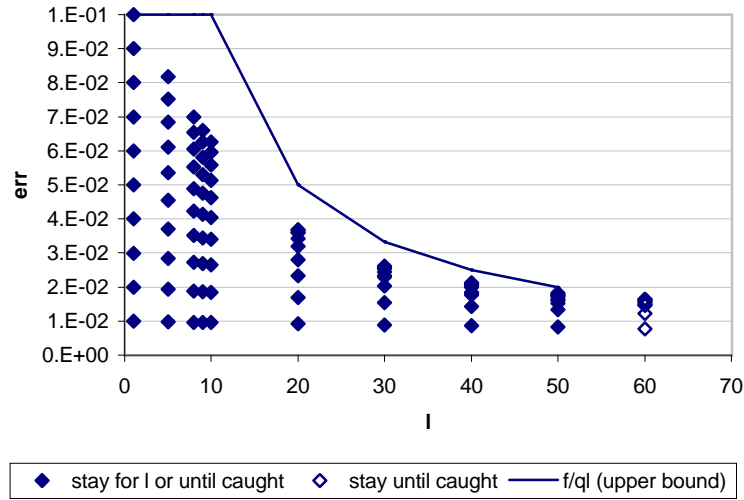


Figure 6-10: Spot-checking without blacklisting: simulated error rates and theoretical upper bound (f/ql) vs. length-of-stay l and various values of s , at $q = 0.1$, $N = 10000$, $P = 200$, $f = 0.1$.

to spot-checking, $\varphi_{scbl} = \varepsilon_{scbl} < \frac{f}{1-f} \frac{1}{qne}$, (from Eq. 6.8) into voting’s error rate as given by Eq. 6.1. Note that although some data points may seem to exceed their corresponding theoretical values, particularly in Fig. 6-11(b), this does not represent a failure in our model, but is most likely because the error rates at these points are close to the limit of precision of our experiments (i.e., 1×10^{-7}) and thus are more prone to variance.

Figure 6-12 shows the case without blacklisting. (For clarity, we just plot the maximum error rate here, instead of showing the error rates at various values of s as we do in Fig. 6-10.) As shown, voting with spot-checking but no blacklisting performs better than simple voting without spot-checking, but worse than voting with spot-checking and blacklisting. Note, however, that the error rate cannot be predicted or bounded by simply substituting $\varphi_{scnb} = err_{scl} < f/ql$ into Eq. 6.1. This is shown by the solid line, which actually represents an even more conservative bound, given by $f/(q \min(l, mN/P))$, to account for the fact that larger l values do not make the error rate smaller when l becomes greater than the work to be done by a worker in a batch, mN/P . As shown, even with this adjustment, the solid line becomes lower than the actual error rate at some point. Why this happens is currently not clear and can be a subject of future research.

Credibility-based voting with spot-checking and blacklisting. Figure 6-13 shows the results of using credibility-based voting and spot-checking with batch-limited blacklisting, using the credibility metric $Cr_P(P)_{scbl}$ from Eq. 6.17. Here, each group of points corresponding to a credibility level is divided into three curves corresponding to $f = 0.2$, 0.1 and 0.05 , respectively. Most significantly, this plot shows that, as intended, the average error rate never goes above $1 - \vartheta$, regardless of s and f .

One thing that is not shown in Fig. 6-13 is that while the maximum error rate remains roughly the same (as limited by $1 - \vartheta$), more and more slowdown is being needed to

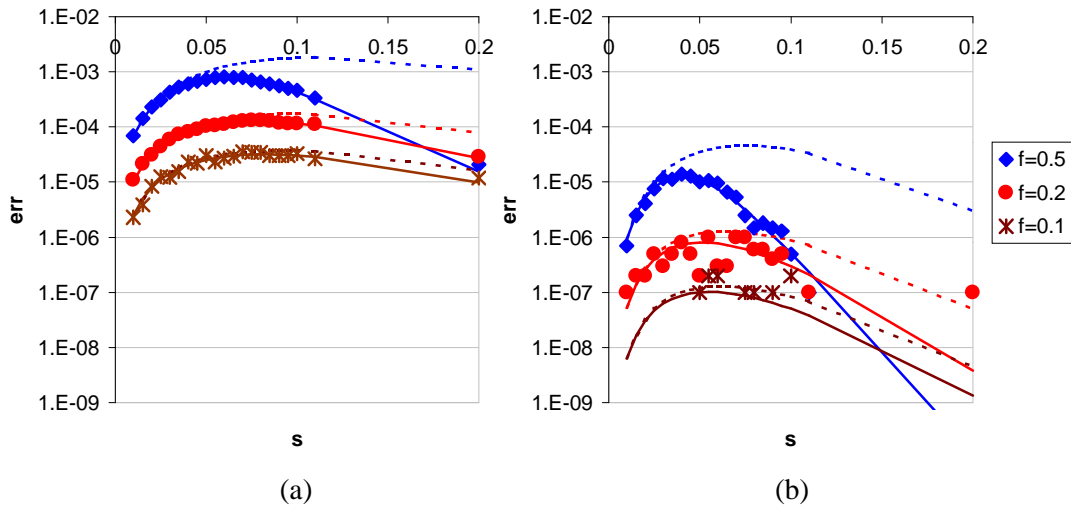


Figure 6-11: Combining voting and spot-checking with blacklisting: simulated and theoretical error rate vs. s at $q = 0.1$, $N = 10000$, $P = 200$ for $f = 0.5, 0.2, 0.1$ and (a) $m = 2$, (b) $m = 3$. The solid curves represent theoretically predicted error rates using the actual observed values of n , while the dashed curves represent a purely theoretical predictions assuming $n = m(N/P)/(1 - q)$.

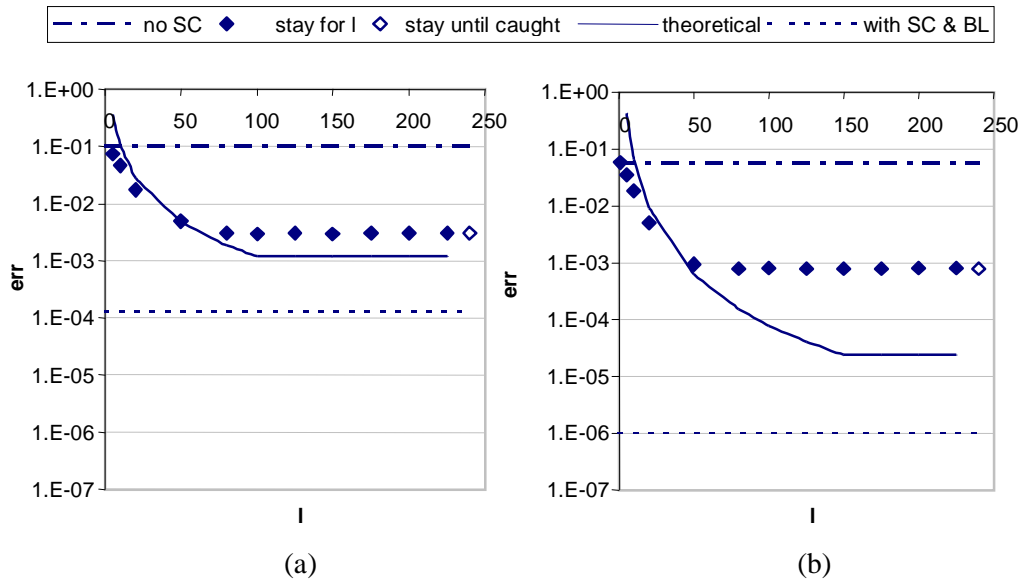


Figure 6-12: Combining voting and spot-checking without blacklisting: maximum simulated error rate vs. l at $f = 0.2$ and (a) $m = 2$, (b) $m = 3$.

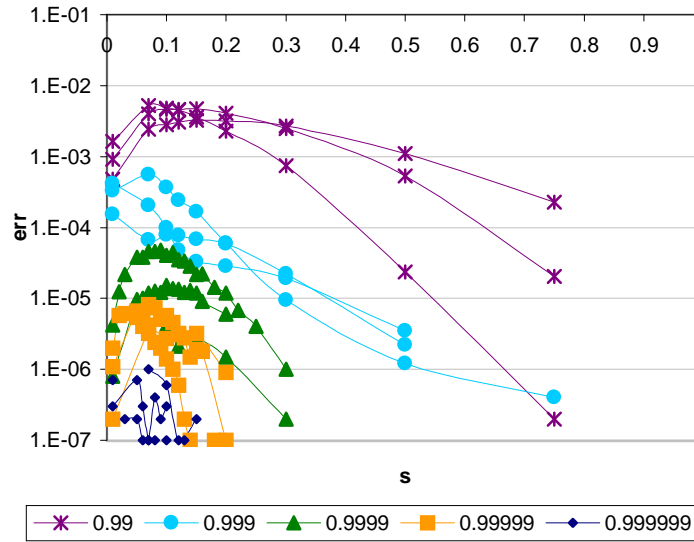


Figure 6-13: Credibility-based voting with spot-checking and batch-limited blacklisting: error rate vs. s at $f = \{0.2, 0.1, 0.05\}$.

guarantee the bounds on the error rate. The slowdown incurred in achieving the *maximum* error rate for a particular value of f and ϑ is shown in Fig. 6-14. Note how the slopes of the lines here are much better than those in simple majority voting, thus allowing us to achieve lower error rates in less time. For example, whereas majority voting would have required a slowdown of more than 32 to achieve an error rate of 1×10^{-6} for $f = 0.2$, here we only need around 3. Furthermore, note that credibility-based fault-tolerance allows us not to use voting at all if it is not necessary, as shown by the points with redundancy less than 2, which represent points where spot-checking was enough to reduce f down to the threshold $1 - \vartheta$ without requiring voting.

Figure 6-15 shows the effect of worker credibility on the slowdown. As shown, it takes less time to achieve the target error rates in the tenth batch than in the first batch. This is because by the tenth batch, good workers have already passed many spot-checks and thus already have correspondingly higher credibilities. These high credibilities allow us to accept results submitted by these workers more quickly, i.e., without having to redo them, and thus lead to finishing the batch sooner.

Credibility-based voting with spot-checking, no blacklisting. Figure 6-16 shows how credibility-based fault-tolerance works even in cases where we do not have blacklisting and saboteurs can leave after doing l work objects, and then come back under a new identity. In this case, we use the credibility metric $Cr_P(P)_{\text{scnb}}$ from Eq. 6.19, and measure the error rate at various values of s for $f = 0.2$ and $\vartheta = 0.9999$. As shown, even without blacklisting, we successfully guarantee that the error rate never exceeds $1 - \vartheta = 1 \times 10^{-4}$, regardless of l .

Interestingly, it seems that in the beginning, error rates are highest at $l = 1$ and decrease with l , as predicted in Sect. 6.3.4. However, at some point above $l = 120$, the error rates get

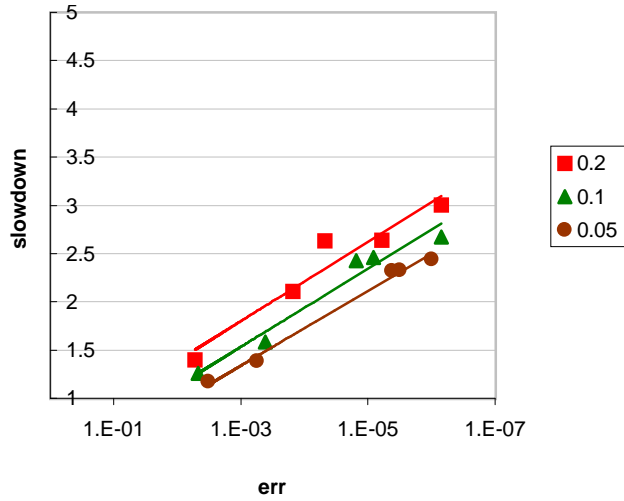


Figure 6-14: Credibility-based voting with spot-checking and batch-limited blacklisting: average slowdown vs. maximum final error rate at $\vartheta = 0.99, \dots, 0.99999$ at various values of f .

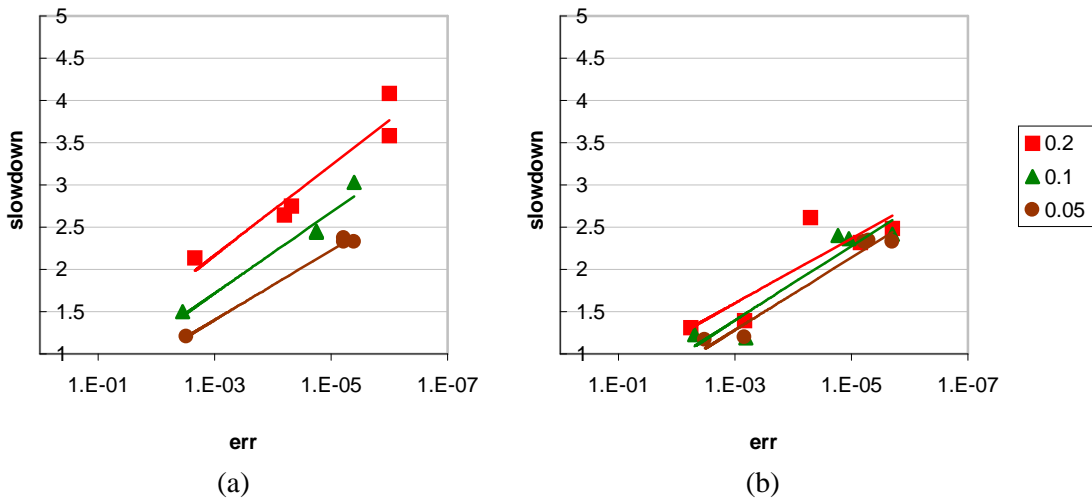


Figure 6-15: Credibility-based voting with spot-checking and batch-limited blacklisting: slowdown vs. maximum final error rate for: (a) first batch, (b) tenth batch.

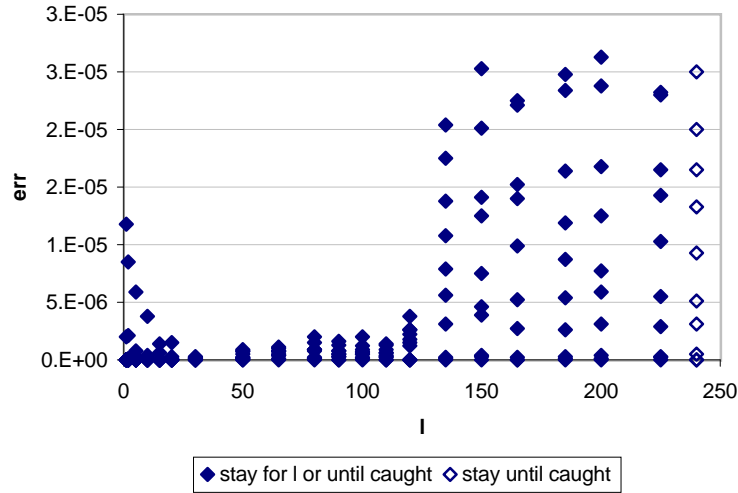


Figure 6-16: Credibility-based voting with spot-checking, no blacklisting: error rate vs. length-of-stay l at $f = 0.2$ and $\vartheta = 0.9999$ averaged over 10 batches.

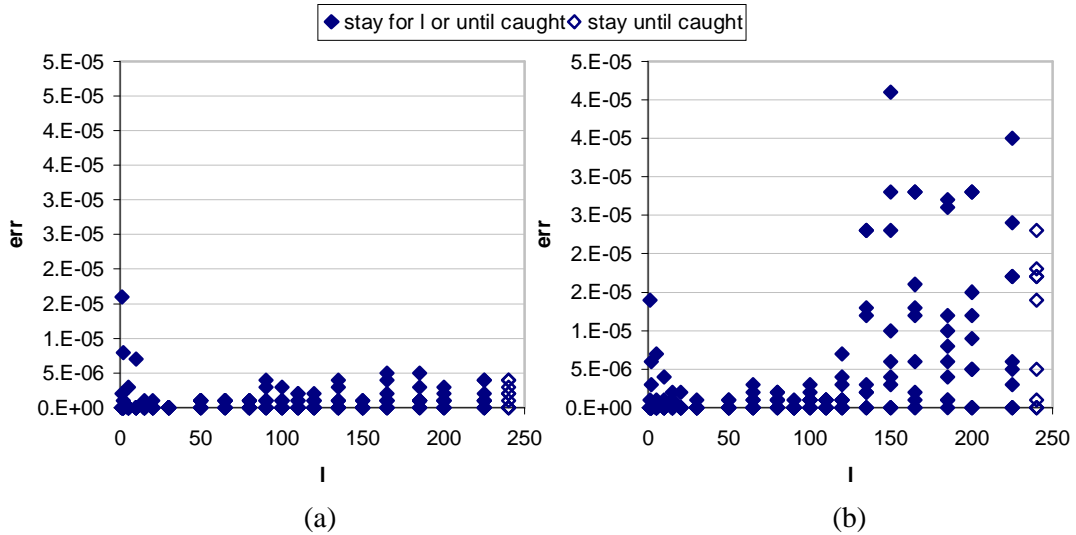


Figure 6-17: Credibility-based voting with spot-checking, no blacklisting: error rate vs. length-of-stay l at $f = 0.2$ and $\vartheta = 0.9999$ for: (a) first batch, (b) tenth batch.

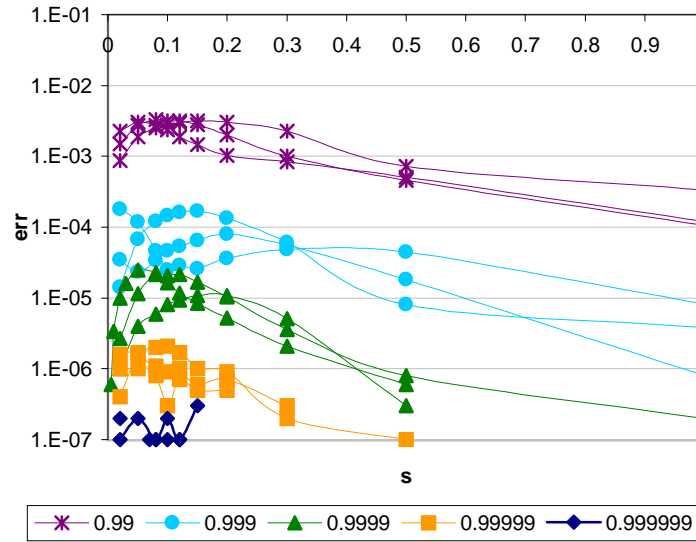


Figure 6-18: Credibility-based voting with spot-checking, no blacklisting: error rate vs. s at $f = \{0.2, 0.1, 0.05\}$, assuming saboteur stays until caught and then rejoins immediately.

dramatically larger, and seems to stay roughly constant. It is not yet clear exactly why this happens, but we can get a clue by looking at the corresponding plots for the first and last batches, shown in Fig. 6-17. Here, we see that the anomaly does not exist in the first batch, but exists in the last batch. (Although not shown, the anomaly exists in other batches after the first as well.) This leads us to suspect that the anomaly has something to do with saboteurs being able to survive until the next batch and keeping the credibility they gained in previous batches. In any case, it would seem from this graph that it is generally advantageous to stay as long as possible and not to quit early. Taking this strategy to the extreme, a saboteur can simply stay in the system and not leave prematurely unless it is caught. As shown in Fig. 6-16, this leads to high error rates relative to fixing l . Thus, for the rest of our experiments, we will assume that saboteurs follow this strategy.

Given this assumption, Fig. 6-18 shows the error rates we get with credibility-based voting and spot-checking without blacklisting, for different values of ϑ , using the credibility metric $Cr_P(P)_{scnb}$. Again, we note that the error never exceeds the threshold $1 - \vartheta$. The slowdown is shown in Figs. 6-19 and 6-20, which show how performance is still much better than with majority voting, although slightly less than that with blacklisting as shown in Figs. 6-14 and 6-15. This is expected since $Cr_P(P)_{scnb}$ is generally lower (and thus more conservative), than the one used with blacklisting $Cr_P(P)_{scl}$, and thus it takes slightly more redundancy to allow it to reach the desired threshold ϑ .

Using credibility-based voting for spot-checking. Finally, Figs. 6-21 to 6-23 show the results of using credibility-based voting to spot-check workers. Figure 6-21 shows how it still guarantees that the error rate threshold is reached, and Figs. 6-22 and 6-23 show the slowdown. As shown, the slope in Fig. 6-22 is even better than that of the case with blacklisting shown in Fig. 6-14. Here, we can now achieve an error rate of less than 1×10^{-6}

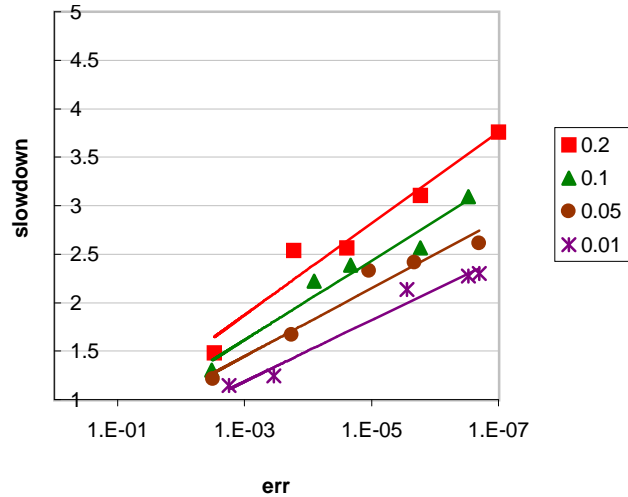


Figure 6-19: Credibility-based voting with spot-checking, no blacklisting: average slowdown vs. maximum final error rate at $\vartheta = 0.99, \dots, 0.999999$ at various values of f , assuming saboteur stays until caught and then rejoins immediately.

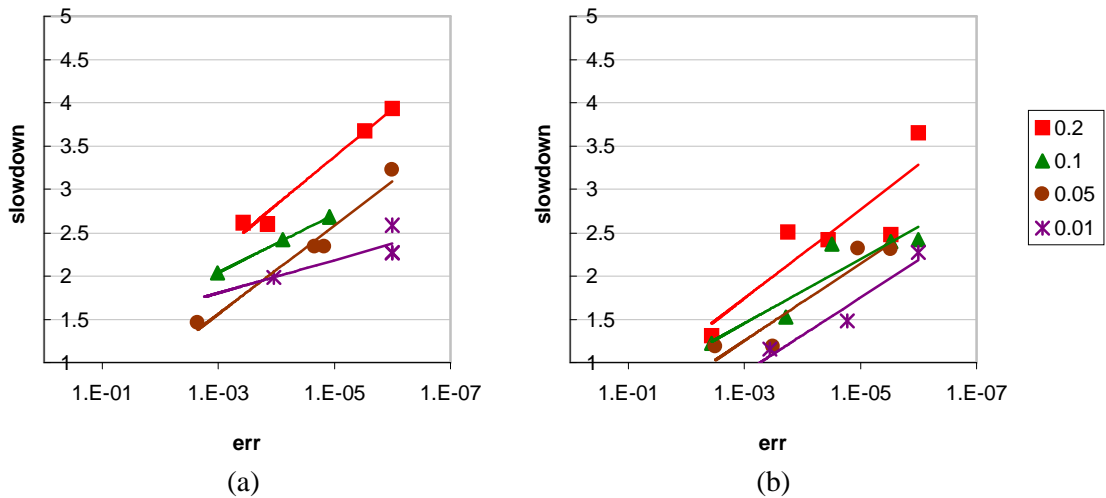


Figure 6-20: Credibility-based voting with spot-checking, no blacklisting: slowdown vs. maximum final error rate for (a) first batch, (b) tenth batch.

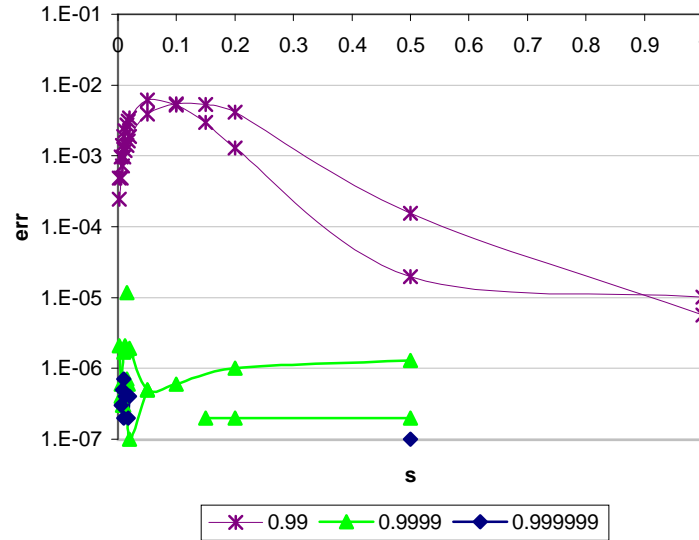


Figure 6-21: Using credibility-based voting for spot-checking, no blacklisting: error rate vs. s at $f = 0.2, 0.1, 0.05$ for various thresholds ϑ , assuming saboteur stays until caught and rejoins immediately.

from $f = 0.2$, with just a little over 2.5 redundancy. Comparing this with majority voting as shown in Fig. 6-8, this shows that for the same redundancy, we get an error rate that is almost 10^5 times better.

6.6 Conclusion

In this chapter, we have proposed new mechanisms for addressing the largely unstudied problem of sabotage-tolerance, and have demonstrated the potential effectivity of these mechanisms through mathematical analysis and simulation. A logical next step for research, therefore, is to implement and apply these techniques to real systems, and start benefitting from them. This should not be too difficult because the master-worker model to which these mechanisms apply is widely used today not only in volunteer computing systems but in other metacomputing and grid computing systems as well.

In the process of applying these mechanisms, questions may arise with respect to assumptions or implementation details. Some questions and variations that we can explore in further research include the following:

- **Handling cases where saboteurs can collude on *when to vote together*.** This would imply a change in $P(G_a \text{ bad})$ in Eq. 6.21.
- **Incorporating the use of checksums.** A worker which submits a result that fails a checksum would be treated as if it had been spot-checked and caught submitting a bad result. (A worker which passes a checksum, however, must still go through the regular spot-checking mechanisms, since we assume that saboteurs may be able to disassemble the code, and forge good checksums for bad data.)

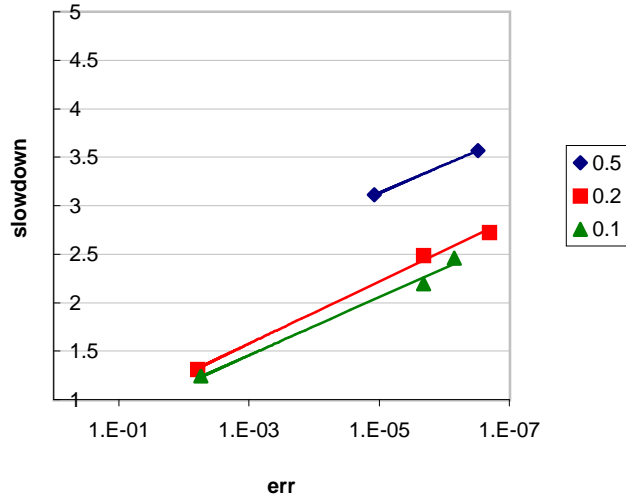


Figure 6-22: Using credibility-based voting for spot-checking, no blacklisting: average slowdown vs. maximum final error rate at $\vartheta = 0.99, 0.9999, \text{ and } 0.999999$ at various values of f , assuming saboteur stays until caught and rejoins immediately.

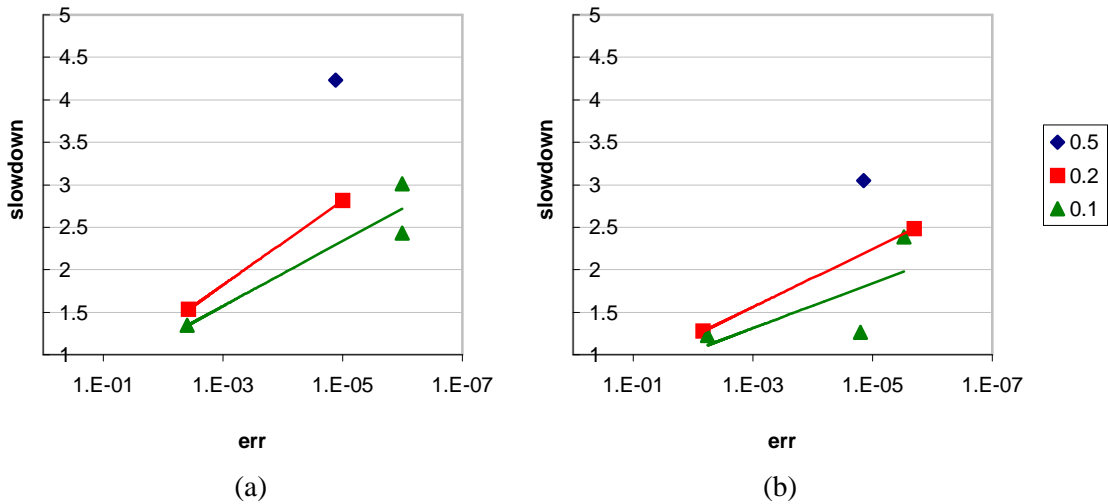


Figure 6-23: Using credibility-based voting for spot-checking, no blacklisting: slowdown vs. maximum final error rate for: (a) first batch, (b) tenth batch.

- **Incorporating the use of periodic obfuscation.** By obfuscating code with a checksum, we can reduce the probability of a saboteur forging a good checksum for bad data. If we can then estimate the probability that a saboteur can successfully forge a good checksum despite the obfuscation, we can use it in calculating the worker's dubiousity, and therefore its credibility. Since this dubiousity should be lower than the case without obfuscation, it should allow us to reach the desired credibility thresholds faster than before.
- **Incorporating the use of encrypted computation techniques.** An example would be using a random number to encode the computation in such a way as to prevent saboteurs from voting together by preventing their bad results from matching. This would imply a change in $P(G_a \text{ bad})$ for any result groups containing more than one result, since the probability that those groups would be formed by bad results would now be very small.
- **Not telling saboteurs when they are caught.** In the case where saboteurs can come back right away, the server can achieve lower error rates by continuing to give work to the saboteur, but ignoring its results. In this way, the saboteur cannot know when it has been caught, and cannot return right away. At best, it can leave after a reasonably short l . We have not analyzed this rigorously, but intuitively, it seems that this should lead to an error rate of around f/ql instead of f/ql for spot-checking without blacklisting. Also, in this case, using a credibility of $1 - f/ke$ instead of $1/fk$ might work now, even without blacklisting.
- **Using credibility in other ways.** So far, our use of credibility has been based on using an eager scheduling work pool. It is also possible to use credibility in other ways. For example, we can keep track of the credibilities of workers and only use workers when they have reached a desired threshold. (Note, however, that it is not clear that this particular scheme will work better than our current scheme, since it seems that it would take longer and would be more prone to non-Bernoulli saboteurs who wait until they get enough credibility to start performing sabotage. It may still be useful to look into this, though, in case other ideas similar to it prove to be useful.)
- **Incorporating other forms of spot-checking.** Another way to do spot-checking is to check a randomly selected fraction h of *all* results instead of randomly checking entire results with a probability q . This was suggested in Sect. 3.5.2, and is also related to the technique in [99] as described in Sects. 3.5.2 and 6.2.1. It would be useful to analyze the expected behavior of this form of spot-checking, and to come up with appropriate credibility metrics for this case. Intuitively, we suspect that in this case, the analysis and the credibility metrics would be very similar, with h replacing q and hk replacing k (since k is now equal to the total number of works done by a worker so far). However, we still need to do this analysis more rigorously.
- **Handling saboteurs that are not Bernoulli processes but who somehow change their sabotage rates in time.** For example, if a saboteur somehow knows in advance the total number of work objects, n , that it will be given in batch, then intuitively, it might be able to increase the error rate by increasing its sabotage rate depending on

how many work objects are left for it to do in a batch. For example, it can set the sabotage rate, s , equal to \hat{s}_{scbl}^* or s_{scnb}^* (from Eqs. 6.6 and 6.11, respectively) using the remaining number of work objects in place of n (such that for example, in the last pieces of work, it just always gives bad answers since it knows that it cannot hurt it anymore to get caught). With credibility-based fault-tolerance, the damage that a saboteur can cause by doing this might be even bigger since the master would trust it more during the later stages, which is exactly when the saboteur is increasing its sabotage rate.

This is an interesting problem for future research. One possible solution is for the master to likewise increase its spot-check rate q as the end of the batch approaches, in order to cancel the saboteurs' increasing s . (In fact, the master can do this more easily than saboteurs since the master actually knows the number of remaining work objects in the batch better than saboteurs do.) Whether or not we employ a countermeasure like this, however, we can still make credibility-based fault-tolerance work by computing a new dubiousity and credibility for each worker, possibly based on the number of objects remaining in the batch. As noted in Sect. 6.4.2, we can also adjust the credibility of results themselves depending on their age and the relative time within the batch that they were done. Finally, for even more security, we can also put a limit to the credibility of a worker by putting a limit, k_{max} , to the value that k can take (this is equivalent to only respecting the last k_{max} spot-checks that a worker has passed). This may make the necessary redundancy slightly larger, but the increase may be an acceptable price to pay for more security. (We can get a hint of how much larger the redundancy would be by looking at the difference between the slowdowns in the first and last batches in Figs. 6-15, 6-20, and 6-23 in Sect. 6.5.2.)

For better performance, we may vary the bounds on k depending on how much we intrinsically trust the worker – e.g., if we have verifiable and hard-to-forge identity and contact information from a worker, then we can give it a higher k bound and possibly also a higher initial credibility (i.e., a lower assumed f in its credibility). Random volunteers who do not give verifiable identity information will still be allowed to volunteer but would have lower k bounds and higher assumed f in their credibilities.

In light of these, and potentially many other, interesting research questions, one of the most significant contributions of this chapter is the *generality* of the credibility threshold principle. That is, credibility-based fault-tolerance is not limited to just using voting or spot-checking as described here, or to making the assumptions we made here, but can be used with other mechanisms and be adapted to other assumptions as well as long as we can derive the net effect of new assumptions or mechanisms on the conditional probabilities of results being correct.

Chapter 7

Conclusion

7.1 Summary

In this thesis, we have aimed to paint a big picture of volunteer computing's potentials not only by presenting its many potential forms and applications, but also by presenting new ways of addressing several technical challenges that need to be overcome in realizing these potentials.

In Chap. 2, we began by showing how the ideas behind volunteer computing apply beyond true volunteer computing systems, and can be used in other forms of volunteer computing as well, including private and collaborative networks, commercial systems, and NOIAs. At the same time, we also showed that realizing these potentials requires addressing new research challenges, particularly in the areas of accessibility, applicability, reliability, and economic issues. In the rest of the thesis, we focused on the first three of these, showing concrete ways of addressing them.

In Chap. 3, we addressed the issue of accessibility by proposing the use of web-based volunteer computing, and developing the Bayanihan system to demonstrate its benefits. Through Bayanihan, we showed how we can allow users to volunteer their machines needing nothing but a web browser, and also showed how doing so not only enables us to form parallel computing networks much faster than possible with more traditional systems, but also enables us to reach users who have previously been unable or unwilling to volunteer. These include such as users who are unwilling to go through the hassle and security risks of installing and uninstalling application software on their machines, and users whose machines and operating systems simply do not allow them to install software on their machines.

At the same time, we also addressed the issue of applicability by showing how using a platform-independent and object-oriented language such as Java can make developing volunteer computing systems easier for programmers and researchers as well. Using a highly modular object-oriented framework design, we successfully implemented several APIs and application frameworks that enable programmers to write a wide variety of applications. Furthermore, we also successfully developed and tested a number of techniques for addressing the various technical issues in volunteer computing, including eager scheduling for adaptive parallelism, voting and spot-checking for reliability, and volunteer

servers for scalability.

In Chap. 4, we used Bayanihan’s runtime system and APIs to explore several classes of master-worker applications, whose coarse-grain and embarrassingly parallel nature make them appropriate for volunteer computing systems. Here, we showed that volunteer computing has many other potential applications other than the *ad hoc* and esoteric ones for which they have been most well known so far. We also demonstrated the real practicality of volunteer computing by showing how it made our research on fault-tolerance mechanisms possible.

In Chap. 5, we took the applicability of volunteer computing even further by showing how applications need not follow the master-worker style or be embarrassingly parallel in order to be implemented on a volunteer computing system. By presenting the Bayanihan BSP programming interface, we showed how it is possible to write message-passing style applications on top of master-worker style volunteer computing systems, and thus expand the range of possible volunteer computing applications even further.

Finally, in Chap. 6, we presented new ideas and results on the largely unstudied problem of achieving reliability despite the presence of malicious volunteers. In addition to discussing general approaches such as choosing naturally fault-tolerant or verifiable computations, using authentication, encryption, and obfuscation, and using randomization and redundancy, we also presented and developed specific techniques such as voting and spot-checking. One of our most promising results is the idea of credibility-based fault-tolerance, which, in addition to allowing us to achieve our desired error rates in a small fraction of the time it would have taken traditional voting techniques, also promises to be adaptable for use with other techniques, or in other situations with different assumptions.

7.2 Future Work

Although we presented many new ideas and techniques in this thesis, these represent only the beginning of a vast space of possible future research in volunteer computing. Future work that we can foresee at this point include:

- **Improving and refining the ideas and techniques presented here.** At the end of each chapter, we mention several open questions and suggestions. We can continue our research by examining these.
- **Applying the ideas here to existing systems.** Most of the ideas presented here, including Bayanihan BSP and our fault-tolerance mechanisms apply to master-worker systems. Since most volunteer computing systems today already support master-worker systems, applying these techniques to other systems should be straightforward. Furthermore, we may also be able to merge these ideas with other ideas implemented by other systems such as Javelin’s scalability techniques and support for branch-and-bound algorithms [105], Charlotte’s distributed shared memory system [12], and the market mechanisms found in other projects [24, 88, 22, 97].
- **Developing mechanisms for better scalability.** Currently, Bayanihan uses a star topology, and is thus not fully scalable. Many other systems today such as

SETI@home also have basically star topologies. To maximize the benefits of volunteer computing, it would be useful to develop techniques for better scalability. Our preliminary experiments using volunteer servers and communicationally parallel applications demonstrate some promising possibilities, and can provide a good starting point for further research. Other approaches we can look into include Javelin's scalability mechanisms [104, 106], and peer-to-peer networks inspired by Napster and Gnutella. The latter includes Entropia, and a growing number of new projects such as those in the Intel [74] peer-to-peer working groups, and others featured in the O'Reilly's P2P web site [112].

- **Addressing economic issues and developing commercial and market-based systems.** In this thesis, we have mostly not discussed the economic issues involved in commercial and market-based systems, and have focused on lower-level technical concerns. Now that we have addressed the latter, it is time not only to apply our results to commercial systems, but also to start focusing on the economic issues and developing mechanisms for addressing these too. In doing so, we should remember that dealing with the economic issues does not only mean developing mechanisms for payment, but also considering hidden costs as described in Chap. 2.
- **Developing the idea of NOIAs.** Many of the ideas we have presented here are already being (independently) implemented by many other research groups and even commercial companies. One particularly promising idea that no one has implemented yet or even studied in depth, however, is that of NOIAs (networks-of-information-appliances), which we proposed in Chap. 2. As commercial volunteer computing systems start becoming more commonplace and popular, NOIAs seem to be the obvious next step. Thus, it would be very interesting and fruitful to start developing technology for NOIAs now. In addition to finding ways of extending the technology already available in today's volunteer computing systems, it may also be helpful to look into new emerging ideas, such as *amorphous computing* [2], that can be relevant to such systems.

Finally, it would also be fruitful to start considering the *social* issues involved in volunteer computing. As sharing one's idle computer time becomes easier and even financially rewarding, more and more people will start volunteering their machines, and more and more companies, organizations, and individuals, would start using, and even depending, on the processing power provided by these volunteers. This opens up a whole new world of possibilities, not only technical or economic but social as well.

7.3 Final Words

In this electronic age, it may be hard to find people still carrying their neighbors' houses on long bamboo poles – even in the rural villages of the Philippines. But in the global village that the Internet has become, the *bayanihan* spirit can live on through volunteer computing. It is our hope that the ideas we have presented here can help form a framework that, much like the bamboo framework of old, will allow people around the world to work together as one, and accomplish the seemingly impossible.

Appendices

Appendix A

Sample Code

This appendix contains sample code for three applications: brute-force factoring (described in Sects. 3.4.2 and 4.2.1), computing π through Monte Carlo methods (described in Sect. 4.6.3), and a sample program using the fault-tolerance simulator (described in Sect. 4.6.4).

A.1 Brute-Force Factoring

A.1.1 Work Object

```
package bayanihan.apps.factor;

import bayanihan.util.*;
import bayanihan.mixer.*;
import bayanihan.api.mw.data.*;
import bayanihan.api.mw.work.*;

public class FactorWork extends BasicWork
{
    // work specs
    // note: these are accessed via package access by FactorResult
    protected long target = 0;
    protected long first = 0;
    protected long last = 0;
    // client-local handles
    protected transient WorkEngineGUIIF weGUI;
    // Constructors //

    // this no-arg constructor is needed by HORB
    protected FactorWork() {}

    public FactorWork( long target, long first, long last )
    {
        this.target = target;
        this.first = first;
        this.last = last;
    }
    ////////////////////////////////////////////////////
    // Interface implementation methods //
    ////////////////////////////////////////////////////

    ////////////////////////////////////////////////////
}
```

```

// Initialization //
////////////////////

/*
 * WorkIF methods
 */

/**
 * Get a handle to the WorkEngineGUI from the Engine, so
 * we can report progress and results to local user.
 * This method is called by Engine when work is received.
 */
public void init( MixoutIF engine, Params params )
{
    super.init( engine, params );

    // get a handle to the WorkEngineGUI from the Engine
    this.weGUI = (WorkEngineGUIIF)engine.getIF( WorkEngineGUIIF.class );
}

////////////////////
// Public Instance methods //
////////////////////

/*
 * WorkIF methods
 */

/**
 * Finds factors of target between first and last (inclusive),
 * and returns this in a FactorResult object.
 */
public Object doWork()
{
    long curnum;
    long runLength = ( last - first + 1 ) / 5;
    long runsleft = runLength;
    FactorResult result = new FactorResult( this );

    for ( curnum = first; curnum <= last; curnum++ )
    {
        if ( ( target % curnum ) == 0 )
        {
            // target is divisible by curnum,
            // add factor pair to result and report it to
            // the local engine GUI

            long othernum = target / curnum;

            Factoring f = new Factoring( target, curnum, othernum );

            result.addFactoring( f );

            if ( weGUI != null )
            {
                weGUI.showResult( f );
            }
        }

        // have local engine GUI report progress every runLength tries
        runsleft--;

        if ( runsleft == 0)

```



```

        {
            runsleft = runLength;

            if ( weGUI != null )
            {
                weGUI.showStatus( "checking " + curnum );
            }

            java.lang.Thread.yield();
        }
        else if (runsleft < 0)
        {
            runsleft = 0;
        }
    }

    return result;
}

/**
 * Returns the number of possible factors tried.
 * This is used for measuring performance.
 */
public long getSize()
{
    return ( last - first + 1 );
}

////////////////////////////////////
// Overridden Object methods //
////////////////////////////////////

public String toString()
{
    return ( "[" + target + "," + first + "," + last + "]" );
}
}

```

A.1.2 Result and Related Objects

Factoring Object

```

package bayanihan.apps.factor;

/**
 * A single result containing one factoring of the target.
 */
public class Factoring implements bayanihan.Serializable
{
    protected long target, num, othernum;

    protected Factoring() {}

    public Factoring( long target, long num, long othernum )
    {
        this.target = target;
        this.num = num;
        this.othernum = othernum;
    }

    public String toString()
    {
        return ( target + ": [" + num + "," + othernum + "]" );
    }
}

```

```

    }
}

```

Result Object

```

package bayanihan.apps.factor;

import java.util.*;

/**
 * This Object, returned by doWork, contains several Factoring Objects
 * consisting of the factorings of the target number within the
 * Work object's range.
 */
public class FactorResult implements bayanihan.Serializable
{
    protected long target = 0;
    protected long first = 0, last = 0;
    protected Vector factorings = new Vector();

    ////////////////
    // Constructors //
    ////////////////

    protected FactorResult() {}

    public FactorResult( long target, long first, long last )
    {
        this.target = target;
        this.first = first;
        this.last = last;
    }

    public FactorResult( FactorWork fWork )
    {
        this( fWork.target, fWork.first, fWork.last );
    }
    // Methods used by FactorWork //

    public void addFactoring( Factoring f )
    {
        this.factorings.addElement( f );
    }

    public int size()
    {
        return this.factorings.size();
    }

    public Enumeration elements()
    {
        return this.factorings.elements();
    }

    public Factoring[] getFactorings()
    {
        Factoring[] fa = new Factoring[ this.size() ];
        int i = 0;

        for ( Enumeration e = this.elements(); e.hasMoreElements(); )
        {
            fa[i] = (Factoring)e.nextElement();
            i++;
        }
    }
}

```

```

    }

    return fa;
}

// toString() method //

public String toString()
{
    return ( factorings.size() + " factors in [" +
            + first + "," + last + "]" for " + target );
}
}

```

A.1.3 Config and Request Object

```

package bayanihan.apps.factor;
import bayanihan.util.*;

/**
 * Used to request a new target; also used as a ResetMsg.
 */
public class FactorConfig implements
    bayanihan.Serializable, Parameterizable
{
    public static final String P_target = "target";
    public static final String P_inc = "inc";

    public long target;
    public long inc;

    protected FactorConfig() {}

    public FactorConfig( long target, long inc )
    {
        this.target = target;
        this.inc = inc;
    }

    public void parameterize( Params params )
    {
        long nTarget = 0;
        long nInc = 0;

        nTarget = params.getLong( P_target, this.target );
        this.target = Util.checkRange( nTarget, 1, Long.MAX_VALUE,
            this.target );

        long targRoot = (long)Math.sqrt( this.target );

        nInc = params.getLong( P_inc, this.inc );
        this.inc = Util.checkRange( nInc, 1, targRoot, targRoot );
    }

    public String toString()
    {
        return ( "Target: " + target + ", inc: " + inc );
    }
}

```

A.1.4 Watcher GUI Object

```

package bayanihan.apps.factor;

import java.awt.*;
import java.util.Enumeration;

import bayanihan.util.*;
import bayanihan.ui.*;
import bayanihan.data.*;
import bayanihan.remote.*;
import bayanihan.fwk.*;
import bayanihan.api.mw.data.*;
import bayanihan.api.mw.watch.*;
import bayanihan.api.mw.request.*;
import bayanihan.api.mw.timed.*;

public class FactorWatchGUI extends BasicWatchGUI
{
    public static final int FACTORS_LIST_HEIGHT = 8;
    public static final int WINDOW_SIZE = 7;

    protected FactorConfig config = new FactorConfig( 0, 0 );
    protected Panel statsPanel;
    protected FactorConfigPanel configPanel;
    protected FactorResultsPanel resPanel;

    ////////////////////////////////////////////////////
    // Constructors //
    ////////////////////////////////////////////////////

    public FactorWatchGUI()
    {
    }

    ////////////////////////////////////////////////////
    // Initialization //
    ////////////////////////////////////////////////////

    /*
     * Parameterizable
     */

    public void parameterize( Params params )
    {
        super.parameterize( params );
    }

    public void init()
    {
        this.setTitle( "Factor Watcher connected to " + this.serverHostName );
        this.setLayout( new BorderLayout() );

        this.statsPanel = this._createStatusPanel();
        this.add( "North", this.statsPanel );

        this.add( "Center", this._createFactorPanel() );

        this.pack();
        repaint();
    }
    public Panel _createFactorPanel()

```

```

{
    this.configPanel = new FactorConfigPanel();
    this.resPanel = new FactorResultsPanel();
    return GUIUtil.mergeNC( this.configPanel,
                           this.resPanel );
}

////////////////////////////////////
// Interface implementation methods //
////////////////////////////////////

/**
 * Hook: draw core object; subclasses should call super
 * at the <em>end</em> if msgCore is of unknown type
 */
protected void _drawWatchMsgCore( MsgIF watchMsg, Object msgCore )
{
    if ( msgCore instanceof FactorResult )      // result msg
    {
        this.drawFactorResult( watchMsg, (FactorResult)msgCore );
    }
    else if ( msgCore instanceof FactorConfig ) // reset msg
    {
        this.reset( (FactorConfig)msgCore );
    }
    else super._drawWatchMsgCore( watchMsg, msgCore );
}

/**
 */
protected void drawFactorResult( MsgIF watchMsg, FactorResult res )
{
    // check for SolverWatchMsg

    long solver = SolverWatchMsg.UNKNOWN_PID;

    SolverWatchMsg swmsg =
        (SolverWatchMsg)DataUtil.getIF( watchMsg, SolverWatchMsg.class );

    if ( swmsg != null )
    {
        solver = swmsg.getSolverID();
    }

    this.resPanel.drawFactorResult( res );
}

////////////////////////////////////
// Other methods //
////////////////////////////////////

public synchronized void reset( FactorConfig fc )
{
    Trace.t( 3, "Resetting ..." );
    this.config = fc;
    this.configPanel.updateConfigPanel( this.config );
    this.resPanel.reset();
    this.repaint();
}

protected void submitConfigRequest()
{

```

```

Trace.t( 3, "Making request ", this.config );
this.makeRequest( this.config );
}

////////////////////////////////
// Inner Classes //
////////////////////////////////

public class FactorConfigPanel extends Panel
{
    protected Button submitButton;
    protected NamedTextField targetTF, incTF;

    //////////////////////////////////
    // Constructors //
    //////////////////////////////////

    public FactorConfigPanel()
    {
        this.init();
    }

    //////////////////////////////////
    // Accessor methods //
    //////////////////////////////////

    //////////////////////////////////
    // Initialization //
    //////////////////////////////////

    public void init()
    {
        this.setLayout( new BorderLayout() );
        this.add( "North", this._createConfigPanel() );

        this.submitButton = new Button("Submit Request");
        this.add( "Center", GUIUtil.mergeNC( this.submitButton, null ) );
    }

    protected Panel _createConfigPanel()
    {
        Panel cPanel = new Panel();

        cPanel.setLayout( new GridLayout( 2, 1 ) );

        cPanel.add( this.targetTF = new NamedTextField( "Target:", 40 ) );
        cPanel.add( this.incTF = new NamedTextField( "inc:", 20 ) );

        return cPanel;
    }

    //////////////////////////////////
    // Event-handling //
    //////////////////////////////////

    public boolean action( Event e, Object arg )
    {
        if ( e.target == this.submitButton )
        {
            this.readConfigValues( FactorWatchGUI.this.config );
            FactorWatchGUI.this.submitConfigRequest();
        }
    }
}

```

```

    }
    return false;
}

//////////
// Other methods //
//////////

/**
 * Reads configvalues from fields and puts them in
 * the parameter config.
 */
public void readConfigValues( FactorConfig fc )
{
    long nTarget = 0;
    long nInc = 0;

    nTarget = Util.parseLong( targetTF.getText(), fc.target );
    fc.target = Util.checkRange( nTarget, 1, Long.MAX_VALUE,
                                fc.target );

    long targRoot = (long)Math.sqrt( fc.target );

    nInc = Util.parseLong( incTF.getText(), fc.inc );
    fc.inc = Util.checkRange( nInc, 1, targRoot, targRoot );
}

/**
 * updates the TextFields in the GUI which show the range
 */
public void updateConfigPanel( FactorConfig fc )
{
    this.targetTF.setText( String.valueOf( fc.target ) );
    this.incTF.setText( String.valueOf( fc.inc ) );
}
}

public class FactorResultsPanel extends Panel
{
    protected long count;
    protected NamedLabel countLabel;
    protected List factorsList;

    //////////
    // Constructors //
    //////////

    public FactorResultsPanel()
    {
        this.init();
    }

    //////////
    // Initialization //
    //////////

    public void init()
    {
        this.setLayout( new BorderLayout() );
        this.countLabel = new NamedLabel( "Factor pairs: ", 20 );
        this.add( "North", countLabel );
        this.factorsList = new List( FACTORS_LIST_HEIGHT, false );
        this.add( "Center", factorsList );
    }
}

```

```

    }

    ////////////////
    // Other methods //
    ////////////////

    public void reset()
    {
        this.count = 0;
        this.countLabel.setText( String.valueOf( this.count ) );
        this.factorsList.clear();
    }

    public void drawFactorResult( FactorResult res )
    {
        this.count += res.size();

        for ( Enumeration e = res.elements(); e.hasMoreElements(); )
        {
            Factoring f = (Factoring)e.nextElement();
            this.factorsList.addItem( f.toString() );
        }
        this.countLabel.setText( String.valueOf( this.count ) );
    }
}
}

```

A.1.5 Program Object (Passive Style)

```

package bayanihan.apps.factor;
import java.util.Enumeration;
import bayanihan.util.*;
import bayanihan.mixer.*;
import bayanihan.remote.*;
import bayanihan.api.mw.*;
import bayanihan.api.mw.program.*;
import bayanihan.api.mw.data.*;
import bayanihan.api.mw.work.*;
import bayanihan.api.mw.watch.*;
import bayanihan.api.mw.request.*;
import bayanihan.api.mw.timed.*;

public class FactorProg extends BasicMWProgram
{
    // default values

    public static final long DIGITS_17 = 12345678901234567L;
    public static final long DEF_TARGET = DIGITS_17;
    public static final long DEF_INC = 1000000L;

    protected FactorConfig config = new FactorConfig( DEF_TARGET,
                                                       DEF_INC );

    ////////////////
    // Initialization //
    ////////////////

    /**
     * Allows FactorConfig object to be parameterized through command-line
     * or .prj file.
     */
    public void parameterize( Params params )
    {

```



```

    super.parameterize( params );

    Params.parameterize( this.config, params );
}

/**
 * Specifies default watcher GUI class
 */
public Object getDefault( String key, Object defValue )
{
    if ( key.equals( BasicMWProj.P_watchGUIClassName ) )
    {
        // use FactorWatchGUI

        return FactorWatchGUI.class.getName();
    } else
    {
        return defValue;
    }
}

////////////////////////////////////
// Interface implementation methods //
////////////////////////////////////

//
// MWProgramIF methods
//

public void startProgram()
{
    this._createNewBatch();
}

/**
 * Posts WatchMsg when a result is accepted
 * from a worker. This allows us to
 */
public void resultFinalized( WorkEntryIF workEntry,
                             ResultMsgIF resMsg )
{
    long workID = workEntry.getWorkID();

    // we get final result from workEntry (after voting and all)
    // and then post a WatchMsg in watch pool for watchers
    watchPool.postWatchMsg( new BasicResultWatchMsg( workEntry ) );
}

/**
 * This is called when all the work objects in the batch are done;
 * in this case, we increment the target of the current config.
 */
public synchronized void allDone()
{
    // report allDone to log File and print results to log File

    Trace.log( 1, "FactorProg.allDone() for target ",
              this.config, true );

    logResults();

    // create a new batch factoring the *next* number

    this.config.target++;
    this._createNewBatch();
}

```

```

}

////////////////////////////////////
// Request Handling methods //
////////////////////////////////////

// RequestHandlerIF
public synchronized void handleRequest( AdvocSLIF advoc, Object request )
{
    Trace.t( 2, "Request received from advoc ", advoc, ": ", request );
    if ( request instanceof FactorConfig )
    {
        this._reset( (FactorConfig)request );
    }
}

////////////////////////////////////
// convenience methods //
////////////////////////////////////
/**
 * Changes target config and then creates and starts new batch.
 */
protected void _reset( FactorConfig req )
{
    // make sure workPool is not distributing work
    // before changing this.config
    this.batchCtrl.setIdle( true );

    this.config = req;
    this._createNewBatch();
}
/**
 * Called in _reset, startProgram, and allDone
 * Puts new work in work pool based on this.config,
 * and starts batch.
 */
protected void _createNewBatch()
{
    // clear work pool and
    // post a BasicWatchMsg with the current config to tell client's
    // watcherGUI to reset with a new config

    this._clearForNewBatch( new BasicWatchMsg( this.config ) );

    this.createWork( this.config );

    this._startNewBatch();
}
/**
 *
 */
protected void createWork( FactorConfig cfg )
{
    long first, last, targroot;
    long numWork = 0;

    targroot = (long)Math.sqrt( cfg.target );

    Trace.log( 1, "Searching for factors from 1 to " + targroot +
        " in blocks of " + cfg.inc + ".", true );

    first = 1;

```

```

do
{
    last = first + ( cfg.inc - 1 );

    if ( last > targroot )
    {
        last = targroot;
    }

    FactorWork w = new FactorWork( cfg.target, first, last );

    WorkEntryIF workEntry = this._addWork( w );

    numWork++;

    first = last + 1;
} while ( first <= targroot );

}

protected void logResults()
{
    Trace.log( 1, "Results of factoring " + config.target, true );

    // _processAllResults is a convenience method that
    // goes through all the final results in the work pool
    // and calls processResult on the supplied WorkEntryProcessor
    // object (here defined as an anonymous inner class)

    this._processAllResults(
        new BasicWorkEntryProcessor()
        {
            public void processResult( Object result )
            {
                if ( result instanceof FactorResult )
                {
                    Factoring[] factorings =
                        ((FactorResult)result).getFactorings();

                    for ( int i = 0; i < factorings.length; i++ )
                    {
                        Trace.log( 1, "", factorings[i], true );
                    }
                }
            }
        }
    );
}
}

```

A.1.6 Program Object (Active Style)

```

package bayanihan.apps.factor;
import java.util.Enumeration;
import bayanihan.util.*;
import bayanihan.mixer.*;
import bayanihan.remote.*;
import bayanihan.api.mw.*;
import bayanihan.api.mw.program.*;
import bayanihan.api.mw.data.*;
import bayanihan.api.mw.work.*;
import bayanihan.api.mw.watch.*;
import bayanihan.api.mw.request.*;

```

```

import bayanihan.api.mw.timed.*;

public class ActiveFactorProg extends ActiveMWProgram
{
    // default values

    public static final long DIGITS_17 = 12345678901234567L;
    public static final long DEF_TARGET = DIGITS_17;
    public static final long DEF_INC = 1000000L;

    protected FactorConfig config = new FactorConfig( DEF_TARGET,
                                                    DEF_INC );

    ////////////////////////////////////////////////////
    // Initialization //
    ////////////////////////////////////////////////////

    /**
     * Allows FactorConfig object to be parameterized through command-line
     * or .prj file.
     */
    public void parameterize( Params params )
    {
        super.parameterize( params );

        Params.parameterize( this.config, params );
    }

    /**
     * Specifies default watcher GUI class
     */
    public Object getDefault( String key, Object defValue )
    {
        if ( key.equals( BasicMWProj.P_watchGUIClassName ) )
        {
            // use FactorWatchGUI

            return FactorWatchGUI.class.getName();
        } else
        {
            return defValue;
        }
    }

    ////////////////////////////////////////////////////
    // Interface implementation methods //
    ////////////////////////////////////////////////////

    //
    // run() method
    //

    public void run()
    {
        // set up new ResultListener as an anonymous inner class
        setupResultListener();

        // the real loop
        while ( true )
        {
            // do parallel step
            FactorConfig request = doParallelFactor( this.config );
            // print results to log file
            logResults();
        }
    }
}

```

```

    // check if parallel step was interrupted by a request
    if ( request != null )
    {
        // if there is a request, print note in log file,
        // then change the config

        Trace.log( 1, "Note: computation was interrupted. " +
            "Factors may be incomplete", true );

        this.config = request;
    } else
    {
        // if there is no request, just increment the current target
        this.config.target++;
    }
}

}

////////////////////////////////////
// Parallel methods //
////////////////////////////////////

/**
 * Finds factors according to FactorConfig until
 * request is made, or all done.
 * Returns request if any, or returns null if all done
 * without any request. Note that processing
 * is not done here but has to be done in resultListener
 * or in run().
 */
public FactorConfig doParallelFactor( FactorConfig cfg )
{
    //--- set-up parallel step ---//

    Trace.log( 1, "Creating new batch: ", cfg, true );

    // clear work pool and
    // post a BasicWatchMsg with the current config to tell client's
    // watcherGUI to reset with a new config
    this._clearForNewBatch( new BasicWatchMsg( this.config ) );
    // create work
    createWork( cfg );

    //--- start parallel step ---//

    _startNewBatch();

    //--- wait for request or allDone ---//
    Object request = null;
    FactorConfig fcRequest = null;

    do
    {
        // waits for request or allDone.
        // Returns null if allDone without request
        request = this._waitForRequest();
        if ( request == null )
        {
            Trace.t( 1, "all done. No request received" );
        }
        else if ( request instanceof FactorConfig )
        {
            Trace.t( 1, "stopping batch" );
        }
    }
}

```

```

        // stops the batch and sets
        // this.waiting for allDone to allow us to fall
        // through below
        _stopBatch();
        fcRequest = (FactorConfig)request;
    }
    else
    {
        Trace.err( "unknown Request type: " + request );
    }
} while ( this.waitingForAllDone );

//--- end parallel step ---//
// at this point, either all done (in which case, fcRequest would be null)
// or batch stopped due to request

return fcRequest;
}

////////////////////////////////////
// convenience methods //
////////////////////////////////////

protected void setupResultListener()
{
    // set up new ResultListener as an anonymous inner class
    this.setResultListener(
        new BasicResultListener()
        {
            public void resultFinalized( WorkEntryIF workEntry,
                                         ResultMsgIF resMsg )
            {
                long workID = workEntry.getWorkID();

                // we get final result from workEntry (after voting and all)
                // and then post a WatchMsg in watch pool for watchers
                _postWatchMsg( new BasicResultWatchMsg( workEntry ) );
            }
        }
    );
}

/**
 * Fills work pool with FactorWork objects according to the
 * target and inc values in the FactorConfig parameter.
 */
protected void createWork( FactorConfig cfg )
{
    long first, last, targroot;
    long numWork = 0;

    targroot = (long)Math.sqrt( cfg.target );

    Trace.log( 1, "Searching for factors from 1 to " + targroot +
              " in blocks of " + cfg.inc + ".", true );

    first = 1;

    do
    {
        last = first + ( cfg.inc - 1 );

        if ( last > targroot )
        {

```

```

        last = targroot;
    }

    FactorWork w = new FactorWork( cfg.target, first, last );

    WorkEntryIF workEntry = this._addWork( w );

    numWork++;

    first = last + 1;
} while ( first <= targroot );
}

protected void logResults()
{
    Trace.log( 1, "Results of factoring " + config.target, true );

    // _processAllResults is a convenience method that
    // goes through all the final results in the work pool
    // and calls processResult on the supplied WorkEntryProcessor
    // object (here defined as an anonymous inner class)

    this._processAllResults(
        new BasicWorkEntryProcessor()
        {
            public void processResult( Object result )
            {
                if ( result instanceof FactorResult )
                {
                    Factoring[] factorings =
                        ((FactorResult)result).getFactorings();

                    for ( int i = 0; i < factorings.length; i++ )
                    {
                        Trace.log( 1, "", factorings[i], true );
                    }
                }
            }
        }
    );
}
}

```

A.2 Computing Pi through Monte Carlo Methods

A.2.1 Work Object

```

package bayanihan.apps.mcpi;

import bayanihan.api.mc.*;

/**
 * Monte-Carlo code for computing Pi
 */
public class PiWork extends MCSingleRun
{
    // members
    protected int iters;

    // Constructors //
}

```

```

/**
 * Needed for HORB compatibility.
 * Defaults to iters = 1;
 */
public PiWork()
{
    this.iters = 1;
}
//////////
// Initialization //
//////////

public void setConfig( Object config )
{
    if ( config instanceof Integer )
    {
        this.iters = ((Integer)config).intValue();
    }
}

//////////
// Overridden abstract methods //
//////////

/**
 * Does work and returns an Object as the result.
 * (The Engine takes care of wrapping the Object in a ResultPacket.)
 */
public Object doWork()
{
    int count = 0;
    PiResult result = null;
    int updateIters = this.iters / 5;

    for( int i = 0; i < this.iters; i++ )
    {
        if ( ( i % updateIters ) == 0 )
        {
            this.showStatus( i + "/" + iters + " iterations done." );
        }

        double x = this.rng.nextDouble()*2 - 1; // random from -1.0 to 1.0
        double y = this.rng.nextDouble()*2 - 1; // random from -1.0 to 1.0

        if ( ( x*x + y*y ) <= 1.0 )
        {
            count++;
        }
    }

    result = new PiResult( this.iters, count );

    return result;
}

/**
 * Returns number of iterations.
 */
public long getSize()
{
    return this.iters;
}

```



```

////////////////////////////////////
// toString() method //
////////////////////////////////////

public String toString()
{
    return "PiWork: seed = 0x" + Long.toHexString( this.rng.getSeed() ) +
        ", iters = " + this.iters;
}
}

```

A.2.2 Result Object

```

package bayanihan.apps.mcpi;

/**
 * Result object for Monte-Carlo Pi program
 */
public class PiResult implements bayanihan.Serializable
{
    protected int iters = 0;
    protected int count = 0;

    /**
     * needed for HORB compatibility
     */
    public PiResult()
    {
    }

    public PiResult( int iters, int count )
    {
        this.set( iters, count );
    }

    public void set( int iters, int count )
    {
        this.iters = iters;
        this.count = count;
    }

    public int getIters() { return this.iters; }

    public int getCount() { return this.count; }

    public double getRatio() { return (double)count / (double) iters; }

    public double getPi() { return this.getRatio() * 4; }

    public String toString()
    {
        return "Pi = (" + count + "/" + iters + ") * 4 = " + getPi();
    }
}

```

A.2.3 Statistics Collector Object

```

package bayanihan.apps.mcpi;

import bayanihan.api.mc.*;

public class PiStat extends MCCollector implements bayanihan.Serializable

```

```

{
    protected DStat pi = new DStat();

    ////////////////////////////////////////////////////
    // Accessor methods //
    ////////////////////////////////////////////////////

    public DStat getPiDStat()
    {
        return this.pi;
    }

    public double getPi()
    {
        return this.pi.getMean();
    }

    ////////////////////////////////////////////////////
    // Abstract method implementations //
    ////////////////////////////////////////////////////

    public void reset()
    {
        this.pi.reset();
    }

    /**
     * Adds iters and count from PiResult
     */
    public void processResult( Object result )
    {
        if ( result instanceof PiResult )
        {
            this.addSample( ((PiResult)result) );
        }
    }

    ////////////////////////////////////////////////////
    // addSample methods //
    ////////////////////////////////////////////////////

    public void addSample( PiStat piStat )
    {
        this.pi.addDStat( piStat.getPiDStat() );
    }

    public void addSample( PiResult result )
    {
        this.pi.addSample( result.getPi() );
    }

    ////////////////////////////////////////////////////
    // toString() method //
    ////////////////////////////////////////////////////

    public String toString()
    {
        return "Pi = " + pi.getMean() + " +/- " + pi.getMeanStdDev() + " std dev.";
    }

    public String fullStatString()
    {
        return "Pi = " + pi.fullStatString();
    }
}

```

```
}

```

A.2.4 Program Object

```
package bayanihan.apps.mcpi;

import bayanihan.util.*;
import bayanihan.api.mc.*;
public class PiProg extends ActiveMCPProg
{
    public static final String P_workSize = "workSize";

    protected int workSize = 1000000;

    protected long batch = 0;

    protected PiStat overallPiStat;

    public PiProg()
    {
        super();
    }

    ////////////////////////////////////////////////////
    // Initialization //
    ////////////////////////////////////////////////////

    public void parameterize( Params params )
    {
        super.parameterize( params );

        this.workSize = params.getInt( P_workSize, this.workSize );
    }

    ////////////////////////////////////////////////////
    // Interface implementation methods //
    ////////////////////////////////////////////////////

    ////////////////////////////////////////////////////
    // run() method //
    ////////////////////////////////////////////////////

    public void run()
    {
        this.batch = 0;
        this.overallPiStat = new PiStat();
        this.programStatus = this.overallPiStat;

        Integer config = new Integer( this.workSize );

        while ( true )
        {
            PiStat batchStat = new PiStat();

            doBatch( PiWork.class, config, batchStat );

            this.overallPiStat.addSample( batchStat );

            Trace.log( 1, "Batch #" + this.batch + " done. ", true );
            Trace.log( 1, "BatchStat: " + batchStat.fullStatString(), true );
            Trace.log( 1, "Overall PiStat ", this.overallPiStat.fullStatString(),
                true );
        }
    }
}

```

```

        this.batch++;
    }
}

```

A.3 Fault-tolerance Simulator

A.3.1 Sample Program Object

```

package bayanihan.apps.ftsim;

import java.io.*;
import bayanihan.util.*;

/**
 * This program performs a parametric analysis by
 * scanning through various combinations of
 * credibility threshold, faulty fraction (f),
 * and sabotage rate (s).
 */
public class CredScanProgNB extends FTsimProg
{
    ////////////////////////////////////////////////////
    // Constructors //
    ////////////////////////////////////////////////////

    public CredScanProgNB()
    {
        super();

        // this class specifies the simulator version
        // which contains the formulas used for credibility
        // in this case, this class uses the credibility
        // metric for spot-checking without blacklisting

        this.simRunClass = SimRunWorkCredNB.class;
    }

    ////////////////////////////////////////////////////
    // Initialization //
    ////////////////////////////////////////////////////

    /**
     * Read parameters from .prj file into this.config
     */
    public void parameterize( Params params )
    {
        super.parameterize( params );

        this.outFileName = "csNB";
    }

    ////////////////////////////////////////////////////
    // run() method //
    ////////////////////////////////////////////////////

    public void run()
    {
        // create a text file with .txt extension
        PrintStream out = this.createTextOutputFile( this.outFileName );

```

```

// create a binary file with .dat extension
ObjectOutputStream oos = this.createObjectOutputFile( this.outFileName );

// the following are the parameter settings to try

double credArr[] = { 0.999999, 0.999999, 0.9999, 0.999, 0.99 };

double fArr[] = { 0.2, 0.1, 0.05, 0.01 };

double sArr[] = { 0.01, 0.02, 0.03, 0.05, 0.07, 0.09,
                 0.1, 0.12, 0.14, 0.16, 0.18,
                 0.20, 0.22, 0.25, 0.3, 0.35,
                 0.5, 0.65, 0.8, 1.0 };

// loop through different values of f
for ( int i = 0; i < fArr.length; i++ )
{
    this.config.fractBad = fArr[i];
    this.config.cF = fArr[i];

    // loop through different values of credibility threshold
    doCredScan( sArr, credArr, out, oos );
}

try
{
    oos.close();
} catch ( IOException e )
{
    Trace.log( 1, "ERROR closing oos: ", e, true );
}

out.close();
}

////////////////////////////////////
// Parallel methods //
////////////////////////////////////

protected void doCredScan( double[] sArr, double[] credArr,
                          PrintStream out, ObjectOutputStream oos )
{
    // loop through different values of credibility threshold

    for ( int c = 0; c < credArr.length; c++ )
    {
        this.config.cThresh = credArr[c];

        // loop through different values of s
        doSScan( sArr, out, oos );
    }
}

protected void doSScan( double[] sArr,
                       PrintStream out, ObjectOutputStream oos )
{
    // loop through different values of s

    for ( int i = 0; i < sArr.length; i++ )
    {
        this.config.probSabotage = sArr[i];

        SimRunResultCollector collector = new SimRunResultCollector( config );
    }
}

```

```
doBatch( this.simRunClass, this.config, this.numRuns, collector );

// write Serialized output
try
{
    oos.writeObject( collector );
    oos.flush();
    // clear oos "memory" to save memory
    oos.reset();
} catch ( IOException e )
{
    Trace.log( 1, "ERROR writing to oos: ", e, true );
}

// write text output
if ( i == 0 )
{
    out.println( this.tabbedHeaders( collector ) );
}
out.println( this.tabbedFullStatsString( collector ) );
out.flush();

Trace.log( 1, "Result: \n", collector, true );

System.gc();
}
}
```

Appendix B

Encrypted Computation and Obfuscation

This appendix contains the contents of an unpublished survey and critique paper [134] that presents and discusses techniques for protecting executable code from malicious users trying to reverse engineer or modify the code for their own purposes. As platform-independent, easily decompilable, and mobile code platforms such as Java come into increasingly widespread use, such techniques are becoming necessary not only to guard against software piracy and the theft of intellectual property, but also to protect users of mobile agents from attacks by malicious hosts trying to influence the agents' behavior. In the context of this thesis, such techniques would also be helpful in preventing computational sabotage, as described in Sects. 2.3.3 and 6.2.

In this paper, we discuss three main techniques – *encrypted computation* (also called *mobile cryptography*) [127], *obfuscation* [30], and *time-limited blackbox security* [71]. We present an overview and a critique of each technique, and conclude with some suggestions on possible directions for future research.

B.1 Motivation

With the growing popularity of Java and other mobile code systems that allow executable code to be automatically downloaded and executed on a user's machine, much attention and research has been focused on protecting users from malicious programs that seek to steal data or do harm to the user's machine. Unfortunately, however, not as much research has been focused on the reverse problem – that of protecting the executable code from attempts by malicious users to reverse engineer or modify the code for their own purposes. This is not so surprising since in the past, reverse engineering from native binary code has generally been a difficult (though not impossible) task. As platform-independent and easily decompilable binary formats such as Java bytecode become more and more common today, however, one can no longer ignore this problem.

In this paper, we provide a survey of techniques currently being developed to address this problem. We discuss three main techniques – *encrypted computation* (also called *mobile cryptography*) [127], *obfuscation* [30], and *time-limited blackbox security* [71]. We present

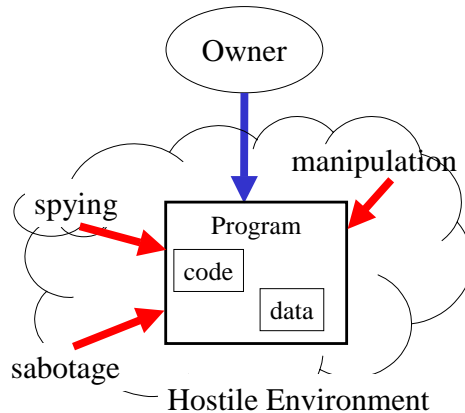


Figure B-1: Programs can be attacked in hostile environments.

an overview and a critique of each technique, and conclude with some suggestions on possible directions for future research.

B.2 Overview

B.2.1 The Problem

In most general-purpose computer systems today, the code and data of an executable program are exposed to the execution environment, and can be read, disassembled, and modified by a determined hacker. As shown in Fig. B-1, this means that once a program owner allows his or her program to run on an untrusted machine, there is usually nothing stopping the owner of that machine from trying to spy on the program's data, manipulate the program's behavior, or damage (sabotage) the program in some way.

The problem, therefore, is to find ways to protect programs from such hostile execution environments. Ideally, we would like to have *blackbox security*¹ – programs must be protected in such a way that a host executing a program it did not create:

1. cannot read, or at least cannot understand, the program's code and data, and
2. cannot modify the program's code and data.

That is, all that a host can do would be to feed the blackbox some input, run it, and get some results.

B.2.2 Applications

An effective blackbox security scheme that would prevent malicious users from reading, understanding, and modifying someone else's programs would have many applications.

¹The following definition is based on Hohl's definition [71], but extends the idea from agents to programs in general (not just agents), and allows code and data to be read as long as it is not understood.

Protection of Intellectual Property. The hiding of trade secrets and proprietary algorithms from competitive espionage has always been a concern in the software industry. In recent years, the problem has been worsened by the growing popularity of Java [59] and other software platforms that have an easily decompilable executable format. (For links to some commercial and non-commercial Java decompilers, see [29].) A technique which can somehow prevent others from understanding the decompiled Java code would thus be very useful for hiding such trade secrets. It can also be used as an additional deterrent to *software piracy* by using it to hide the workings of mechanisms for registration keys, trial versions, software watermarks [32], etc.

Protection of Mobile Agents. As more and more people start using autonomous mobile agents, it is becoming vital to find ways to protect these mobile agents from attacks by the hosts on which they run [127, 71]. In an e-commerce application, for example, where an agent may visit different vendors' host machines while shopping for the best offer, there are many ways in which a dishonest vendor may try to manipulate the agent into buying from it [71]. If the malicious vendor can understand and modify the code, for example, he can directly change the behavior of the agent to make it buy from him, regardless of better offers from other vendors. If the vendor cannot change the code directly, but can read and change the data, he can still manipulate the agent by changing the values of critical data, such as the best price seen by the agent so far, in his favor. Worse, the vendor may be able to steal secret information from the agent such as its private keys, digital signing functions, and digital cash, and use these to make purchases and transactions at the agent's owner's expense.

Market-based Volunteer Computing. Blackbox security would also be useful in *market-based volunteer computing*, where a parallel computation is done by users offering use of their machines' spare cycles in exchange for compensation. In such cases, we need to prevent dishonest users from trying to get paid without actually doing their work, and from generating erroneous results intended to sabotage the entire parallel computation.

B.2.3 Possible Solutions

In the following sections, we will examine three different approaches to the problem of protecting programs from hostile environments: *encrypted computation*, *obfuscation*, and *time-limited blackbox security*. For each of these techniques, we will give an overview, a critique of its strengths and weaknesses, and give some suggestions for future research.

B.3 Encrypted Computation

Encrypted computation seeks to provide true blackbox security by employing provably secure cryptographic techniques to allow an untrusted host to perform useful computation without understanding what it is computing.

Figure B-2 shows one form of encrypted computation. Here, Alice has a secret function f that she wants to run on Bob's machine using some input x provided by Bob. To prevent

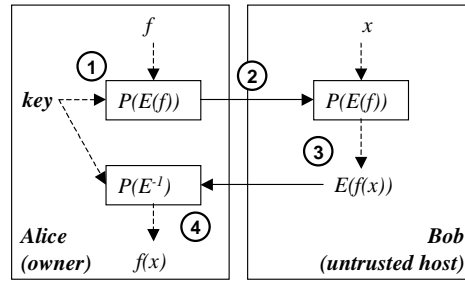


Figure B-2: Computing with encrypted functions (CEF).

Bob from understanding or modifying f , Alice encrypts f using her private key and an encryption scheme E to obtain a program $P(E(f))$ for computing $E(f)$ (step 1). She then sends this program to Bob (step 2), who receives the program and runs it on his input x . The program produces the *encrypted* result $E(f(x))$, which Bob sends back to Alice (step 3). Finally, Alice runs a decryption program $P(E^{-1})$ on $E(f(x))$ to obtain the desired value $f(x)$ (step 4). In this way, Bob is able to do useful computation without learning either f or $f(x)$.

This example depicts what is sometimes known as *computing with encrypted functions* (CEF) [128]. Here, f is encrypted, but x is known to Bob. CEF can also be extended to what Sander and Tschudin call *function hiding* [129], where the value of $f(x)$ is returned to Bob after step 4 in Fig. B-2. Function hiding can be used to protect intellectual property from theft and piracy. Suppose f is some proprietary algorithm Alice has developed for solving a problem of interest to many people. With function hiding, Alice can sell $(P(E(f)))$ and let users run it on their own machines using their own input. The program would produce encrypted results, which users can then ask Alice to decrypt. In this way, Alice can protect her intellectual property (f), and at the same time charge users for its use through a sort of “pay-per-use” scheme.

Another form of encrypted computation is *computing with encrypted data* (CED), shown in Fig. B-3. Here, Alice provides the input data x as well, but encrypts it so that Bob cannot learn its value. (She may or may not allow Bob to understand f .) CED is useful in applications such as market-based parallel computing, where Alice would like to use Bob’s computational power without allowing Bob to learn what Alice is doing. Note that CED is can be seen as a special case of CEF if the function f can be redefined to include the data x .

B.3.1 Basic Examples

As a simple example of encrypted computation, consider the case where $f(x)$ is the function $x + c$. In this case, Alice can encrypt f by simply adding some random number r to c to get the encrypted function $E(f(x)) = x + c'$, where $c' = c + r$. She can then decrypt the output by simply subtracting r from it. Since r can be any number, Bob cannot deduce c from c' and therefore cannot learn the original f . Neither can he manipulate the computation to make it produce a specific output value since he does not know what number Alice would subtract from his output. In this way, Alice is able to protect f from spying

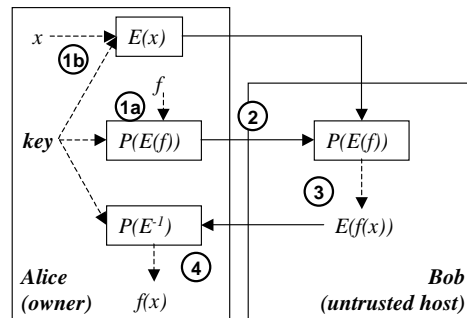


Figure B-3: Computing with encrypted data (CED).

and manipulation. Note however, that this scheme is not secure against sabotage. That is, there is nothing preventing Bob from returning a random number to Alice instead of actually performing the computation. Also, this scheme is not efficient since it costs Alice more computation to encrypt and decrypt the data than to compute the function itself. Thus, it would be cheaper for Alice to just ask Bob for x and compute the function herself.

A similar, but better example is outlined in [136], and shows CED applied to the discrete logarithm problem. This algorithm is much more efficient, since the discrete logarithm computation (which will be solved on Bob's machine) is a hard problem, and is thus likely to dwarf Alice's encryption and decryption overhead. Moreover, Alice can easily check the answer using exponentiation (which is much easier than solving for the discrete logarithm), so the algorithm is safe from sabotage. (Note, that this example is slightly different from the first example since it is doing CED, not CEF.)

B.3.2 Sander's and Tschudin's work

Recently, Sander and Tschudin, have proposed the application of encrypted computation techniques to the problem of protecting mobile code from malicious hosts [128, 127], as well as protecting intellectual property and secret functions in general [129]. So far, they have developed techniques for the encrypted computation of polynomials. In this section, we give an overview and critique of their techniques.

Polynomial Evaluation using Homomorphic Encryption Schemes

Sander and Tschudin show how encrypted polynomial evaluation can be done using *homomorphic encryption schemes*.

An *algebraically homomorphic* encryption scheme E is one where there exists easily computable operations PLUS and MULT which take encrypted inputs $E(x)$ and $E(y)$ and can produce encrypted outputs $E(x+y)$ and $E(xy)$ respectively without having to decrypt and learn x and y . With such a scheme, encrypted polynomial evaluation becomes easy – Alice need only give Bob a program where all numerical values in the polynomial (i.e., the coefficients and the input data) are encrypted, and where addition is replaced by PLUS and multiplication is replaced by MULT. Unfortunately, however, no one has yet discovered an algebraically homomorphic encryption scheme.

Sander and Tschudin note, though, that if Bob provides the input x , then Bob does not need a MULT operation, but only a MIXEDMULT operation, which can compute $E(xy)$ using $E(y)$ (from Alice), and x (from Bob), without revealing y or xy to Bob. Since MIXEDMULT in turn can easily be implemented using $O(\log^2 N)$ PLUS operations (by a process similar to longhand binary multiplication), this means that all that is needed to do encrypted polynomial evaluation (on integers, modulo some integer N whose factors are small primes), is an *additively homomorphic* encryption scheme – a scheme which has a PLUS, but not necessarily a MULT operation. Such schemes, unlike algebraically homomorphic schemes, are known to exist [15, 92], and thus make encrypted computation of polynomials possible.

Composition-based approach

Sander and Tschudin also propose another way to perform encrypted computation using *function composition*: suppose f is a rational function (the quotient of two polynomials), and suppose that Alice can generate a random rational function s which is easily invertible. Then, Alice can give Bob the function $B = s \circ f$. When Bob returns $y = B(x)$, Alice can then retrieve $f(x)$ by computing $s^{-1}(y)$. The trick here, of course, is to be able to generate easily invertible rational functions s at random. Sander and Tschudin cite some known techniques but note that these are not completely secure, and more research needs to be done in this area.

One application of functional composition is in the implementation of *undetachable signatures*. Undetachable signatures address the following problem: suppose Alice has a function f , which she wants Bob to perform on some input x from Bob, and a signing function s , which she wants to run on Bob's machine to sign the output $f(x)$ for her. However, Alice wants to keep s secret so that Bob cannot use it to sign other values (not produced by f) – and thus impersonate her. Using function composition as described above, Alice can give Bob $s \circ f$ instead of s and ask Bob to run both f and $s \circ f$ on his input x to produce $f(x)$ and its signature. Since the decomposition of multivariate rational functions is a hard problem in general, this would prevent Bob from discovering s and generating valid signatures for arbitrary results.

Using Randomization and Redundancy

The encrypted computation techniques discussed so far are susceptible to some attacks. For example, when used in function hiding, the homomorphic encryption techniques are susceptible to *coefficient attacks* [129] that would allow Bob to decrypt the coefficients of f by simply sending Alice the encrypted forms of the coefficients, as if they were the result $f(x)$. The composition scheme for undetachable signatures is prone to many attacks such as *left decomposition* (Bob can determine s given both f and $s \circ f$), *interpolation*, and *inversion* [127]. Sander and Tschudin propose solutions to these problems where the basic idea is to use *tuples* in the functions, where some elements of the tuples are randomly generated by Alice and kept secret from Bob. This randomized information prevents Bob from being able to manipulate the computation to his advantage. It also prevents sabotage, since if Bob randomly changes part of the computation, Alice would be able to detect it, since the

“hidden checksum” or signature will not match the output. The cost, of course, is the redundancy and inefficiency introduced by using tuples of random data.

B.3.3 Critique

Sander and Tschudin’s work is very interesting because it gives us hope of achieving true blackbox security *with theoretically provable strength*. That is, unlike the obfuscation techniques to be discussed below, the difficulty of breaking the security of encrypted computation techniques can be predicted formally based on information-theoretic limits or the well-studied computational difficulty of certain mathematical problems. Unfortunately, however, their work in its current state does not yet seem to be useful in real applications for several reasons.

Limited Functionality. First of all, their approach is currently limited to the evaluation of polynomials and rational functions. In fact, it is not clear that they can really handle rational functions either since they note that they still need to come up with a good way to find easily invertible rational functions. They note that others [1] have developed some theoretical methods for the encrypted evaluation of boolean circuits in general, but that these are very inefficient and require a lot of rounds of communication.

Poor Efficiency. Secondly, their techniques still seem to be too inefficient for practical use. To be practical, the encryption and decryption overhead on Alice’s side should not be greater than the cost of computing the non-encrypted function completely on Alice’s side. Otherwise, it would be more efficient to just use the *perfectly secure* solution of simply having Bob send Alice his input x and let Alice do all the computation.

Unfortunately, Sander and Tschudin do not provide a formal complexity analysis of their techniques in order to quantify Alice’s overhead cost and prove that their techniques are practical. Even if we assume that their techniques can be shown to be practical at least asymptotically, however, there is a question of how big the constant factors are. Since simple integer addition and multiplication can be done very quickly in today’s processors, the cost of computing an integer polynomial is quite small, and only becomes significant when dealing with very high-degree polynomials. Thus, it would be interesting to see how large the polynomials have to be before their encrypted computation becomes practical.

The encryption overhead can be amortized away if the same encrypted polynomial is used by Bob on several inputs. This may make encrypted computation more practical. Care must be taken, however, that Bob does not learn much from the values that the encrypted function produces. Sander and Lipton propose an additively homomorphic encryption scheme that claims to be safe in this respect [92]. In the case of function hiding, though, where Alice decrypts $f(x)$ and returns the value to Bob, no matter what encryption scheme is used, Bob can always completely interpolate f by simply asking for $f(x)$ at $k + 1$ points, where k is the degree of the polynomial. Thus, in this case, encrypted computation is not practical if the polynomial has to be reused more than k times to make it efficient.

Another source of inefficiency are the techniques needed for hidden checksums, and undetachable signatures. First of all, the redundancy added by using randomized tuples

increases the overhead on both Alice’s and Bob’s sides, as well as the total time it takes to perform the computation. Secondly, the encryption schemes they currently use seem to be very inefficient. The scheme described in the function hiding paper, for example, performs the encryption one bit at a time, and thus seems very inefficient.

Impracticality for Function Hiding and Market-Based Computation. Although Sander and Tschudin’s general idea of using *function hiding* for protecting intellectual property and as a basis for “pay-per-use” computing is very interesting, their example using polynomial evaluation does not seem to be a good one. As noted above, the encryption and decryption overhead seems to be extremely high, such that the only way to make it practical would be to re-use the polynomial many times. However, this would allow Bob to interpolate f , since Bob receives the decrypted $f(x)$ from Alice. Because of these problems, it seems that it would be faster and safer for Alice to just perform the computation completely by herself.

For similar reasons, Sander’s and Tschudin’s current scheme for encrypted polynomial evaluation does not seem to be practical for market-based computation either. In addition to the overhead on Alice’s side which can prevent us from getting any speedup at all from parallelization, there is also the added cost of the PLUS and MIXEDMULT operations compared to simple addition and multiplication, which, while still making parallelization possible, would require a larger number of worker nodes to achieve the same performance. To be fair, however, we should note that Sander and Tschudin did not consider market-based computation as an application in their papers.

Summary of Critique. To summarize, in their current form, Sander’s and Tschudin’s proposed encrypted computation techniques seem usable, if at all, only for applications requiring the computation of polynomials on the host (Bob’s) side where Alice (the server) would like to know the value of some secret rational function $f(x)$ for some input x from Bob, but does not care how long Bob takes to compute it. Applications where Alice would like to achieve speedup, such as in market-based computation, seem unlikely to benefit from these techniques due to the high overheads involved. Applications such as function hiding, where Alice returns decrypted values to Bob need to be carefully analyzed to make sure that: 1) the overhead is less than the cost of having Alice compute everything herself, and 2) Bob cannot learn f or use Alice’s decryption “service” for his own purposes. Currently, polynomial evaluation does not seem to be a good example in this respect.

In the future, it would be interesting to see encrypted computation made more efficient and extended to more versatile functions than polynomials and rational functions. Meanwhile, it would also be helpful to do a more formal and comprehensive complexity analysis of Sander’s and Tschudin’s techniques, as well as come up with real practical applications of their techniques.

B.4 Obfuscation

Encrypted computation attempts to achieve blackbox security by using mathematically based cryptographic techniques to generate a new encrypted function $E(f)$ from a secret

function f , to make it difficult for a malicious user to understand, manipulate, and sabotage the computation of f . Another technique, called *obfuscation*, uses heuristic techniques to modify the *program* for computing f by transforming it into a “messy” form that is very hard to understand, but performs the same function as the original program. While the security properties of obfuscation are not as easy to analyze formally as those of encrypted computation, obfuscation has the advantage of being more generally applicable. That is, while encrypted computation is currently limited to specific problems such as polynomial evaluation, obfuscation automatically applies to any computation that can be expressed as a program.

Obfuscation is an old technique that has been around since programmers first started to worry about protecting their intellectual property from reverse engineering by competitors. In 1992, Cohen studied the application of obfuscation towards protecting operating systems from attacks by hackers or viruses [28]. (At around the same time, these same techniques were actually being used already by virus writers themselves in *polymorphic* or *evolutionary* computer viruses.) More recently, obfuscation has gained a lot of interest with the advent of Java, whose bytecode binary executable format is very easy to decompile back into its original source.

B.4.1 Obfuscation Transformations

In this light, Collberg, Thomborson, and Low have compiled a taxonomy of obfuscation techniques [30], classifying obfuscation transformations into four general types: *layout*, *control*, *data*, and *preventative* transformations. They further identify four main criteria that can be used to measure the quality of an obfuscation transformation: *potency*, *resiliency*, *stealth*, and *cost*. Potency refers to how much the transformation confuses a human reader, while resiliency refers to how much it confuses an automatic deobfuscator. Stealth [31] refers to how well the obfuscated code blends in with the rest of the program and thus avoids detection and focused attack. Finally, cost can be measured in terms of increased code size and execution time.

Layout Transformations

Layout transformations are transformations that change the way the code *looks* to a human reader. A simple example would be removing the source code formatting information sometimes available from Java byte code. Another example is that of scrambling identifier names to confuse a human reader. The latter is more potent than the former since identifier names carry a lot of pragmatic information that would help human programmers understand the code better. Many popular obfuscators today, such as Crema [152], depend on layout transformations. In general, layout transformations are simple to apply and cost very little in terms of added code size and execution time, but have relatively low potency, since the structure of the code and data are still mostly preserved and can be analyzed and eventually understood despite the change in layout.

Control Transformations

Control transformations aim to obscure the control-flow of the original program. Examples of control transformations include the following:

Computation Transformations. These transformations change the *apparent* computation being performed by the program. Examples include:

- inserting dead or irrelevant code,
- using more complicated but equivalent loop conditions,
- using language-breaking transformations that take advantage of constructs available in the binary code but not at the source level (e.g., `goto`'s in Java),
- removing library calls and programming idioms,
- implementing a virtual machine with its own instruction set and semantics,
- adding redundant operands, and
- converting a sequential program into a multithreaded one by using automatic parallelization techniques or by creating dummy processes that do not perform anything relevant.

To these we may add some techniques mentioned by Cohen, such as using different but equivalent machine instructions, and using equivalent instruction sequences [28].

Aggregation Transformations. These transformations make code harder to understand by breaking the abstraction barriers represented by procedures and other control structures. Examples include:

- *inlining* subroutine calls and *outlining* sequences of statements (turning them into a subroutine),
- interleaving methods,
- cloning methods, and
- loop transformations.

Ordering Transformations. These transformations change the order of statements, blocks, or methods, within the code, being careful to respect any data dependencies. These transformations may have low potency, but have high and *one-way* resilience, since in most cases, it is impossible for an automatic deobfuscator to restore the code ordering to its original form. Code ordering transformations have also been proposed as a method for *software watermarking* and *fingerprinting* software [35]. Specifically, the ordering of a program's code can be made into a unique identifier that can be used to track sources of pirated software.

Data Transformations

Data transformations obscure the data structures used by the application. Examples include the following:

Storage and Encoding Transformations. In many cases, there are “natural” ways to encode and store particular data items in a program. Storage and encoding transformations obscure code by deviating from these and choosing unnatural and encodings for the program’s data. Examples of these include:

- changing encoding of primitive data types,
- promoting variables from more specialized forms (e.g. a primitive type) to more generalized ones (e.g., a class),
- splitting variables, and
- converting static to procedural data.

Aggregation Transformations. These transformations change the way data items are grouped together. Examples include:

- merging scalar variables,
- restructuring arrays, and
- modifying inheritance relations.

Ordering Transformations. These transformations randomize the order in which data is stored in the program. These include

- randomizing the declarations of methods and instance variables within classes,
- randomizing the order of parameters within methods,
- using a mapping function to reorder data within arrays.

Preventative Transformations

Preventative transformations are different from control and data transformations in that they do not aim to confuse a human reader, but to make *automatic* deobfuscation difficult. Examples include the following:

```

{ int v, a=5, b=6;
  v = a + b; /* v is 11 here. */
  if (b > 5) ... /* True */
  if (random(1,5) < 0) ... /* False */
  if ( ... ) ...
    ... code keeps a and b unchanged ...
  if (b < 7) /* True */
    a++;
  v=(a > 5)?v=b*b:v=b /* v is 36 here */
}

```

Figure B-4: Examples of trivial opaque constructs.

Inherent Preventative Transformations. These transformations introduce problems that are known to be hard to solve with known automatic deobfuscation techniques. Some examples include:

- adding aliased formals to prevent *program slicing*,
- adding variable dependencies to prevent program slicing,
- adding bogus data dependencies,
- using opaque predicates with side-effects,
- making opaque predicates using difficult theorems (e.g., $(x*x==(7*y*y-1))$ is known to be false for any integers x and y).

Targeted Preventative Transformations. These transformations target known specific weaknesses in existing deobfuscation programs. For example the HoseMocha program [87] inserts extra instructions after `return` statements in the source program. This has no effect on the program execution, but causes the Mocha decompiler [153] to crash.

Opaque Variables and Predicates.

A key tool in implementing these transformations (and especially control transformations) is the idea of *opaque variables* and *opaque predicates*. An opaque variable is one which has some property that is known *a priori* to the obfuscator, but difficult for a deobfuscator to deduce. Similarly, an opaque predicate is one whose outcome is known ahead of time to the obfuscator, but difficult for the deobfuscator to deduce. Figure B-4 shows some examples of trivial opaque constructs (from [30]). (Here, we assume `random(a,b)` is a standard library function whose semantics is known to both the obfuscator and the deobfuscator.) These opaque constructs have low resilience since they can easily be broken by an automatic deobfuscator using simple global static analysis.

Since most deobfuscators are likely to employ static analysis techniques, Collberg et al. propose ways to generate opaque predicates based on problems that these techniques cannot handle well. In particular, they present ways of taking advantage of the intractability

of the static analysis of pointer-aliasing problems and multithreaded programs. They note that by manipulating pointers seemingly randomly at many points in the program, for example, we can generate opaque predicates of the form $(p==q)$ or $(p==null)$, where p and q are pointers (object references in Java), whose values would be very hard or impossible to determine statically. And, by employing concurrent threads with carefully controlled, but seemingly random race conditions, we can complicate the process even further.

Opaque variables and predicates can be used in many ways. Opaque predicates, which can be used to direct the control-flow in a program, can be used for many control transformations, such as inserting dead or irrelevant code. Opaque predicates and values can also be used in data transformations to obscure the encoding techniques and mapping functions used in the program.

Building an Obfuscator

Collberg et al. propose to build an obfuscator that will apply all these techniques repeatedly until the desired levels of potency, resilience, stealth, and cost, are achieved. Their proposed obfuscator will not just blindly apply these techniques to the code arbitrarily, but will attempt to analyze the code and determine appropriate transformations to apply to different parts of the code. For example, when obfuscating a part of the code with tight loops, care will be taken to use low-cost transformations, and when obfuscating a computational part without pointers, stealth would be maintained by employing opaque predicates using numbers instead of those based on pointer-aliasing.

B.4.2 Critique

Collberg et al.'s taxonomy of obfuscation techniques is very impressive. They provide a wide collection of techniques which, it seems, when used together and iteratively as they propose, should make code very, very difficult to understand even with the help of automatic deobfuscators. Furthermore, they identify some concrete formal metrics of program complexity from software engineering research, such as program length, cyclomatic complexity, nesting complexity, data flow complexity, fan-in/out complexity, etc. and propose to measure them and use them in their obfuscation program [30]. This may be a significant advantage in their approach over those of others such as Hohl, who so far only estimate the effectivity of obfuscation techniques using informal analysis.

Measuring Effectivity. The effectivity of their proposed techniques, however, remains to be seen. In general, the main weakness of obfuscation as an approach to blackbox security² is the difficulty of measuring how much these transformations really make it harder for a *human* user (fully armed with deobfuscation tools) to understand the obfuscated code. That is, even if their software is able to produce code that is highly obscured according to some formal metrics, it is still unclear whether or not these would really be effective in protecting the code against a focused attack from determined hackers.

²Although Collberg et al., like Sander and Tschudin, do not use the term *blackbox security*, which was coined by Hohl, their goals are very similar.

For example, although using pointer aliasing for generating opaque predicates seems like a very promising idea (and one of the significant and unique contributions of their work), the specific techniques (based on graph structures) that they propose for generating these opaque predicates, may not be as effective against human attackers as they are against automatic deobfuscators. A patient and determined human, for example, may be able to notice that some pointers seem to be used in predicates a lot, follow the assignments to these pointers, and learn their opaque values. Also, presumably, the statements in the program that will manipulate the values of these pointers will be generated automatically by an obfuscation program that will follow certain rules to guarantee the desired outcome. If a human can somehow discover these rules, then he could generate a program that uses pattern matching to break all opaque predicates using those rules.

Moreover, often, it is not necessary to discover all the rules and break all the opaque predicates to successfully attack a piece of code. For example, the mechanisms used for enabling or disabling features in trial versions of some software are sometimes based on a few simple `if` statements or on setting a few specific data locations to particular values. Thus, a determined hacker need not deobfuscate the whole program, but only needs to learn enough to determine these crucial statements in the code and focus his attack on them.³

Finally, even if the attacker cannot deobfuscate any part of the program at all, it is still possible for him to *sabotage* the program by making random changes in the code or data. Obfuscation alone would not protect against this. (Although it may be used to hide redundant and randomized checksum or signature computations similar in spirit to those used by Sander and Tschudin in encrypted computation).

In any case, it would be useful to look into these questions once Collberg et al. finish Kava, the obfuscation program they are building based on their proposed techniques. Perhaps the best way to test it would be to set up a challenge similar to the RSA decryption challenges [126], where hackers will be invited to crack programs obfuscated by Kava, and cause them to behave a certain way or produce a certain output. This would provide empirical evidence to support their theories, and may even be useful to other researchers in software engineering who are developing formal complexity metrics for software.

Parameterized Obfuscation. Another, somewhat orthogonal, direction for possible future research is that of developing more concrete schemes for *parameterized obfuscation*. Most current obfuscation programs have fixed strategies for obfuscating a particular piece of code. Such programs would generate the same obfuscated code given the same input program. What would be much more useful would be a program that is capable of generating many different versions of the obfuscated code depending on some parameter such as an integer. Such programs would have many applications. As will be described in the next section, they are necessary for time-limited blackbox security. They can also help track software piracy: a software vendor can sell each new customer a different version generated via parameterized obfuscation using a unique serial number. That way, if the vendor discovers a pirated version, she can trace it to the original purchaser.

Collberg et al. mention this potential application of obfuscation in [30], and note

³Thanks to Diego Sevilla for pointing this out.

that one can parameterize their obfuscator by choosing the random number seed in the `SelectTransform` algorithm, which selects obfuscation transforms from a pool of appropriate transforms. Different seeds would then lead to different sequences of obfuscation transforms being applied to the code. Since their obfuscator has a large number of transforms to choose from, this technique should be able to produce a large variety of different obfuscated versions of a program – much better than most current obfuscators which can only produce one.

In addition to randomizing or parameterizing the *selection* of transforms, however, it would also be useful to parameterize the workings of *individual* transformations as well. For data transformations, for example, it should be possible to use an integer parameter to generate different encoding maps or order-mapping functions. We can create similar variations within control transformations as well.

As an example, consider one type of control transformation described by Cohen [28], where parts of the original code are replaced by different, but equivalent instruction sequences – e.g., a sequence which performs the computation $x=x+17$ can be replaced by a sequence which first performs $x=x+20$, then $x=x-3$. It is easy to see how such a scheme can be parameterized using a single integer parameter p : if the original computation was $x=x+q$ for some constant q , then the transformation would generate $x=x+p$; $x=x-r$, where r is the (precomputed) constant $p - q$. We can parameterize this further by using other integers to determine the type or order of the replacement operations. For example, another parameter can be used to determine whether the subtraction of r is done before or after the addition of p .

Of course, this is just a small example, and it is easy to extend the idea and imagine countless ways to use parameters to generate many different obfuscated versions of a program. At the highest, most abstract level, such parameterization would be equivalent to Collberg et al.'s idea of simply setting the random number seed in the `SelectTransform` algorithm, since we can view each different version of the transformation (e.g., each p and r in the example above) to be entirely different transformations. However, the difference here is similar to the difference between implementing some function $f(x)$ as a *lookup table* indexed on x , or as a more compact algebraic *formula* based on x – given the same amount of space, the latter can handle a much wider range of inputs, or in the case of an obfuscator, it can produce a much wider variety of obfuscated programs.

It may be that Kava is already being implemented using such parameterized obfuscation. In a paper on software watermarking [32], for example, Collberg et al. present some techniques for encoding integers as graph structures. It is not hard to see that these and similar techniques can also be used as a basis for parameterizable data obfuscation transformations, so it seems safe to assume that they are already using them for this purpose. In any case, though even if they are already using parameterized obfuscation in Kava, a description of how different obfuscation transformations can be parameterized would be a useful supplement to their already-comprehensive taxonomy.

B.5 Time-limited Blackbox Security

Because they work on computer programs in general, obfuscation techniques are much more widely applicable than encrypted computation techniques, which currently work only for specific mathematical constructs such as polynomials. Thus, at present, obfuscation seems to be the more promising path to achieving blackbox security. Unfortunately, however, as we have seen, obfuscation is not as provably secure as encrypted computation.

Time-limited blackbox security, proposed by Hohl [71], recognizes this shortcoming of obfuscation by restricting the definition of blackbox security defined in Sect. B.2.1 as follows: programs must be protected in such a way that a host executing a program it did not create:

1. cannot read, or at least cannot understand, the program's code and data, and
2. cannot modify the program's code and data
3. – *for a certain known time interval.*
4. *After this time interval, attacks are possible, but these attacks would have no effects.*

Limiting the protection interval in this manner, allows us to use weaker security mechanisms such as obfuscation in implementing time-limited blackbox security.

B.5.1 Overview

Hohl presents time-limited blackbox security primarily as a way to protect mobile agents from malicious hosts. In this context, time-limited blackbox security can be achieved by first “messing-up” the agent code and data to make it hard to analyze, and then attaching an *expiration date* to indicate its protection interval.

Mess-up Algorithms. Hohl's idea of “mess-up algorithms” is essentially the same as obfuscation. In fact, most of the algorithms he proposes are already mentioned by Collberg et al. (of whom Hohl was not aware). The main idea here is to make the agent's code very hard to understand such that it will take some time before someone can understand it and be able to read it, manipulate it, or sabotage it. This minimum time required to successfully “break” the agent can then be used as the protection interval.

Of course, as in obfuscation, there is still the question of how long it would take a malicious host to break a messed-up agent. Hohl admits that this is an open question, although he points out that we only need to concern ourselves with estimating the time it would take for an *automatic* deobfuscator to break the code, arguing that a human alone cannot possibly do the analysis by himself in time.

Deposited Keys. One mess-up technique proposed by Hohl but not by Collberg et al. is that of *deposited keys*. Here, we encrypt parts of the agent, or remove some critical information needed by the agent and place it on a trusted server. As the agent runs, these parts remain encrypted or missing. Then, when the agent reaches a certain state, it asks the server for the decryption key or the missing parts. In this way, the host cannot analyze

the agent before running it and allowing it to come to the required state. (Note: there are some similarities between this technique and that described in [33], which also involves encrypting parts of the program, and decrypting it as the program runs.)

Expiration Dates. After creating an obfuscated version of the agent with the desired protection interval, the agent's owner adds the protection interval to the current time to get an *expiration date*. This expiration date is then included as part of a digitally signed certificate that the agent carries and must present to any other party that it wants to communicate with. Parties who receive such a communication request would first check the certificate to determine if the agent is still valid. If the agent has already expired, communication is refused it, on the assumption that the agent could already have been broken by the malicious host and is thus untrustable. (Note that even if the agent is broken, the certificate itself containing the expiration date still remains valid since it is *cryptographically* signed by the agent's creator and is thus presumably secure for a very long time.) In this way, even if the agent is successfully broken after the expiration date, it cannot cause harm since it cannot communicate with anyone else. (A presumption here is that all receiving parties need to be able to get the correct global time in order to check the expiration date.)

If the agent to be sent out contains some *token* data – i.e., self-contained documents that depend on the identity of the owner and can be traded independently of the issuer, such as digital cash, secret keys, etc. – then the owner must also attach expiration dates on these tokens and digitally sign them. He must do this *before* the agent is obfuscated so that the obfuscation algorithms will also scramble the way the tokens are stored in the agent and ensure that a malicious host cannot get the tokens before the protection interval. Then, even if a malicious host successfully breaks an agent *after* the protection interval and manages to get its tokens, no harm is done, since the tokens would already be expired and unusable.

Finally, note that the expiration date protects the agent even if it migrates to another host. And, if the agent needs to extend its life for some reason, then it can get “recharged” by returning to its owner (or some other trusted host), which can then *reobfuscate* it using different parameters, and then give it a new expiry date.

Limitations and Open Problems. At present, the main problem in implementing time-limited blackbox security is the difficulty of estimating and guaranteeing the protection interval of an obfuscated agent. However, Hohl also recognizes some other limitations and open problems, including the following:

- There is a need for a standard *global* time that can be used to check expiration dates. This requires either some mechanism for distributed clock synchronization, or communication with a trusted timekeeper, which may increase communication overhead and limit scalability.
- The types of tokens that can be protected using expiration dates are limited. For example, tokens which need to be protected past the expiration date (e.g., the owner's long-term private keys) should not be carried by time-limited agents.

- Obfuscation leads to longer transmission and execution time for agents, which may limit the applicability of this technique to applications where speed is not a critical factor.
- Time-limited blackbox security (and blackbox security in general) may still be susceptible to *sabotage*, where the host destroys parts of the agent, and *blackbox test* attacks, where the host tries to deduce the internal workings of the agent by running it many times on many different combinations of inputs. Hohl notes that we may be able to protect against the former by embedding checksum code within the algorithm. If the agent is truly a blackbox, then sabotage is likely to lead to random damage, and can thus be detected by checking the checksums.

B.5.2 Critique

The idea of time-limited blackbox security is a very promising one. It aims to achieve the same goals sought by Sander and Tschudin in mobile cryptography, but seems to offer a much better chance at actually becoming implementable and usable in the near future.

Better Obfuscation, Deposited Keys, and Interactive Techniques. Hohl's list of mess-up algorithms is somewhat limited, and can benefit a lot from incorporating Collberg et al.'s obfuscation transformations, especially if the latter can be parameterized as suggested earlier.

At the same time, it would also be useful to further study and develop the idea of deposited keys and similar techniques. This idea is different from the other techniques discussed by Sander and Collberg in that it is *interactive*. This is both an advantage and a disadvantage. The disadvantage is that it requires communication with a trusted server, and can thus increase execution time and limit scalability. The advantage, however, is that one can rely on more *information-theoretic*, and not just complexity-based security against attacks. That is, if parts of the code are actually missing and cannot be predicted, then it is not possible to analyze the code *regardless of how much time an attacker may have*.

Sander and Tschudin seem to think that the cost of extra communication outweighs the advantages of interactive techniques, and thus consider only non-interactive techniques for their mobile cryptography. There may be applications, however, where the extra communication may be acceptable. For example, if an agent needs to communicate with the server at certain points anyway, then we can piggyback the deposited keys with the normal communication packets at these points. In this way, we can use interactive techniques to provide significantly more security at minimal additional cost.

Applications. In his paper, Hohl focuses exclusively on using time-limited blackbox security for protecting mobile agents against malicious hosts. It would be interesting to see if his ideas can also be applied to other applications as well, such as the protection of programs and intellectual property from reverse engineering and piracy, and market-based parallel computation.

As is, time-limited blackbox security does not seem directly applicable to the problem of protecting software from piracy and intellectual property theft. Unlike mobile agents,

which are typically sent out to other hosts for relatively specific and short tasks (e.g., to find the best deal on an item), software has a much longer intended life and thus needs a longer protection interval. Unfortunately, the task of guaranteeing a protection interval by using obfuscation becomes much harder for longer intervals. This is because as the intervals get longer, the human advantage over automatic deobfuscators becomes a more significant factor. If a program's time limit is in the order of seconds or minutes, for example, (a reasonable time for a mobile agent to achieve its task), then it is safe to assume, as Hohl does, that we only need to worry about automatic deobfuscators since it is simply impossible for a human to process the information in such a short time. It takes time, for example, for a human just to *read* the text of the program, let alone understand it and modify it. If the program's time limit needs to be extended to the order of months, or even just days, however, then considering the human ability to deobfuscate code becomes necessary. This is a much harder problem. While complexity analysis may be used to predict the time it would take for certain deobfuscation algorithms to unravel a piece of obfuscated code, it is much harder to analyze the human thinking process in a similar way.

It is also important to note that time-limited blackbox security applied to a program does not mean that the program becomes unusable or self-destructs after the protection interval. On the contrary, it means that we assume the user can do *anything* it wants with the program after the protection interval. This presents another problem if the program we are trying to protect is a commercial software package and not just an agent. With an agent, we are able to guarantee that no harm is done after the protection interval by simply preventing other parties from communicating with the agent. Thus, even if the malicious host is able to understand all the algorithms contained in the agent, it cannot use them. In contrast, with commercial software, if a malicious user is able to reverse engineer a program after the protection interval, then harm has already been done. The user can then, for example, rewrite the program, remove its security mechanisms, and then sell copies of the modified program to others. Since many commercial programs do not depend on communication with other parties to operate (e.g., word processors), there is no way to prevent the broken programs from being used in the same way that we prevented expired agents from being used.⁴

Fortunately, while time-limited blackbox security cannot be used for protecting intellectual property, it can be useful for market-based parallel computing. The idea here is to give the worker hosts obfuscated and time-limited code for performing part of the parallel computation. The worker runs the code until the time limit, at which point, the server gives it a new reobfuscated version of the same code (or the code for a different part of the parallel computation). Here, as with mobile agents, the code given to the hosts is specific to the current problem being solved, and has a relatively short lifetime. Thus, it should be easier to predict and guarantee protection intervals for it. Furthermore, as with agents, we can prevent malicious hosts from doing harm by simply ignoring any communication requests or any results produced by expired code. Of course, the server should also take care to change the data encoding, or to change the computation itself, when giving new code to the worker after the previous time limit expires. In this way, even if the malicious

⁴One way to make time-limited blackbox security work here may be to *require* even ordinarily non-communicative programs to communicate with a server to operate. This leads to an idea similar to Sander and Tschudin's function hiding.

worker understands the previous version of the code, he cannot use that knowledge to affect the results of the new version.

It is important to note, however, that the cost of obfuscation is a more significant factor when applying time-limited blackbox security to market-based parallel computation. The need to periodically give new code to the workers incurs a certain amount of communication overhead. In the case of Java code, it can also mean a reduction in performance since frequently changing the code prevents us from taking full advantage of just-in-time (JIT) compilation techniques, which take a certain warm-up time, but speed up the code execution considerably. In applications where the correctness and security of the computation is a major concern, however, the cost may be worth it. It may be better, for example, to have the computation take 10 times longer but be guaranteed correct, for example, than to risk having a small error due to one sabotaged worker invalidate and waste the work of 1000 other workers.

B.6 Suggestions for Future Research

The problem of protecting programs from attacks by malicious users is a relatively new research area, and there is much space left for further exploration. Some ideas for further research have already been mentioned, including the following:

- **Parameterized Obfuscation.** It would be useful to describe concrete ways not only to *select* between different obfuscation transformations as Collberg et al. suggest, but also to parameterize the *individual* transformations themselves. This (in addition to Collberg et al.'s parameterized selection) would lead to being able to generate more variations on a piece of code, and would be useful for implementing time-limited obfuscation.
- **Holding a cracking contest to measure effectivity.** The main disadvantage of obfuscation is that unlike mathematically analyzable techniques such as encrypted computation, it is hard to predict the effectivity of obfuscation transformations. One way to *empirically* measure such effectivity may be to hold a publicized contest inviting hackers to crack a certain obfuscated program and make it behave in a certain way. This would be similar to the successful decryption contests held by RSA [126].
- **Interactive techniques.** Sander and Collberg consider mostly non-interactive techniques, in order to avoid the extra cost incurred by requiring the program to communicate with the server. In cases where such communication is already required anyway, then it might be profitable to apply interactive techniques, such as Hohl's deposited keys idea, to achieve much better security.

In addition to these, there are at least two other general ideas which deserve further study: using redundancy and randomization, and combining encrypted computation and obfuscation.

B.6.1 Redundancy and Randomization

A potentially very useful technique is that of using redundancy and randomization as a way of further enhancing security. For example, Sander and Tschudin use such techniques when addressing the problems with undetachable signatures and function hiding [127, 129]. Their solutions work by giving the host a new function where each coefficient is replaced by a *tuple* containing elements not only from the original function, but also from other randomly chosen functions whose answers are either known, or can easily be evaluated. This makes it much harder for an attacker to understand the structure of the original function or to generate a manipulated answer. Unless the attacker can deduce *all* the other functions, *and* identify which one is the original one, any changes that the attacker makes to the function is likely to cause an inconsistency in the answer that the owner can detect. Similar techniques may be used in time-limited blackbox agents to implement the embedded checksums suggested by Hohl as a protection against sabotage attacks.

Further research in this area may also benefit from the results of research in *digital watermarking*, and specifically, of the watermarking of *active intellectual property* such as software and hardware designs.⁵ The idea behind such watermarking is to embed some property in the software or hardware design that is checkable by the owner using certain key information, but unnoticeable to people without the required key information. The watermark should also be resilient to attempts to remove it from the watermarked object. Collberg et al. classify some common watermarking techniques for software and suggest some of their own in [32]. A group at UCLA is also currently studying the watermarking of hardware designs and solutions to graph theory problems [78, 84, 147].

Most watermarking techniques are designed primarily as protection against intellectual property theft, and thus probably cannot be applied directly in their present form to the problem of hiding checksum computations in blackbox code. However, some ideas from these techniques may be helpful. One technique presented in [84], for example, protects an FPGA design by randomly selecting a number of normally invisible (internal) nodes to be made visible in addition to many more normally visible nodes. This allows the owner to check for design theft, since the chance that another honest designer would come up with a design that includes not only the normally visible nodes, but also the same set of normally invisible nodes is very small. While this exact implementation may not be directly applicable to blackbox security, similar techniques involving the addition of a randomized set of unnecessary information may be useful.

Another example of a watermarking technique that may be useful is that of *constraint-based watermarking* [78]. Here, additional constraints are added to the original problem based on a certain private key. A solution is then generated to fulfill these and the original constraints. Because of the additional constraints, the resulting solution space would be restricted. Checking for the watermark can then be done by checking if a certain solution satisfies the additional constraints. If the original solution space is large, and the constraints are chosen randomly,⁶ then the chances of an honest designer coming up with his own solution to the original problem which also happens to satisfy the extra constraints will be small.

⁵as opposed to the watermarking of *artifacts* such as digital images.

⁶but carefully, to avoid constraining the problem too much that it becomes unsolvable.

A similar technique may be used for protecting mobile agents and market-based parallel computing systems against sabotage by malicious hosts. The idea is to ask the hosts to solve a *harder* (more constrained) problem whose answers will have properties that the server can easily check. Thus, if a host's answer does not contain the desired property, we can assume that it did not run the code it was given correctly. Sander and Tschudin's idea of using tuples may be considered one example of this idea.

B.6.2 Combining Encrypted Computation and Obfuscation

Finally, it would also be interesting to find ways to combine the ideas of encrypted computation and obfuscation to get better security.

In looking for such solutions, it would probably be helpful to assume that the server expects to receive *encrypted* data from the host, and the host does not need to understand the results it produces. This is because allowing the host to understand the meaning of even just the results makes the scheme susceptible to many different learning attacks such as interpolation and the coefficient attack, noted earlier in Sect. B.3.2. Fortunately, this assumption is not too restricting. Although it does not allow us to protect commercial software from piracy (since commercial software in general are designed to produce unencrypted data for the host's use), it can still be used in mobile agent applications, and in market-based parallel computation, where the host does not need to understand the results generated by the code.

Given this assumption, some possible approaches include: obfuscating encrypted functions, using encryption as a part of the obfuscation process, and using obfuscation transforms as encryption schemes.

Obfuscating Encrypted Functions

One straightforward way to combine encrypted computation and obfuscation is simply to first use encrypted computation to encrypt the function to be computed (assuming it is a polynomial or rational function), and then obfuscate the program afterwards. By obscuring the workings of the PLUS operator, and hiding the coefficients, this would add an extra layer of security and make it harder (though not impossible) for a malicious host to take advantage of attacks such as coefficient attacks, which are based on knowledge of the encrypted coefficient values and the structure of the function to be evaluated.

Encryption as Part of the Obfuscation Process.

It may also be possible somehow to use encryption techniques to determine the reordering and regrouping of the program's code and data. In [32], for example, Collberg et al. talk about watermarking software by taking two secret primes, encoding their product as graph structure, and hiding the structure in the program heap among other structures. It is not yet clear, however, if and how this idea can be applied to achieving blackbox security.

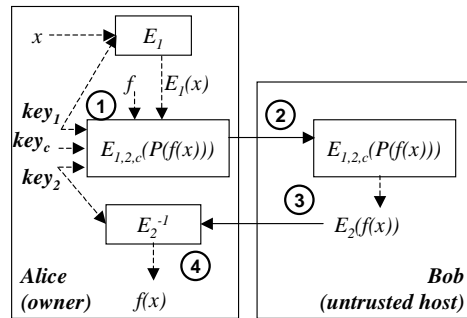


Figure B-5: CED using obfuscation as an encryption scheme.

Obfuscation as Encryption

Finally, a most intriguing and promising possibility is that of using obfuscation itself as a form of encryption.

Implementation. Suppose, for example, that we consider data obfuscation transformations, such as changing data encoding, splitting and merging variables, and reordering and restructuring arrays and other data structures, as encryption schemes. Then, we can do computing with encrypted data (CED) as shown in Fig. B-5. Here, Alice chooses two secret “keys” or parameters and uses them to select (or generate) two data obfuscation schemes, E_1 and E_2 . E_1 is used for encoding the input data, while E_2 is used for encoding the output. Using these keys, together with a code obfuscation parameter, key_c (for added variety), Alice then generates a program (step 1), $E_{1,2,c}(P(f(x)))$, which encodes and stores x according to E_1 and works on it to produce a result encoded according to E_2 without revealing the original value of x . Alice gives this program to Bob (step 2), who runs it on his own machine to produce the encoded answer $E_2(f(x))$ (step 3). Bob then returns the answer to Alice, who runs the decoding program $P(E_2^{-1})$ on Bob’s result to get the original value of $f(x)$ (step 4). As in the CED example described in Sect. B.3.2, Bob only sees encrypted forms of the data and the code, and thus never learns x or $f(x)$.

While this scheme is very similar to those described in Sect. B.3.2, there are a few important differences.

First, unlike the techniques in Sect. B.3.2, this scheme is not limited to using algebraic functions and algorithms to perform the encrypted (homomorphic) operations. Thus, while no one has yet found a traditional encryption scheme where one can perform *both* the PLUS and MULT operations homomorphically – let alone perform generalized computation – doing so is relatively straightforward with obfuscation. Since the program $P(E_{1,2,c}(f(x)))$ can be *any* program that can be expressed as a sequence of machine instructions, we know that we can implement any operator that we need as long as it is computable.

Second, note that in Sander’s and Tschudin’s schemes, the homomorphic operators PLUS and MIXEDMULT were *fixed* and *well-known*, even to Bob. It is only the data’s encrypted form that changes, based on the private key. In contrast, here, not only the data, but also the code given to Bob, change depending on the private keys.

Finally, since $P(E_{1,2,c}(f(x)))$ is not limited to being mathematically describable and

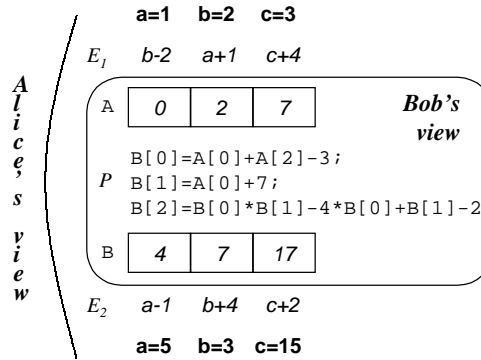


Figure B-6: Computing $a=b+c$; $b=b+1$; $c=a*b$; using obfuscated data.

fixed, its operations do not have to be homomorphic. That is, as shown in Fig. B-5, the encoding scheme for the result, E_2 , does not have to be the same as the encoding scheme for the input E_1 . In Sander's and Tschudin's scheme, the PLUS operator had to be homomorphic because it was used many times on Bob's side in the process of calculating the polynomial. In fact, finding a way to do encrypted polynomial evaluation using just one homomorphic operator (PLUS) was one of Sander and Tschudin's major achievements, since homomorphic operators, and schemes which support several of them, are generally hard to find. If, as described above, however, we have complete freedom in writing the encrypted program $P(E_{1,2,c}(f(x)))$, then limiting ourselves to homomorphic operators is not necessary, and we may be able to get more flexibility and more security by allowing the output to be encoded differently from the input.

Example. Figure B-6 shows an example. Here, Alice wants to hire Bob to compute the following computation for her without allowing Bob to understand the computation,

$$a = b + c; \quad b = b + 1; \quad c = a * b;$$

As shown, she first chooses two different encoding schemes – E_1 for the input, and E_2 for the output – which scramble the data by adding random numbers to them and placing them in an array in random order. Given these two schemes, she then generates a program, P , to perform the desired computation using the new encodings, and gives this program to Bob together with the encoded input array A . (Note that Alice can make it even harder for Bob to understand the computation by employing more complex data encoding schemes as well as applying code obfuscation transformations on P .) Bob then runs the program to produce the output array B , which he returns to Alice. Finally, Alice computes the inverse of E_2 on B to get the desired values of a , b , and c .

By thus allowing Bob to see only encoded versions of the data and the code, Alice is able to make it hard, or even impossible, for Bob to understand the computation. Moreover, unlike in traditional encrypted computation, Alice can have Bob perform *arbitrary* secret computations for her, since all she needs to do is to rewrite the original computation to work with the scrambled data. This example, for instance, shows Alice having Bob perform a secret computation using addition and multiplication operations without the need for an algebraically homomorphic encryption scheme.

Of course, since this scheme is based on obfuscation techniques, the problem of predicting its security remains a problem. There is one significant advantage here, however. In most traditional applications of obfuscation, such as in protecting commercial software programs, the obfuscated program has to receive *unencoded* data from the user, process it, and then produce *unencoded* output. This means that even if most of the internal code and data are obfuscated, there is still at least some code needed for converting between the unencoded and encoded forms of the data. If Bob can identify and understand this part of the code, he can understand the data encoding, and gain a much better chance of understanding the code itself. In contrast, the scheme in Fig. B-6 never allows Bob to see any data in unencoded form. Thus, even if he can understand how P works, he can never be totally sure that his understanding is correct.

For example, suppose, using a process like the one to be described below, he is able to analyze P and come to the conclusion that the code is performing a computation which looks like:

```
a = b + c; b = b + 1; c = a * b;
```

He still cannot be sure of this analysis, since if the array orderings were slightly different, the computation can also mean:

```
c = b + a; b = b + 1; a = c * b;
```

(with a and c swapped), which would have a different meaning if a and c are used in other parts of the program. Of course, in reality, Bob's job is even harder, since the constant numbers added to the variables can be random, so he can never be sure whether each statement involves constants, and if so, what their values are.

Caveats. It is important to note that although hiding the data encoding schemes from Bob makes it much harder for Bob to understand the obfuscated program P , Bob may still be able to break the code given some information about the code. For example, if he knows that the pattern $b = b + 1$ is common in the program, and that other similar patterns such as $b = a + 1$ or $b = a + 3$ are very rare, then he may be able to deduce that $A[1]$ contains an encoded version of the original b and $B[1]$ will hold an encoded version of $b + 1$. If he further knows that Alice has the habit of writing code such that statements with additions and multiplications involving two variables never involve constants, then he may be able to deduce the structure of the first and third statements, and then deduce the constants in the encoding. Finally, by looking at other parts of the code, he may be able to deduce that Alice is more likely to compute $c = a * b$ than $a = c * b$ and thus identify the proper array ordering.

This problem is similar to that of old cryptographic techniques such as the substitution cipher, which can be broken by noticing the frequencies of different encoded characters and matching them with known frequencies of letters in the original language. Modern-day cryptographers have gotten around this problem by developing techniques designed to produce random-looking ciphertexts without the statistical patterns of the original language. Applying the same idea to obfuscation, this means that we should choose data and code obfuscation transformation schemes that would hide statistical patterns in the original code. Ideally, an obfuscated program P , should be the *possible* result of *many* different

original code sequences, each *with equal probability*. In this way, even if Bob is able to generate all possible deobfuscations of P , he still will not know which one is the correct one, since they are equally likely.

A further way to mask statistical patterns is to interleave different and independent computations. For example, to prevent Bob from deducing the structure of the code above, Alice may choose to interleave it with the code for solving another problem where b is not always incremented by 1, multiplications and additions can involve constants as well as variables, and a is more likely than c to contain a product.

In general, suppose Alice has two different and independent computational problems C_1 and C_2 , which taken several steps to compute. She can interleave the computations either:

- in space – at each step, give Bob a program with the code for a step in C_1 and a step in C_2 .
- in time – at each step, give Bob a program with the code for a step in either C_1 or C_2 , but without letting Bob know which one.
- or both – at each step, give Bob a program which may perform zero or more steps from either C_1 or C_2 .

Interleaving code in either space or time helps hide statistical patterns in C_1 and C_2 if these have different patterns. Interleaving computations in time makes analysis even harder by obscuring the data flow in the code. Suppose for example that a program Bob receives for a step writes to a variable a , and the program for the next step reads and performs computation on a . Bob cannot conclude anything from this since the second program may actually belong to a totally independent computation where a has a different meaning and value.

Summary. Whether these and other techniques would be enough to make obfuscation feasible as a way of doing encrypted computation is still an open question, but it looks very promising at the moment and thus deserves further investigation. At the very least, hiding both the input and output data encoding schemes from the host (Bob), will almost certainly make deobfuscation significantly more difficult, and can thus be used as a way of increasing the protection intervals needed in time-limited blackbox security.

B.7 Conclusions

In this paper, we discussed different ways of protecting programs such as commercial software and mobile agents from attempts by malicious users to understand, modify, and manipulate their behavior. Using mathematical principles, encrypted computation promises the potential of finding provably secure ways to hide the details of a function from the host executing it, but currently works only for a few specialized functions such as polynomial evaluation, and seems to be too inefficient to be practical in real applications. Obfuscation offers more versatility and efficiency, but its security properties are harder to analyze due to the heuristic nature of both the obfuscation and deobfuscation processes. Time-limited

blackbox security recognizes this limitation in obfuscation by preventing mobile code from communicating with other parties after a certain protection interval based on an estimate of the time it would take to deobfuscate the code.

All these techniques are relatively new, and provide a lot of opportunity for further research. Some ideas worth looking into include parameterized obfuscation, interactive techniques, using redundancy and randomization techniques such as checksums and watermarks, and combining encrypted computation and obfuscation. In the end, a successful solution to this problem will probably be achieved by combining all these techniques – that is, by using parameterized obfuscation as an encryption scheme (with randomized parameters as the private keys) to prevent the malicious host from understanding and manipulating the code, and by using checksums and watermarks to detect attempts to change the code.

Bibliography

Bibliography

- [1] M. Abadi, and J. Feigenbaum. Secure circuit evaluation, in *Journal of Cryptology*, 2(1), 1990.
- [2] H. Abelson, et al. Amorphous Computing, in *Communications of the ACM*, May 2000. URL: <http://www.swiss.ai.mit.edu/projects/amorphous/>
- [3] *Proc. of the ACM 1997 Workshop on Java for Science and Engineering Computation*, June 1997. Also published as *Concurrency: Practice and Experience*, 9(11), Nov. 1997. URL: <http://www.cs.rochester.edu/u/wei/prog.html>
- [4] *Proc. of ACM 1998 Workshop on Java for High-Performance Network Computing*, Feb./Mar. 1998. Also published as *Concurrency: Practice and Experience*, 10(11-13), 1998. URL: <http://www.cs.ucsb.edu/conferences/java98/program.html>
- [5] *Proc. of ACM 1999 Java Grande Conference*, June 1999. Also published as *Concurrency: Practice and Experience*, 12(6-8), 2000. URL: <http://www.cs.ucsb.edu/conferences/java99/program.html>
- [6] A.D. Alexandrov, M. Ibel, K.E. Schauser, C.J. Scheiman. SuperWeb: Research Issues in Java-Based Global Computing, in *Concurrency: Practice and Experience*, June 1997. URL: <http://www.cs.ucsb.edu/~schauser/papers/96-superweb.ps>
- [7] T.E. Anderson, D.E. Culler, D. A. Patterson, and the NOW Team. A case for networks of workstations: NOW. *IEEE Micro*, Feb. 1995. URL: <http://now.cs.berkeley.edu/Case/case.html>
- [8] D. Arnow, G. Weiss, K. Ying, D. Clark. SWC: A Small Framework for WebComputing, in *Proc. of the International Conference on Parallel Computing (ParCo99)*, August 1999. URL: <http://www.sci.brooklyn.cuny.edu/~kevin/>
- [9] M. Baker, B. Carpenter, S. Ko, and X. Li. mpiJava: A Java interface to MPI, in *Prof. of 1st UK Workshop on Java for High Performance Network Computing, Euro-Par '98*, Sep. 1998.
- [10] M. Baker, ed. Special Issue: Message passing interface-based parallel programming with Java. *Concurrency: Practice and Experience*, 12(11), 2000.

- [11] J.E. Baldeschwieler, R.D. Blumofe, and E.A. Brewer. ATLAS: An Infrastructure for Global Computing, in *Proc. of the Seventh ACM SIGOPS European Workshop: Systems Support for Worldwide Applications*, Sep. 1996.
- [12] A. Baratloo, M. Karaul, Z. Kedem, P. Wyckoff. Charlotte: Metacomputing on the Web, in *Proc. of the 9th International Conference on Parallel and Distributed Computing Systems*, Sep. 1996. URL: <http://cs.nyu.edu/milan/charlotte/>
- [13] A. Baratloo, M. Karaul, H. Karl, and Z.M. Kedem. An Infrastructure for Network Computing with Java Applets, in *Proc. of ACM 1998 Workshop on Java for High-Performance Network Computing*, Feb./Mar. 1998. Also published in *Concurrency: Practice and Experience*, 10(11-13), 1998.
- [14] A. Baratloo. *Metacomputing on Commodity Computers*. PhD Thesis. New York University, May 1999. URL: http://www.cs.nyu.edu/csweb/Research/Theses/baratloo_arash.pdf
- [15] J. Benaloh. Dense probabilistic encryption, in *Proc. of the Workshop on Selected Areas of Cryptography*, 1994.
- [16] A.F. Bielajew, H. Hirayama, W.R. Nelson, D.W.O Rogers. History, overview and recent improvements of EGS4. Technical Report SLAC-PUB-6499. Stanford Linear Accelerator Center, June 1994. URL: <http://ehssun.lbl.gov/egs/egs.html>
- [17] R.D. Blumofe, C.F. Joerg, B.C. Kuszmaul, C.E. Leiserson, K.H. Randall, Y. Zhou. Cilk: An Efficient Multithreaded Runtime System, in *Proc. of the 5th ACM SIGPLAN Symposium on Principles of Parallel Programming (PPOPP '95)*, July 1995. URL: <http://supertech.lcs.mit.edu/cilk/>
- [18] R.D. Blumofe and P.A. Lisiecki. Adaptive and Reliable Parallel Computing on Networks of Workstations, in *Proc. of the USENIX 1997 Annual Technical Symposium*, Jan. 1997.
- [19] T. Brecht, H. Sandhu, M. Shan, J. Talbot. ParaWeb: Towards World-Wide Supercomputing, in *Proc. of the Seventh ACM SIGOPS European Workshop: Systems Support for Worldwide Applications*, Sep. 1996.
- [20] BSP Worldwide Home Page. URL: <http://www.bsp-worldwide.org/>
- [21] M. Bucayan, J. Punzalan, E. Vidal. Projects for a course on Internet-based Parallel Computing. Ateneo de Manila University, Oct. 1999.
- [22] R. Buyya, D. Abramson, and J. Giddy. An Economy Driven Resource Management Architecture for Global Computational Power Grids, in *Proc. the 2000 International Conference on Parallel and Distributed Processing Techniques and Applications (PDPTA 2000)*, June, 2000.
- [23] 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. of ACM Workshop on*

- Java for Science and Engineering Computation*, June 1997. Also published in *Concurrency: Practice and Experience*, 9(11), Nov. 1997.
- [24] P. Cappello, B. Christiansen, M.O. Neary, and K.E. Schauer. Market-Based Massively Parallel Internet Computing, in *Proc. Third Working Conf. on Massively Parallel Programming Models*, pages 118–129, Nov. 1997.
- [25] D. Caromel, J. Vayssière. A Java Framework for Seamless Sequential, Multi-threaded, and Distributed Programming, in *Proc. of ACM 1998 Workshop on Java for High-Performance Network Computing*, Feb./Mar. 1998. Also published in *Concurrency: Practice and Experience*, 10(11-13), 1998.
- [26] N. Carriero, D. Gelernter. Linda in Context. *Comm. of the ACM*, Apr. 1989.
- [27] H. Casanova and J. Dongarra. NetSolve: A Network Server for Solving Computational Science Problems. Technical Report CS-95-313. University of Tennessee, Nov. 1995.
- [28] F.B. Cohen. Operating System Protection Through Program Evolution. 1992. URL: <http://all.net/books/IP/evolve.html>
- [29] C. Collberg. Christian Collberg's Software Watermarking and Obfuscation Page. URL: <http://www.cs.arizona.edu/~collberg/Research/Obfuscation/>
- [30] C. Collberg, C. Thomborson, and D. Low. A Taxonomy of Obfuscating Transformations. Technical Report 161. Department of Computer Science, The University of Auckland, New Zealand, July 1997.
- [31] C. Collberg, C. Thomborson, and D. Low. Manufacturing Cheap, Resilient, and Stealthy Opaque Constructs, in *Proc. of ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL'98)*, Jan. 1998.
- [32] C. Collberg and C. Thomborson. Software Watermarking: Models and Dynamic Embeddings, in *Proc. of ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL'99)*. San Antonio, Texas, Jan. 1999.
- [33] T.E. Cooper, H.W. Philips, R.F. Pryor. Method and apparatus for enabling trial period use of software products: Method and apparatus for utilizing a decryption block. US Patent 5,598,470. Assignee: IBM Corporation, Jan. 1997.
- [34] P. Dasgupta, Z. Kedem, and M. Rabin. Parallel processing on networks of workstations: A fault-tolerant high performance approach, in *Proc. 15th IEEE International Conference on Distributed Computing Systems*, 1995. URL: <http://cs.nyu.edu/milan/milan/>
- [35] R.L. Davidson and N. Myhrvold. Method and system for generating and auditing a signature for a computer program. US Patent 5,559,884. Assignee: Microsoft Corporation, Sep. 1996.

- [36] D. Dean, E.W. Felten, and D.S. Wallach. Java Security: From HotJava to Netscape and Beyond, in *Proceedings of the IEEE Symposium on Security and Privacy*, May 1996. URL: <http://www.cs.princeton.edu/sip/Publications.html>
- [37] D. Delorie. djgpp home page. URL: <http://www.delorie.com/djgpp>
- [38] A.K. Dewdney. Computer Recreations, in *Scientific American*, 253(2), pages 16–24, Aug. 1985.
- [39] K. Dincer. Ubiquitous Message Passing Interface Implementation in Java: jmp_i, in *Proc. IPPS'99*. URL: <http://ipdps.eece.unm.edu/1999/papers/039.pdf>
- [40] distributed.net home page. URL: <http://www.distributed.net>
- [41] distributed.net. Press release, Oct. 1997. URL: <http://www.distributed.net/pressroom/56-announce.html>
- [42] distributed.net. Personal Proxies. URL: <http://www.distributed.net/download/proxies.html>
- [43] distributed.net. Project OGR. URL: <http://www.distributed.net/ogr/>
- [44] distributed.net. Project RC5. URL: <http://www.distributed.net/rc5/>
- [45] Entropia, Inc. Entropia home page. URL: <http://www.entropia.com/>
- [46] J. Farrington. A Web Computer: a Super Computer for the Masses. UCL research note RN-1996-58, 1996. URL: <http://www.wizzo.demon.co.uk/webcom/wc02.html>
- [47] A. Ferrari. JPVM: Network Parallel Computing in Java, in *Proc. of ACM 1998 Workshop on Java for High-Performance Network Computing*, Feb./Mar. 1998. Also published in *Concurrency: Practice and Experience*, 10(11-13), 1998. URL: <http://www.cs.virginia.edu/~ajf2j/jpvm.html>
- [48] S. Fields. Hunting for Wasted Computing Power: New Software for Computing Networks Puts Idle PC's to Work. Research Sampler. University of Wisconsin-Madison. 1993. URL: <http://www.cs.wisc.edu/condor/doc/WiscIdea.html>
- [49] FightAIDS@home home page. (Also featured in Entropia home page.) URL: <http://www.fightaidsathome.com/>
- [50] I. Foster, and C. Kesselman. Globus: A Metacomputing Infrastructure Toolkit. *Intl J. Supercomputer Applications*, 11(2), 1997, pages 115–128. URL: <http://www.globus.org/>
- [51] I. Foster, and C. Kesselman, eds. *The Grid: Blueprint for a New Computing Infrastructure*. Morgan Kaufmann Publishers, Inc., 1998.
- [52] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1995.

- [53] B. Gates. *The Road Ahead*. Viking, a division of Penguin Books, USA, 1995.
- [54] A. Geist, A. Beguelin, J. Dongarra, W. Jiang, R. Manchek, V. Sunderam. *PVM: Parallel Virtual Machine: A User's Guide and Tutorial for Networked Parallelism*. MIT Press, 1994. Book URL: <http://www.netlib.org/pvm3/book/pvm-book.html>. PVM Home page: <http://www.epm.ornl.gov/pvm/>
- [55] D. Gelernter and D. Kaminsky. Supercomputing out of recycled garbage: Preliminary experience with Piranha. *Proc. of the 1992 ACM International Conference of Supercomputing*, July 1992.
- [56] W. Gibbs. CyberView. *Scientific American*, May 1997.
- [57] P. Gladychew, A. Patel, D. O'Mahony. Cracking RC5 with Java applets, in *Proc. of ACM 1998 Workshop on Java for High-Performance Network Computing*, Feb./Mar. 1998. Also published in *Concurrency: Practice and Experience*, 10(11-13), 1998.
- [58] Gnutella home page. URL: <http://gnutella.wego.com/>
- [59] J. Gosling, B. Joy, and G. Steele. *The Java Language Specification*. Addison-Wesley, 1996. URL: <http://java.sun.com/>.
- [60] Gotong royong. In *Bali Folder: the Reference of Bali Indonesia.*, 1999. URL: <http://www.balifolder.com/reference/folks/07.shtml>.
- [61] P.A. Gray, and V.S. Sunderam. IceT: Distributed Computing and Java, in *Proc. of ACM Workshop on Java for Science and Engineering Computation*, June 1997. Also published in *Concurrency: Practice and Experience*, 9(11): 1139 - 1160, Nov. 1997.
- [62] A.S. Grimshaw, W.A. Wulf, J.C. French, A.C. Weaver, and P.F. Reynolds, Jr. A Synopsis of the Legion Project. CS Technical Report CS-94-20, University of Virginia, June, 1994. URL: <http://www.cs.virginia.edu/~legion/>
- [63] W. Gropp, E. Lusk, and A. Skjellum. *Using MPI*. MIT Press, 1994.
- [64] Y. S. Gutfreund, J. Nicol, R. Sasnett, V. Phuah. WWWinda: An Orchestration Service for WWW Browsers and Accessories, in *WWW Conference '94: Mosaic and the Web*, 1994. Paper URL: <http://www.ncsa.uiuc.edu/SDG/IT94/Proceedings/Agents/gutfreund/gutfreund.html>. Home page: <http://www.fsz.bme.hu/~molnar/java/WWWinda/Orchestrator/documentation/WWWinda.html>
- [65] J. Hagimoto. Construct Java applications through distributed object technology, in *JavaWorld*, Dec. 1997. URL: <http://www.javaworld.com/javaworld/jw-12-1997/jw-12-horb.html>
- [66] P. Hellekalek, et al. The WWW Virtual Library: Random Numbers and Monte Carlo Methods. Institute of Mathematics, University of Salzburg, Austria. URL: <http://random.mat.sbg.ac.at/links/monte.html>

- [67] J.M.D. Hill, S.R. Donaldson, T. Lanfear. Process Migration and Fault Tolerance of BSPLib Programs Running on Networks of Workstations, in *Proc. of Euro-Par'98*, volume 1470 of LNCS. Springer, 1998.
- [68] S. Hirano. HORB: Distributed Execution of Java Programs, in *Proc. of WWCA'97*, volume 1274 of LNCS, pages 29–42, Springer, 1997. URL: <http://www.horbopen.org/>
- [69] S. Hirano, Y. Yasu, and H. Igarashi. Performance Evaluation of Popular Distributed Object Technologies for Java, in *Proc. of ACM 1998 Workshop on Java for High-Performance Network Computing*, Feb./Mar. 1998. Also published in *Concurrency: Practice and Experience*, 10(11-13), 1998.
- [70] R. V. Hogg, J. Ledolter, *Applied Statistics for Engineering and Physical Scientists*, Macmillan Publishing Co., 1992.
- [71] F. Hohl. Time-limited blackbox security: Protecting mobile agents from malicious hosts, in *Mobile Agents and Security*, G. Vigna, ed. Springer, 1998.
- [72] J.H. Holland. *Adaptation in Natural and Artificial Systems*. University of Michigan Press, 1975. (2nd ed.: MIT Press, 1992).
- [73] W. Hoscheck. The Colt Distribution: Open Source Libraries for High Performance Scientific and Technical Computing in Java. URL: <http://nicewww.cern.ch/~hoscheck/colt/index.htm>
- [74] Intel Corporation. Intel Forms Peer-To-Peer Working Group. Press release. Intel Developer Forum Conference, San Jose, CA. Aug. 24, 2000. URL: <http://www.intel.com/pressroom/archive/releases/cn082400.htm>
- [75] P. Jansson. Project 1: Gamma flux from a nuclear fuel. (Also featured in Process Tree home page.) URL: <http://www.jansson.net/distributed/project1/>
- [76] JARS home page. URL: <http://www.jars.com/>
- [77] G. Judd, M. Clement, and Q. Snell. DOGMA: Distributed Object Group Management Architecture, in *Proc. of ACM 1998 Workshop on Java for High-Performance Network Computing*, Feb./Mar. 1998. Also published in *Concurrency: Practice and Experience*, 10(11-13), 1998.
- [78] A.B. Kahng, J. Lach, W.H. Mangione-Smith, S. Mantik, I.L. Markov, M. Potkonjak, P. Tucker, H. Wang, and G. Wolfe. Watermarking Techniques for Intellectual Property Protection, in *Proc. of 35th Design Automation Conference Proceedings*, 1998.
- [79] M. Kantrowitz. Frequently Asked Questions: Genetic Algorithms, Aug. 1997. URL: <http://www.cs.cmu.edu/Groups/AI/html/faqs/ai/genetic/top.html>
- [80] M. Karaul. *Metacomputing and Resource Allocation of the World Wide Web*. PhD Thesis. New York University, May 1998. URL: http://www.cs.nyu.edu/csweb/Research/Theses/karaul_mehmet.pdf

- [81] K. Keahey and D. Gannon. PARDIS: CORBA-based Architecture for Application-Level PARallel DIStributed Computation, in *Proc. of Supercomputing '97*, Nov. 1997. URL: <http://www.supercomp.org/sc97/program/TECH/KEAHEY/INDEX.HTM>
- [82] C.W. Keßler. NestStep: Nested Parallelism and Virtual Shared Memory for the BSP model, in *Proc. of PDPTA'99*. CSREA Press, June/July 1999. URL: <http://www.informatik.uni-trier.de/~kessler/neststep.html>
- [83] Keynote Systems home page. URL: <http://www.keynote.com/>
- [84] D. Kirovski, Y. Huang, M. Potkonjak, and J. Cong. Intellectual Property Protection by Watermarking Combinational Logic Synthesis Solutions, in *Proc. of IEEE/ACM International Conference on Computer Aided Design*, Nov. 1998.
- [85] E. Korpela, D. Werthimer, D. Anderson, J. Cobb, and M. Lebofsky. SETI@home: Massively distributed computing for SETI. *Computing in Science and Engineering*, 3(1), Jan. 2001. URL: <http://www.computer.org/cise/articles/seti.htm>
- [86] M. LaDue. Hostile Applets Home Page. URL: <http://www.cigital.com/hostile-applets/>
- [87] M.D. LaDue. HoseMocha. URL: <http://www.cigital.com/hostile-applets/HoseMocha.java>, Jan. 1997.
- [88] S. Lalis and A. Karipidis. An Open Market-Based Architecture for Distributed Computing, in *Proc. of IPDPS 2000*, volume 1800 of LNCS, pp. 61 ff. Springer, May 2000.
- [89] S. Lawrence. The Net World in Numbers. *The Standard*, Feb. 7, 2000. URL: <http://www.thestandard.com/research/metrics/display/0,2799,10121,00.html>
- [90] T. Leighton. The Gambler's Ruin, in Lecture Notes for 6.042/18.062J: Mathematics for Computer Science. Massachusetts Institute of Technology, Dec. 1997. URL: <http://theory.lcs.mit.edu/classes/6.042/spring01/handouts/fall97/lecture26.ps.gz>
- [91] S. Levy. Wisecrackers. *Wired*, issue 4.03, Mar. 1996. URL: <http://www.wired.com/wired/archive/4.03/crackers.html>
- [92] R. Lipton and T. Sander. An additively homomorphic encryption scheme or how to introduce a partial trapdoor in the discrete log. Submitted for publication, Nov. 1997.
- [93] N. A. Lynch. *Distributed Algorithms*. Morgan Kaufman Publishers, Inc., 1996.
- [94] N. Metropolis. The Age of Computing: A Personal Memoir, in *A New Era in Computation*, N. Metropolis and G. Rota, eds., pages 119–130, MIT Press, 1993.

- [95] Microsoft Corporation. MSN "Kasparov vs. the World" Online Chess Match On Pace to Be Largest Interactive Competition in History. Press release, Redmond, WA. July 22, 1999. URL: <http://www.microsoft.com/presspass/press/1999/Jul99/InteractivePR.asp>
- [96] Microsoft Corporation. Hotmail: The World's FREE Web-based E-mail. URL: <http://www.hotmail.com/>
- [97] M.S. Miller and K.E. Drexler. Markets and Computation: Agoric Open Systems, in *The Ecology of Computation*, Bernardo Huberman, ed. Elsevier Science Publishers/North-Holland, 1988. URL: <http://www.agorics.com/agoricpapers.html>
- [98] M. Mitchell. *An Introduction to Genetic Algorithms*. MIT Press, 1996.
- [99] F. Monrose, P. Wyckoff, and A.D. Rubin. Distributed Execution with Remote Audit, in *Proc. of the 1999 ISOC Network and Distributed System Security Symposium*, Feb. 1999. URL: <http://cs.nyu.edu/~rubin/audit.ps.gz>
- [100] Napster home page. URL: <http://www.napster.com/>
- [101] *National Geographic Magazine*, Mar. 1977.
- [102] NCSA. The Metacomputer: One from Many, Nov. 1995. URL: <http://www.ncsa.uiuc.edu/Cyberia/MetaComp/MetaHome.html>
- [103] C.G. Ndiritu. In the spirit of harambee. Foreword in *Harambee: Pulling Together for Kenya*. University of Missouri-Columbia, Nov. 1997. URL: <http://www.ssu.missouri.edu/IAP/Harambee/>
- [104] M.O. Neary, S.P. Brydon, P. Kmiec, S. Rollins and P. Cappello. Javelin++: Scalability Issues in Global Computing, in *Proc. of ACM 1999 Java Grande Conference*, June 1999. Also published in *Concurrency: Practice and Experience*, 12(6-8), 2000. URL: <http://javelin.cs.ucsb.edu/docs/javelinplusCPE.ps>
- [105] M.O. Neary and P. Capello. Internet-Based TSP Computation with Javelin++, in *Proc. 1st Intl. Workshop on Scalable Web Services (SWS 2000)*, Aug. 2000. URL: <http://javelin.cs.ucsb.edu/docs/tsp.pdf>
- [106] M.O. Neary and P. Capello. Javelin 2.0: Java-Based Parallel Computing on the Internet, in *Proc. of Euro-Par 2000*, Aug./Sep. 2000, volume 1900 of LNCS, Springer, Aug. 2000. URL: <http://javelin.cs.ucsb.edu/docs/europar.pdf>
- [107] N. Negroponte. *Being Digital*. Vintage Books, a division of Random House, 1995.
- [108] M.V. Nibhanupudi and B.K. Szymanski. Adaptive Parallelism in the Bulk Synchronous Parallel model, in *Proc. of Euro-Par'96*, volume 1124 of LNCS. Springer, 1996.
- [109] R. van Niewpoort, J. Maassen, H.E. Bal, T. Kielmann, R.Veldema. Wide-Area Parallel Computing in Java, in *Proc. of ACM 1999 Java Grande Conference*, June 1999. Also published in *Concurrency: Practice and Experience*, 12(6-8), 2000. URL: <http://www.cs.ucsb.edu/conferences/java99/program.html>

- [110] M. Oguchi, T. Shintani, T. Tamura, and M. Kitsuregawa. Characteristics of a Parallel Data Mining Application Implemented on an ATM Connected PC Cluster, in *Proc. of HPCN Europe '97*, pages 303–317, April 1997. URL: <http://www.tkl.iis.u-tokyo.ac.jp/~oguchi/PAPERS/DOCUMENTS/HPCN.ps>
- [111] H. Pedroso, L. M. Silva, and J. G. Silva. Web-Based Metacomputing with JET, in *Proc. of the ACM 1997 Workshop on Java for Science and Engineering Computation*, June 1997. Abridge version also published as *Concurrency: Practice and Experience*, 9(11), Nov. 1997. URL: <http://www.npac.syr.edu/projects/javaforcse/acmprog/prog.html>
- [112] O'Reilly & Associates, Inc. OpenP2P home page. URL: <http://openp2p.com>
- [113] Parabon Computation, Inc. Parabon home page. URL: <http://www.parabon.com/>
- [114] Parabon Computation, Inc. Compute Against Cancer. URL: <http://www.parabon.com/cac.jsp>
- [115] Parabon Computaton, Inc. Frontier: The Premier Internet Computing Platform. White paper, 2000. URL: <http://www.parabon.com/clients/internetComputingWhitePaper.pdf>
- [116] S.H. Paskov. New Methodologies for Valuing Derivatives, in *Mathematics of Derivative Securities*, S. Pliska and M. Dempster, eds., pages 545–582. Cambridge University Press, UK, 1997.
- [117] K. Pearson. Internet-based Distributed Computing Projects. URL: <http://www.nyx.net/~kpearson/distrib.html>
- [118] Popular Power home page. URL: <http://www.popularpower.com/>
- [119] Popular Power. Simulation of the Immune Response to Influenza Vaccination. URL: <http://www.popularpower.com/applications/influenza.html>
- [120] T. Priol and C. René. Cobra: A CORBA-compliant Programming Environment for High-Performance Computing, in *Proc. of Euro-Par '98*, volume 1470 of LNCS, Springer, Sep. 2000.
- [121] Process Tree home page. URL: <http://www.processtree.com/>
- [122] A. Pryke. The Data Mine: Data Mining and KDD Information. URL: <http://www.cs.bham.ac.uk/~anp/TheDataMine.html>
- [123] B. Ramkumar and V. Strumpfen. Portable Checkpointing for Heterogeneous Architectures, in *Proc. of 27th Intl. Symposium on Fault-Tolerant Computing FTCS-27*, June 1997. URL: <http://supertech.lcs.mit.edu/~strumpfen/ftcs27.ps.gz>
- [124] M. Rappa. Solomon's House in the 21st century, in *Technika: A Review of Technology, Innovation, and Management*. North Carolina State University. Nov. 1994. URL: <http://technika.ncsu.edu/research/mrappa3.pdf>

- [125] D. Rossi. Jada home page. URL: <http://www.cs.unibo.it/~rossi/jada/>
- [126] RSA Security. RSA Laboratories Challenges. URL: <http://www.rsasecurity.com/rsalabs/challenges/>
- [127] T. Sander and C.F. Tsuchdin. Towards Mobile Cryptography. Technical report 97-049. International Computer Science Institute. Berkeley, CA, Nov. 1997. Also published in *Proc. of Security and Privacy '98*, May 1998. URL: <http://www.icsi.berkeley.edu/~tschudin/ps/tr-97-049.ps.gz>
- [128] T. Sander and C.F. Tschudin. Protecting Mobile Agents Against Malicious Hosts, in *Mobile Agents and Security*, G. Vigna, ed. Springer, 1998. URL: <http://www.icsi.berkeley.edu/~tschudin/ps/ma-security.ps.gz>
- [129] T. Sander and C.F. Tschudin. On Software Protection via Function Hiding, in *Proc. of the 2nd Workshop on Information Hiding*, volume 1525 of LNCS, Springer, April 1998. URL: <http://www.icsi.berkeley.edu/~tschudin/ps/ihws98.ps.gz>
- [130] L. Sandon. Advanced Undergraduate Project. MIT Dept. of Electrical Engineering and Computer Science, Jan. 1998.
- [131] L.F.G. Sarmenta. Volunteer Computing. Preliminary Concept Paper and Project Proposal. Unpublished draft, 1996.
- [132] L.F.G. Sarmenta. Bayanihan: Web-based volunteer computing using Java. Draft distributed at MIT research exhibit booth at *Supercomputing '97 (SC'97)*, Nov. 1997. Final version published in *Proc. of WWCA '98*, volume 1368 of LNCS, pages 444–461. Springer, Mar. 1998. URL: <http://www.cag.lcs.mit.edu/bayanihan/>
- [133] L.F.G. Sarmenta, S. Hirano, S. A. Ward. Towards Bayanihan: Building an Extensible Framework for Volunteer Computing Using Java, in *Proc. of ACM 1998 Workshop on Java for High-Performance Network Computing*, Feb./Mar. 1998. Also published in *Concurrency: Practice and Experience*, 10(11-13), 1998.
- [134] Luis F. G. Sarmenta. Protecting programs from hostile environments: Encrypted computation, obfuscation, and other techniques. Area Exam Paper. Submitted to Dept. of Electrical Engineering and Computer Science, MIT. July 1999.
- [135] M. Sato, H. Nakada, S. Sekiguchi, S. Matsuoka, U. Nagashima, and H. Takagi. Ninf: A Network based Information Library for a Global World-Wide Computing Infrastructure, in *Proc. of HPCN'97*, volume 1225 of LNCS, pages 491–502, Springer, 1997.
- [136] B. Schneier. *Applied Cryptography*, 2nd ed. John Wiley & Sons, 1996.
- [137] SETI@home. SETI@home home page. URL: <http://setiathome.ssl.berkeley.edu/>.
- [138] SETI@home. Frequently Asked Questions about SETI@home. URL: <http://setiathome.ssl.berkeley.edu/faq.html>.

- [139] D. Skillicorn, J.M.D. Hill, W.F. McColl. Questions and answers about BSP. Technical Report 96-15, Oxford University Computing Laboratory, Nov. 1996. Improved version published in *Scientific Programming*, 6(3), pp. 249–274, 1997. URL: <http://www.cs.queensu.ca/home/skill/QandA.ps>
- [140] D.B. Skillicorn. Strategies for Parallel Data Mining. Technical report TR1999-426. Queen's University, Ontario, Canada, May 1999. Also published in *IEEE Concurrency*, Oct.–Dec. 1999. URL: <http://www.cs.queensu.ca/TechReports/Reports/1999-426.ps>.
- [141] L. Smarr. Toward the 21st Century. *Communications of the ACM*, 40(11), Nov. 1997, pages 29–32.
- [142] SolidSpeed Networks, Inc. Probestor: Measuring Performance from the Edge of the Internet. White paper, Feb. 2001. URL: <http://www.solid-speed.com/solution/whitepaper/probestor.pdf>
- [143] V. Strumpen. Coupling Hundreds of Workstations for Parallel Molecular Sequence Analysis. *Software: Practice and Experience*, 25(3), 1995, pages 291–304
- [144] Sun Microsystems, Inc. Java Remote Method Invocation: Distributed Computing for Java. White Paper. URL: <http://java.sun.com/marketing/collateral/javarmi.html>
- [145] Sun Microsystems, Inc. Jini home page. URL: <http://www.sun.com/jini/>
- [146] H. Takagi, S. Matsuoka, H. Nakada, S. Sekiguchi, M. Satoh, and U. Nagashima. Nin-flet: a Migratable Parallel Objects Framework using Java, in *Proc. of ACM 1998 Workshop on Java for High-Performance Network Computing* Feb./Mar. 1998. Also published in *Concurrency: Practice and Experience*, 10(11-13), 1998.
- [147] UCLA IPP Group Publications Page. URL: <http://www.cs.ucla.edu/~gangqu/ipp/>
- [148] United Devices, Inc. United Devices home page. URL: <http://www.uniteddevices.com/>
- [149] United Devices, Inc. Cancer Research. URL: <http://www.uniteddevices.com/projects/cancer/>
- [150] L.G. Valiant. A bridging model for parallel computation, in *Communications of the ACM*, 33(8), pages 103–111, 1997.
- [151] L. Vanhelsuwe. Create your own supercomputer with Java, in *JavaWorld*, Jan. 1997. URL: <http://www.javaworld.com/javaworld/jw-01-1997/jw-01-dampp.html>
- [152] H.P. Van Vliet. Crema: The Java obfuscator. Original site no longer available. See instead: E. Smith. Mocha, the Java Decompiler. URL: <http://www.brouhaha.com/~eric/computers/mocha.html>

- [153] H.P. Van Vliet. Mocha: The Java decompiler. Original site no longer available. See instead: E. Smith. Mocha, the Java Decompiler. URL: <http://www.brouhaha.com/~eric/computers/mocha.html>
- [154] G. Voelker, D. McNamee. The Java Factoring Project. HTML document with Java alpha version applets, Sep. 1995. (URL no longer available.)
- [155] E. Walsh. Design and Implementation of a Framework for Performing Genetic Computation Through a Volunteer Computing System. Research Science Institute summer mentorship program report. MIT, Aug. 1998.
- [156] WebTV home page. URL: <http://www.webtv.com/>
- [157] B. Wilkinson and M. Allen. *Parallel Programming: Techniques and Applications Using Networked Workstations and Parallel Computers*. Prentice Hall, 1999.
- [158] G. Woltman. Mersenne.org Main Page. URL: <http://www.mersenne.org/>
- [159] W. Wulf. The Collaboratory Opportunity. *Science*. Aug. 1993.
- [160] XPulsar@home home page. URL: <http://www.xpulsar.de/>
- [161] Maryalice Yakutchik. Amish online: The barn raising. In *Discovery Channel Online*, 1997. URL: <http://www.discovery.com/area/exploration/amish/barnraising/main.html>.
- [162] Y.Yasu, et al. Evaluation of Gigabit Ethernet with Java/HORB, in *Proc. of Intl Conference on Computing in High Energy Physics (CHEP98)*, Aug./Sep. 1998, also published in KEK Preprint 98-181, Japan, November 1998.
- [163] Y. A. Zuev, The Estimation of Efficiency of Voting Procedures, in *Theory of Probability and its Applications*, 42(1), March 1997. URL: <http://www.siam.org/journals/tvp/42-1/97594.html>

Note on URLs. Where possible, we provide a URL with each entry, pointing to a web site containing a reference to the cited work, and possibly other related ones. All URLs here have been verified valid as of March 2001. To search for papers without URLs, or for further related papers, we recommend trying NEC's ResearchIndex (aka CiteSeer) at <http://researchindex.org/> or <http://citeseer.nj.nec.com/>.

Contact Information. The author may be contacted by email at lfgs@alum.mit.edu, which is an email-for-life address. The Bayanihan home page shall be maintained as long as possible at <http://www.cag.lcs.mit.edu/bayanihan/>