A Programming Model and Performance Model for Cycle Stealing

Jiro Sumitomo

May 2006

A dissertation submitted in partial fulfilment of the requirements for the degree of DOCTOR OF PHILOSOPHY

School of Software Engineering and Data Communications
Faculty of Information Technology
Queensland University of Technology
Brisbane, Australia
Copyright © Jiro Sumitomo, 2006. All Rights Reserved.

The author hereby grants permission to the Queensland University of Technology to reproduce and distribute publicly paper and electronic copies of this thesis document in whole or in part.
Keywords

Cycle stealing, Cycle scavenging, Volunteer Computing, Programming Model, Performance Model, Performance Analysis, Simulation, Distributed Computing
Abstract

This work describes a programming model and performance model for cycle stealing on the Internet.

Cycle stealing is the use of otherwise idle computers to perform work, and promises high performance computing at relatively low cost. The Internet, being the largest pool of potentially idle computers, is an obvious target for cycle stealing. However, computers connected to the Internet are often protected by firewalls, preventing point-to-point communication between them. The fluctuating availability of computers for cycle stealing as they move in and out of an idle state, combined with the restricted communication of the Internet environment, means that programming models and abstractions suitable for programming supercomputers and clusters are not ideal. Therefore, I have created a programming model for cycle stealing which reflects the types of parallel applications that are suitable for execution using idle computers connected to the Internet. The model is designed for use by non-expert parallel programmers, and I will show how it simplifies the development of cycle stealing applications, enabling rapid application development, and straightforward porting of existing sequential applications. This simple to use programming model, combined with the low cost of cycle stealing, improves the accessibility of high performance computing to non-traditional users of supercomputers and clusters.

Deployment on the Internet, and the need to navigate through firewalls, suggests a web based framework using common web protocols, web servers and web browsers. Part of this work investigates the feasibility of web based approaches to cycle stealing, from the setup of a cycle stealing system, application development and deployment, and connection of potentially idle computers. I designed and implemented a cycle stealing framework, deployable on the web, to meet expectations of performance, reliability, ease of use and safety.
Existing cycle stealing frameworks emphasise the need for applications to be decomposed into a set of jobs that execute for a long period, that is, a job should have a computation time sufficient to justify its communication cost. However, there are no tools available for users to determine what an appropriate computation time might be, given a job’s data communication requirements. To date, deciding the granularity of jobs has been a matter of intuition. Therefore, a user may experience uncertainty as to the benefit of cycle stealing for their particular application, especially if the applications will have relatively short-lived jobs. Based on performance analysis of my framework, I have developed an analytical model and simulator, which can be used to predict, and help to optimise, the performance of user applications, and show the feasibility of executing a particular application using the cycle stealing framework.
# Table of Contents

## CHAPTER 1  INTRODUCTION  ........................................................................................................................................1

1.1  MOTIVATION .................................................................................................................................................2

1.1.1  Innovation Diffusion Theory ..................................................................................................................4

1.2  TOWARDS SIMPLE, FLEXIBLE, PREDICTABLE FRAMEWORKS ..................................................................7

1.3  CONTRIBUTIONS ...........................................................................................................................................10

1.4  THESIS OUTLINE .........................................................................................................................................11

## CHAPTER 2  FRAMEWORK OVERVIEW ..................................................................................................................13

2.1  G2 CYCLE STEALING INFRASTRUCTURE .................................................................................................13

2.1.1  Client Component ..................................................................................................................................15

2.1.2  Volunteer Host ......................................................................................................................................15

2.1.3  Server .......................................................................................................................................................15

2.2  JOB EXECUTION ...........................................................................................................................................17

2.3  FRAMEWORK PROGRAMMING ..................................................................................................................18

2.3.1  Jobs ........................................................................................................................................................19

2.4  UNOBTRUSIVE PARTICIPATION ....................................................................................................................19

2.4.1  Definition of idleness and its implication for Jobs ................................................................................20

2.5  RELATED WORK .........................................................................................................................................22

2.5.1  Cycle Stealing Frameworks ...................................................................................................................22

2.5.2  Architecture ..........................................................................................................................................29

2.6  CONCLUSION ...............................................................................................................................................32

## CHAPTER 3  A MODEL OF CYCLE STEALING .........................................................................................................33

3.1  PROGRAMMING MODEL .................................................................................................................................34

3.1.1  XML Web Services ..................................................................................................................................36

3.1.2  ASP.NET Web Services Programming Model .......................................................................................36

3.1.3  Parallel Programming Using the Web Services Model .............................................................................39

3.1.4  Beyond Master-Worker ..........................................................................................................................42

3.1.5  Dataflow ..................................................................................................................................................53

3.1.6  Type-safe Dataflow ..................................................................................................................................58

3.1.7  Propagating Asynchronous Result Handles ..........................................................................................60

3.1.8  Data Parallel Programming in G2 ...........................................................................................................64

3.2  DEPLOYING AND USING A CYCLE STEALING SYSTEM .............................................................................77

3.2.1  Installers and Administrators ................................................................................................................78

3.2.2  Resource Donors .....................................................................................................................................78

3.2.3  Resource Consumers ..............................................................................................................................82

3.3  EASE OF USE THROUGH AUTOMATION .....................................................................................................83

3.4  RELATED WORK ..........................................................................................................................................88

3.5  CONCLUSION ...............................................................................................................................................93

## CHAPTER 4  REALISING AN INTERNET BASED CYCLE STEALING FRAMEWORK 97

4.1  EFFICIENCY AND PERFORMANCE ...............................................................................................................98

4.1.1  Efficient Communication ........................................................................................................................99

4.1.2  Mitigating the Cost of Communication ..................................................................................................102

4.2  RELIABILITY ...............................................................................................................................................105

4.2.1  Job Tracking .........................................................................................................................................105

4.2.2  Scheduling ..........................................................................................................................................107

4.2.3  Volunteer Checkpointing .......................................................................................................................113

4.3  SAFETY .......................................................................................................................................................123

4.4  PERFORMANCE EVALUATION ......................................................................................................................125

4.5  RELATED WORK .........................................................................................................................................134

4.5.1  Volunteer Idle Period ............................................................................................................................134

4.5.2  Minimising the Cost of Remote Job Execution ....................................................................................136

4.5.3  Broadening the Volunteer Base ..............................................................................................................141

4.5.4  Reliability ............................................................................................................................................141
 CHAPTER 5 PERFORMANCE MODELLING ........................................ 147
  5.1 PERFORMANCE EVALUATION METHOD ........................................ 148
  5.2 G2 SIMULATOR ........................................................................ 151
    5.2.1 Simulation Components .......................................................... 152
    5.2.2 Simulator Communication Model ............................................. 155
    5.2.3 G2 Specific Components ......................................................... 156
    5.2.4 Comparison .......................................................................... 162
  5.3 SIMPLIFIED SIMULATOR .............................................................. 164
    5.3.1 Service Time for web methods ................................................. 165
    5.3.2 Think Times ......................................................................... 168
  5.4 ANALYTICAL MODEL .................................................................... 170
    5.4.1 G2 Queueing Network Model .................................................. 170
    5.4.2 Mean Value Analysis ............................................................... 171
    5.4.3 Multiclass models ................................................................. 172
    5.4.4 Multiclass MVA extensions for G2 ............................................ 173
    5.4.5 Throughput - X ................................................................. 174
    5.4.6 Service Time - S ................................................................. 180
    5.4.7 Response Time - R ............................................................ 181
    5.4.8 Queue Lengths - Q .......................................................... 185
    5.4.9 Volunteer Volatility .............................................................. 186
    5.4.10 G2 MVA Equations ............................................................ 192
    5.4.11 Job Throughput ................................................................. 193
    5.4.12 Comparison ......................................................................... 193
    5.4.13 Job Latency ......................................................................... 194
  5.5 PROFILING METHOD ................................................................. 197
  5.6 VALIDATION .............................................................................. 198
  5.7 USAGE SCENARIOS ................................................................. 202
  5.8 RELATED WORK ........................................................................ 213
  5.9 CONCLUSION ............................................................................ 217

 CHAPTER 6 CONCLUSION .................................................................. 221
  6.1 SUMMARY .............................................................................. 221
  6.2 FUTURE WORK ......................................................................... 223

APPENDIX .................................................................................. 229
BIBLIOGRAPHY ........................................................................... 256
Table of Figures

Figure 2-1: The Internet with nodes behind firewalls .......................................................... 14
Figure 2-2: G2 star network topology ................................................................. 14
Figure 2-3: Pull communication model showing job submission and fetching ...................... 18
Figure 3-1: Continuations using subjobs ............................................................................. 46
Figure 3-2: A job may be broken down into subjobs, but appears as one job to its caller ....... 52
Figure 3-3: DAG of jobs ................................................................................................. 53
Figure 3-4: DAG of a PIFS application. Arrows indicate input data ..................................... 56
Figure 3-5: Generic, web page embedded G2 volunteer component .................................... 80
Figure 3-6: Client Operation Schematic ........................................................................... 84
Figure 3-7: Volunteer Operation Schematic ...................................................................... 87
Figure 4-1: Three threads fetch jobs, execute jobs and submit results ............................... 103
Figure 4-2: Pipelined job transmission ............................................................................. 103
Figure 4-3: A standard class, implementing the serialization interface, is turned into a mobilized class through the use of a Mobilization Tool ........................................ 120
Figure 4-4: TSP performance compared to linear speedup ................................................. 127
Figure 4-5: TSP 1000 city tour ......................................................................................... 129
Figure 4-6: Ray tracing application .................................................................................. 130
Figure 4-7: TSP speedup over a range of job execution times ........................................... 131
Figure 4-8: TSP Speedup over a range of job input and output data sizes ......................... 132
Figure 4-9: Speedup over a range of volunteer numbers ................................................... 133
Figure 5-1: Workstation component shown here with the network ................................. 153
Figure 5-2: Simulator Architecture with G2 specific thread logic ................................... 157
Figure 5-3: Listen thread state diagram .......................................................................... 157
Figure 5-4: Worker thread state diagram ........................................................................ 158
Figure 5-5: Submit Job thread state diagram ................................................................. 159
Figure 5-6: Fetch Results thread state diagram ............................................................. 160
Figure 5-7: Fetch Job thread state diagram .................................................................... 161
Figure 5-8: Execute Job thread state diagram ............................................................... 161
Figure 5-9: Submit Result thread state diagram ............................................................. 161
Figure 5-10: Simulator screenshot showing one client, one server, and four volunteers .... 162
Figure 5-11: Measured result throughput ...................................................................... 163
Figure 5-12: Simulated result throughput .................................................................... 163
Figure 5-13: Simplified simulator architecture .................................................................. 165
Figure 5-14: Predicted result throughput from the simplified simulator ......................... 170
Figure 5-15: Simple queueing network with multiple clients and volunteers .................... 173
Figure 5-16: Fetch Job probability graph with stable volunteers ...................................... 177
Figure 5-17: Result throughput predicted by the analytical model ................................... 194
Figure 5-18: Measured and predicted speedup, varying job execution time ..................... 199
Figure 5-19: Varying volunteer numbers ....................................................................... 200
Figure 5-20: Varying job and result data size. Execution time is one minute ..................... 201
Figure 5-21: Performance model accuracy over fast execution times ............................. 202
Figure 5-22: TSP speedup with up to 65 volunteers ....................................................... 204
Figure 5-23: Cross sections of Figure 5-22 .................................................................... 205
Figure 5-24: Predicted performance with up to 32 volunteers ........................................ 206
Figure 5-25: Smaller jobs can overwhelm the server if there are too many volunteers ..... 207
Figure 5-26: The server is overloaded with many volunteers and short execution times ... 208
Figure 5-27: Efficiency of the TSP application ............................................................. 209
Figure 5-28: Efficiency with 20 volunteers over a range of data sizes and job execution times ................................................................. 210
Figure 5-29: Cross sections of Figure 5-28 .................................................................... 211
Figure 5-30: Relationship between execution time, data size and volunteer volatility .... 212
Figure 5-31: Cross sections of Figure 5-30 .................................................................... 213
Figure C-1: Direct Dependencies of the Submit Job Response time equation .................. 245
Figure C-2: Second Generation Dependencies ................................................................ 246
Figure C-3: Full Dependency Graph ............................................................................. 247
Table of Listings

Listing 3-1: Classic web service method that adds two numbers ........................................ 36
Listing 3-2: Asynchronous web method invocation ............................................................ 38
Listing 3-3: Asynchronous web method invocation with callback .................................. 39
Listing 3-4: The Add web method in Visual Basic. This will execute as a G2 job as the G2 
framework supports and language with a compiler targeting the .NET runtime .......... 40
Listing 3-5: The code to create a G2 job is identical to web method invocation. Shown here 
in VB.NET ................................................................................................................................ 41
Listing 3-6: Coding for server-side asynchronous execution .......................................... 48
Listing 3-7: Recursive Fibonacci ..................................................................................... 49
Listing 3-8: Code for simple dataflow example .................................................................. 54
Listing 3-9: Example Dataflow code of a Partitioned Iterated Function System (PIFS) .... 57
Listing 3-10: Undesirable coupling between method implementations. To extract the result 
cepsulated in the G2AsyncResult object (called "pool"), a call to EndGetDomainPool 
is needed. .................................................................................................................................. 58
Listing 3-11: Example of Type Safe Data flow method signature ....................................... 58
Listing 3-12: The variable domainPool is declared to be of type G2AsyncResult. But 
BeginFindPairing expects a parameter of type domainBlock[]. ....................................... 59
Listing 3-13: The two automatically generated BeginFindPairings methods in the proxy...... 60
Listing 3-14: The standard EndPIFS method ...................................................................... 61
Listing 3-15: PIFS method that returns result handles to the caller .................................. 61
Listing 3-16: Block and Wait .............................................................................................. 63
Listing 3-17: Registering as a Listener ............................................................................... 63
Listing 3-18: The client part of a G2 application that renders the Mandelbrot set. The user 
specifies the range and the desired dimensions of the output image ............................. 65
Listing 3-19: Delegate invocation, illustrating two asynchronous invocation options....... 67
Listing 3-20: Using the Map construct, the creation of all of the jobs and obtaining the 
results is reduced to one line. ............................................................................................... 69
Listing 3-21: Proxy implementation of Map ...................................................................... 70
Listing 3-22: A Map overload to handle an array of single parameters ............................ 71
Listing 3-23: The proxy’s callrecord data structure is used to track results, to store the 
delegate, which is used to extract results, and to record the appropriate insertion point 
in the results array ................................................................................................................... 71
Listing 3-24: Methods of the Tile Class. Note that the data needed to execute methods 
comes from the method parameters as well as the data members ................................. 74
Listing 3-25: Automatically generated TileProxyCollection class ................................... 74
Listing 3-26: Proxy code for object based data flow. This is machine generated .............. 75
Listing 3-27: Serialization of Proxy objects ....................................................................... 76
Listing 3-28: Client side code for data parallel render .................................................... 77
Listing 3-29: Charlotte Parallel Step .................................................................................. 90
Listing 3-30: Job creation using Cilk-NOW ....................................................................... 91
Listing 3-31: Jicos Client Code ......................................................................................... 92
Listing 3-32: Jicos Volunteer Code .................................................................................... 92
Listing 4-1: Implementing the custom serialization interface ........................................... 119
Listing 4-2: Recursive Fibonacci (Checkpointed) ............................................................... 123
Listing 4-3: Parabon Frontier Checkpointing .................................................................... 144
Statement of Original Authorship

The work contained in this thesis has not been previously submitted to meet requirements for an award at this or any other higher education institution. To the best of my knowledge and belief, the thesis contains no material previously published or written by another person except where due reference is made.

Jiro Sumitomo

May 2006
Acknowledgements

A thesis is supposed to document one’s own work. However, this work is not possible without the guidance and support of others. I take this opportunity to express my gratitude to those to who have assisted me over the years.

I thank my supervisors, Wayne and Paul, for their invaluable advice, and for periodically hoisting me up, so that I too, could appreciate the view from 20 000 feet. My many collaborators: Dominic, who showed me the ins and outs of his mobilization framework; Richard, for many discussions, and his insight on cycle stealing and distributed computing; and Ben, for helping me wrestle with betas in the early days.

I also thank my fellow students and cubicle mates, Simon, Jens, Aaron, Greg, Richard, Doug, Gavin, Joel, Simeon, Asbjørn and Alex, for making my days at QUT that much more rewarding.

Finally I thank my parents. Without their support I might have had to go out and get a real job a long long time ago.
Chapter 1  Introduction

The combined processing power of idle computers has long been recognised as a potential resource [1]. Projects to harness the dormant power of idle machines have progressed from proof of concept systems, to successful, globally distributed cycle stealing systems, used for a range of computing challenges as diverse as protein folding [2] to searching for alien intelligence [3].

General purpose cycle stealing frameworks have been created with the goal of enabling their users to make use of idle computers, without having to develop their own cycle stealing systems. These frameworks consist of a software infrastructure that is deployed across a network of computers to form a cycle stealing network. Cycle stealing frameworks also incorporate an API, or their own programming language, for the development of cycle stealing applications.

Cycle stealing can provide high performance computing at a relatively low cost, compared to purchasing and deploying a supercomputer or cluster. This has prompted some organisations to deploy a cycle stealing system across their network of computers, to replace or augment their existing high performance computing resources. This low cost also provides an opportunity for high performance computing to those who may not have had access to such resources in the past. Without access to an organisational wide network of potentially idle machines, the Internet becomes the obvious source of idle computers and their cycles. However, many computers are shielded by firewalls, which place a restriction on the communication that is possible between the computers connected in a cycle stealing network. This restriction in communication, combined with the fluctuating availability of computers as they transition in and out of an idle state, makes the Internet a unique environment for parallel computing, which is quite different from that provided by a supercomputer or cluster. Therefore, programming models designed for supercomputers or clusters are not ideal, and a programming model specifically for Internet based cycle stealing is desirable. I have created such a model, with suitable abstractions for parallel programming on the Internet,
which I believe is easy to use, flexible and expressive, and easier to use than existing programming models for Internet based cycle stealing.

One problem with existing cycle stealing frameworks is that their users face uncertainty as to the performance benefits that cycle stealing can provide for their applications. A high performance cycle stealing application comprises a set of jobs, which are executed in parallel on volunteered and otherwise idle computers. The ratio between the communication time and the computation time of jobs dictates the performance of the application. For the users of existing cycle stealing frameworks, there is uncertainty as to how they should decompose their applications into jobs to achieve their performance goals, or whether their performance goals are even possible. To address this uncertainty, I have created a performance model based on the execution and analysis of applications using a cycle stealing framework that I developed, called G2. The G2 performance model quantifies the benefits of application execution using the G2 framework, by providing throughput estimates of jobs and their results, and it can be used to assist in the decomposition of an application to meet user performance goals.

As such, I have created a programming model, which simplifies the development of parallel applications for an Internet based cycle stealing system, and I have shown how a performance model of a cycle stealing system can be developed and used. These are the major contributions of this thesis.

1.1 Motivation

If a user has a problem that is computationally solvable, but the time needed to run their application is prohibitive using a single desktop computer, then the traditional solution is through the parallel execution of their application using a supercomputer or cluster. An alternative is parallel execution using cycle stealing. Cycle stealing is also known as cycle scavenging or volunteer computing.

The advantage of cycle stealing is its relative low cost, and as a consequence, high performance computing has the potential to be widely accessible, no longer lim-
itted to just those with access to supercomputers or clusters. Cycle stealing supports simple parameter sweep parallelism, where a sequential application is executed multiple times in parallel, and it supports the execution of parallel applications, where a single application executes on multiple processors. If a user needs to write and execute a parallel application, the easiest way to make use of cycle stealing is through a general purpose cycle stealing framework.

Existing cycle stealing frameworks each have their own programming model for application development. I wanted my programming model to be high-level and simple to use, enabling non-experts to develop parallel applications. I wanted it to be a model that supports rapid application development, enables existing applications to be easily ported, and present a simple and familiar desktop approach, in terms of how applications are developed, deployed and launched. The programming model also had to be suited to the intended execution environment; that is, on computers connected to the Internet. Coupled with the relatively low cost of cycle stealing, an easy to use programming model should make high performance computing accessible to a wider range of users than current users of supercomputers and clusters.

However, programming abstractions provided by existing cycle stealing frameworks are similar to that of supercomputers or clusters, requiring programmers to explicitly construct jobs or tasks. This is a problem because parallel programming is different from the programming of ordinary sequential applications, which is familiar to most programmers. Difficulties can arise because an application needs to be decomposed into parts that are capable of parallel execution, and then those parts need to be specified somewhere in code. If this is difficult or tedious, then people will be less inclined to make use of a cycle stealing framework.

When developing parallel programs for supercomputers, or clusters with dedicated nodes, programmers decompose applications using abstractions such as processes, and manage operations such as message passing and synchronization. In a cycle stealing environment, the execution of an application, or parts of an
application are dependent on the presence of idle computers. As a result, it is not ideal in this environment to decompose applications into concurrently executing processes, because it is difficult to ensure that parts of applications will execute at the same time. Similarly, operations such as synchronisation and message passing are not desirable, especially in an Internet environment where close coordination between processes is limited by high latency and firewalls.

Therefore, I have raised the level of abstraction in my programming model, especially in the way that programmers express the decomposition of an application, which simplifies the programming of a parallel application. My other goal for a programming model is to enable program compositions that go beyond master-worker. I therefore had to enable programmers to express relationships between jobs in a simple manner. I have used code annotation, in the form of attributes, to enable programmers to specify such dependencies in their application.

My intention was to incorporate the programming model into a new cycle stealing framework, and I wanted this framework to be used as a high performance computing platform. However, given the cost advantages, cycle stealing seems to be underachieving in the rate at which it is being adopted. So, to identify the features that a cycle stealing framework should have, and what may be lacking in existing frameworks, I considered the principles of Innovation Diffusion Theory.

1.1.1 Innovation Diffusion Theory

Innovation Diffusion is a theory that explains the characteristics of a technology that promote its adoption among potential users. The theory can be applied to a particular technology to identify the features that promote or hinder its adoption.

In his book, Diffusion of Innovations, Everett M. Rogers identifies five general factors that influence the rate of adoption of an innovation [4].

1) *Relative Advantage* is the degree to which an innovation is perceived as better than the idea it supersedes.
2) **Compatibility** is the degree to which an innovation is perceived as consistent with the existing values, past experiences, and needs of potential adopters.

3) **Complexity** is the degree to which an innovation is perceived as relatively difficult to understand and to use.

4) **Trialability** is the degree to which an innovation may be experimented with on a limited basis.

5) **Observability** is the degree to which the results of an innovation are visible to others.

Existing frameworks satisfy the requirements of relative advantage and observability. Therefore, I have focused on compatibility, complexity, and trialability.

**Compatibility**

For a cycle stealing framework, there are two aspects of compatibility.

1. How does the flexibility and performance of cycle stealing framework compare to a supercomputer or cluster?

2. How does the use of a cycle stealing framework compare to a desktop in the development, deployment and the way in which applications are launched? In other words, how does the use of a cycle stealing framework compare to the relative ease of using a desktop?

As a high performance computing system, users will expect a cycle stealing framework to provide functionality similar to the alternatives, such as supercomputers or clusters. In practice this is the ability to execute parallel applications reliably and quickly. These users will expect that a framework be reasonably flexible, and not limit them to a small subset of parallel applications. As a cycle stealing system relies on donated computers, users expect that applications will run safely, without harming those computers.
A significant proportion of potential users will not have experience with supercomputing and parallel programming. As such, a cycle stealing framework should also be compatible with their needs, and their experiences. These users are probably not experienced parallel programmers, and would prefer to use a programming style and language that is familiar to them. They may have existing applications that they wish to port. The types of applications they run are not grand challenge type applications such as SETI@Home, so their applications may have jobs that are relatively short lived. They may be accustomed to developing applications through rapid prototyping, so instead of deploying a finished application on a cycle stealing system, they may want to continually redeploy and test versions of their application on the system. And these users are accustomed to doing everything from the comfort of their own desktop, without having to log into other computers to run applications.

**Complexity**

Typically, applications have to be written to target a particular cycle stealing framework, so application development has a significant influence on the complexity in using a framework. The other factors are the way in which the cycle stealing infrastructure is deployed and maintained, and how applications are deployed, launched and managed. Trialability of a cycle stealing framework is linked to the complexity of deployment of the system and the development and deployment of applications. By reducing complexity in these areas, trialability is also enhanced.

Given the expectations of compatibility and the complexity of current frameworks, I believe that cycle stealing frameworks do not adequately support many potential users. I have created techniques to meet these requirements and have integrated them into the G2 cycle stealing framework.
1.2 Towards Simple, Flexible, Predictable Frameworks

Reducing Complexity

I have mentioned that application development for cycle stealing can be simplified. Programs written for current cycle stealing frameworks resemble traditional parallel programs, explicitly broken down into a set of tasks or jobs. Programmers may have had to learn a new programming language, or otherwise have had to craft the separate parts of the application that execute remotely, and then manually submit them for execution. I have developed a programming model that presents a higher level of abstraction. This model does not require programmers to implement interfaces or learn a new language. This model is language independent, uses the proxy programming pattern [5], and resembles that of ASP.NET web services, which is not a parallel programming model, but may be familiar to many users. The model is flexible and simplifies the creation of master-worker type applications as well as supporting data flow style applications. It can be easily extended, which I demonstrate by creating abstractions for the development of data parallel applications. The combination of simplicity and flexibility reduces complexity, as well as increasing the types of applications that the G2 framework can support.

Enhancing Compatibility

To be compatible with supercomputers or clusters, cycle stealing systems have to provide high performance computing. Perhaps the most successful cycle stealing system is SETI@Home, which has demonstrated the enormous potential of networks of workstations, in its case, computers connected to the Internet. The SETI@Home project was initially an application specific cycle stealing system, but has since migrated to a general purpose framework called BOINC [6]. BOINC is developed by the authors of SETI@Home and is suited to that type of parallel application.

SETI@Home, and other @Home style systems that have subsequently been developed, are intended to solve highly processor intensive applications that exe-
cute for months, if not years, involving thousands of computers that execute work units that run for many hours. These systems achieve high performance though the high throughput and duplicate execution of a large number of jobs using a large number of computers. It is not critical to performance to ensure minimum job latency, so that individual jobs return their results as fast as possible. This is due to the sheer scale of SETI@Home project, and the composition of the SETI@Home application. SETI@Home is involved in the analysis of radio telescope data. Each SETI@Home work unit performs analysis on a recorded signal, and the result of the work unit, the analysis, is the desired end product. In other words, the SETI@Home application has only one phase. There are no subsequent phases that rely on the initial analysis, so the next phase, and the application as a whole, is not held up due to a delay in the completion of a work unit. Also, the duration of the SETI@Home project means that a work unit could potentially take days or weeks longer than expected to return a result, but this will not significantly affect the performance of the application.

I mention SETI@Home here, and BOINC, its general purpose counterpart, because it is a cycle stealing system, deployable on the Internet, and it provides high performance. However, it is limited in that it supports only one type of parallel application, and therefore, it is not compatible with supercomputers or clusters because it does not offer the flexibility of those platforms. Although I do not claim that G2 supports the range of parallel applications supported by supercomputers or clusters, it is capable of efficiently executing applications that have relatively short running times, and applications with dependencies between jobs, where the execution of some jobs cannot proceed until other jobs have been executed. For these types of applications, it is desirable to produce and return a result as fast as possible, so I have designed and implemented techniques to satisfy this requirement.

One of the reasons that SETI@Home and other such projects have jobs that execute for many hours is to offset the communication cost of the job. All cycle steal-
ing frameworks, and parallel systems in general, have a requirement where efficient execution, characterised by speedup, depends on the processing benefits of parallel execution using multiple processors being sufficient to compensate for the communication cost of getting the job to the processors. This communication cost is not borne by a non-parallel application executing using one processor. The usual advice given to users of current frameworks is to say that execution time of jobs needs to be relatively long compared to their communication time. This advice is not sufficient and does not provide users with meaningful information as to how they should configure their applications. It does not give an indication of the benefits, if any, that the use of a cycle stealing system can provide for a particular application. The result is that users have to hope that their application’s job execution time is long enough, compared to job communication time.

To address this problem, I have developed both an analytical model and simulator, based on performance analysis of G2, which models the execution of parallel applications, and can be used to predict their performance on a given G2 system. This enables users to evaluate the benefits of cycle stealing, and to estimate and optimise the runtime behaviour of their applications.

If a person has a problem that they would like to solve computationally, then a typical scenario might be that a person writes an application, runs it on their computer, and if there are problems, they test and refine their application until it is correct. For a cycle stealing framework this scenario raises deployment and versioning issues, because the correct version of the application has to be deployed on all of the relevant computers. For a user working on a single machine, each new prototype replaces the old version which was deployed on the local disk, so the user does not have to take care of versioning or deployment, and can simply launch their application. I wanted to support this development scenario with G2. Therefore, I have included mechanisms to assign a globally unique name to each version of an application, and to ensure that the latest version is always deployed throughout a G2 network. The deployment of applications happens automati-
cally, and users are free to continually refine their applications without bothering to manually redeploy their application on the G2 network. G2 applications are launched from the user’s own machine, so launching a G2 application resembles launching an ordinary application.

1.3 Contributions

The following are the major contributions of this thesis:

**A programming model for Internet based cycle stealing**

The execution of applications on a set of volunteers that come and go, and the limits placed on communications between them by firewalls, places a restriction on the types of applications that can be supported on an Internet deployed cycle stealing system. I have shown how an existing programming model which is unrelated to parallel programming, can be adapted to develop a parallel programming model that is simple to use, is suited for the development of cycle stealing applications for the Internet, and supports parallelism beyond master-worker.

**Performance modelling and analysis**

Although there is some literature on the performance analysis of cycle stealing, rarely is any such analysis performed on a real world system. I demonstrate the application of a simple and well known analysis technique, mean value analysis (MVA), for the analysis of a real cycle stealing system and show how the standard MVA equations may be modified to fit the behaviour of a particular system to develop an analytical model of that system. I have developed a simulation framework for distributed systems and use it to produce a simulator of the G2 cycle stealing system. I show how these performance models, (the simulator and the analytical model), can enhance the use of a cycle stealing framework by enabling programmers to optimise their applications for execution using the framework, and providing insight into the relationship between job data size, job execution time, and the number and availability of idle workstations.
**Web based cycle stealing**

Some earlier frameworks made use of web browsers and applets to deliver frameworks components and jobs to volunteers. I have extended this idea to take a fully web based approach to cycle stealing, making better use of web server and content delivery mechanisms than did earlier frameworks, and using communication technology, in the form of XML web services, that is fully compatible with existing web standards. I demonstrate the feasibility of this web based approach to cycle stealing, and propose techniques to ensure that application execution using cycle stealing on the web is efficient, reliable and safe, as well as being simple.

**1.4 Thesis Outline**

This dissertation is organised as follows. In Chapter 2, I describe the architecture of the G2 framework, its three main components: Clients, Volunteers and Servers, explain the rationale behind its architecture and provide a comparison to other general purpose cycle stealing frameworks.

In Chapter 3, I describe my approach to the way a cycle stealing system could be used, and how this approach is manifested in the G2 framework. The first half of this chapter describes the G2 programming model, which I hope will make cycle stealing more accessible due to the ease with which it allows programmers to write parallel applications. I have made the model expressive, so that programmers can develop parallel applications that go beyond embarrassingly parallel, master-worker style applications.

I want the use of a G2 system to be analogous to the use of an electrical power grid. When a person wants to use an electrical appliance, they simply plug it into the grid and turn it on. Similarly, a parallel application taps into the power of idle volunteers, by simply connecting it to a G2 system. This approach is incorporated into the programming model, and it is apparent in the way that G2 networks are constructed, and how the G2 framework is able to propagate users’ application code to the server and volunteers.
In addition to ease of use, cycle stealing frameworks need to be efficient, reliable and safe. The features I have incorporated into the G2 framework to meet these requirements, along with the results of performance validation experiments are described in Chapter 4. One of the costs incurred through the use of a cycle stealing system, which is not incurred by a sequential application running on one computer, is the need to transmit jobs and results between clients and volunteers. I have devised means to make this communication as efficient as possible. By reducing communication costs, I can provide users with a wider range of options as to the granularity in which their applications are decomposed into jobs. Making communication more efficient also reduces the overall amount of traffic, which minimises the effect that G2 has on the other uses of a network. I describe the mechanisms in place to prevent loss of jobs and results in an environment where the volunteers can come and go at any time. Also included is an evaluation of the commonly used Eager scheduling algorithm and how I found that it can be detrimental in some circumstances, and an investigation into the possibility of adding job checkpointing capabilities into G2.

I performed extensive performance analysis and modelling of the G2 framework which is detailed in Chapter 5. The purpose of this work is to provide insight into the interaction between clients, volunteers and the server. A user may be unsure as to whether they can decompose their application in such a way, where parallel execution provides a performance benefit over sequential algorithms. Or they may have an application which is capable of being decomposed in various ways and don’t know which is best. My performance analysis and modelling resulted in an analytical model and simulator that can be used to demonstrate the performance gains that users can expect, and can used to optimise the decomposition of an application for execution on a particular G2 network.

I present my conclusions in Chapter 6, including avenues for future work.
Chapter 2 Framework Overview

A cycle stealing framework consists of software components that when deployed on a distributed set of computers, creates a network that is capable of hosting and executing applications on behalf of client users. A framework typically includes class libraries, and sometimes its own programming language, which programmers use to write applications for execution on the network.

In this chapter, I provide an overview of the G2 framework, its software components, and the architecture of a G2 cycle stealing network.

2.1 G2 Cycle Stealing Infrastructure

G2 provides high performance by enabling an application to be divided into parts that execute in parallel. Each of these parts is known as a job. The role of the G2 infrastructure is to provide a means to transport jobs from job producers, to job executors. Jobs are produced by a client computer running a user application, which incorporates a G2 infrastructure component called the client component. Clients connect and communicate with the remainder of the G2 network through the client component. Job executors, or volunteers, donate CPU cycles through the execution of a G2 infrastructure component called the volunteer host. Clients and volunteers are also known as nodes.

The largest source of potential volunteers is the Internet, so I wanted any computer on the Internet to be a potential node in a G2 network. However, many computers connected to the Internet are also behind firewalls, which often prevent unsolicited messages originating outside the firewall, from reaching their intended recipient behind the firewall. This precludes direct, bidirectional communication between nodes, because computers are not always able to receive messages. Therefore, an intermediary, capable of receiving and responding to requests, is needed to facilitate the movement of jobs and results between nodes.
This request-reply model is exactly the communication model of the web, and as most Internet aware computers have web access by default, I chose to make G2 a web based system. Therefore, G2 has a client-server architecture where the G2 server, which has a web interface, provides a bridge between participating nodes. To connect to a G2 network, a computer only needs to be able to connect to a G2 web server.
2.1.1 Client Component

A programmer enables their application to submit work to a G2 network, by incorporating a G2 client component into their application. Programmers do not explicitly construct job objects, instead the client component performs job creation and submission to the G2 network. The client component retrieves the results of jobs as they are computed and make those results available to the programmer’s application. The way in which programmers trigger the creation of jobs is described in detail in Chapter 3.

2.1.2 Volunteer Host

CPU cycles are donated to a G2 network through the volunteer host component. When active, a volunteer connects to the server and requests jobs. If a job is available, the volunteer will receive that job and execute it. The computed result is then submitted to the server. To minimise interference with the normal use of the volunteer node, the volunteer host is active only when the node is unattended, much like a screen saver. Therefore, volunteers are considered to be unreliable because their availability for job execution fluctuates over time, and a volunteer that has fetched a job may cease execution of the job and not return a result.

2.1.3 Server

One of the reasons I chose web based deployment for G2 is the availability of HTTP based web services. I used the ASP.NET web services implementation, which is used within the G2 framework as a communications mechanism. A G2 server hosts a web service exposing operations that enable nodes to submit and retrieve jobs, and to submit and retrieve results. Being HTTP based, this communications mechanism adheres to the request-response paradigm, and is no different to standard web traffic.

Although the server acts as a link between clients and volunteers, this does not mean that it provides a link between clients and specific, identifiable volunteers.
The server provides the client with access to a pool of volunteers. So, when a client submits a job, it does not submit it for execution by a particular volunteer, it submits the job for execution by some volunteer within a G2 network. The server acts as a clearinghouse for jobs, and also for results once they have been computed. As such, the communication of jobs between clients and volunteers is asynchronous, and it is necessarily this way due to the dynamic availability of volunteers. Although individual volunteers are dynamic and unreliable, the volunteer pool as a whole is considered static and reliable. Therefore, clients target the volunteer pool via the server, and not individual volunteers.

The role of the server as clearinghouse has other advantages in addition to handling a pool of dynamic volunteers. The server’s position as the centre of a G2 network makes it an ideal location from which to manage a network. In addition to its role facilitating communication, the server is responsible for, data distribution, reliability, scheduling and load balancing.

Data Distribution

The approach that I have taken with G2 jobs differs from the approach taken by other systems [7, 8], in that the code, and any data needed to execute a job, does not form part of the job payload. A G2 job consists of a serialized method call and some parameters. The actual implementation of the method is not part of each job, and is lazily distributed to volunteers. There is a one-time requirement for the client to submit code and data needed to execute the client’s jobs to the server. After which the server assumes the burden of distributing code and data to volunteers as required. The separation of method implementation from jobs, and the use of the server as the communication link, has the beneficial side effect of enabling clients to take advantage of the network bandwidth provided by the server. The server enables clients with modest network capacity to take advantage of many volunteers. Without this data distribution service, the number of volunteers that a client can use becomes limited by the network capacity of the client’s host machine.
Scheduling for Reliability and Load Balancing

The server, as the distributor of jobs is responsible for assigning them to volunteers and to implement policies that ensure that a particular job is executed. Thus the server is responsible for scheduling, and through the scheduler, reliability and load balancing.

Efficient scheduling can be complex because the scheduler operates in an uncertain environment. It does not know the job submission behaviour of clients in terms of quantity and frequency, nor the computation time of each job. The number of clients and volunteers will fluctuate, and the duration of availability, and capability, of a particular volunteer is uncertain. A centralised server architecture that distributes jobs among the available volunteers is efficient in this environment, because it allows the scheduler to continually adjust the number of volunteers executing jobs for a particular client, in effect redistributing volunteers among clients [9]. Contrast an architecture where volunteers connect directly to clients. In such an environment, it is more difficult to react to changing conditions, such as the arrival of new clients, or the availability of volunteers, because the connections between volunteers and clients have to be reorganised, rather than simply reassigning jobs to volunteers. The centralised nature of the scheduler gives complete control and tracking of all of the available volunteers for the duration that they are connected to the system.

2.2 Job Execution

Parallel execution using G2 starts when a client application, through the G2 client component, creates and submits a number of jobs to the server. Available volunteers fetch these jobs and execute them. Due to the request-response model of communication, volunteers pull jobs, which provides load balancing across volunteers because volunteers fetch jobs at the rate that they are able to execute them.
The execution of a job by the volunteer generates a result which it submits to the server. The server will hold the result, until it is retrieved by the relevant client. Result submission and retrieval is not shown in Figure 2-3, but it is similar to job submission except volunteers submit results to the server through a web service request, and clients retrieve results by requesting results from the server.

### 2.3 Framework Programming

A cycle stealing framework that supports the execution of parallel applications usually requires those applications to be written using that framework’s API. Frameworks differ in their concept of jobs, the way in which jobs are executed, their architectures, and so on. The consequence of these differences is that each framework requires a programming model and API tailored to its pattern of operation. The pattern that I have adopted for G2 is that a job is the evaluation of a user defined function. These jobs are executed on behalf of a client by a set of volunteers, each producing a result which is then returned to the client.

To develop applications that run on a G2 network, a programmer supplies a file containing the compiled functions that they wish to run in parallel. A tool parses the file and generates a proxy class, which the programmer incorporates into their application. The programmer then links their application to the G2 class libraries and then jobs can be created and submitted to the server by invoking
methods on an instance of the proxy class. This process is described in detail in Chapter 3.

2.3.1 Jobs
The data payload of a Job consists of the name of a function, the parameters to be used in the evaluation of the function, and some tracking data used by the G2 server. Once submitted, a job is added to a queue of jobs, and is assigned to volunteers on request. Jobs are assigned in a FIFO manner, but jobs may require differing amounts of computation to execute, and they are executed on volunteers with varying levels of capability. In addition, volunteers can start executing a job, but subsequently disconnect from a cycle stealing network. The consequence is that the execution order of jobs is non-deterministic, and to minimise the negative effect on performance of volunteer disconnections, a job may be redundantly assigned to multiple volunteers, and hence, executed to completion more than once. A G2 application programmer cannot rely on jobs being executed in a particular order, nor can they assume that jobs will even be executed concurrently. So a function should be programmed such that once it has been assigned to a volunteer, its successful evaluation does not rely on any other job. This reinforces the requirement imposed by the default assumption that volunteers cannot directly communicate with each other. That is, that executing jobs cannot communicate with other executing jobs. Jobs need to be able to execute completely in isolation, and apart from the computation of the result, the execution of the job should not otherwise affect the state of the application. A benefit of the isolated execution of jobs, is that a failed job execution caused by an active volunteer disconnecting and leaving the volunteer pool, can be recovered simply by assigning that job to another volunteer.

2.4 Unobtrusive Participation
There are many reasons why a computer may be connected to a cycle stealing network as a volunteer. Perhaps that computer is part of an organization and
connection to the cycle stealing network is mandatory. Perhaps the owner of the computer is selling their spare cycles. Or perhaps, the owner is participating for interest or altruistic reasons. Regardless, it is important that the process of volunteering does not interfere with the ordinary use of the volunteered computer, and so cycle stealing should be as unobtrusive as possible. The web based approach that I have taken helps to minimise any effects of cycle stealing on the volunteer computer, and the computer owner’s local processes. Firstly, the volunteer host component is embedded in a web page, so it does not need to be installed and take up an exclusive portion of a donated computer’s hard disk space. That is not to say that it will not take up any space at all. The volunteer host component may be stored in the browser’s cache as a normal part of the browser’s operation. A beneficial side effect of this approach is that a user will not encounter any versioning problems, because the latest version will always be downloaded from the web page. Secondly, browser based volunteering gives the donor full control over when the volunteer is operational. If so desired, the donor can configure their computer\(^1\) to have a browser to connect to G2 whenever the computer is idle, much like a screen saver. The volunteer host component can be terminated by simply closing the web page.

### 2.4.1 Definition of idleness and its implication for Jobs

To minimise the effect on the normal use of the volunteer computer, I have assumed that the volunteer host will only be active when the volunteer computer is unattended. So the definition of “idleness” that I use, is when its owner is not physically interacting with their computer, such that the volunteer host runs like a screensaver.

Other cycle stealing systems, such as SETI@Home, have another operating mode to maximise the use of the volunteer computer. Their volunteer components are

\(^{1}\) In Windows, this is done by creating a new Scheduled Task that executes when the computer is idle.
always active, but execute as low priority processes [10]. They are able to skim the CPU cycles not used by local processes by taking a fine grain view of idleness. In terms of CPU usage, the interference to local processes by the activity of the volunteer component is hardly noticeable, but the volunteer component does maintain a constant memory footprint. For application specific systems such as SETI@Home, the characteristics of each job, and the memory requirements to execute them, are well known. For SETI@Home, the memory footprint of the volunteer application is about 15MB which has little effect on modern computers.

On general-purpose cycle stealing frameworks however, the memory requirements of jobs are not known and they may be significantly large, causing too much interference to other applications running on the volunteer computer. It is possible to have a volunteer host that is capable of controlling the memory use of jobs, but such a volunteer host component is not conducive to web based, browser-integrated deployment. Therefore, I believe that perpetually active volunteers are not generally suitable cycle stealing frameworks striving for unobtrusive participation. The owner of a volunteer computer should always have the option of running the volunteer host only in a screensaver mode.

One of the consequences of screensaver mode is that like a screensaver, the volunteer host ceases activity and job execution terminates when the volunteer owner reclaims their computer. This behaviour has been described in the literature as immediate eviction [11]. My implementation of this policy is that the volunteer terminates and releases resources immediately. This policy means that the volunteer cannot persist data on the local disk, nor can it send data to another location once the host computer has been reclaimed. A consequence of immediate eviction is that it prevents a volunteer from being able split the execution of a single job over more than one idle interval; any incomplete work performed on a job prior to the donor reclaiming their computer is lost. The implication is that the probability that a job will not be executed to completion increases with the execution time of a job.
Faced with these conditions, many cycle stealing frameworks, including G2, have adopted a policy that encourages jobs of relatively short duration, where jobs that fail to complete are simply restarted on another volunteer [12]. To use this simple error recovery mechanism, the framework must guarantee that such restarts, (resulting in the duplicated executions of jobs, or parts of jobs), do not cause errors in the application. Providing such a guarantee becomes difficult if the partial execution of a job is able to affect the state of the distributed application. This difficulty adds further justification for jobs not to be able to communicate with other jobs and to make jobs function evaluations, instead of operations on persistent remote objects. Idempotent jobs capable of fully independent, isolated execution allows a simple error recovery mechanism, without the need for elaborate rollback schemes, or other mechanisms to identify duplicated operations to change the state of an application.

### 2.5 Related Work

#### 2.5.1 Cycle Stealing Frameworks

In this section I present a selection of related Cycle stealing systems. More detailed comparisons to related work are provided in later sections where relevant.

#### 2.5.1.1 Condor

One of the earliest cycle stealing systems was Condor [13]. Rather than execute parallel programs, Condor allows users to execute sequential programs in parallel on idle machines on the network. It is a batch processing system, whose focus is on high throughput, not high performance. Condor handles long running programs, and machine failures with a system of checkpointing and job migration. Jobs would be matched to appropriate volunteers through a matchmaking system called ClassAds. This resource matchmaking system requires jobs to specify their execution requirements, and volunteers to specify their available resources, such as the presence of a JVM, and accordingly matches jobs to volunteers. I/O occurs through files, all of which are stored on the client’s computer. This means that a
Condor user has access to all the data produced by their applications at all times, removes the need for the client to have read and write permission on volunteers, and the owners of volunteers are not burdened with foreign files taking up space on their disks. Initially designed to be deployed on a local area network, Condor has since been expanded to operate over wide area networks, and to participate in a global grid [14].

2.5.1.2 Nimrod

Nimrod [15] enables the execution of parametric studies across distributed computers. A parametric study involves an application, or applications, being executed multiple times across set of input parameters. By using available volunteers, a parametric study can be conducted in parallel, providing performance benefits to users. The Nimrod system manages the distribution of necessary files, the execution of applications and the recovery of results. Instead of performing an exhaustive search of the parameter space, a search for parameters to minimise or maximise an output variable can be performed using a tool called Nimrod/O [16], and a variant, Nimrod/G [17], is able to be connected to a global grid.

2.5.1.3 SETI@Home and BOINC

SETI@Home [3] is perhaps the most visible example of the potential of cycle stealing. Using millions of volunteered workstations radio telescope data is analysed for signs of extra terrestrial intelligence. Although this is arguably not the most practical use of so much computing power, it was probably the choice of the SETI application that attracted so many people to participate in the project. Each volunteer downloads and installs an application which connects to a central SETI@Home server and downloads a work unit. After the work unit is executed, the result is uploaded to the server. To ensure accuracy the processing of each work unit is duplicated by multiple independent users. The success of the SETI@Home project inspired similar projects, typically associated with pharmaceutical research, such as Cancer@Home. The developers of SETI@Home have
since created a general purpose framework, called BOINC [6] enabling anyone to roll their own SETI@Home like project.

2.5.1.4 Cilk

Cilk [18] is a parallel multithreaded extension of the C language. A runtime system, called Cilk-NOW [19] was created to run Cilk-2 programs on a network of UNIX machines. When a workstation became available, it would join the computation, and retreat when its owner returned to it. Other features of Cilk-NOW were its work stealing scheduler [20], a system whereby idle machines could “steal” work from its neighbour’s work queue, and a fault tolerance mechanism that detected and recovered from failures without the application programmer having to handle it themselves. In other words, Cilk programs are fault oblivious, as it is the role of the runtime system to detect and recover from failures. This greatly simplified programming for the system. Cilk-NOW programs could be written as if they were intended for execution on one machine, and the runtime system would schedule execution across the network.

2.5.1.5 Charlotte

Charlotte [21] was the first cycle stealing system to use web based deployment. It leveraged Java enabled web browsers by deploying applications as Java applets embedded in web pages. Charlotte programs are written as sequential programs with parallel steps. These parallel steps consist of a set of routines which are executed on available computational nodes. Data is passed to and from volunteers through distributed shared memory (DSM) managed by the client. Apart from distributing jobs as embedded applets, Charlotte also introduced eager scheduling which simplified volunteer management. Instead of having to detect whether a particular volunteer had ceased execution of an applet before reissuing the same job to another volunteer, the eager scheduler distributed jobs to any available volunteer regardless of whether it had issued that job previously, and whether another volunteer was working on that job. Although creating redundancy, eager
scheduling promoted reliability while eliminating the need by the system to track the status of volunteers.

2.5.1.6 Knitting Factory

The Knitting Factory [22] extensions to Charlotte enhanced its scalability and provided IPC mechanisms using Java RMI. The original implementation of Charlotte relied on a central web server to serve out applets. This approach was abandoned by Knitting Factory by having Clients run a lightweight applet server, called a KF Class Server, to serve out applets. Volunteers were matched to Clients through a directory service, the KFDS. The purpose was to improve scalability by decentralizing the distribution of applets. Each KFDS maintains a list of clients as well as links to other directory servers. New clients register with a known KFDS on initialisation. When a volunteer connects to a KFDS, it is connected to an available client, or it directed to another KFDS. IPC was realized through volunteers registering an RMI object when they connected to a client. That client would then propagate the RMI handle to the other volunteers. By using the then current version of Java RMI, the developers of Knitting Factory were able to circumvent host of origin communication restrictions that usually apply to applets, and to enable direct communication between volunteers.

2.5.1.7 Piranha

Piranha [23] is an implementation of the Linda concurrent programming model in Java. Linda provides an abstraction for concurrent programming though tuples. A tuple is a series of typed fields stored in a tuplespace that is accessible by all processes. Originally intended to be an alternative scheduling strategy for multiprocessor machines, the idea was applied to cycle stealing. The Piranha project introduced the term “adaptive parallelism” which describes the execution of a program of a dynamic set of processors [24]. In Piranha, the master process known as the feeder, connects to the tuplespace and inserts a live tuple, or job. Volunteers run a piranha daemon that connects to the tuplespace, randomly se-
lects an application and forks a process to execute it. Results are returned to the Feeder via the tuple space. The executing process can store its state as a tuple which can be reactivated at a later time for checkpointing and migration purposes. However, this must be explicitly controlled by the programmer, that is, the runtime system is not able to take a generic job and store its state as a tuple.

2.5.1.8 Javelin

Like Charlotte, the original Javelin [25] system distributed jobs by embedding applets into web pages. Unlike Charlotte, where a central server both hosted and distributed applets, Javelin clients deploy applets on any web server and register the URL with the Javelin server, known as the broker. Volunteers communicate results back to the client through the client specified web server.

Clients in Javelin++ [26] did away with applets and distributed tasks as java applications. Hosts (Javelin volunteers) retrieved and executed jobs by registering with a broker and being directed to a Client. Brokers could act as a cache for Clients, so that the client was not solely responsible for the distribution of code. This is intended to eliminate the bottleneck created if many volunteers tried to simultaneously connect to the client. When a volunteer connects to a client, it is inserted into a heap managed by the client. The volunteer “steals” work from its parent as specified by the heap, and them from any children. This is the basis of the “work stealing” scheduler introduced by Cilk and incorporated into Javelin.

Javelin++ facilitates scalability by maintaining a limit on the number of connections to each broker. The original Javelin prototype had a single broker that suffered in performance if more than 60 hosts connected to it. Javelin++ has a network of brokers of which there are two types. Primary brokers start up without logging onto any other brokers. They link to other brokers by reading a configuration file – these primary brokers make up the backbone of the system. Secondary brokers are normal Javelin++ hosts that connect to their local broker. If a broker reaches its connection limit, one of the connected hosts can become a new
broker if it is prepared to take on that role. To be eligible, a host must have a permanent internet connection and its owner must agree to only disconnect from the Javelin++ network gracefully, allowing the system to replace it. Javelin 2.0 [27] adds the Broker Name Service that assists in the discovery of new brokers, and the ability to branch and bound computations to minimise unnecessary computation.

2.5.1.9 CX

CX [28] was developed from Javelin and differs only in the way the brokers, or task servers, are arranged. A task server and its associated producers make up an “isolated cluster”. Arranged as a sibling connected fat tree, tasks diffuse through the network when each server pings its siblings and queries its state. If there are excess tasks, then they “diffuse” to the less loaded task server. The computational model is similar to Cilk threads. It incorporates a continuation style model that can be represented with a directed acyclic graph. The underlying communication model is built using Jini [29].

2.5.1.10 Jicos

Jicos [30] is the latest incarnation of CX and provides a simplified API as well as a different task server arrangement. At the top level are Host Service Providers (HSP). On creation, task servers register with an HSP. Consumers (clients) submit work to the HSP, which sends it to a task server. Producers (volunteers) connect to a task server after being directed by an HSP.

The Jicos API provides for three types of tasks – root tasks, decompose and compose tasks. (Same class, different constructors) Root tasks are created by the Consumer and create other tasks on execution. Decompose tasks can complete a calculation, create new tasks, or both. Compose tasks receive results from the created tasks and can be executed when all pending results are received.
2.5.1.11 Bayanihan

Bayanihan [8] is a “Web-based volunteer computing system” that uses Java applets to distribute code to volunteers, and HORB [31], a Java based object request broker, for communication to the server. Being web based, volunteering involves connecting a workstation to a Bayanihan server and viewing a web page with an embedded chassis applet. When the chassis registers with a server, the server creates a “work advocate” that acts as the contact point between the server and the volunteer. The work advocate, which is essentially a proxy, contains information, such as a process id, which is used to manage the volunteers. The Chassis contains a work engine which repeatedly makes remote calls to the work advocate, which relays work data objects from the server’s work manager to the volunteer chassis’ work engine.

The watch engine is a separate chassis applet that connects to the watch manager on the server via an advocate, and can receive result messages, usually with a result object, from the server. The watch manager can also be used to interactively control the parameters of the computation as it occurs.

To increase scalability, the creation of volunteer servers is possible. These are special Java applications with a simple HTTP applet server that volunteers can download and execute. These connect to the main server and request a set of jobs that it can pass on to any connecting volunteers.

2.5.1.12 Bayanihan Computing .NET

Bayanihan Computing .NET [32] has parallels to the current implementation of G2. It uses web services as the communications bridge to connect applications with volunteers. Using web methods, a task pool is created on the server and work is added to them. A dll containing the volunteer code is also uploaded. To volunteer, users download an executable file, which connects to the server and its task pool via web methods. If it does not already exist, a local copy of the dll is made, the worker object instantiated and the doWork method is called on it.
Bayanihan Computing .NET relies on the security features of the CLR to protect volunteer nodes, as well as requiring all users to log on, and only trusted users can submit work.

2.5.1.13 **Alchemi.NET**

Like G2 and Bayanihan Computing .NET, Alchemi.NET [33] is based on the .NET framework with an additional ability to be integrated into a global grid. Programmers use the Alchemi.NET API to decompose their application, called a “Grid Application” into a set of independently executing jobs, called “Grid Threads”. These threads are farmed out to Alchemi Executor nodes via an Alchemi Manager Node. Alternatively, precompiled executables can be “grid-enabled” and executed using Alchemi.NET. The executable to be run, along with any input parameters are specified using an XML document, which is submitted to an Alchemi Console Interface or an Alchemi Cross Platform Manager. These convert the XML representation of a job into Grid Threads which are subsequently executed on Executor nodes. An Alchemi Cross Platform manager provides integration into global grids by exposing a web interface which is used to trigger the execution of Grid Applications.

2.5.1.14 **United Devices Grid MP, Parabon Frontier**

Grid MP from United Devices [34]; and Frontier, from Parabon [35], are two commercially released frameworks. Both feature a centralized job scheduler which receives and distributes jobs out to available nodes.

2.5.2 **Architecture**

The majority of existing cycle stealing frameworks use a centralised server of some description, but the role of the server differs between frameworks.

In the Condor [13] system, a set of volunteers, are organised into Condor Pools each controlled by a central manager. Clients host an agent, which announces the availability of jobs, along with the resource requirements of each job. The agent coordinates with a management process called a Matchmaker [36] to match vol-
unteers to clients. Volunteers advertise their capabilities to the matchmaker through a resource matching mechanism called ClassAds (Classified Advertisements) [37]. The Matchmaker connects volunteers with clients, and then the client is responsible for sending jobs to volunteers, including the files and other data needed by volunteers to execute the jobs. In this sense, the client assumes some of the responsibilities that would be performed by a G2 server, including maintaining its queue of jobs and scheduling job allocations to volunteers. The results of jobs are typically output files written directly to a file system share on the client.

In contrast to G2, the direct connection between volunteers and clients, the client as the distributor of data, and the requirement of a shared file system makes this type of system suitable more for a LAN environment than the Internet.

Delegating the role of data distributor to the client machine was also used by Charlotte [21]. Like G2, Charlotte is web based but it requires the client host computer to be running a web server to serve jobs to connecting volunteers. This integrates the roles of client and server into a single machine, and means that the client must be able to create direct connections with volunteers.

Like Condor, Javelin [25] separated the roles of matchmaker and job server. Javelin clients use a web server to distribute jobs, but the matchmaker role is fulfilled by a network of brokers. Clients advertise their need for volunteers on a broker, which includes the URL of their job server. Volunteers connect to the brokers and are redirected to the client specified job servers. This mechanism potentially allows the client to delegate the role of job server (and data distributor) to a machine other than its host computer. Making data distribution the responsibility of the client does have the advantage of eliminating the bottleneck of having a centralised server, but it does make the usage of Javelin more complicated than G2. A person wishing to submit jobs must also be running, or have some control of the content of a web server, and deploying an applications requires the extra step of making their data available for download by volunteers.
Direct communication between volunteers is generally not permitted, but an early system, Knitting Factory [22], enabled volunteers to communicate through Java RMI. Whenever a volunteer connected, it received handles to all other volunteers, enabling communication between executing jobs. This functionality is generally not supported by cycle stealing frameworks due to difficulties that arise when volunteers leave the network. This information must be propagated to the remaining host machines, and becomes especially complicated if the host machine leaves during the execution of a job that communicated with another. This loss can compromise the integrity of the remaining jobs, which must be able to cope with the loss or be rolled back to some safe state. Knitting Factory, as an experimental addition to Charlotte, did not deal with this problem.

Other systems allow direct communication between volunteers but not communication between executing jobs. The ability to communicate between volunteers is used for a scheduling algorithm introduced by Cilk [19] called work stealing [38]. In work stealing, each volunteer maintains a queue of jobs. If a volunteer exhausts its local queue, it connects to another volunteer and “steals” some jobs. Using this method, jobs can diffuse throughout a Cilk network making work stealing an effective method to load balance jobs among volunteers. Later versions of Javelin and related frameworks, CX [28] and Jicos [7] make use of work stealing. Work stealing delegates the role of job server from dedicated servers, to executing volunteers. This again has the benefits of removing reliance on a central node, and work stealing also acts to load balance jobs among volunteers. However, work stealing depends on volunteer nodes being aware of each other and being able to communicate with each other, which is not always possible.

G2 is a fully web based framework, using XML web services on HTTP as the communications model. Although this approach enables excellent connectivity, a disadvantage is that messages can be verbose. Alchemi.NET [33] is a framework capable of Internet deployment that has a server component to which nodes can communicate using TCP. The server also has a web service interface, thus avoid-
ing the verboseness of HTTP and SOAP where possible, but providing the fall-back option of HTTP web services where required.

2.6 Conclusion

The architecture of the G2 framework is influenced by the goal to be easily deployable on the Internet. The Internet is a network where security considerations necessitate people to closely control the types of communication allowed on their computers. Often this control is exerted through firewalls, which may not be under the direct control of all the individual users behind them. Firewalls are often configured to block unsolicited messages from the outside, which hinder peer to peer communication, and in the case of G2, direct communication between nodes. However, access to the web is commonplace. Therefore, I have concentrated on web based deployment, which not only maximises the number of potential volunteers, and also enables users to quickly and easily form ad-hoc cycle stealing networks.

There are however, disadvantages that arise from using the web, such as the lack of direct connectivity between volunteers, and communication occurring only after initiation by nodes. There are ways to get around such limitations, such as requiring all computers to run web servers, but I have rejected this as self defeating – it is too complicated, and may not work behind firewalls in any case. Chapter 4 describes the features that I have incorporated into the framework to overcome some of the difficulties encountered by being web based, but I believe the advantages conferred by the accessibility and simplicity of web based deployment far outweigh any disadvantages.
Chapter 3 A Model of Cycle Stealing

Cycle stealing promises low cost, high performance computing. This low cost makes cycle stealing attractive to organisations; it also makes high performance computing accessible to those who in the past, may not have had access to high performance computing systems. The scale of deployment may be different, in that large organisations can, and are, deploying cycle stealing systems across their intranets, providing a high performance facility to their employees, whereas individuals may only be able to rely on a few computers, and the computers of their friends or colleagues. But regardless of scale, the principles are the same; take advantage of the cycles yielded by otherwise idle computers.

If cycle stealing can be done cheaply, then everybody has the opportunity to deploy, or have access to a cycle stealing system. But cost and availability are not the only barriers to high performance computing. To take advantage of a cycle stealing system, a person needs to be able to express their problem as a parallel, cycle stealing application. If programming such applications is complex, then this can prevent people from taking advantage of cycle stealing. Reducing complexity also makes application development faster and less error prone.

Therefore, cycle stealing is more useful to more people if it is also easy to use. This ease should characterise not only application development, but the entire process of cycle stealing, from system deployment, to application deployment and execution, and the connection of idle computers as volunteers. I wanted the use of G2, to be analogous to the way a person might use electrical power. They simply plug their appliance into the system. They don’t have to tell the power company what they will be using, and where the power should come from. Similarly, a G2 user should be able to plug their application into the G2 system, without needing to pre-deploy parts of their application, or configure the system for their application. This chapter describes my work to enable users to simply plug into a cycle stealing system. In particular, a programming model for Internet based cycle stealing, with abstractions suited to the types of applications supported by
Internet based cycle stealing frameworks. This model is one of the contributions of this thesis. It provides a simple interface to the G2 system, while providing flexibility for programmers. The latter part of this chapter describes the features that I have incorporated into the G2 framework that simplify the process of cycle stealing, from system set up, application deployment, to connecting volunteers. I describe a method of deploying and launching applications on a cycle stealing system, which is largely automated and enables users to delegate deployment and versioning to the framework.

3.1 Programming Model

Like most other cycle stealing frameworks, G2 provides high performance by executing parallel applications. This gives rise to two factors that influence, and potentially complicate application development for G2. The first is the requirement that the programmer write parallel applications. This is more complicated than writing applications that execute sequentially, and I do not assume that G2 programmers are experienced parallel programmers. The second, is that parallel applications are executed on a set of processors that change over time; Adapting to an ever changing set of processors is known as adaptive parallelism [24] and this is something that existing frameworks can handle without programmer intervention. However, while programmers don’t have to do anything in code to handle volunteer disconnections, adaptive parallelism does have implications that affect the types of applications that a cycle stealing framework can support.

A multiprocessor machine or cluster runs parts of an application (jobs) in parallel, with each job running on a separate processor. Close interactions, such as messaging between jobs, is plausible because each processor is reliable, and programmers can be sure that when a job tries to message another job, the target job will be active on another node. In contrast, a cycle stealing environment relies on relatively unreliable volunteers to execute the jobs of an application. Allowing close interactions between jobs can be problematic when one volunteer inevitably disconnects from the cycle stealing system while an application is executing. Recov-
ery from such an occurrence is costly and expensive, and is expected to occur too frequently to bear. This encourages a style of parallelism where the execution pattern of applications is simple, where interactions between executing jobs, if any, are severely limited, and the consequences of volunteer disconnection are few.

Further impetus to this style of parallelism is added if recovery from a volunteer disconnection is the responsibility of the cycle stealing system. System managed adaptive parallelism is easier to design and implement if interactions between jobs are limited, thereby quarantining the effects of a volunteer disconnection to a single job.

Thus, irrespective of the fact that web based volunteers may not be able to communicate with each other directly, I have adopted a common cycle stealing approach of disallowing interactions between executing jobs. Often, this approach restricts programmers to embarrassingly parallel, master-worker type applications. I wanted to provide support for applications that go beyond this simplest form of master-worker, to enable a hierarchical model where jobs are able to create subjobs which are potentially executed on another processor.

So the challenge was to create a programming model that is simple and fits within these bounds, but also enables programmers to exploit parallelism and express applications that are more than embarrassingly parallel. My approach was to simplify programming by making the G2 programming model analogous to an existing and simple programming model, thereby exploiting programmer familiarity, as well as its inherent simplicity. As it turns out, the ASP.NET web services programming model suited this purpose.

As the G2 programming model is analogous to that of ASP.NET web services, I will start with a brief discussion on the ASP.NET model, before describing the G2 model.
3.1.1 XML Web Services

A [web] service is generally implemented as a coarse-grained, discoverable software entity that exists as a single instance and interacts with applications and other services through a loosely coupled (often asynchronous), message-based communication model[39].

A web service exposes its functionality through a standard protocol, typically SOAP. SOAP works with existing web standards and protocols, such as XML, HTTP and TCP/IP, and this allows programs written using different languages using different platforms to communicate in a standards based way [40].

3.1.2 ASP.NET Web Services Programming Model

A web service implemented using ASP.NET consists of a software component, hosted by a web server, Internet Information Services (IIS), which exposes a set of methods that can be invoked remotely over HTTP. A web service is implemented as a class with one or more “web methods”- methods exposed to invocation by remote processes. A web method is implemented by annotating an ordinary method with the WebMethod attribute as illustrated in Listing 3-1.

```csharp
public class MathService
{
    [WebMethod]
    public int Add(int x, int y)
    {
        return x + y;
    }

    ...
}
```

Listing 3-1: Classic web service method that adds two numbers

A new instance of the web service class is instantiated for every web method invocation. The instance exists only for the duration of the method invocation, and any changes to the internal state of the instance do not persist between web method calls. So my view of web methods, which is reflected in the programming model of G2, is that web methods are analogous to procedures, and ASP.NET web services provide an RPC mechanism for the web.
Invoking Web Methods

A web service exposes a service description (typically provided as an XML document called a web services description language (WSDL) document), which provides consumers of the web service with all of the information they need to use the service. The web service is accessed through a URL, and web methods are invoked by sending a HTTP request to the URL with the appropriate headers and SOAP encoded message body. To simplify this process for programmers, there are tools that parse the WSDL and generate a proxy class that has public methods with names and signatures that correspond to the web methods exposed by the web service. Programmers can invoke remote web methods by invoking the methods on a local instance of the proxy class. If a web service exposes a method called, Add, to invoke this web method, all the programmer has to do is to instantiate a proxy, and invoke its Add method with the appropriate integer parameters.

Internally, and invisible to programmers, the proxy creates a SOAP encoded message which is sent to the server via HTTP. The server receives and decodes this message, and then invokes the appropriate web method. The results of this method are encoded, again using SOAP, and returned to the proxy as the reply to the original HTTP request. The proxy decodes the results and returns them as the results of the method call.

The basic mechanism described thus far is the invocation of a web method synchronously on the client side. For each web method, the proxy also generates a pair of methods that are used by clients to invoke the web method asynchronously. These methods are named by prefixing the words Begin and End to the ordinary name of the web method. So, for the Add method described in Listing 3-1, the tools that generate the proxy provide it with a BeginAdd method, and an EndAdd method. The use of these methods for asynchronous invocation is demonstrated in Listing 3-2.
// create proxy
private myproxy = new MathProxy();

// synchronous invocation
result = myproxy.Add(x, y);

// asynchronous invocation. Null params explained later
handle = myproxy.BeginAdd(x, y, null, null);

...  
// block until result is available
handle.WaitHandle.WaitOne();

// get result
finish(handle);

void finish (IAsyncResult ar)
{
  actual_result = myproxy.EndAdd(ar);
  ...
}

Listing 3-2: Asynchronous web method invocation

After using the BeginAdd method to invoke the Add web method asynchronously, programmers have two available options to obtain the result of Add. The first is illustrated in Listing 3-2. The call to BeginAdd immediately returns a handle, known as an asynchronous result handle, which is a handle to the result of the Add web method call. An asynchronous result handle also exposes a synchronization object, called a wait handle, which the caller uses in this example to block until the Add web method is executed. When the caller resumes, the other proxy generated method, EndAdd, is used to obtain the result.

The alternate way that programmers can use to receive the result is by providing a callback. This way does not involve blocking, so the caller is free to perform other work while the result is computed. The .NET runtime will invoke the callback when the result is available. In Listing 3-3, the call to BeginAdd includes a callback, or more specifically, an asynchronous callback delegate. A .NET delegate is similar to a function pointer in C, and points to the method to be executed when the result is available.
// create proxy
private myproxy = new MathProxy();

// synchronous invocation
result = myproxy.Add(x, y);

// some state information
object stateInfo = 1;

// asynchronous invocation
myproxy.BeginAdd(x, y, new AsyncCallback(finish), stateInfo);

... // do other work without blocking
...

void finish (IAsyncResult ar)
{
    actual_result = myproxy.EndAdd(ar);
    object stateInfo = ar.AsyncState;
    ...
}

Listing 3-3: Asynchronous web method invocation with callback

Again, the EndAdd method is used to ultimately retrieve the result using the asynchronous result handle. Along with the callback, some state information can be provided, in this case the stateInfo object, which can be recovered when the callback is invoked. The state information can be used to store metadata about the result for example. In Listing 3-2, a callback and state information were not needed, hence the call to BeginAdd used null values as arguments in place.

3.1.3 Parallel Programming Using the Web Services Model

Consider an application that consumes a web service. The application as a whole, logically consists of two parts. One part is the client application, the other part is the remote web service. The types of applications that are executed by G2 can also be seen to consist of two parts. One part is the client application, the other part runs remotely and in parallel on multiple volunteers. I exploit this similarity by making G2 application programming analogous to ASP.NET web service programming. The parts of a G2 application that run in parallel on volunteers, are written as if they were a web service, but when deployed are not a true web services. When invoked, these pseudo web services execute as jobs on volunteers.
The analogy with web services is maintained when these pseudo web services are actually invoked. Web methods are invoked through a locally instantiated proxy, which hides the remote nature of the web service from the method caller. I leverage this model of web method invocation to simplify the job creation mechanism for programmers.

**G2 Application Programming**

To write a G2 application, a programmer decomposes their problem by deciding which parts of the application will execute in parallel on volunteers. The logic for these parts is then encapsulated into web methods, which make up a pseudo web service. Note that the way in which these web services are written is identical to the way in which an ordinary web service is written. As I have shown, an ASP.NET web service consists of a class with methods annotated with the `[WebMethod]` attribute. Therefore, development of a G2 application may be as simple as adding these annotations, or by creating web method wrappers to existing code. So writing a G2 application may not involve starting from scratch. If new code is to be written, one of the advantages gained from using the .NET framework is that programmers can write in any language that has a compiler that targets the .NET runtime as illustrated in Listing 3-4.

```vbnet
Public Class MathService

    <WebMethod()>
    Public Function Add(ByVal i As Integer, ByVal j As Integer) As Integer
        Return i + j
    End Function

End Class
```

**Listing 3-4: The Add web method in Visual Basic. This will execute as a G2 job as the G2 framework supports and language with a compiler targeting the .NET runtime.**

As I mentioned earlier, the use of web services can be simplified through the use of tools. For example, the .NET framework comes with a tool that parses the WSDL document exposed by a web service, and produces source code for a remote proxy [5]. Programmers can incorporate the remote proxy into their appli-
cations and invoke web methods through it. The proxy sends the call to the web service for remove invocation, and receives the result and returns it to the original caller. Similarly, I have developed a tool that provides programmers of G2 applications with machine generated proxy classes that work in the same way as their web service invoking counterparts. To the programmer the use of a G2 proxy is identical to a web service proxy.

```vbnet
Function UseAddService(ByVal i As Integer, ByVal j As Integer) As Integer
    'Create proxy
    Dim proxy As localhost.MathService = New localhost.MathService

    'Invoke Web service / Start G2 job
    Dim ar As IAsyncResult = proxy.BeginAdd(i, j, Nothing, Nothing)
    ar.AsyncWaitHandle.WaitOne()
    Return proxy.EndAdd(ar)
End Function
```

Listing 3-5: The code to create a G2 job is identical to web method invocation. Shown here in VB.NET

The difference between standard web service proxies and G2 proxies is that G2 proxies are not surrogates for web methods implemented and executed on a server. G2 web methods are queued on the server, and executed on volunteers. As an aside, I have used “real” web services as the communication mechanism between the server, clients and volunteers. So, when a method is invoked on a G2 proxy, it serializes the method name and input parameters into an XML encoded string, and uses a web service hosted on a G2 server to submit a new job. A real web service proxy is used within a G2 proxy to interact with the server.

By invoking G2 proxy methods asynchronously, in a manner identical to the asynchronous invocation of standard web methods, a number of jobs can be created and executed in parallel. In effect, G2 presents a web service façade to a cycle stealing system. The remote proxy enables programmers to create jobs indirectly, by invoking methods with names and signatures that are application meaningful, instead of explicitly creating jobs, and programming jobs with generic interfaces with methods such as “Run” [35]. Writing master-worker style
applications using G2’s programming model is simply a matter of writing an application that invokes methods via a proxy.

### 3.1.4 Beyond Master-Worker

A consequence of cycle stealing is that the set of machines used to execute an application may change during the course of the computation. Individual machines can join and leave the computation, often abruptly, making it difficult for volunteers to maintain communication channels with one another and impossible for a machine to depend on the existence of a particular volunteer. In an Internet setting, volunteers may not be able to communicate with one another due to the presence of firewalls. It is for this reason that existing cycle stealing frameworks, especially those supporting use over the Internet, require programmers to adopt the master-worker programming paradigm. This paradigm lends itself well to cycle stealing environments as it permits workers to perform jobs completely in isolation, after receiving them from the master process.

However, not all problems can be expressed cleanly as master-worker type applications, and master-worker is limiting from an expressivity point of view, as it requires the master to expose all of the parallelism that the application hopes to exploit. To a caller, a function should be a unit of abstraction, requiring a set of arguments to be provided, and should itself produce a result consistent with its specification. The actual implementation of the function should be irrelevant to the caller. Specifically, whether the implementation of the function exploits some parallelism inherent to the function, or the parallel processing capabilities of the underlying platform, should be irrelevant, and therefore transparent to the caller.

As jobs in G2 correspond to function evaluations, it follows that a job running on a volunteer should have the capability to create new jobs (or subjobs), and thereby evaluate in parallel. In other words, job creation should not be exclusive to the clients, or the master process. A corollary is that if a job internally creates multiple subjobs, it should be its responsibility to retrieve the results of those sub-
jobs and process them into a result to be returned, thus hiding its internal implementation from its caller.

This more general programming model requires jobs to be able to create subjobs, and the results of those subjobs to be collected and combined into a result for the original job. Due to the simplicity of the basic programming model, enabling jobs to create subjobs is relatively straightforward. A job running on a volunteer mimics the job creation model used by the master; that is, it creates an instance of a proxy and invokes methods on it. The complexity lies in the mechanism that controls the collection and aggregation of subresults.

### 3.1.4.1 Issues raised by jobs creating jobs

**Non-Blocking Jobs**

In a typical master-worker program, the master process generates a collection of jobs and then blocks until the results of those jobs become available. This presents no problem as the single master executes on the client computer as a regular user process. As such, its execution does not depend on the idleness of the host machine; it does not need to complete execution within a limited period and may continue to execute alongside other local processes.

This approach is not appropriate for volunteers because, like many other similar systems, G2 has a simple fault recovery mechanism whereby if a job is interrupted, due to a volunteer somehow disengaging from the cycle stealing network, it is simply re-executed from its beginning on some other volunteer. Such a simple mechanism is possible because the execution of a job does not have any effect on the state of the application until the job has completed. Job execution is idempotent, and therefore, no rollback mechanisms, or other restoration of state is required. This simple fault recover mechanism encourages programmers to create relatively short-lived jobs; this increases the probability that execution will complete within the idle period of a volunteer, and minimises the amount of computation time wasted if a job is interrupted and restarted.
So, if a job executing on a donated computer were to wait for the results of subjobs, it may find itself blocking for a considerable period of time – until the remaining volunteers (if any) find time to process each of its subjobs. This of course violates the assumption that each job is relatively short lived.

My solution is based on the premise that a process that creates jobs has two active phases. The first phase includes the creation of jobs; the second phase includes the collection of results. These two phases are separated by a period of waiting while the jobs are executed elsewhere. The problem with jobs creating subjobs is that the intermediate waiting period (while the subjobs are executed), is unbounded and impossible to control. The approach I have adopted to enable the creation of subjobs, is to split the logical processes of creating subjobs and recovering results into two physical jobs. The job that creates subjobs terminates instead of waiting for results. The job that recovers the results runs only when the results have been computed, thereby avoiding any need to wait. This approach is inspired by continuation passing style programming and Cilk [18].

**Preventing Redundant Subjobs**

As mentioned earlier, I have adopted a simple fault recovery mechanism for G2 whereby jobs are restarted if they fail to execute to completion. This mechanism assumes that job execution is idempotent. However, the creation of a subjob changes the distributed state of an application, which would appear to invalidate the assumption. Specifically, if partially executed jobs can create subjobs, and those subjobs persist regardless of the successful execution of the parent job, then the simple fault recovery mechanism can hinder application execution. The mechanism will not produce errors, and applications will still produce a correct result, but they will do so less efficiently. The creation of subjobs by jobs, has the potential for producing identical subjobs, or orphaned subjobs, if the job that creates those subjobs is executed, or partially executed more than once. If these redundant subjobs are subsequently executed, the overall efficiency of the G2 sys-
tem will suffer. The problem increases exponentially if these orphaned subjobs themselves create subjobs.

The problem is one of ensuring that redundant subjobs are not executed, which I achieve by binding the possibility of execution of a subjob, to the completion of the job that created it (its parent job). The mechanism I use is that of transactions. Recall that a job that creates subjobs consists of two logical parts that are embodied in two physical jobs. The first of these parts creates all of the subjobs. As this part executes, it opens a transaction, submits subjobs to the server, and when it finishes execution, the server commits the transaction. Only after the transaction has been committed will these new subjobs be made available for execution on volunteers.

A common fault tolerance mechanism for cycle stealing systems is to redundantly assign the same job to multiple volunteers. I have adopted this approach and so it is sometimes possible for multiple volunteers to execute the same job simultaneously. The implication is that if that job is a parent job, then redundant sets of subjobs are created and introduced into the system. This is undesirable, so each execution of that parent job is treated as a separate execution by the server; that is, each set of subjobs is stored independently. The server retains the set of subjobs created by the first parent job to complete (and commit the transaction), and discards all others. This scheme ensures that all of the subjobs originate from the same execution of a job, which is significant if the number or function of the subjobs created by a job is nondeterministic.

**Subjob mechanism**

As mentioned earlier, if programmers wish to exploit some internal parallelism within a G2 pseudo web method, they implement that web method in two parts.
The first part creates new jobs in the same way that a new job is created by clients. Somewhere within the implementation of the first part, a proxy is instantiated and its methods are invoked. On instantiation, a proxy is able to detect whether it is running on a client or volunteer, and this information is relayed to the server when the proxy submits the new job to the server. If the job originated from a volunteer, then it is a subjob, and the server will not release it for execution on other volunteers until the job that created the subjob is known to have completed. This is different from jobs that originate from clients, as these are immediately made available for execution on volunteers.

The second part runs when all of the subjobs have been executed, and it combines the results of these subjobs into a result that is returned to the creator of the original job. These parts follow exactly the standard pattern used for implementing ASP.NET web services that execute asynchronously on the server side.

**Asynchronous Web Services in ASP.NET**

Previously, I had discussed the asynchronous *invocation* of web methods. This asynchronous invocation occurs on the client side, and enables the client to continue other processing while waiting for results from a web method call. ASP.NET web services provides a mechanism whereby web methods are *executed* asynchronously on the server side. This functionality is described in this section.
A web server allocates resources to handle each invocation of a web method. For example, IIS uses one of a limited number of worker threads to handle each web request. If the invocation of a web method relies on the use of other servers and services external to IIS, then the asynchronous execution of web methods enables worker thread resources to be freed while the external processing occurs. This is useful in many situations, such as invoking other web services in a service composition scenario, or the execution of complex database queries.

It is important to note that the asynchronous execution of a web method is completely orthogonal to whether the client chooses to invoke that web method synchronously or asynchronously. In other words, web methods that are implemented to execute asynchronously can be invoked by the client either synchronously or asynchronously. This is no different to web methods implemented for synchronous execution. Whether or not a web method executes synchronously or asynchronously is purely an implementation detail and is transparent to the caller.

Web methods in ASP.NET can be implemented to execute asynchronously by dividing their implementation into two parts. The first part is presented as a standard web method implementation but with the word Begin prefixed to the web method’s name as known to the client. This Begin method takes the same arguments as the original web method plus two additional arguments, a callback delegate and a state object. Rather than returning the actual result, the Begin method returns an asynchronous result handle. The second part of the method is presented as a web method implementation with the word End prefixed to the web method’s name as known to the client. The End method takes the asynchronous result handle returned by the Begin method and returns an actual result. Although the web service implementer is required to code two methods, they should be thought of as two parts of a single web method implementation.
In Listing 3–6, the web service implementer writes a `BeginFoo` and an `EndFoo` method. However, the method exposed, and thus visible to users of the web service, is called `Foo`. The signature of `Foo`, is a method that takes `someArg` as a parameter and returns a string. In other words, the two-part implementation for asynchronous execution is hidden to its caller.

The `AsyncCallback` parameter to the `Begin` method is supplied by the .NET framework and is a callback to the corresponding End method. This callback is typically invoked by some server-side, third-party controlling process when that process determines that the information needed to complete the method invocation, and hence invoke the End method, has become available. The `Begin` method returns an asynchronous result handle, and this handle is the input parameter to the `End` method. Any data that is required by the `End` method that is created or accessible to the `Begin` method can be stored and transferred using this handle.

---

2 The .NET supplied callback has a dual purpose. It frees the programmer from having to manually tie the `End` method to the `Begin` method, and more importantly, it provides a means to tie the `Begin` and `End` parts, together with the session information of the original caller of the web method. The callback does not directly point to the programmer supplied `End` method. Instead, the target of the web method is a method called “Callback” which is a member of the `System.Web.Services.Protocols.AsyncSessionlessHandler` class. This class is a subclass of the `SyncSessionlessHandler` class, which in turn is a subclass of the `WebServiceHandler` class. The `Callback` method, which is inherited from the `WebServiceHandler` class, calls a method called `DoCallback`, which is what triggers the execution of the programmer supplied `End` method.
Adapting Asynchronous Method Implementation to Create Subjobs

G2 web methods that create subjobs are implemented in a manner analogous to the implementation of ASP.NET web methods that execute asynchronously. The G2 server handles the two parts of an asynchronously executing web method, the Begin part and the End part, as two separate but linked jobs. The creation of subjobs is performed during the execution of the Begin method – by instantiating and invoking methods on a proxy. The invocation of these methods each returns an asynchronous result handle, which are used during the execution of the End method to retrieve the results of the subjobs. To pass these handles to the End method, the handles are incorporated into the result handle (IAsyncResult) that is returned as the result of the Begin method.

The execution of the End method occurs when the G2 server determines that each subjob has been completed. The End method will be made available and fetched by a volunteer. The parameter to this End method is the result handle that was initially returned by the Begin method, and it encapsulates the computed results from the subjobs. The subresults can be extracted from this result handle and processed. The return value of the End job is returned as the result of the two-part web method.

```csharp
[WebMethod]
public IAsyncResult BeginFib(int i, AsyncCallback cb, object state)
{
    if (i <= 2)
    {
        return new G2AsyncResult(i);
    }
    else
    {
        G2AsyncResult ar1 = proxy.BeginFib(i-1,null,null);
        G2AsyncResult ar2 = proxy.BeginFib(i-2,null,null);
        return new G2AsyncResult(ar1, ar2);
    }
}
[WebMethod]
public int EndFib(G2AsyncResult ar)
{
    if (ar.CompletedSynchronously)
    {
        return (int)ar.result;
    }
    else
    {
        return proxy.EndFib(ar.handle[0])+proxy.EndFib(ar.handle[1]);
    }
}
```

Listing 3-7: Recursive Fibonacci
Listing 3-7 above shows the implementation of a two-part web method that uses recursion to calculate the \(i^{th}\) Fibonacci number. There are a few things to note about this listing. Asynchronous invocation of a web method, through the use of a Begin method, always immediately returns a G2AsyncResult object. The G2AsyncResult class implements a standard .NET class library interface called IAsyncResult. This interface has a Boolean property called CompletedSynchronously, which indicates whether an asynchronous invocation of a method resulted in the actual asynchronous execution of that method, or whether the method executed synchronously. In the context of G2, the CompletedSynchronously flag indicates whether a new job was actually created. This recursive Fibonacci example illustrates how this may be useful. The first two Fibonacci numbers are both 1, hence to calculate the first or second Fibonacci number it is more efficient to simply return 1, rather than to create a new job, which will evaluate to 1\(^3\). However, the return type of a Begin method is IAsyncResult. I have used the CompletedSynchronously property, and overloaded constructors to the G2AsyncResult object to enable programmers to obtain the result.

If a G2AsyncResult object is constructed with a parameter of any type other than a variable length parameter array of type G2AsyncResult, that parameter is assumed to be the result of a synchronous invocation, and the result is stored and the CompletedSynchronously property is set to true. This occurs on line 5. If the first or second numbers in the Fibonacci sequence are requested, then no further processing is required. The G2 framework detects that the method was completed synchronously, and the volunteer will immediately execute the EndFib web method. The programmer can check for this condition and extract the result, shown in lines 18 and 19.

If the requested Fibonacci number is not one of the first two numbers in the sequence, then two new jobs that return the two preceding Fibonacci numbers are

\[3 \text{ This simple program will not calculate the } 0^{th} \text{ Fibonacci number which is 0.}\]
created in lines 8 and 9. Job creation by volunteers is the same as by clients, by invoking a method of the proxy. (The instantiation of the proxy is not shown here.\(^4\))

It is important to recall that the proxy exposes methods that enable the *invocation* of Fib to be synchronous or asynchronous, hence `proxy.BeginFib` and `proxy.EndFib`. The methods exposed by the proxy are different to the `BeginFib` and `EndFib` methods that make up the web method implementation. If the caller of the `Fib` method desires synchronous invocation, then they can call `proxy.Fib`. If the above Fibonacci function was implemented as part of a real web service, the web method exposed by that service would be called `Fib`, despite the actual implementation for asynchronous *execution* in two parts, `BeginFib` and `EndFib`.

Asynchronous invocation through `proxy.BeginFib` returns a `G2AsyncResult` object, which encapsulates handles to the results of subjobs created during the execution of the `BeginFib` job. All `Begin` jobs return a `G2AsyncResult` and so one is created in Line 11, containing handles to the yet-to-be-computed results of the subjobs created. Note that the constructor takes a variable length argument list of `G2AsyncResult` objects. A `G2AsyncResult` constructed with this list holds handles to subjobs and so the value of its `CompletedSynchronously` property is false.

Once the sub jobs created in lines 8 and 9 have been executed and results computed, the `EndFib` part of the web method can be invoked. This `End` part is not directly invoked from user code; the G2 framework and the server are responsible for creating the appropriate `EndFib` job at the appropriate time, leading to its execution on some volunteer. As previously stated, the `G2AsyncResult` parameter of the `EndFib` method encapsulates handles to the results of the sub-

\(^4\) As the proxy is used in multiple methods, a programmer may prefer to instantiate it in the constructor of the containing class. Suppose the proxy is an instance of the class, `FibProxy`, then a proxy is instantiated, in this case using C# syntax, by: `FibProxy proxy = new FibProxy();`
jobs created during the execution of `BeginFib`. If the result was not completed synchronously, then the encapsulated result handles are extracted and used to obtain the computed result. Results of subjobs are extracted by calling the proxy’s `EndFib` method (Line 21), with the appropriate result handle as the parameter. Recall that the standard job creation pattern in G2 is to invoke a proxy’s `Begin` method, and then to invoke the `End` method to retrieve the result. This is what is occurring in Line 21. Again, it should be noted that the proxy’s `End` method is being called here, and not the `EndFib` part of the web method implementation.

![Diagram](image)

**Figure 3-2: A job may be broken down into subjobs, but appears as one job to its caller**

The two phase implementation of web methods that ultimately generate subjobs is used by the G2 framework to prevent the execution of redundant subjobs. When a volunteer has finished executing the `Begin` part of the method, it will submit a job corresponding to the `End` part. The volunteer submits this job automatically; it is not explicitly submitted by programmers. The receipt of the `End` job by the server *commits the transaction*, and signifies that the `Begin` part has successfully executed.
3.1.5 Dataflow

The mechanisms that connect Begin and End jobs, and detect and delete redundant subjobs, all rely on the pattern where subjobs are created only in the Begin part of methods. With this pattern alone, only one tier of subjobs is possible because the results of the subjobs are collected by the End part, and programmers cannot further subdivide the logic of the End part into a set of subjobs. This does not restrict the types of applications that are possible, because the client can always submit more jobs based on the results of prior jobs, but it can be more convenient and efficient if programmers could construct multiple tiers of subjobs.

I have therefore modified the general Begin-End pattern to support a more general mechanism whereby the programmer can express the results of subjobs to be inputs into subsequent subjobs. This allows arbitrary dataflow DAGs (Directed Acyclic Graphs) of subjobs and subresults to be constructed.

As a simple example, suppose there are three methods, Foo, Bar and Baz, where Bar and Baz correspond to subjobs created by the parent job, Foo. The results computed by Bar, are used as inputs to Baz. It is the results of Baz that are ultimately collected by EndFoo and returned to the client. The DAG of this application will be as follows.

![Figure 3-3: DAG of jobs](image-url)
In this example, the execution of the BeginFoo method creates five subjobs. Three of these jobs execute the Bar method. The results of the execution of Bar are inputs to the two Baz methods. It is the results of the two Baz method calls that are processed by the EndFoo method and returned to the original caller of Foo.

```csharp
[WebMethod]
public G2AsyncResult BeginFoo(...) {
    G2AsyncResult a1, a2, a3, b1, b2;
    a1 = proxy.BeginBar(1, null, null);
    a2 = proxy.BeginBar(2, null, null);
    a3 = proxy.BeginBar(3, null, null);
    b1 = proxy.BeginBaz(a1, a2, null, null);
    b2 = proxy.BeginBaz(a2, a3, null, null);
    return new G2AsyncResult(b1, b2);
}

[WebMethod]
public in EndFoo(G2AsyncResult ar) {
    subresult1 = proxy.EndBaz(ar.handle[0]);
    subresult2 = proxy.EndBaz(ar.handle[1]);
    ...
}

[WebMethod]
public int Bar(int i) {
    ...
}

[WebMethod]
    int i1 = proxy.EndBar(ar1);
    int i2 = proxy.EndBar(ar2);
    ...
}
```

**Listing 3-8: Code for simple dataflow example**

In Listing 3-8, three subjobs to execute the Bar method are created in lines 6,7 and 8. Lines 10 and 11 show the creation of subjobs corresponding to the Baz method that takes result handles to the Bar subjobs as parameters. The logic for this particular application is that the Baz subjobs created in lines 10 and 11 should not be dispatched to volunteers and executed until the Bar subjobs have been executed and their results made available. This dependency between the Baz and Bar subjobs is indicated by the programmer by annotating formal parameters of
Baz with the [G2] attribute. In the definition of Baz on line 31, the G2AsyncResult parameters are annotated with the [G2] attribute. These annotated result handles are handles to the results of the execution of the Bar subjobs. This indicates the dependency between Bar and Baz to the framework, and ensures that Baz will not be executed prematurely.

For a practical example, suppose a researcher wants to test a new algorithm for fractal image compression that generates and uses a domain pool consisting of a set of overlapping rectangles of variable dimension\(^5\).

Initially, a job that that runs a method called PIFS is created. The Begin part of this method creates many subjobs. The first subjob is the division of a given input image into a suitable set of domain blocks, that is, the generation of the domain pool. This computed set of blocks is then used as an input parameter into many other subjobs that run the FindPairing method. The FindPairing method tries to find the optimum transformations for a domain block to best resemble a region block.

\(^5\) Fractal image compression uses transforms of a simple base image to describe another much more detailed image. Each image is separated into a set of region blocks, and for each region block, a transformation of the base image is created, such that the transformation applied to the base image produces an image that resembles the region block. The final image is produced by iterating through the set of transforms, using the latest produced image as the base image. Partitioned Iterated Function Systems (PIFS) describe transforms on parts of the base image, called domain blocks, to produce the final image, instead of transforms of the entire base image \cite{41}. The set of domain blocks is known as the domain pool. The domain pool generation algorithm mentioned is hypothetical.
The BeginPIFS method creates a GetDomainPool job, and one or more FindPairing jobs. Two are pictured in Figure 3-4. Each FindPairing job depends on data generated during the execution of the BeginPIFS method, as well as the results of the GetDomainPool method, which is computed as a separate job. Thus, like the Baz method in the previous example, FindPairing jobs have a dependency on the results of another subjob. (Although not shown in Figure 3-4, the FindPairing method is recursive, and can use quadtree partitioning to further divide a job into four subjobs.)

```csharp
[WebMethod]
public G2AsyncResult BeginPIFS(Bitmap image, int rangeWidth)
{
    G2AsyncResult domainPool = proxy.BeginGetDomainPool(image, null, null);
    Bitmap[] RangeBlocks = this.SplitImage(image, rangeWidth);
    G2AsyncResult[] resultHandles = new G2AsyncResult[RangeBlocks.Length];
    for (int i = 0; i < resultHandles.Length; i++)
    {
        Bitmap b = RangeBlocks[i];
        resultHandles[i] = proxy.BeginFindPairing(b, domainPool, null, null);
    }
    return new G2AsyncResult(resultHandles);
}

[WebMethod]
public Transform[] EndPIFS(G2AsyncResult ar)
{
    ...
}

[WebMethod]
public DomainBlock[] GetDomainPool(Bitmap image)
{
    //Find Domain Set
    ...
}
```
The first job runs the `BeginPIFS` method. The execution of this job results in the submission of a new subjob to find a set of suitable domain blocks. (Line 4). The result of this subjob, is to be used as input to a multiple number of subjobs that find a suitable pairing between a transformed domain block and a region on the image to be compressed. Each of these subjobs is created in Line 12, and the set of result handles from these subjobs are returned by the `BeginPIFS` method. Only after `BeginPIFS` completes will the server assign the subjob corresponding to the `GetDomainPool` method (created on line 4) to a volunteer. The server however, will not assign any subjob corresponding to the `FindPairing` method (created on line 12). This is because the `FindPairing` method depends on the results of the `GetDomainPool` method, and hence cannot be executed until `GetDomainPool` has produced a result. Note the [G2] annotation on the formal parameters of `BeginFindPairing` on Line 31. Without the attribute, the server would be free to assign the `BeginFindPairing` jobs to volunteers, as soon as the job corresponding to `BeginPIFS` completed execution. In which case the `G2AsyncResult` parameter would be treated like any other parameter, and be a handle to a non-executed job. There may be cases where the programmer desires this behaviour, but it would be incorrect in this case.
3.1.6 Type-safe Dataflow

A problem with the mechanism described so far is that it is the responsibility of successor jobs to extract the results from predecessor job result handles. To do so, the successor needs to know which method generated the asynchronous result handle in order to know which End method to use to extract the result. This coupling between function implementations itself breaks the principle of abstraction.

```csharp
{
    DomainBlock[] domainPool = (DomainBlock[])proxy.EndGetDomainPool(pool);
    ...
}
```

Listing 3-10: Undesirable coupling between method implementations. To extract the result encapsulated in the G2AsyncResult object (called “pool”), a call to EndGetDomainPool is needed.

This coupling is illustrated in Listing 3-10, where the code in the BeginFindPairing method makes explicit reference to the GetDomainPool method.

Therefore, I have incorporated an additional mechanism that passes results to successor methods as arguments of the specified type. To make use of this mechanism, programmers again annotate formal parameters of the successor method, but this time they specify the type of the input parameters to be of the expected incoming type, rather than the generic G2AsyncResult type.

```csharp
public G2AsyncResult BeginFindPairing(Bitmap image, [G2]DomainBlock[] domainPool)
{
    ...
}
```

Listing 3-11: Example of Type Safe Data flow method signature

In Listing 3-10, the BeginFindPairings method accepted a generic result parameter of type G2AsyncResult. The programmer then had to use the appropriate proxy method to extract the result and cast it to the expected type. Not only does this introduce an undesirable coupling between method implementations, the onus falls on the programmer to ensure that the type of the result encapsulated within the result handle is of the type that they expect. Using the type safe mechanism, demonstrated in Listing 3-11, the G2 framework volunteer component takes care of converting the result data received from the server into the
type expected by the BeginFindPairing method. This can fail if the GetDomainPool method does not return a result of the type expected by BeginFindPairing, but the framework guarantees that if the BeginFindPairing method is executed, the input parameters are of the type expected. Therefore, programmers do not have to perform their own runtime checks.

The [G2] attribute still performs the same role, in that it signifies to the framework that the annotated parameter is the result of a subjob, and BeginFindPairing jobs will not be executed until the result from the GetDomainPool subjob is available.

Referring back to Lines 4 and 12 in the BeginPIFS method, excerpted in Listing 3-12, the type of domainPool is G2AsyncResult, and yet the new signature of BeginFindPairing expects an array of type DomainBlock.

<table>
<thead>
<tr>
<th>Line</th>
<th>Code</th>
</tr>
</thead>
<tbody>
<tr>
<td>04:</td>
<td>G2AsyncResult domainPool = proxy.BeginGetDomainPool(image);</td>
</tr>
<tr>
<td>05:</td>
<td></td>
</tr>
<tr>
<td>06:</td>
<td>Bitmap[] RangeBlocks = this.SplitImage(image, rangeWidth);</td>
</tr>
<tr>
<td>07:</td>
<td>G2AsyncResult[] resultHandles = new G2AsyncResult[RangeBlocks.Length];</td>
</tr>
<tr>
<td>08:</td>
<td></td>
</tr>
<tr>
<td>09:</td>
<td>for (int i=0; i&lt;resultHandles.Length; i++)</td>
</tr>
<tr>
<td>10:</td>
<td>{</td>
</tr>
<tr>
<td>11:</td>
<td>Bitmap b = RangeBlocks[i];</td>
</tr>
<tr>
<td>12:</td>
<td>resultHandles[i] = proxy.BeginFindPairing(b, domainPool);</td>
</tr>
</tbody>
</table>

Listing 3-12: The variable domainPool is declared to be of type G2AsyncResult. But BeginFindPairing expects a parameter of type DomainBlock[].

This inconsistency, (which leads to a compiler error) is resolved by overloading the BeginFindPairing method on the proxy. When the proxy is generated, overloaded BeginFindPairing methods are be created.

---

* Like normal job parameters, input parameters that are results of subjobs arrive on the volunteer as XML encoded strings. The expected parameter types of methods are obtained through reflection and the framework deserializes the encoded strings into the correct type.
public G2AsyncResult BeginFindPairings(Bitmap arg1, G2AsyncResult arg2, AsyncCallback cb, object state)
{
    //stub only, no body
}

public G2AsyncResult BeginFindPairings(Bitmap arg1, DomainBlock[] arg2, AsyncCallback cb, object state)
{
    ...
}

Listing 3-13: The two automatically generated BeginFindPairings methods in the proxy

The first method in Listing 3-13 is never called and is a stub only. The second method is the one that is actually called. Note the callback and state parameters which are not used, (null is passed) in the example code.

3.1.7 Propagating Asynchronous Result Handles

A G2 job encapsulates a function evaluation. The function specification indicates the result or results that a function may return, and this result does not reflect the way in which the function was evaluated. For example, if the function is recursive, then the form of the result should not depend on whether the function was evaluated recursively or not. Similarly, if a G2 job creates subjobs, then this fact is hidden from the creator of the original job. The results of those subjobs are typically amalgamated into a single result and then returned to the original job creator. This behaviour is all designed to preserve the abstraction provided by functions – the creators of jobs aren’t interested in how those jobs are executed, just the results. However, there are circumstances where the programmer desires to break the abstraction barrier, and be exposed to the existence of subjobs. I have made this possible and the mechanism relies on the propagation of asynchronous result handles.

Consider again the image compression example. After all of the subjobs created in the BeginPIFS part have been executed and their results computed, the job corresponding to the EndPIFS method can be assigned to a volunteer where it is executed.
The **EndPIFS** method in Listing 3-14 simply combines the set of transforms computed by the subjobs into a single array. Instead of executing this as a job on a volunteer, it would be more efficient if the set of result handles from invocations of **proxy.BeginFindPairings**, which were created during the execution of **BeginFIPS**, were all returned directly to the master process. The master could then use those handles to retrieve the transforms as they were computed.

This, more efficient execution method is realised by having jobs return result handles as the result of their execution, as shown in Listing 3-15.

As the PIFS method is not needed to be implemented in two parts, there is no need to keep the Begin prefix. As with methods implemented in two parts, the subjobs created will not be made available to volunteers until this PIFS method has completed and the server receives the result, which is an array of result handles.

Like any ordinary result, the G2 client component will retrieve these handles from the server as the result of the PIFS job, and they will be made available to
the client side part of the user application. The problem is that the client part of
the user application now has a set of result handles to jobs that it did not directly
create. For the G2 client component to retrieve the results of these subjobs, it
needs to be registered on the server as a “listener” to the results of the subjobs.
For this to occur, programmers do not have to do anything that deviates from or-
dinary asynchronous programming using the .NET framework.

Registering as a Listener

When a G2 client first connects to a G2 server, it receives a unique client identi-
fier, which is used by the G2 framework to match jobs with their owners. This
identifier is submitted by the client each time it submits a new job, and whenever
it attempts to fetch results. In G2, the entity that is interested in the result of a job
is known as that job’s Listener.

If a client submits a job, then that client becomes a listener for that job. When a
volunteer submits a subjob, then another subjob can become the listener, for ex-
ample the job corresponding to the End method. A job does not always have a lis-
tener. The subjobs created in the PIFS method above is an example. As there is no
End method, the FindPairings jobs do not have listeners.

A G2 Client receives result handles for subjobs as a result of the execution of a job
as illustrated by the PIFS method. To the G2 client, a result is what is produced
by job execution, and it does not differentiate between types of results. It does not
know that the result of the PIFS method is a set of result handles to subjobs. For
the G2 client to retrieve these results, two pieces of information are required.
First, the user application has to inform the G2 client component that it is inter-
ested in the result corresponding to the result handle. Second, the G2 client needs
to know what to do with the result after it is retrieved.

Both pieces of information are provided by the programmer when their applica-
tion registers itself as the listener for those jobs. As part of the registration proc-
ess, the client component informs the server that it is now listening to those jobs,
and the results will be retrieved as part of its normal Fetch results mechanism. The registration process is initiated by the programmer in one of two ways.

Firstly, the user application (the master) can block on the wait handle exposed by a result handle. This wait handle is part of the .NET IAsyncResult interface which is implemented by G2AsyncResult. Thus, this method is a standard way to wait for the completion of an asynchronous method invocation and is shown in Listing 3-16.

```csharp
G2AsyncResult[] arList = proxy.EndFIPS(ar);
arList[1].WaitHandle.WaitOne();
...
```

Listing 3-16: Block and Wait

If `WaitOne` is called on the wait handle, the client, through its proxy will add the result handle to a list of pending results maintained by the fetch results mechanism. The G2 client proxy will then register the client as a listener to that subjob. The result of the subjob will be retrieved if available during the next fetch results cycle. The client also knows what to do when the result is retrieved. It signals the wait handle, and any thread blocking on it will be released.

The `WaitOne` method applies to one result handle and may not be appropriate if many subjob result handles are retrieved. The .NET framework provides static functions, `WaitAny` and `WaitAll`, that take an array of WaitHandles as input. These static methods can be used when there are many result handles, but they, like `WaitOne` are still blocking calls. So I have provided a second, non-blocking registration method, shown in Listing 3-17, where the client can use the asynchronous result handle to register itself as a listener to a job and provide a callback delegate to be invoked when the result is returned.

```csharp
G2AsyncResult[] arList = proxy.EndFIPS(ar);
ar.RegisterCallback(new AsyncCallback(this.ProcessResult, null);
...
```

Listing 3-17: Registering as a Listener
This second mechanism breaks from the standard .NET asynchronous programming model, because I had to introduce a new method to the result handle. However, it does provide a means to retrieve the result without blocking, and enables the user application to do useful work while waiting for subresults.

3.1.8 Data Parallel Programming in G2

As programmers write G2 applications, they might find themselves relying on common execution patterns, or rewriting common fragments of code. The G2 framework can be extended with new constructs that provide new abstractions for these coding and execution patterns. The constructs provide a façade to the existing job creation mechanism, and their implementation should require little or no changes to the framework. As an example, I have implemented two alternate constructs to facilitate data parallel programming, requiring minor changes only to the automatic proxy generator. These constructs are built on top of the existing framework, so they do not provide additional functionality, but they provide better abstractions for this style of programming.

If a programmer writes a data parallel application, then they will typically have some collection of data elements, each of which is to be processed in the same way. Their application will iterate over the elements of the collection, creating a new job for each element.
Listing 3-18: The client part of a G2 application that renders the Mandelbrot set. The user specifies the range and the desired dimensions of the output image

Listing 3-18 shows a Mandelbrot rendering example where the region of interest to the user is divided up into a set of subregions, which are stored in a collection called regions. The actual renderer takes one of these subregions and returns a bitmap. When all of the sub-image results have been computed, they are obtained by the client and pieced together to create the final image. Notice that the regions collection is used twice in the code. It is initially used to specify the subregion to render, and then used to establish the location of the rendered subimage when piecing together the final image.

Sub-images are rendered as the result of jobs, which are created by invoking a proxy method using the elements of the regions collection as parameters. Each invocation returns a result handle, which is stored in an array, and each result handle exposes a wait handle, which is stored in a separate array of WaitHandles. This WaitHandles array is used to block until all of the jobs have been completed, and their results retrieved by the client proxy. The proxy is subsequently used to extract the results of the jobs from their corresponding result handles. Although the code is not long, it represents a non-trivial sequence of events that has to be
managed by the programmer. The G2 framework supports extensions and I have created examples that simplify this process. They rely on the .NET asynchronous programming model, specifically, asynchronous method invocation using delegates.

**Asynchronous Programming using Delegates**

Earlier, I demonstrated the asynchronous invocation of web service methods using a proxy. But using the .NET framework, any method can be asynchronously invoked through the use of a delegate. Delegates in the .NET framework are data structures with references to a static method, or a method of an instance of a class. They can be thought of as function pointers with some additional data. Delegates are instantiated with the name of the method to which they are bound.

When a programmer declares a delegate type, the compiler adds two methods, BeginInvoke and EndInvoke as public methods of that type. These methods enable asynchronous invocation of the delegate. As with asynchronous programming using proxies, there are two options for asynchronous invocation, which I repeat here:

Firstly, the BeginInvoke method takes the method parameters as well as an optional Callback method, and a state object. This callback is also a delegate. The BeginInvoke method returns a handle to the result of the method to which the delegate is bound. This result handle is of type IAsyncResult. If a callback is supplied, the callback method will be invoked after the delegate has actually been completed. This callback method takes a single IAsyncResult as its only parameter.

Alternatively, using the second invocation method, the caller can block on the result handle returned by the BeginInvoke method. The caller unblocks when the method to which the delegate is bound has been executed.

To obtain the result of the method invoked via the delegate, programmers invoke the delegate’s EndInvoke method, with the result handle as the only parameter. If
BeginInvoke was called with a callback, the callback is invoked when the result becomes available.

Listing 3-19 below illustrates a common scenario where the original delegate, which was used to begin asynchronous invocation, is not in scope in the body of the callback method. Without the original delegate, no EndInvoke method can be called. Therefore, the .NET framework embeds the original delegate in the result handle. The IAsyncResult returned from the BeginInvoke method, is actually of type AsyncResult (which implements the IAsyncResult interface). The AsyncResult type has a data member called AsyncDelegate, which is used to obtain the original delegate.

The Listing 3-19 illustrates the use of a delegate.

```csharp
// The method being invoked through the delegate
private int Add(int i, int j)
{
    int result;

    // declare delegate type
    private delegate int AddDelegate(int i, int j);

    // invoke and block
    public void InvokeOption1(int i, int j)
    {
        // instantiate delegate
        AddDelegate ad = new AddDelegate(this.Add);
        IAsyncResult ar = ad.BeginInvoke(i, j, null, null);

        // block until a result is available
        ar.AsyncWaitHandle.WaitOne();
        result = ad.EndInvoke(ar);
    }

    // invoke with callback
    public void InvokeOption2(int i, int j)
    {
        ad.BeginInvoke(i, j, new AsyncCallback(this.ProcessResult), null);
    }

    private void ProcessResult(IAsyncResult ar)
    {
        // Recover the original delegate
        ad = (AddDelegate)((AsyncResult)ar).AsyncDelegate;
        result = ad.EndInvoke(ar);
    }
```

Listing 3-19: Delegate invocation, illustrating two asynchronous invocation options

Asynchronous programming using delegates is similar to asynchronous web method invocation which was illustrated earlier. The only difference is that web method invocation occurs via a proxy forgoing the need for a delegate.
Constructs for Data Parallel Programming

In the previous Mandelbrot example, the programmer created a series of jobs, all running the same method, using parameters that were stored as elements in a collection. I simplify the programming of applications following this pattern by providing a Parallel Map function that is used for the bulk creation of jobs and result recovery.

Parallel Map

A Map\(^7\) function takes a function that takes some parameter, and applies that function to a collection of parameters, returning a collection of results. Thus, the G2 parallel map function, takes a delegate (to a proxy method), and a collection of parameters. As the invocation of a proxy method results in the creation of a job, using the map function causes a new job to be created for each element in the collection. The map function is also responsible for collecting the results of the jobs when they are executed, and will populate the result collection as results are retrieved. The creation of jobs and the recovery of results are hidden from the programmer, who just uses the map function and receives a collection of results. Referring back to the earlier Mandelbrot example in Listing 3-18, the same application using the Map function is shown in Listing 3-20 and becomes:

\[
\text{map } f \ x s = \{ f \ x \mid x \leftarrow x s \} \quad \text{(Haskell Definition)}
\]

\(^7\) map \( f \ x s = \{ f \ x \mid x \leftarrow x s \} \quad \text{(Haskell Definition)}\)
private delegate Bitmap CalcRegionDelegate(RenderRegion block);

public void Mandelbrot(Point topLeft, Point bottomRight, int width, int height)
{
    Bitmap totalImage = new Bitmap(width, height);
    RenderRegion[] regions = GetRegions(topLeft, bottomRight, width, height);
    CalcRegionDelegate crDel = new CalcRegionDelegate(proxy.CalcRegion);
    object[] results = proxy.Map(crDel, regions);
    for (int i=0; i<blocks.Length, i++)
    {
        Bitmap subImage = (Bitmap)results[i];
        this.PositionBitmap(totalImage, subImage, blocks[i]);
    }
    return totalImage;
}

Listing 3-20: Using the Map construct, the creation of all of the jobs and obtaining the results is reduced to one line.

The parallel map function is a powerful construct that hides job creation and result recovery, reducing them to one call.

Parallel Map Internals

The Map function is a public member of any proxy generated by the G2 proxy generation tool and shown in Listing 3-21. It has two overloads but the general Map function, that takes a delegate and an array of arrays, is shown here. The points of interest in the following code, is the use of reflection to obtain the delegate’s BeginInvoke Method, and the invocation of the BeginInvoke method, with a callback to a private method, EndMap. Each invocation of BeginInvoke creates a job, and the EndMap callback is invoked whenever a result is retrieved. Note that this is the internal implementation of the Map function that I include here for completeness.
public object[] Map(System.MulticastDelegate method, object[][] parameters)
{
    completed = 0;
    results = new object[parameters.Length];

    Type myDel = method.GetType();
    int callParamLength = 0;
    for (int i = 0; i < parameters.Length; i++)
    {
        callParamLength = parameters[i].Length + 2;
        object[] callParams = new object[callParamLength];
        parameters[i].CopyTo(callParams, 0);
        callParams[parameters[i].Length] = new AsyncCallback(this.EndMap);
        callParams[parameters[i].Length + 1] = new CallRecord(i, method);
        mi.Invoke(method, callParams);
    }
    are.WaitOne();
    return results;
}

private void EndMap(IAsyncResult ar)
{
    CallRecord record = (CallRecord)ar.AsyncState;
    int resultNo = record.Number;
    System.MulticastDelegate method = record.Delegate;

    Type mydel = method.GetType();
    results[resultNo] = mi.Invoke(method, new object[]{ar});
    if (++completed == results.Length)
    {
        are.Set();
    }
}

Listing 3-21: Proxy implementation of Map

The EndMap method takes the latest result, and places it in the appropriate place
in the results array. Listing 3-23 shows the implantation of the CallRecord struc-
ture used in Listing 3-21.

The overload in Listing 3-22 was used in the Mandelbrot example in Listing 3-20.
If the delegate is bound to a method that takes a single parameter, then progra-
mers can supply an array of single parameters, instead of an array of parameter
arrays.
```csharp
public object[] Map(System.MulticastDelegate method, object[] parameters)
{
    object[][] arrayOfParams = new object[parameters.Length][];
    object[] param = new object[1];
    for (int i = 0; i < parameters.Length; i++)
    {
        param[0] = parameters[i];
        arrayOfParams[i] = param;
    }
    return Map(method, arrayOfParams);
}
```

Listing 3-22: A Map overload to handle an array of single parameters

```csharp
private struct CallRecord
{
    public int Number;
    public System.MulticastDelegate Delegate;

    public CallRecord(int n, System.MulticastDelegate d)
    {
        Number = n;
        Delegate = d;
    }
}
```

Listing 3-23: The proxy’s callrecord data structure is used to track results, to store the delegate, which is used to extract results, and to record the appropriate insertion point in the results array

### Data Parallelism and Collections of Objects

Standard ASP.NET web services are implemented as classes, exposing web methods as non static public members. A new instance of the class is constructed whenever a web service client invokes a web method; no two web method invocations involve the same instance of the web service class. Thus, despite their implementation as objects, web services are stateless, because web service instances do not persist between method calls. An invocation of a web method is more a remote procedure call, than a remote method invocation. As I have based the G2 programming model on ASP.NET web services, the web services and web methods written by G2 application programmers are functions, and not methods that act on some persistent object. This approach is reflected in the behaviour of volunteers. Whenever a volunteer receives a new job, it will create a new instance of the user’s web service class, and then invoke the method specified by the job.
This style of programming however is inconsistent with the object oriented style of programming favoured by many programmers. If G2 application programmers have to implement web services classes, then ideally, they should be able to think about web methods as if they operated on some persistent instance of the class. Web method invocations should be more like RMI, and less like RPC. But RMI implies object state persistence between method calls. The reality however, is that these objects do not persist. For standard web services, persisting state between method invocations has the potential to make the results of web method invocations unpredictable as multiple users invoke methods on the same object. In G2, persisting objects on volunteers is impractical, as volunteers can come and go.

To simulate the persistence of objects, and to enable an object oriented approach to programming, I have added an operating mode where the state of the object is communicated with each method call and returned with each result. To use this mode, programmers add public data members to their web service class, which represents the object state that they wish to persist between calls. These public data members are incorporated into that class’s proxy, as generated by the G2 proxy generation tool. In this way, the proxy continues to act as a surrogate for the programmer’s web service class. Any changes to the public data members, either on the client, or during method execution of volunteers, will persist for the life of the proxy.

Persistent object state changes the nature of proxies from the model of G2 programming described thus far, and has implications for the way programmers use proxies. Instead of instantiating a single proxy, and executing methods on it in parallel, programmers may wish to have a collection of proxies, and execute methods across all of the proxies in parallel. Therefore, I have implemented an extension where the generated proxy includes specialized collection classes for the invocation of methods in parallel across a collection of proxies. I have restricted the extended RMI style of programming to use of these collection classes because
the object persistence required differs from the standard ASP.NET model. In the
standard model, which does not have object persistence, users can modify in-
stance data members with impunity and can assume that the value of these mem-
bers will reset to the default value whenever a new web method is executed. If
state persistence is not intended by programmers, then such persistence has the
potential to introduce errors, and can result in unnecessary communication. Ob-
ject state becomes an implicit in/out parameter passed to each method.

**Preserving Object State**

Objects do not persist on volunteers, so persistence is simulated. Object state is
transmitted as part of each job, and transmitted back to the client as part of each
result. The object is serialized prior to transmission, and deserialized at the desti-
nation. Only public data members are serialized, so any data that is not to be per-
sisted is stored as private data fields. If a job includes a serialized object, then vo-
unteers deserialize the object and execute methods on it, instead of constructing
a new object. If a result contains a serialized object, the client deserializes the ob-
ject and inserts it into the appropriate place in the specialized object collection.
This additional step required only minor changes to the pre-existing G2 frame-
work.

**Using the Specialized Collection Classes**

When the G2 proxy generator creates the definition of a proxy of a user’s web
service class, it also creates a definition for a class, consisting of a collection of
these proxies. The collection class has public methods corresponding to the web
methods of the user’s web service class. The collection class methods are used to
invoke methods on all of the proxy instances within the collection.

As an example, suppose a G2 application parallelises the rendering and post
processing of an image. The desired image is divided into a set of smaller seg-
ments called tiles. Each tile represents a part of a larger image and has public
members which store its location in that larger image.
class Tile : System.Web.Services
{
    public Bitmap image;
    public int x, y;

    [WebMethod]
    public int Render(int viewX, int viewY, int viewZ)
    {
        int startX = x;
        ...
    }

    [WebMethod]
    public void Blur(int radius)
    {
        ...
    }
}

Listing 3-24: Methods of the Tile Class. Note that the data needed to execute methods comes from the method parameters as well as the data members.

When the proxy for the Tile class is generated, it will contain a TileProxyCollection class, shown in Listing 3-25. This collection class has Render and Blur (a post processing operation) methods, corresponding to the web methods of the same name, from Listing 3-24.

class TileProxyCollection
{
    public int[] Render(int viewX, int viewY, int viewZ)
    {
        ...
    }

    private void EndRender(G2AsyncResult ar)
    {
        ...
    }

    public void Blur(int radius)
    {
        ...
    }

    private void EndBlur(G2AsyncResult ar)
    {
        ...
    }
}

Listing 3-25: Automatically generated TileProxyCollection class

This application, and in particular the proxy class, is different from what I have described thus far. Previously, I had been using a proxy class to be a proxy to the user’s stateless web service class, and typically, only one proxy was instantiated. Every tile was represented not as an object, but as some set of parameters. Here, I
treat the user’s web service class as a remote object, and each web service class represents a single tile. Thus, a proxy is instantiated for every tile that I wish to represent. To invoke the `Render` method on each tile, instead of separately invoking the `Render` method on each proxy instance, the `TileProxyCollection` class is used to invoke `Render` across all of the proxies with one call.

**Inside the Proxy**

Taking a closer look at the `Render` method of the collection class, as each element in the Tile collection is a tile proxy, new jobs are created by invoking a `Begin` method on each proxy. To collect results, a callback is supplied to a private method called `EndRender`, which puts results as they are fetched into a results array. The `Render` method blocks until all results have been retrieved. The `Render` method of the collection class takes the same parameters as the `Render` web method, in this case, to indicate the coordinates of the camera. Note that the code in Listing 3-26 and Listing 3-27 is contained in the autogenerated proxy class.

```csharp
public int[] Render(int viewX, int viewY, int viewZ) {
    for (int n = i; i < tileProxies.Length; i++)
    {
        TileProxy t = (TileProxy)tiles[i];
        AsyncCallback cb = new AsyncCallback(EndObjectRender);
        t.BeginObjectRender(viewX, viewY, viewZ, cb, new ProxyRecord(i, t));
    }
    results = new int[tileProxies.Length];
    are.WaitOne();
    return results;
}

private void EndRender(G2AsyncResult ar) {
    ProxyRecord record = (int)ar.State;
    TileProxy proxy = record.Proxy;
    int number = record.ProxyNumber;
    results[resultNo] = proxy.EndObjectRender(ar);
    if (++completed == results.Length)
    { are.Set(); }
}
```

**Listing 3-26: Proxy code for object based data flow. This is machine generated.**

To ensure that the state of each proxy is sent as part of the job, the collection class creates jobs by calling `BeginObjectRender`. This is one of a pair of additional
methods created by the proxy generator that serializes or deserializes a Tile-Proxy when a job is submitted or a result returned. The following code sample shows parts of the `BeginObjectRender` and `EndObjectRender` methods. Only those parts associated with object serialization are shown.

```csharp
public G2AsyncResult BeginObjectRender(int viewX, int viewY, int viewZ, AsyncCallback cb, object state)
{
    StringWriter writer = new StringWriter();
    XmlSerializer ser = new XmlSerializer(typeof(Tile));
    ser.Serialize(writer, this);

    // the serialized state and the serialized parameters go into an
    // array of string called parameters
    return base.BeginInvoke(parameters, cb, state);
}

public int[] EndObjectRender(G2AsyncResult ar)
{
    string[] results = ar.Result;
    string serializedTile = results[0];
    System.IO.StringReader reader = new StringReader(serializedTile);
    XmlSerializer ser = new XmlSerializer(typeof(Tile));
    Tile tile = (Tile)ser.Deserialize(reader);
    this.Bitmap = tile.Bitmap;
    this.x = tile.x;
    this.y = tile.y;

    // Extract the result and populate the results array.
}
```

**Listing 3-27: Serialization of Proxy objects**

The collection’s `Render` method, calls the `BeginObjectRender` method with a callback to `EndObjectRender`. Whenever a Tile is rendered by a volunteer, and a result is received, the `EndObjectRender` method is called. A serialized object, which makes up part of the result, is deserialized, and the local object state is updated accordingly. Any changes to the public members of the Tile object that occurred during the execution of the `Render` method are preserved.

The addition of object data with each job and result required some changes to the underlying G2 framework. I changed the format of the job data structure, and the logic of the G2 volunteer to extract and deserialize an object if one was included in the job. The job is then executed using the deserialized object, and after execu-
tion, the object is serialized and returned as part of the result. Similarly, the proxy recognizes that a serialized object may be included in the result from the server.

When programmers use collection classes to invoke methods on the proxies, the asynchronous invocation of proxy methods and their parallel execution on remote volunteers is hidden. The collection classes can be reused to invoke a sequence of methods. In Listing 3-28 a TileProxyCollection is created and two operations are performed on each tile proxy in the collection. In this case, the tiles are all rendered in parallel, and then a post processing operation is performed on each tile, again in parallel. The programmer does not have to create jobs, or manage result handles and result retrieval.

```java
public void BulkRender(int viewX, int viewY, int viewZ, int radius) {
    // url is a field storing the location of the G2 Server
    TileProxyCollection tiles = new TileProxyCollection(url);

    // Populate the collection
    for (int x = 0; x < imageSize.Width; x += division ) {
        for (int y = 0; executing && y < imageSize.Height; y += division ) {
            int stopX = Math.Min(x + division, imageSize.Width);
            int stopY = Math.Min(y + division, imageSize.Height);
            tiles.Add(new TileProxy(x, stopX, y, stopY));
        }
    }

    // Render in parallel. Calls the render method on all of the TileProxies
    // in the tiles collection
    int[] results = tiles.Render(viewX, viewY, viewZ);

    // Postprocess in parallel
    tiles.Blur(radius);
    Display(tiles);
}
```

**Listing 3-28: Client side code for data parallel render**

### 3.2 Deploying and Using a Cycle Stealing System

Before anyone can start taking advantage of idle cycles, a cycle stealing infrastructure needs to be set up. Ideally this process is simple. A client-server based system such as G2 has an infrastructure component for the server, a volunteer component for computers that donate resources, and a client component for computers that use resources. On the server, somebody installs the server software onto a computer and administers it. To donate cycles, somebody, presuma-
bly the owner of the computer, obtains and executes some software that connects their computer to the cycle stealing network. Similarly, to use those donated cycles, a user somehow connects their application to the cycle stealing network. All of these steps are potential sources of complexity, which I have tried to simplify in the G2 framework.

3.2.1 Installers and Administrators
Installers and administrators are the people responsible for initial deployment and continual maintenance of the G2 server. G2’s central server component consists of a database, with a web server based front-end. I implemented the server component using Sql Server 2000 and IIS using web services, but there is no reason why these cannot be substituted with alternatives. For example, there is a version of G2 that uses an Oracle database, and other database products such as MySql could be used. The use of these commodity software components means that there will be scenarios where G2 can be deployed on a computer where the required software already exists. In that case, deploying a G2 server consists of running an installer. Another advantage of the use of commodity components is that the familiarity, optimisation and performance of these components are leveraged by G2. The administration of the G2 server components therefore requires nothing in addition to the normal administration of IIS and Sql Server.

3.2.2 Resource Donors
Resource donors are those individuals who donate their computer and its unused cycles to the cycle stealing network. I use the term “donor” because in G2, like in most other cycle stealing systems, a person receives no tangible benefit from making their spare cycles available for others. While there has been some work in commoditisation, marketing and sale of cycles [42, 43], for the most part, donors currently contribute cycles because of interest in a particular project, or for altruistic reasons. In some organisations, employees may be mandated to make their spare cycles available, but in general, there is little personal motivation for a per-
son to contribute to a cycle stealing effort. This implies that donors will have little motivation to overcome any difficulties that they should encounter in connecting their computer to a cycle stealing network. For this reason, the need to keep their experience simple and straightforward is vital.

A donated computer becomes part of a cycle stealing system by running a cycle stealing infrastructure component. In G2 terminology, this component is called the volunteer. A running volunteer actually consists of two parts. The volunteer host is one part, it is generic and is supplied by the G2 framework. Donated cycles are consumed by user supplied G2 applications, so these application specific sections of code make up the other part of a running volunteer. The resource donor initially performs some action that starts running the volunteer host on their computer. In G2, this means opening the G2 volunteer web page. The volunteer host connects to the G2 server and downloads the application specific components that it needs. This approach differs from systems such as SETI@Home and its general purpose derivative, BOINC [6], where the infrastructure component and the application specific components are integrated into one package. Also it should be noted that unlike BOINC and SETI@Home, G2 is capable of supporting multiple clients running multiple different applications on the same cycle stealing network. As such, G2 volunteers are intended to be able to quickly switch from one application to the next. The separation of the generic component, and the downloading of application specific components without further donor intervention makes this possible, and means that the steps taken by donors to connect their computers to a G2 network is always the same, irrespective of whichever applications are running at the time.

**Web Browser Based Resource Donation**

Donors connect their computer to a G2 system, and more specifically, to a G2 server, through a web browser. The volunteer host is implemented as a .NET user control embedded in a web page, and it starts running when the donor opens that page. As the volunteer host is generic, the web page is static, with a well-known
URL. The server does not need to do any further processing to deliver the page to the donor’s browser. This is probably the simplest method of connecting to a cycle stealing system. To start contributing cycles to G2, no pre-installation of G2 software is required.

![Figure 3-5: Generic, web page embedded G2 volunteer component](image)

The volunteer host connects to the server, finds a job that needs to be executed, and downloads the application specific components that it needs to execute that job. These application specific components take the form of user created .NET assemblies.\(^8\) Assemblies downloaded in this manner are stored in the downloaded assembly cache, which is a feature of the .NET framework. Assembly caching enables volunteers to switch between different classes of G2 applications, without needing to re-download application specific code. The volunteer host can switch applications without needing web pages to be refreshed, or any human interven-

---

\(^8\) All programs written using the .NET framework reside in modules. An assembly is a logical collection of one or more modules, and is the base unit of deployment for modules. Therefore, code needed to execute jobs are packaged somewhere inside an assembly, and these assemblies are eventually downloaded by volunteers prior to job execution.
tion. Versioning and name clashes are handled through the optional use of strongly named assemblies. A strong name consists of the assembly’s name, version number, culture information\(^9\), a public key and a digital signature. Strong names give each version of an assembly a globally unique name eliminating the possibility of name clashes.

The fact that the volunteer is not installed and executed as a local application means that the donated computer is protected by default from malicious code. The .NET framework uses a security model called code access security [44], where access permissions are based on the identity of the assembly. All loaded assemblies exist in some security context, which controls the resources available to that assembly. Anything obtained from the Internet, by default, exists in a restrictive context, which prevents free access to the file system and other system resources, thereby totally preventing harm to the donated computer.

Browser based volunteering was first introduced by the Charlotte [21] project, which embedded Java applets in web pages. These applets had to be supplied by the application programmer and conform to Charlotte specifications. The lack of separation between the generic component and the application specific component, and perhaps limited by the then available methods of content delivery, meant that each Charlotte application had to host its applet in its own web page with a unique, static URL. While this enabled donors to choose the applications to which they wished to provide resources, it created potential inefficiencies because volunteers could not switch between Charlotte applications without user intervention. Therefore, it was possible for donated computers to sit idle because an application had terminated, while another application, hosted on a different web page did not progress due to lack of volunteers.

\(^9\) The culture or language supported by the assembly. This is an RFC1766 identifier. http://www.ietf.org/rfc/rfc1766.txt
3.2.3 Resource Consumers

The resource consumers are the users who wish to run an application that makes use of the computing resources provided by the cycle stealing network. I have simplified the use of the G2 framework for these users, by reducing the steps that they need to perform prior to starting a cycle stealing application.

For a donated computer to start executing jobs, it must first have access to all of the code and data required to execute that job. As mentioned earlier, Charlotte, and some other miscellaneous one-off systems [45], achieved this by packaging jobs up as applets and hosting them on a web page. Other systems have implemented distributed shared memory, Tuple Spaces [Carriero, 1989 #27; Wyckoff #28] or similar methods, or required the user to host the data on their computer, and make the data somehow accessible to volunteers [13]. These methods require the user to perform an additional step, such as copying files onto a server, or creating file system shares, or maintaining some other means to serve files from their computer. These methods have other prerequisites, such as needing accounts on the server, negotiating security configurations, or even the need to run and maintain a web or FTP server, and they may not work without reconfiguring existing firewalls.

In G2, all of the data needed by volunteers is encapsulated in assemblies, which are distributed to volunteers by the server. The framework also automatically propagates these assemblies from the computer of the resource consumer onto the server.

Part of the process of writing a G2 application involves linking to the G2 framework class library and instantiating a proxy. The proxy is part of the G2 infrastructure component known as the client component, which runs on the computer of the resource consumer. When the client component is initialized, it determines the assemblies that are needed by the volunteer to execute jobs, and automatically uploads them to the server from the client’s host computer. During this upload process, the client component uses reflection to recursively determine
the dependencies of each assembly, to ensure that all of the required user created assemblies are uploaded to the server. This scheme takes care of assembly distribution on behalf of resource consumers, who do not need accounts or other privileges on the server.

To run an application using G2 all that is required by consumers is to start the application on their local machine. The framework takes care of data distribution, and recognises assembly versions and strong names to ensure that the correct version of assemblies is present on the server for distribution to volunteers. The G2 framework enables users to plug in their computers to a distributed system, so users can start applications from their own PCs.

### 3.3 Ease of use through automation

In the previous section, I mentioned that I had tried to simplify the use of the G2 framework by having the client components automatically upload client applications to the server. Such automation enables users to concentrate on developing and running their applications rather than managing the deployment of those applications and making sure that the correct versions of components are installed on each volunteer. This level of automation can encourage experimentation and prototyping, because deploying new versions of applications becomes trivial from a user perspective. All it takes is to rerun the updated application from the user’s computer. Without such automation, a user might be required to manually deploy the new application on the server, or worse still, manually deploy the updated application on each volunteer.

It is not only deployment of user applications that are automated by the G2 framework components. Before jobs are executed on volunteers, a client application has to be connected to a cycle stealing network and a job has to be created and submitted. The connection to the network, and the subsequent submission of jobs, also requires the use of some mechanism that ensures that the results of the job are returned to the correct recipient. These steps could be the responsibility of
the programmer, but in G2, they are largely automated through the G2 client and volunteer components.

To make use of the G2 framework, a proxy to the programmer’s G2 “web service” is created by using the G2 proxy generation tool. This proxy encompasses the G2 client component, and its operation is described below in Figure 3-6.

1. In this example, G2 proxy generation tool has generated a proxy class called MyProxy. The first instantiation of this proxy from the programmer’s code, is also the instantiation of the G2 client component and the first step in the automation process. MyProxy is actually a subclass of a G2 framework class called G2ClassProxy. The G2ClassProxy contains a reference to a G2ClientProxy instance, which is usually instantiated with the G2ClassProxy. The ClassProxy, and ClientProxy make up what I had referred to earlier as the G2 Client Component. On any client, there is only one ClientProxy instance for each G2 server that is used by the application. (It is possible for a single application to submit jobs to multiple G2 servers.) Therefore, if the programmer were to create another instance of MyProxy, the two instances of MyProxy would share the same ClientProxy if they used the same server. The purpose of sharing is to aggregate messages to the server, and to minimise the load that one client will put on a server.

2. MyProxy is instantiated with a URL of a known G2 server. When a ClientProxy is instantiated, it contacts the G2 server referenced by the URL and registers itself as a new client. Client registration is used to match clients with jobs and
results. The ClassProxy contacts the G2 server and registers the programmer’s G2 web service class. This web service class is the class that was used to generate MyProxy. It is possible that the G2 web service class has already been registered, which would be the case if the user application has been executed in the past, or the same class was used in another application. In that situation, the server takes no action and ignores duplicate registration information.

3&4. The G2 server acts as a repository for the code needed to execute jobs to volunteers. This code is the implementation of the user’s web service class, and is contained within .NET assemblies. If a class had not been registered, then the server does not have a copy of the assemblies, and so the ClassProxy uploads them to the server. An application may consist of multiple assemblies needed to execute jobs, so multiple assemblies are able to be uploaded during this phase. The server may periodically deregister classes and remove unused assemblies from its cache, so it is conceivable that an assembly will be registered and uploaded more than once in its lifetime.

5&6. Job creation and submission occurs as a result of the programmer invoking a method on MyProxy corresponding to their “web method”. In this case, BeginAdd. The class proxy creates the job payload, which contains the name of the method to be invoked, and XML serialized method parameters. The job is then passed to the ClientProxy. The ClientProxy is responsible for job submission and result retrieval. The ClientProxy is registered with the server, and so the ClientProxy performs any operations that require a client to identify itself. The server uses this information to route the result, when it is eventually computed, to the correct client.

7. The ClientProxy submits the job onto the server, and receives a job identifier. It then creates a result handle which is returned to the programmer, via the ClassProxy, as the result of the BeginAdd call. This result handle, and any callbacks, are stored by the Client proxy, they are used to return the final result to the user once the job is complete.
8. The server receives the job and adds it to the list of jobs on the database. It will be dispatched according to its scheduling rules when a volunteer connects and requests a job.

9. After a job has been submitted, there is at least one outstanding result that must be retrieved by the client. This triggers the fetch results process, which is responsible for requesting and retrieving results from the server. When the ClientProxy receives a result, it will extract the appropriate result handle from its list of stored result handles and signal it as completed. If the programmer elected to block on the wait handle exposed by the result handle, then the blocked thread will resume. If a callback delegate was supplied, the callback will be invoked, and the programmer’s callback method executed. The fetch results process continues until no more results are outstanding.

**Volunteer Component Operation**

A donor makes their computer available to a G2 network to execute jobs, by opening a web page hosted on a G2 server. The web page, contains an embedded .NET component, with a GUI, which is the G2 volunteer host component. I wanted there to be only one URL that embedded the volunteer host, and I wanted the volunteer host to be able to switch between applications relatively quickly. That is, donors should not have to a URL specific to a particular application, and programmers should not have to supply fully contained components or applets for their application, as was the case in Charlotte. This called for a design with a generic volunteer host, which is capable of acting as a container for user supplied application code. This code does not have to implement any G2 specific interfaces.

The operation of the volunteer host component is described below.
1. The volunteer host opens within the web page displayed by a web browser (currently, Internet Explorer is supported). The volunteer host connects to the server via its web service interface, registers itself, and requests a job. If a job is not available, then it will pause and try again. The retry-pause period increases exponentially to a maximum specified duration. Exponential backoff is used as there may be cases where jobs are available, but a request failed due to some contention on the server. In which case the volunteer should try again after a relatively short time. If there are repeated failures, due to server problems, or no jobs being available, then unnecessary or wasteful communication attempts can be prevented by increasing the timeout.

2. If a job is available it will be fetched by the volunteer host. A job typically consists of the name of the class that is required, the method name and the parameters. (The job may also contain a serialized object on which to invoke a method as was discussed in the data parallelism section earlier, and it may contain a checkpoint, which will be discussed in Chapter 4). The server will also add the URL of the assembly required by the job and the class identifier to the job. If the specified assembly has dependencies on other assemblies, then those assemblies will also be downloaded.

3&4. To execute the job, the volunteer host needs to load the assembly that contains the code for the user’s class. If the .NET framework encounters an assembly that is located remotely, it will download it, and put it in its downloaded assem-
bly cache. Therefore, if the volunteer had previously executed methods on this class, then the assembly may exist in its cache. An assembly will only be downloaded if it does not exist in the assembly cache. The process of loading an assembly involves loading all the assemblies that it depends on. So, the .NET framework will recursively download and open all of the assemblies needed to execute the job.

Fetching a job is actually a two step process. First the volunteer requests the Class Id of a class that has jobs available. When this Id has been obtained, the volunteer requests jobs belonging to that class. This scheme is part of a strategy to minimise the downloading of assemblies to the volunteer. Once a volunteer has received a class id, it will continue to request jobs for that class until the server stops assigning jobs for that class. If a fetch attempt for a job fails, (because all of the jobs have been completed, or the server wishes to force the volunteer to start working for another class), the volunteer will try to obtain another Class Id. The server will only respond with Class Ids that have jobs available for execution.

5. An instance of the class whose method invocation is encapsulated by the job is constructed. The method parameters are XML deserialized, and the method invoked.

6. Method execution yields a result which is submitted to the server.

7. Results are stored in the server database until they are retrieved by clients.

Using the G2 framework components, much of the work of connecting applications to a G2 network, creating jobs and retrieving results, is automated and is not explicitly performed by users. To add their computer to the G2 volunteer network, a donor has only to open a web page.

3.4 Related Work

I have tried to simplify programming by taking an existing programming model for the development and invocation of web services, and adapting it for parallel computing. This makes programming easy by first increasing its familiarity to us-
ers, that is, if a programmer is familiar with programming ASP.NET web services, which is the model on which G2 is based, then they should be able to quickly produce parallel applications. Secondly, the adapted programming paradigm (ASP.NET web services) is quite simple.

One of the advantages of the .NET framework is its language independence. Although this is true of the JVM to some extent (some languages have compilers that target the JVM, e.g. Component Pascal), the .NET runtime was designed to support multiple languages and considerable effort has been made to produce compilers that target the .NET framework. The G2 framework is able to leverage this work and enable programmers to write parallel applications in any of these languages. This is more user friendly than requiring a programmer to learn a new language, which is the approach of Cilk [18] and Mianjin [48], the language supported by the original Gardens Project [49], the predecessor of G2.

Many systems adopt Java as the programming language and provide libraries that enable programmers to wire their applications to cycle stealing infrastructure. Examples are Javelin and its derivatives [26, 27], and Bayanihan [8]. Some commercial frameworks, such as Frontier, from Parabon [35], are also Java based. The advantage of Java is that it is wide use, familiar to many programmers, and available on many platforms.

The programming simplicity makes these frameworks more accessible but less powerful than parallel systems supporting programs written using specialised parallel programming tools such as MPI and PVM. PVM is partially supported by Condor but much of the message passing capabilities of PVM are restricted [50]. G2 abstracts the creation of jobs and their remote execution, with a method call on a local proxy. Other systems feature explicit job creation.

**Condor**

Condor [13] is capable of executing native code, so applications do not have to be written specifically for execution using Condor. However, these are sequential,
not parallel applications. To make use of some features of Condor such as check-pointing, applications have to be relinked to Condor libraries using the condor_compile command. Launching an application requires an additional job description file, which has to be supplied by the user. An API, called MW, is available to enable the execution of parallel master-worker applications using Condor [51]. Programmers are required to re-implement a set of virtual classes representing an MWMaster, an MWTask, and an MWWorker, corresponding to clients, jobs and volunteers using G2 terminology. Although different types of applications such as those with branch-and-bound optimisation, can be shoehorned into MW applications, there is no built-in support for applications other than standard master-worker.

**Charlotte**

A Charlotte [21] program consists of alternating sequential and parallel steps. A parallel step is a list of routines, where each routine is executed on an available volunteer, and exhibits master-worker style parallelism. Therefore, a Charlotte program is similar to a Bulk Synchronous Parallel (BSP) application. The syntax for a parallel step is:

```plaintext
parBegin();
//Routine List;
   void addDroutine(Droutine obj1, int numRoutines);
   ...
   Void addDroutine(Droutine objN, int numRoutines);
parEnd();
```

**Listing 3-29: Charlotte Parallel Step**

A routine list is a sequence of addRoutine statements, where the first argument is a subclass of a Charlotte class called Droutine, and must implement the method drun().
Cilk-NOW

```
thread Fib (cont int k, int n)
{
    if (n < 2)
        send_argument (k, n);
    else
    {
        cont int  x, y;
        spawn_next Sum (k, ?x, ?y);
        spawn Fib (x, n-1);
        spawn Fib (y, n-2);
    }
}
thread Sum (cont int k, int x, int y)
{
    send_argument (k, x+y);
}
```

Listing 3-30: Job creation using Cilk-NOW

This Cilk-NOW [19] Fibonacci routine is written in continuation passing style and takes a continuation and the Fibonacci number to be calculated. The `send_argument` call sends a value to a waiting thread. The `spawn_next` function creates a successor thread, or a continuation, and the `spawn` function creates child threads. In this manner threads in Cilk, like G2 jobs, have the ability to spawn new jobs. It should be noted that later versions of Cilk have a more advanced API that simplifies job creation, but those programs execute on dedicated multiprocessor machines, and not in a cycle stealing environment.

**Jicos**

In Jicos [7], the programmer is required to connect to server elements, and explicitly create jobs.
package jicos.examples.helloworld;
import jicos.examples.helloworld.*;
import jicos.system.*;
import java.rmi.*;

final class Application
{
    public static void main (String args[]) throws RemoteException
    {
        // get a remote reference to an HSP
        HSPAgent agent = new HSPAgent( args[0] );
        HSP hsp = agent.getHSP();

        // register with HSP
        ClientInfo clientInfo = new ClientInfo( null, null );
        hsp.registerClient( clientInfo );

        // start computation
        hsp.compute(new HelloTask(), 0 );

        // retrieve, process Result
        Result result = hsp.getResult();
        System.out.println( (String) result.getValue() );

        // unregister from HSP
        hsp.unregisterClient();
    }
}

Listing 3-31: Jicos Client Code

package jicos.examples.helloworld;
import jicos.system.*;

final class HelloTask extends Task
{
    public Object execute ( Host host )
    {
        return "Hello, world!";
    }
}

Listing 3-32: Jicos Volunteer Code

The application registers with a host service provider, and submits a task object. Unlike G2, these steps are explicitly performed by the programmer. Volunteers receive an object of type task, and run the execute function. Jobs in Jicos consist of task objects that encapsulate application logic. Therefore the software installed on the volunteer machines is merely a shell for task objects and do not need to be tailored for each type of job. In addition to standard master-worker, Jicos has built-in support for Branch-and-Bound optimisation. [JICOS: A Java Centric network computing service] Computation in Jicos, as in G2, is modelled as a directed
acyclic graph. A task may simply return a result, or decompose into a sequence of subtasks. The result of such a task is another task called a compose task. Each subtask produces an output that is an input to this compose task, and the output of the compose task is the expected output of the original task. Therefore, to a caller, there is no difference between task that decompose, and tasks that do not. To support Branch and Bound optimisation, the current state of a computation needs to be communicated to all nodes. Therefore, Jicos tasks have access to a common environment, in the form of a common Environment object. The value of this shared object may be changed, and such changes are propagated asynchronously. The programmer has to supply logic which indicates whether an incoming change notification is “newer” than the current known value of the shared object.

Parabon

Many of the steps hidden to programmers in G2 are exposed in Parabon [35]. This complicates the programming of applications, but this complexity can make the system more flexible. However, only standard master-worker style applications are directly supported.

An example Parabon application is demonstrated in their online tutorial. Note: Some Frameworks use a similar naming convention where a Job is an application submitted for execution. A Job is decomposed into a series of executable units, called a task. It is the tasks that are executed on volunteer nodes. The code for even the basic application is quite long and is included in Appendix F.

3.5 Conclusion

I created a model for simple cycle stealing which, in a way, is analogous to the use of an electrical power grid. A user simply plugs in their applications to a G2 network, and anonymous sources provide power. Like an electricity grid, the source of the power may change over time, but this is hidden and irrelevant to the user. The network can support a wide range of applications, and does not need to be configured for any one in particular.
Deploying and maintaining a G2 server is no more complex than deploying and maintaining a web server and database. Connecting volunteers is quick and simple due to webpage based delivery and execution of the G2 volunteer host component. Unlike similar web based cycle stealing frameworks, a single G2 network, with one server, is capable of supporting multiple applications where a volunteer can automatically switch between them as needed.

One of the key aspects of making a cycle stealing framework easy to use is the programming model. The G2 model has a high level of abstraction, and is based on an existing, well known programming model, that of ASP.NET web services, which simplifies the development of applications that are suitable for execution using cycle stealing. The G2 model enables the straight-forward creation of master-worker, hierarchical master-worker, data parallel and data flow type applications whose complexity goes beyond the standard type of embarrassingly parallel, master-worker application supported by most existing frameworks.

Cycle stealing environments are characterised by the property that volunteers, or processors, can come and go at any time. In such an environment, G2, and other cycle stealing frameworks, have a restrictive approach to job execution compared to supercomputers and clusters. In the G2 model, jobs are units of an application capable of independent execution. Thus the concepts of processes executing on different nodes, their synchronisation, and passing messages between them, are not appropriate for the G2 environment. The only “operations” on jobs permitted in the G2 model are submission to the G2 server, and the recovery of their results. As such, I was able to eliminate the job construct from the G2 programming model, because if jobs do not expose any functionality aside from their submission and result recovery, then programmers should not have to deal with jobs at all. Thus, G2 application programmers, unlike programmers for other cycle stealing frameworks, do not have to manually create job objects. Job creation is implied when a “web method” is invoked via a proxy.
In this way, I have created a programming model that simplifies application development through an appropriate but higher level of abstraction than those for cycle stealing frameworks, and supercomputers and clusters. The model is ideally suited for cycle stealing using computers connected to the Internet, which is the target deployment scenario for G2.
Chapter 4  Realising an Internet Based Cycle Stealing Framework

The purpose of cycle stealing systems is to provide users with high performance or high throughput computing\textsuperscript{10}. This means they must provide users with capabilities that exceed those of an ordinary desktop computer. G2 is a high performance computing system, concerned with the execution of user applications as fast as possible. This is achieved by decomposing an application into a set of jobs, which are independently executed in parallel. This means that a G2 system needs to be able to execute jobs efficiently, thereby solving the user's problem faster than a sequential application running on single computer. Maximising efficiency is not the only goal however. The execution of these jobs also needs to be reliable. This means that the computed result of jobs needs to be accurate, and jobs should not be lost. Where there are multiple jobs and user applications, starvation should not occur, and all jobs need to be executed in a timely manner. These jobs execute on computers that have been temporarily provided by others. So, there is a safety requirement, because the donated computers need to be protected. Mechanisms are needed to ensure that the execution of jobs have no detrimental side effects on donated computers.

This chapter describes the techniques that I have developed and used to ensure that G2 applications execute efficiently, reliably, and safely. I have focused on making communications more efficient because the need to communicate jobs and results has a large effect on performance, but I also examined the use of the common Eager scheduling algorithm, and investigated the possibility of incorporating job checkpointing into G2 volunteers.

\textsuperscript{10} High Performance Computing (HPC) environments are concerned with delivering the most performance in the shortest period of time, whereas High Throughput Computing (HTC) is concerned with delivering large amounts of processing capacity over a long period of time [52]. If a user wants an application to run as fast as possible, then a HPC environment is suitable. If a user wants to quickly run a series of applications, and the execution of any one application does not have to be as fast as possible, then a HTC environment is suitable.
4.1 Efficiency and Performance

Evaluating performance involves a comparison of the running time of a parallel G2 application, compared to a sequential version of the application running on a single machine. Parallelising an application and its execution introduces overheads that are not incurred by a sequential application. For example, a parallel application needs to decompose itself into a set of independently executing parts, called Jobs in G2, and then the results of these jobs have to be later aggregated. To prepare these jobs for execution on remote nodes, they have to put into a form that is capable of being transmitted, that is, serialized, and then deserialized at the remote node. The results of the execution of these jobs similarly need to be transmitted and hence are also serialized and deserialized. In addition there is the actual cost of communicating jobs and results. For the execution of a parallel application to be worthwhile, that is, to yield a performance benefit over a sequential application, the benefits of parallel application have to offset the overheads caused by parallel execution.

Changing the execution of a sequential algorithm into a parallel version involves the decomposition of an application, and the amalgamation of results. These steps may carry an application specific runtime cost, but I have assumed that this cost is not significant. To communicate jobs and results, there is a serialization and deserialization cost, but I found, following the performance analysis of G2 (described in Chapter 5), that the actual network communication cost constitutes the bulk of the overhead of parallel execution.\(^{11}\)

For a parallel application to exhibit a performance benefit over its sequential counterpart, jobs must have a sufficient execution time to offset their communication costs. The larger the computation time, compared to the communication

---

\(^{11}\) Phillipson et al. [53] found that 25-50% of the cost of Java RMI was incurred through serialization. This may have been the case for Java in 1999 when the paper was published, but it probably does not hold in 2005. It is certainly not the case for .NET serialization on 100Mbit networks.
time, that is, the ratio between computation compared to communication time, the larger the amount of speedup that a parallel application will exhibit.

So given an application which decomposes into a number of jobs, all requiring some communication time, there is a minimum computation time that these jobs have to exceed for the execution of the application to be worthwhile. The efficiency of the communication mechanism determines this minimum computation time. Efficiency therefore not only leads to better performance, but increases the range of applications that are feasible to execute using a cycle stealing system.

4.1.1 Efficient Communication

In G2, the transmission of each job and result carries an inevitable serialization and deserialization cost. However, these are relatively small compared to the other costs associated with communicating them between clients, volunteers and the server. Communication involves physical communication and marshalling, and it is these processes that I have tried to make more efficient within the G2 framework.

Asynchronous Communication

The G2 framework client and volunteer components communicate with the server via a web service hosted on the server. A request-response protocol such as HTTP increases the likelihood that computers behind firewalls will be able to participate as clients or volunteers and connect to a G2 server. However, a problem arises because the G2 server cannot initiate communication with a client or volunteer. The problem becomes apparent in those situations where a status change on the server is ideally communicated instantly to interested nodes. For example, when a volunteer submits a result to the server, that result should instantly be relayed to the appropriate client. However, as the server cannot initiate communication, the client can only receive the result after it requests that result from the server. The interval between the instant that a result becomes available
on the server, and its actual receipt by the client, is a delay that adds to the over-all communication time of a job.

If the server cannot unilaterally send the result of a job to a client, or otherwise notify a client that a result is available, the obvious approach to have the client poll for results. However, this is problematic because choosing an appropriate polling period is difficult. If the period is short, then the client will be able to retrieve results soon after they become available, but there is a danger of clients inundating the server, and the network, with superfluous requests, effecting overall server and network responsiveness. If the interval is too long then the outcome is an increase in the communication time of jobs.

Fortunately, there is a mechanism that can be used to eliminate the problems associated with polling, and enables the server to return a result to a client as soon as that result is received from a volunteer. I have used the asynchronous execution capabilities of ASP.NET web services (described in chapter 3) to effect an asynchronous communication mechanism over HTTP. Using this mechanism, a client attempts to fetch results by making a fetch results request to the server. This is a polling approach, but the request will block on the server side if a result is not yet available. When a volunteer submits a result for that client, the fetch result request resumes, thereby returning the result to the client as soon as it is received on the sever. When the client receives the result, it again attempts to fetch results from the server. Using this mechanism, I am able to have a G2 server control when results are returned to clients, and yet work within a request-response pattern, where clients have to initiate communication.\(^{12}\)

---

\(^{12}\) The G2 server’s Fetch Results web method is implemented in two parts following the ASP.NET’s standard pattern for implementing asynchronously executing web methods. The Begin Part of the method takes an ASP.NET generated callback to the End part of the method as an input parameter. If a result is available, the callback is immediately triggered, executing the End Part. The End Part causes a result or results for the client to be read from the database, packaged into a XML serialized string and returned. If there are no results, the callback is stored until a result becomes available.
One Client Proxy per server per application

G2 applications connect to a G2 server via a client proxy. An application has only one instance of the client proxy for each G2 server to which it connects. If multiple class proxies exist and connect to the same G2 server, they will share a single instance of the client proxy. This sharing enables the aggregation of messages, such as requests to fetch results, thereby minimising the amount of traffic between a client and a G2 server.

FetchResults Retrieves Multiple Results

It is possible that more than one result is available for collection when a fetch results request is made. In that case, the server aggregates the results into a single message, which is returned to the client. The sharing of a single Client Proxy instance among multiple Class Proxies on a client means that a fetch results request made by a Client Proxy is capable of receiving results for any of that client’s Class Proxies. Therefore, aggregated results received from the server may contain results for multiple Class Proxies. The Client Proxy is able to separate the aggregated result message into single results, and ensure that those results are returned to the correct Class Proxy.

Retrieving multiple results improves performance, because the client application will generally retrieve results faster than if it had to fetch each result individually. It also reduces the number of requests that the server has to handle. However, simply attempting to retrieve all of the available results at once can be problematic. Firstly, a large amount of data may need to be transmitted, leading to a greater probability of transmission problems. Secondly, the client computer may

The callback is stored in memory, in the G2 web service’s Application State. This is a feature of all ASP.NET web services, and enables a web service to share data between web method calls. Whenever a result is submitted, the server checks whether a callback for a fetch results request, relevant to the newly submitted result, exists in the Application State. If a callback exists, it is removed from the Application State and invoked, triggering the execution of the End part of the asynchronous Fetch Results request.
not have the memory to process such a large amount of result data at once. Therefore, the client and server coordinate to tune the number of results that are retrieved per request. The tuning process works best if all results are of similar size, but this is not essential. This process is driven by the Client Proxy, which, as part of the fetch results request, informs the server on the number of results it is prepared to accept. This number is initially unlimited, and while successful, the client records the largest number of successfully fetched results from one request. On a failure, the client proxy assumes that it has tried to fetch too many results, and limits the next fetch attempt to the largest, successfully fetched number of results. If this too fails, the largest number retrieved is reset and the client requests only one result. While the number of results actually received equals the number of results requested, the number of results requested is increased. Eventually, the client proxy settles for a consistently reliable number of results.

4.1.2 Mitigating the Cost of Communication

The work encapsulated in a job, will take longer for a volunteer to perform than if it were performed on the client (assuming equivalent specifications for clients and volunteers). The reason is that the job needs to be communicated to the volunteer, and the result needs to be communicated back to the client. This communication cost I not be incurred if the client were to perform that work locally. Around the actual execution of a job, there is a period of communication, which adds to the overall time needed by G2 to “process” a job.

If there are many jobs available for execution on the server, an opportunity arises to greatly reduce the effect of the communication cost incurred by volunteers. By enabling volunteers to have multiple jobs, volunteers can be continually executing jobs while communication occurs in the background.

I have therefore incorporated a pipelining approach to job execution on volunteers, where each volunteer maintains a local queue of jobs. The queue maximises the time a volunteer spends executing jobs, as opposed to waiting to receive jobs.
and submit results. While a job is executing, background threads on the volunteer attempt to fetch jobs and submit results.

![Diagram of job execution process]

**Figure 4-1:** Three threads fetch jobs, execute jobs and submit results.

Without this pipelining capability, the time that a volunteer takes to execute a set of jobs becomes much greater.

![Diagram comparing pipelining vs. no pipelining]

**Figure 4-2:** With pipelined job transmission, volunteers are able to execute jobs without an intervening interval to submit a result and fetch a new job.

Pipelining can only provide benefits if volunteers remain connected to G2 long enough to execute multiple jobs. As jobs are abandoned by a volunteer whenever
it disconnects from the G2 network, pipelining can provide no benefit if volunteers generally connect long enough to execute one job.

**Code Caching**

One of the issues for general purpose cycle stealing is how the users’ application code, needed to execute their jobs, is propagated to volunteers. In some systems, [8, 35] compiled code is incorporated as part of the job payload. This can be convenient because each job is then an independent executable element. In contrast, a G2 job consists of a globally unique class identifier, method name, a serialized set of parameters and some data identifying the origin of the job. A job is therefore a set of instructions telling a volunteer which method to execute and the parameters to use, but a G2 job does not include the user’s application code, that is, the actual implementation of the method. I chose this design because the physical separation between jobs and method implementations enables the caching on volunteers of user application code, which will be used by multiple jobs. This reduces the amount of data transmitted with each job.

In the .NET framework, class implementations, including their methods, are contained within modules, which are in turn contained within assemblies. The .NET CLR (Common Language Runtime) running on each volunteer, maintains a downloaded assembly cache that, as its name suggests, stores downloaded assemblies. Therefore, any assemblies downloaded by the volunteer, such as user-provided assemblies needed for jobs, are automatically inserted into this cache. When a volunteer receives a job, it foregoes the downloading of the assemblies needed to execute the job if they are already present in the cache, and can immediately start executing that job. To take advantage of the cache, I have designed G2’s scheduling algorithm to attempt to assign jobs to volunteers, based on heuristics such as previously assigned jobs sharing the same assemblies. However, prior assignment of jobs sharing the same assemblies is not the only heuristic used to schedule job execution, and a volunteer may be assigned to another job where it needs to download new assemblies. This could happen, for example, when it is
necessary to balance the number of volunteers among clients with different applications (and therefore different assemblies). If in future, a volunteer is reassigned back to jobs from the earlier application, then the cache enables it to switch almost instantaneously, without having to re-download assemblies. The presence of the cache gives the scheduler the flexibility to redistribute volunteers among different applications without a large performance penalty.

4.2 Reliability

Reliability is the ability to produce a result consistent with the application code as written. A G2 application decomposes a problem into a set of jobs which are executed by volunteers.

Hence, reliability in the context of G2 means:

1. Jobs, once submitted to the server, will not be lost.
2. Ensure jobs are eventually executed (provided volunteers capable of executing those jobs are available).
3. Prevent saboteurs or random computing errors\textsuperscript{13} from affecting the outcome of an application.

4.2.1 Job Tracking

The life cycle of a job in the G2 system is as follows:

1. The job is created by an application running on client.
2. The job is submitted and recorded on the server.
3. When a volunteer requests a job, the job is assigned to that volunteer.
4. The volunteer executes the job and submits a result.

\textsuperscript{13} Hardware errors can be a source of incorrect results. For example, there is a very low probability that an ordinary CPU will make an arithmetic error. In a system such as SETI@Home, the number of participants means that at least one such error is encountered daily. The only defence is to duplicate the execution of jobs, which is the approach taken by SETI@Home. The problem becomes more frequent on computers with overclocked processors [54].
5. The client requests results its outstanding jobs.
6. The server sends the available result to the client.
7. The server confirms that the client received the result and the result can safely be removed.

The client learns that a job has been successfully submitted when the server returns a job id. From this initial job submission, until successful result retrieval, the progress of the job is tracked by the server. All operations on a job, such as fetching by volunteers, change the job’s state and are recorded by the server. This state information is used, for example, to assign jobs to volunteers, and to determine which results are available to be returned to clients. The integrity of state information is protected by transactions, and the server needs to be able to detect whether they completed successfully in order to commit the transaction, or to perform a rollback. These operations are initiated by web service request, and an operation is complete when a reply is returned to the web service caller.

Confirming whether the reply was successfully received and processed is handled differently depending on the operation. All operations, except fetch results, use a passive mechanism, where exceptions, raised by the .NET framework, are used to detect and handle errors. A communication error such as a timeout will raise an exception on a client or volunteer, which will recover by retrying the operation. Such an error is not detected by the server, which assumes the client or volunteers successfully received a reply. This passive mechanism is only used where the assumption of success by the server does not result in any loss of reliability. At worst, a duplicate job may be submitted to the server, but this does not result in a loss of data, or otherwise affect the reliability of an application.

An active mechanism uses a separate acknowledgement message that confirms, or denies the successful completion of an operation. Generally, volunteers use passive mechanisms, and only actively report errors if there was some problem during the execution of a job – such as unhandled exceptions from user code. Passive
mechanisms are sufficient for the volunteer initiated operations, Fetch Job and Submit Results, because jobs will not be lost if failures in these operations are not detected. I also assume that volunteers are unreliable, so the server does not rely on volunteers to report that there has been an error.

The mechanism used by clients to fetch results, on the other hand, does require the client to actively confirm the successful transmission of results from the server. As described earlier, the client indicates to the server the maximum number of results that it is prepared to receive during a fetch results request, but the server determines which results are to be returned (if any are available). Therefore, the server needs to know which results were successfully retrieved. For a client to successfully retrieve results, not only must the results be transmitted correctly, but the client needs to successfully deserialize and process the results. The server can detect a transmission error, but not a client processing error. Therefore, the client needs to acknowledge that result transmission was successful, and if a failure occurs, the server needs to resend those results. To ensure successful result transmission, the server assigns a consignment number to each batch of results that it sends to the client. The client echoes this consignment number on the next fetch results request. The client only echoes the last successfully retrieved and processed consignment, enabling the server to detect any errors, and resend results as required. A successfully retrieved result means that corresponding job is now complete and the server can deal with it according to its configuration. By default completed jobs are deleted, but the server can be configured to cache results – cached results could be used to for example to speed up subsequent executions of a user application that share common jobs with prior runs. Such functionality can easily be integrated into the G2 framework.

4.2.2 Scheduling

To ensure the successful execution of jobs, cycle stealing systems need to deal with the volatility of the donated computers that execute those jobs. In G2, the primary mechanism used to recover from volunteer errors is incorporated into
the scheduler, which handles fetch job requests from volunteers. Unlike schedulers for non-cycle stealing systems, the G2 scheduler has to contend with the situation where allocated jobs routinely do not complete, while also attempting to maximise performance. Another hurdle is that is has no knowledge of the capabilities of the donated computers nor the computation time of jobs on which to base scheduling decisions. In other words, it has to attempt to efficiently schedule the use of resources, of which it has very little information.

Running parallel applications on a set of changing processors is known as Adaptive Parallelism [24], and faced with its difficulties, many cycle stealing systems use the Eager scheduling algorithm to manage the use of the pool of volunteers. First introduced by the Charlotte [21] general purpose cycle stealing framework, eager scheduling is a means to ensure the completion of jobs in a volatile volunteer environment without the need to closely monitor the status of each volunteer, and without the need to know the capabilities of the computers in the processing pool.

The basis of eager scheduling is to redundantly assign jobs to any available volunteer. The advantage of this approach is that there is no need to monitor the status of volunteers – all available processors are executing a job, and it decreases the possibility of less capable computers slowing the progress of an application, as jobs can be simultaneously allocated to multiple volunteers of differing capabilities.

14 Using volunteer components embedded in web pages, it is difficult store benchmark information locally on the volunteer. It is possible for the server to store volunteer profiles, on which to base scheduling decisions. If this capability were desired, it can be supported in the G2 framework, although these profiles are only useful after they have been refined over time.
Eager Scheduling and G2

One of my goals for the design of G2 server was to make it fully componentised, where the server interface, the data store and the server control were connected via standard interfaces. Such architecture would provide a framework for researchers to perform experiments on aspects of cycle stealing, as well as enabling users to plug in components that were appropriate for their environment. The implementation of the server in this way is not fully complete, but it is still highly compartmentalised, which enabled me to experiment with different scheduling strategies by modifying a small, highly localised part of the server.

The first scheduling algorithm I tried was eager scheduling, using a basic implementation that maintained an allocation count on each job, giving priority to jobs that had been allocated the least number of times, and otherwise using a FIFO allocation strategy. Using this algorithm, applications were successfully executed using volatile volunteers, that is, volunteers randomly disconnected, discarded their assigned jobs, then reconnected.

However, I found that this basic implementation of Eager scheduling can be detrimental to overall performance under the following four conditions:

1. **Limited number of volunteers compared to jobs**

   One of the problems that I encountered was that if a client were to submit a series of jobs, and the period between job submissions was relatively long compared to the interval between the fetch job requests of different volunteers, than the first job in the series is allocated to those volunteers. Often subsequent jobs submitted by the client would be unallocated because all available volunteers were redundantly executing the same job. The redundant execution of jobs is both the strength and weakness of eager scheduling. If there are many more volunteers compared to jobs, than the negative effects of redundant execution is minimised. The detrimental effect of the redundant allocations is naturally dependent on the duration of the job, and the expected duration of the application as a whole, but
one can imagine a scenario where relatively long intervals between job submissions by a client, can lead to many, if not all of the volunteers being assigned the same long running job, to the performance detriment of the entire application.

2. Relatively Reliable Volunteers

In situations where the volunteers are relatively reliable, then the benefits of redundant allocation to ensure reliability are diminished, as redundant computation occurs with no gain.

3. Homogenous Volunteers

Where the volunteers are relatively equal in performance and capability, there are no suboptimal scheduling or inappropriate allocation issues for eager scheduling to mitigate.

4. Relatively long running jobs

If jobs are short lived, the redundant allocation of jobs to multiple volunteers results in less wasteful work duplication, in terms CPU seconds. If the jobs are long, then this potentially leads to more work being duplicated.

The problem of redundant allocation is amplified in G2 due to the job pipelining employed by volunteers. To mitigate the communication costs, a volunteer will overlap job requests, result submissions and job execution, so that it always has a job ready to execute when one is completed. This means that a volunteer may have multiple redundant jobs in the pipeline.

Volunteer Pulse

If the server can detect whether a volunteer has left the computation, then eager scheduling is not essential to ensure reliability. I therefore implemented volunteers to periodically send a pulse to the server. Not only does this inform the server whether a volunteer is still active, it provides information to the server and potential users about the number of available volunteers, and the average time that a volunteer is connected. This information could be used by the server for
example, to dynamically modify its scheduling strategy, and by users to adjust the decomposition of their applications.

The disadvantage of the pulse is that it introduces another network operation, and has the potential to congest the server. Therefore, the interval between pulses can be dynamically adjusted by the server – the response to each pulse by the server indicates the desired pulse interval. The server tries to keep this short, such that a volunteer failure can be quickly detected, but long enough not to inundate the server and network with pulses from many volunteers. Other messages can be relayed with the pulse, such as the status of the jobs currently held by the volunteer, and whether they are still required to be executed. This pulse strategy can still lead to the redundant execution of jobs, because an intermittent network problem may prevent some pulses from reaching the server, leading to an erroneous assumption by the server that a volunteer has failed.

**Modified Eager Scheduling**

The G2 scheduler does not have data on the relative power of each volunteer, and does not have data on the relative complexity of each job, so sub optimum job allocations are inevitable. By eagerly assigning a job to multiple volunteers, the effects of inappropriate job allocations are mitigated. For example, one that allocates a long running job to a computer, when there are much more powerful computers available.

Even if the scheduler had complete information on the relative power of each volunteered computer and the complexity of each job, sub optimum job allocations are still possible because at any time, a volunteer could be executing local applications as well as executing jobs, splitting system resources between them. This diminishes the utility of such information.

For the G2 scheduler, I do not try to ensure optimum scheduling, rather, I would like to make sure all available volunteers are working. This mitigates the negative effects of volunteer termination, sub-optimum allocation of long running jobs to
less powerful volunteers, through the redundant execution of jobs. Also, a job should not be assigned to more than one volunteer if there are other jobs that have not been assigned at all. I altered the behaviour of the basic eager scheduler to introduce a minimum delay between the time a job is assigned, and when it can be reassigned. The duration of the delay is currently user configurable but it is ideally tuned by the server. The delay should be sufficient to allow a client to submit jobs, one at a time, without a job being assigned more than once while subsequent jobs are unassigned. This can still lead to redundant allocations, but such allocations can be corrected by using the pulse mechanism to send control messages to the volunteer. The pulse message is a web service request, so the server can send messages to a volunteer in the reply. One such control message forces a volunteer to abandon a job, and to fetch a new one. The combination of the minimum delay between allocations of a particular job, and control messages via the pulse mechanism, are used to reduce the amount of wasted work performed by volunteers.

**Duplication for Integrity**

Duplication is not always undesirable, and is widely used for integrity checking. SETI@Home ensures correct results by ensuring that each work unit is executed a number of times by different users, and such duplication can be used to detect saboteurs [55]. G2 takes advantage of any duplication to check the correctness of submitted results, and users can specify the amount of duplication desired if any. Integrity checks and sabotage detection are outside the scope of this research, but I try to take advantage of duplication when it occurs, by comparing the results of jobs.

**Server Reliability**

The centralised server approach as used by G2, simplifies the detection and recovery of errors that occur on the server. The principal function of the server is to act as a clearinghouse for jobs and their results. This takes the form of a database stored in non-volatile memory, which prevents data loss during any event that
causes the server to fail. The basic operations on the server, accessed through web methods, relate to jobs and therefore alter the state of the database. The operations on the database are performed synchronously, and as transactions, which can easily be rolled back in the event of an error. These operations are typically synchronous, so that clients and volunteers get immediate feedback if an error occurs and can respond accordingly. The server is otherwise stateless, and a restart of the server will never result in the loss of critical data, or cause the database to enter an inconsistent state. If a server crashes, or restarts, clients and volunteers will automatically reconnect to the server, once it comes back on-line.

4.2.3 Volunteer Checkpointing

Earlier, I mentioned that the communication to computation time ratio of jobs affects the amount of speedup provided by a cycle stealing framework. So, by increasing the execution time of jobs, a user can increase the performance of their application by adjusting the computation time to communication time ratio in their favour. However, the ability to enhance performance in this way is limited.

An important principle of cycle stealing is to minimise interference with the donor’s use of their computer. Ideally, a donor should be able to connect their computer to a cycle stealing system and donate cycles, without any negative consequence to their use of the computer. That is, the execution of their applications should not be compromised by the operation of the cycle stealing volunteer component. To adhere to this principle, a common approach is for the volunteer component to be active only when its host computer is unattended, and the volunteer immediately ceases activity when the donor interacts with their computer. This approach, known as immediate eviction, ensures that the volunteer does not contend for resources while donors are using their computer. However, immediate eviction has the greatest potential for lost or wasted work, because any work performed to complete the execution of a job is lost when “eviction” occurs. Although a G2 volunteer may be configured to continue to execute jobs despite
coming out of the idle state, I have assumed that volunteers are configured with an immediate eviction policy.

A consequence of immediate eviction is that it forces the execution of a job to be completed within one continuous idle period. This introduces the possibility that long running jobs will not be successfully executed, which limits the way a programmer can adjust the computation to communication ratio to their advantage.

One way to work around the limitation imposed by immediate eviction is through the use of checkpoints on volunteers.

In this context, a checkpoint is the saved state of a job at a certain point in its execution, from which the execution of the job can be resumed at a future time. Therefore, checkpointing enables the execution of a single job to span multiple volunteer idle periods.

For cycle stealing systems that do not have an immediate eviction strategy, checkpoints may still be necessary to ensure application progression. Volunteers can still be powered down, which will terminate a cycle stealing period. Where a cycle stealing system supports very long running jobs, such as SETI@Home, checkpoints play an important role in ensuring the execution of jobs.

Given its benefits, I decided to investigate a checkpointing mechanism for incorporation into G2. The three main issues that arose were:

1. Where to store the checkpoint data
2. When to checkpoint i.e. the frequency of checkpoint generation
3. How to create checkpoints (The representation of checkpoints)

1. **Checkpoint Storage**

Once a checkpoint is created, it needs to be stored and this can be either on the machine that created the checkpoint, or at some remote location. In the case of G2, the volunteer host component that creates the component is assumed not to have access to the local disk and cannot store checkpoints locally. Therefore,
checkpoints are transmitted to a remote location, which is logically the G2 server. The requirement to transmit checkpoints to the server, and for volunteers to receive checkpoints to restart jobs, adds to the communication cost of jobs, so checkpointing is not a silver bullet solution to performance. In fact, checkpointing may be prohibitive if the size of the checkpoint is relatively large compared to the capacity of the network link between volunteers and the server.

2. Checkpoint Frequency

On one hand, it is desirable to increase the checkpoint frequency to decrease the amount of work lost when the volunteer is evicted and disconnects from a cycle stealing network. But, if the process of checkpointing is expensive, in terms of CPU or communications cost, then there is a conflicting interest to checkpoint less frequently. The optimum checkpoint frequency also depends on the behaviour of the volunteer when it is reclaimed by its owner. As stated earlier, G2 has an immediate eviction policy, but other systems have a pause and migrate policy, which creates a checkpoint and stores it before terminating. Cycle stealing systems with such a policy should not have to checkpoint as frequently as systems with immediate eviction, because the eviction process does not result in a loss of work.

3. Checkpoint Generation

A checkpoint is any type of data from which a computation may resume: there are various forms that checkpoints may take, and various ways in which they are created. A checkpoint may take the form of a set of parameters: the value of these parameters are output by the function as a checkpoint, and when used as input to that function, enables a function evaluation to resume. Checkpoints in Parabon [35] take this form. A checkpointing API is provided where the programmer provides the executing state that they wish to preserve, and they write functions that take this state as input. BOINC [6] provides a checkpoint mechanism that writes the checkpoint (state) to the local disk. According to the BOINC API [56], “The state file must include everything required to start the computation again at the
same place it was checkpointed. On startup, the application reads the state file to determine where to begin computation.” As highlighted by the authors of CX [28] a continuation is also a form of checkpoint. The work to be performed by a job can be written as a series of continuations, where each continuation represents the state or phase of job execution. A more complex option is to save the complete executing state of a process, including stack and thread state, into a form that can be stored, recreated and resumed. Condor [57] supports this kind of checkpointing.

With the exception of Condor, these checkpoint mechanisms are examples of manual checkpointing, because the onus is on the programmer to create checkpoints, and to write their applications and jobs in such a manner that supports resumption from checkpoints. Manual checkpointing can be complex, and the use of ordinary programming constructs such as loops can become tedious if the programmer has to deconstruct the loop to checkpoint, and recreate it to resume from a checkpoint. The differences between the computers in the volunteer pool, in terms of their idle times and performance, means that there may be wide variations between volunteers in the amount cycles that they can contribute. A manual checkpoint mechanism requires the programmer to tune their checkpoint frequency to account for this diversity. However, it is difficult for programmers to control the checkpoint frequency, as checkpoints are created not by specifying the elapsed time between checkpoints, but by specifying points in the code where checkpoints are to be created. The checkpoint frequency becomes dependent on the power of the volunteer’s host computer, and how quickly these points in the code are reached. A programmer may also need to include checkpoint calls in different parts of their code, which raises the difficulty of creating checkpoints where the type, as well as the value of data, depends on the checkpoint location.

**4.2.3.1 Automatic Checkpointing**

The difficulties raised by manual checkpointing can be overcome by automatic checkpointing mechanisms. Automatic checkpointing is the capability of a run-
time system to create checkpoints without programmer intervention. Automatic checkpoint mechanisms typically store the complete, reproducible state of an executing job, because no programmer supplied cues are available, and it is difficult to automatically ascertain what data is important and what can be discarded. The advantage of automatic checkpointing is that a checkpoint can be created at anytime, regardless of the point of execution of the job, enabling the runtime to determine checkpoint frequency, and freeing the programmer from the task of handcrafting a checkpoint.

Condor has an automatic checkpoint capability that serializes the state of a job and moves it onto the client computer, or to a checkpoint server if available. These checkpoints may be many megabytes in size, so the costs of transmission of checkpoints can be significant. Condor generally supports native executables as well as Java applications, but checkpointing support is only provided for native applications compiled with the Condor Checkpoint libraries [51].

Automatic checkpointing for Java applications is non trivial and is the subject of much research as surveyed in [58]. The existence of the virtual machine, including undefined stack and heap formats, and JIT compilation, introduces complexities that are not present when executing native binaries. Although G2 does not use Java, the use of .NET and its VM, called the Common Language Runtime, made it worthwhile to investigate Java checkpointing approaches.

Attempts to include persistence in Java have taken one of three approaches [58]. The first, is to provide libraries that are used to specifically compile, or to post-process and rewrite applications to implement support for checkpointing. The benefit of this approach is that it requires no modification to the Java language, or to the Java virtual machine. The disadvantage is that they do not support reachability (not every referenced object is checkpointed), type-ortogonality (persistence is not available to all objects regardless of type), or thread persistence.
The second approach is to modify the JVM such that the state of the job, including the thread state is easily accessible and preserved. The disadvantage of this approach is that checkpoints can only be supported on those volunteers running on a modified JVM which will limit the volunteers where checkpoints are available.

The third approach is to run the JVM on an OS that supports persistence. In this scenario, the entire JVM can be persisted. Again the disadvantage lies in limiting the number of potential volunteers to those running the correct OS, in addition to having to checkpoint the entire VM.

**Automatic Checkpointing for G2**

From a programmer perspective, automatic checkpointing provides a simple means for job execution to survive volunteer termination. I wanted to avoid manual checkpointing because it is too complicated, and a form of manual checkpoint is already supported through the use of continuations. Automatic checkpoint capability was therefore incorporated into G2.

My approach is similar to the first Java persistence approach mentioned above. In particular:

- I leverage an existing mobilization framework [59] to add checkpointing and job migration features to G2. This mobilization framework operates on top of the standard .NET runtime.

- A mobilization tool automatically generates mobile versions of the user’s classes.

- I incorporated the mobilization framework directly into G2 volunteers which frees programmers from having to handle checkpoints and allows dynamic tuning of checkpoint frequencies.

This work was performed in conjunction with Dominic Cooney, whose agent mobilization framework provided the basis of G2 checkpointing. This pre-
existing mobilization framework and some of its runtime functionality was incorporated into the G2 volunteer component.

The first step in the mobilisation of applications is to suspend an executing application and convert it into a form that can be migrated. This form is essentially a checkpoint because the execution of the application can be resumed from the point of suspension after it reaches its migration destination. Therefore, the processes involved in the mobilization of applications can be used to enable checkpointing in a cycle stealing system.

To “mobilize” a G2 volunteer application, the programmer implements a simple serialization interface (modelled on and identical to the standard .NET serialization interface and shown in Listing 4-1) which assists the framework to create the checkpoint. A custom serialization interface is required to work around the security constraints imposed by the standard .NET serializers, and helps to ensure that unnecessary data is not included in the checkpoint, which minimises checkpoint size. It is important to note that the custom serializer does not compromise the security of volunteers.

```java
[Atomic]
public void GetObjectData(PickleWriter p)
{
    p.Write(number);
    p.Write(result);
}

[Atomic]
public void SetObjectData(PickleReader p)
{
    number = (int) p.Read();
    result = (int) p.Read();
}

Listing 4-1: Implementing the custom serialization interface
```

This code snippet demonstrates the custom serialization interface. The use of the “atomic” custom attribute signals to the mobilization framework that execution should not be suspended (and a checkpoint created) during the execution of these methods. This interface is similar to standard .NET serialization interfaces.
The mobilized version of the application is then generated, when the programmer uses a tool to convert their application to create a *mobilized* version that is compatible with the G2 checkpoint mechanism. This mobilized version is the version that is transmitted to volunteers. For programmers, this is all that is required to enable checkpointing in their application.

Adding support for automatic checkpointing to G2 required small modifications to the standard G2 volunteer host component. Jobs in G2 consist of a method execution of a user defined class. If that class incorporates the custom serialization interface, then this will be detected by a volunteer, which will attempt to create checkpoints during job execution when appropriate. As programmers do not specify when checkpoints are to be created, G2 volunteers have a checkpoint period that controls the interval between checkpoint generation. At present, this period is static, but it could be dynamically adjusted to suit the volunteer and application.

After a checkpoint is created, it is uploaded and stored on the G2 server. This makes the checkpoint available to other volunteers, any of which could download the checkpoint and resume execution of the job.

A checkpoint consists of a serialized object, and the data needed to restore the execution of a job: local variables, evaluation stack and point of execution. This data is known as the object’s *mobilization context*. The checkpoint is packaged into the form of a G2 job, so the server’s basic mechanism to receive and distribute jobs was not changed to accommodate checkpoints.
When a volunteer retrieves a job that is a checkpoint, the serialized object is de-
serialized, its mobilization context is recreated, and job execution is resumed. This design simplified the integration of checkpointing into the existing frame-
work, reducing the changes needed to the volunteer component and the server. The server does draw a small distinction between ordinary jobs and checkpoints. When a checkpoint is submitted, it is labelled as a job that has already been as-
signed to the submitting volunteer, (which is presumably still executing the job after submitting the checkpoint). Therefore, the checkpoint will not be assigned to a volunteer, if there are other jobs that have not been assigned.

The G2 server may assign the same job to multiple volunteers, which raises the possibility that checkpoints originating from the execution of the same job, but from different volunteers, can be submitted to the server. As the volunteers have differing performance, and as checkpoint frequency is dependent on wall clock time, and not processor cycles, these checkpoints will represent the execution of the job at different stages. For the sake of efficiency, I would like the server to keep the checkpoint from the most advanced execution and discard the rest. Recall that for programmers to make their applications checkpoint compatible, a mobilization tool was used to insert code into their compiled applications to make them checkpoint friendly. For G2 checkpointing, the mobilization tool also inserts code at each branch point that increments a virtual program counter. This counter is read by the volunteer host, and its value is submitted to the server when the checkpoint is submitted. The server uses the counter to estimate the most advanced checkpoint\textsuperscript{15}. If a volunteer attempts to submit a checkpoint that represents a less advanced point of execution than a checkpoint already submitted, or if a result to that job exists, then the server will reject that checkpoint.

\textsuperscript{15} The virtual program counter can only be used to estimate the most advanced execution, because the execution of a job may be non-deterministic. In that case the checkpoint with the highest counter value, may not represent the most advanced point of that job’s execution.
As the submission of a checkpoint occurs through a web method, the server uses the reply to inform the volunteer whether or not it should continue to execute the job. If the server rejected the submitted checkpoint, then the volunteer will abandon execution of the job.

**High Throughput Computing with Checkpoints**

Condor is one of the more successful cycle stealing systems as it is being used by a large number of users. The aim of the Condor project is High Throughput Computing, which is to provide users with “large amounts of processing capacity over long periods of time”[52]. For users, the difference between G2 and Condor is that through the execution of parallel applications, G2 aims to minimise the wall-clock time of applications, that is, the total time that an application executes. In contrast, Condor and HTC is concerned with maximising the throughput of multiple sequential applications, over minimising the wall-clock time of a single application. For that reason, Condor is described as a batch processing system. A typical usage scenario in Condor might be that a user has multiple sequential applications that they wish to execute. Instead of executing them one at a time on their own machine, they can have their applications executed on the Condor pool. The advantage is that the set of applications will be executed faster using Condor, not that the execution of any particular applications will be faster. This is not to say that a user cannot achieve high performance from Condor. Condor has numerous extensions including an API called MW [51]for writing Master-Worker style applications. In addition, a user can always decompose an application into a series of applications and aggregate the results, but in my opinion, G2 is a much simpler framework for parallelising applications.

With the integration of checkpoints into the G2 framework, high throughput computing becomes possible. Checkpoints are necessary, because the execution time of applications that users are likely to submit will typically span more than a single volunteer idle period. Programmers still need to write an application, or a wrapper to an application, that can be executed as a G2 job. That is, it has at least
one web method. That application will be executed as a job and a result returned when it has been completed.

**Checkpoint Sample**

The following code sample in Listing 4-2 is a function that is executed as a job, and is supported by automatic checkpointing. Although this Fibonacci number generator is hardly a prime candidate for parallelisation, it requires the automatic checkpoint mechanism to handle recursion, and shows that programmers do not need to insert checkpoint related instructions into their code.

```csharp
[WebMethod]
public int Fib(int n)
{
    if (n == 0)
        return 0;
    else if (n == 1)
        return 1;
    else
        return Fib(n-2) + Fib(n-1);
}
```

**Listing 4-2: Recursive Fibonacci (Checkpointed). This is no different to an ordinary Fibonacci routine.**

Ordinarily the job will execute until the \( n \)th Fibonacci number is calculated, but with automated checkpointing and using a short checkpoint frequency, the calculation of the \( n \)th Fibonacci number is periodically interrupted and a checkpoint is created. (In the current implementation, the volunteer framework creates a continuation during this checkpointing phase which is submitted as a new job onto the server.) When the \( n \)th Fibonacci number is eventually calculated, the result is returned to the original client and the application terminates.

### 4.3 Safety

Safety is the ability of the framework to protect the volunteer host machine from potential harm caused by the execution of jobs. Ideally all jobs will be well-behaved, and their execution will have no residual side effects on the volunteer host. In practice, this ideal situation will not exist, and it is inevitable that by accident or by intent, some jobs will exhibit behaviour that is damaging to volunteer host machines. To ensure their participation, volunteer host machines must
be protected and their owners assured that participation in cycle stealing is harmless.

Ensuring safety is a requirement for all cycle stealing systems, but the level of sophistication required to protect volunteers will vary from system to system. Consider an application specific system such as SETI@Home. The developers of SETI@Home provide the jobs that are executed by volunteers, and can test to ensure that no harm results from their execution. A person who donates their computer to SETI expects that the process is safe, but does not expect SETI@Home to provide sophisticated runtime security during the execution of a work unit. The user does not expect such security, because they have a reasonable expectation that the people running the SETI@Home project will not harm their computers.

In contrast, general purpose frameworks enable anybody to write and deploy cycle stealing applications. For G2, donors may have no knowledge of the nature of the jobs that their computers are executing, nor the producer of those jobs. I have assumed that jobs will behave maliciously, either by accident or design, and the volunteer component must have facilities to protect its host. One way to provide such security is through the use of managed execution environments, such as those provided by Java and .NET. These environments execute code originating from untrusted sources in a sandbox. For G2, this means that jobs do not have access to anything that can potentially harm the host computer. The alternative to managed execution, that is, allowing the execution of native code, typically involves security restrictions such as limiting only authorised users to submit jobs, and pre-executing a sample of submitted jobs to test for errant behaviour. This type of verification is time consuming and often requires human intervention. Managed execution removes the need for manual testing and provides protection for donated computers, from any job submitted by any user.

One criticism levelled at managed execution environments such as Java and the .NET CLR, relates to their performance compared to the execution of native code. This issue is particularly relevant for high performance computing. Some bench-
marks show that for sequential, numerical applications, the performance of the JVM and the .NET CLR is comparable to native code implementations [60, 61]. Sequential benchmarks are relevant for G2 as I anticipate that jobs will execute sequential user code most of the time. For cycle stealing, I believe that the runtime security provided by the .NET CLR, and the ability to develop applications using a number of languages, outweigh any performance disadvantages of managed execution.

A potential security feature is to enable donors to execute jobs only for users that they trust. The G2 server hosts a web site that enables users and volunteer owners to input their details which enables individuals to track the number of jobs they have submitted or executed. The site also displays jobs, job classes, and the users who are currently using the system. This website could be used by donors to choose which class of jobs, and whose jobs, they are willing to execute. When the volunteer host component tries to fetch a job, only those jobs allowed by the donor will be assigned to the volunteer.

### 4.4 Performance Evaluation

A consequence of the client-server architecture of G2 is that the server is a potential bottleneck, particularly when a large number of volunteers are present. Earlier in this chapter I described the features that were incorporated into the G2 framework to improve its efficiency. Here I describe the experiments that I performed to demonstrate that a G2 server is capable of providing high performance using a non trivial number of volunteers. The experiments consisted of the execution of two applications using two different pools of volunteers.

The first application was a TSP (Travelling Salesman Problem) solver using a genetic algorithm [62] over a tour size of 1000 cities. In this application, each job represents a population which is evolved for some number of generations. The result of each job is an evolved population. This evolved population, is combined by the client, with another evolved population (which was received as the result of
another job) and the resulting new population is then incorporated into a new job. Thus, the client is continually submitting jobs and fetching results from the server.

The volunteer pool used for this experiment consisted of up to 50 Pentium 4, 2.6 GHz computers each with 1 GB of RAM running Windows XP. The server used in all experiments had dual Pentium 3, 1.2 GHz processors with 1GB RAM, running Windows Server 2003. The client machine had identical specifications to the volunteer machines, and did not function as a volunteer. The client, volunteers and server were all connected to a 100 Mbit Ethernet network, where the client and volunteer machines were all located within a single room. The server was in a separate physical location with a 1-Gigabit optical fibre link connecting the server and client/volunteer Ethernet networks. This is a shared network and may have been in other use during the execution of the experiment.

For the TSP application, the size of a job and its result are identical. Each job is about 128 Kbytes in size and executes in approximately 160 seconds. These job characteristics were appropriate given the size of the volunteer pool. I chose these characteristics so that the transmission of jobs and results between the volunteers and the server would not result in the formation of bottlenecks. A user of a G2 system would have to make similar decisions when configuring their own application. However, these jobs are not trivial; the execution times are not so long as to make requests from volunteers to the server negligibly infrequent: there is some load on the server. The jobs are also short enough that they could be executed to completion when volunteered computers are idle, possibly during office hours, when I assume that idle times will be relatively short. The data size of jobs and results are such as to require a communication cost that is realistic given the computation time of each job. The computation to communication ratio is favourable, but not unrealistically biased to guarantee good performance.

I executed a sequential version of the TSP application using the machine designated as the client, to obtain a single machine benchmark to which the parallel
G2 version would be compared. Figure 4-4 graphs the speedup obtained by executing the parallel G2 version on up to 50 volunteers.

![Graph showing speedup with up to 50 volunteers.](image)

**Figure 4-4: TSP performance compared to linear speedup.**

The volunteers in this pool were dedicated G2 volunteers, that is, they were not used for any other purpose for the duration of the experiment. Volunteers were simulated to be volatile; a controlling process would terminate a volunteer then immediately restart it. The average idle time, (the average period between restarts) was configured to be thirty minutes. This period is relatively long compared to the execution time of each job; hence I did not expect volatility to have a significant effect on speedup.

The second application was a ray tracing application which rendered a scene containing a lattice of over 8000 spheres. A sequential version of the ray tracer rendered the scene in about four hours and twenty minutes using a Pentium 3 Xeon 2.4 GHz computer with 2GB of RAM. Although this computer was a dual processor machine, only one processor was used. The volunteer pool used to execute a parallel G2 version of the ray tracer consisted of 38 Pentium 4, 2.6 GHz machines,
and 22 Pentium 4, 2.0 GHz machines, all with 1GB of RAM. I executed the client component on a Pentium 4, 2.0 GHz machine; this machine did not also serve as a volunteer. The client and volunteers were all wired to a 100 Mbit Ethernet network, but again, an optical fibre link separated the client and volunteers with the G2 server.

The parallel ray tracer separated the scene into small blocks, which were rendered using the sixty available volunteers. Each block took on average, a little over three minutes to render, and contained about 200 Kbytes of data. Twenty such scenes were rendered producing twenty frames of “animation”, although a rather uninteresting one as the camera and objects were stationary. The render time was one hour, thirty-six minutes. Compared to the four hour, twenty minute render of one frame using one machine, the parallel execution provided around a 54 times performance benefit.

A screenshot of the TSP application is shown in Figure 4-5; a screenshot of the ray tracing application is shown in Figure 4-6.
Figure 4-5: TSP 1000 city tour
One feature of the travelling salesman application is that its parameters can easily be adjusted to create jobs with the desired execution time or input and output data sizes. Using a volunteer pool consisting of up to 32 volunteers, I executed the travelling salesman application over a range of execution times, input and output data sizes, and volunteer numbers. The experimental conditions, such as the specifications of the computers and their network environment were the same as those described above for the earlier TSP experiment. However, the average idle time of volunteers was lowered from thirty to eight minutes. An eight minute idle time is comparable to the job execution times that I tested and illustrates the effect that volunteer volatility can have on performance.
Figure 4-7: TSP speedup over a range of job execution times

Figure 4-7 graphs the observed speedup of the TSP application using 32 volunteers. The left side of the graph shows that speedup, compared to a sequential version of the TSP application, increases with the execution time of jobs. This is due to a more favourable computation to communication time ratio as execution times are increased. However, lengthening a job’s execution time leads to a decrease in the probability that the job will be completed within a volunteer’s idle period; this can result in a gradual decrease in overall speedup, which is evident as the speedup curve plateaus and then declines. The amount of performance loss due to volunteer termination depends on the relative duration between job execution times and volunteer idle periods. As Figure 4-4 showed, if the average volunteer idle time is relatively long compared to job execution time, (30 minute idle time compared to an approximate 3 minute job execution time) the effect of volunteer termination is negligible.
Figure 4-8 graphs the speedup obtained as job input and output data size was increased. Again, 32 volunteers were used with an average idle period of eight minutes. For this experiment, I set the job execution time to be one minute, as a short execution time would highlight the negative effect on performance due to an increase in job data: a short execution time ensures that the server becomes overloaded as the frequency and size of messages increase. As job execution time was set to be about one minute, an eight minute idle period was not a significant factor in determining the speedup obtained. As expected, speedup suffers as the data costs associated with distributing jobs increase. When I performed this experiment, I observed a small increase in performance as the job data size was increased from 90 to 128 Kbytes. This anomaly could be due to a number of factors, such as changing network conditions, as the experiment was performed on a shared network, or the random occurrence of volunteer terminations.
Large job and result data sizes are detrimental to achieving speedup in two ways. The first is latency, that is, that the larger the data size, the longer it takes to transmit the job or result. The second is the contribution to network congestion and the formation of bottlenecks on the server. Figure 4-9 graphs the speedup obtained from the TSP application using different sized volunteer pools. The input and output data size is 128 Kbytes and the job execution time is about 20 seconds. Up to around twenty volunteers, speedup is respectable and comparable to linear. This indicates that the job and result data sizes did not cause communication latency that was detrimental to speedup on this network. However, as more volunteers were added, the effect of network congestion, and potential server bottlenecks, resulted in a falloff in speedup. The cause of this falloff is due to the communication from multiple volunteers potentially occurring in bursts contributing to bottlenecks, and the limitations of the underlying network infrastructure. The short execution time of jobs coupled with their proportionally large data re-
quirements, makes the effect of congestion apparent even with a relatively small number of volunteers.

4.5 Related Work
Cycle stealing frameworks have features that reflect the goals of their designers. As such, G2 has a set of features that make it easy to use, deployable over the Internet, and capable of supporting applications where job latency is important, because these were my design goals. The approaches that I have taken to enhance the performance, reliability and safety of G2, reflect these goals, and may not be applicable to all cycle stealing frameworks. Similarly, other frameworks designed with goals different from G2, have features that are not necessarily suitable for incorporation into G2. In this section I describe different systems and their techniques for achieving performance, reliability and safety, as these concepts apply to their intended patterns of usage.

4.5.1 Volunteer Idle Period
A volunteer executes jobs, when it is otherwise idle. This raises three issues.

1. What does it mean for a volunteer to be idle?
2. How does the framework detect the idle condition?
3. How does the framework enforce execution only during the idle period?

The definition of “idle” that I have chosen is that a computer is idle when its owner has stopped physically interacting with it for some user defined period. This is the definition of “idle” that a screensaver application uses. The benefit of this definition is that it is the most potentially unobtrusive behaviour of the volunteer component. In addition, enforcing execution only during an idle period is a responsibility that can be delegated to the Operating System. There is still a potential effect on the volunteer owner’s applications because a person can be using their computer without using input devices. For example, they may be running applications that don’t require continuous human intervention, including applica-
tions that not only use the CPU but make use of the network. I don’t take these into consideration, because web based delivery means that the G2 volunteer component will usually execute in a restricted security environment where it cannot detect the presence of other running applications.

However, other frameworks have different approaches to a volunteer's idle state.

**Continuously Executing Low Priority Volunteers**

Many existing cycle stealing systems have jobs that execute over many hours or days, for example, SETI@Home and Folding@Home. For these systems, simply restarting a job whenever the volunteer terminates is not a realistic option. The volunteer component for these systems can be configured in a mode other than screensaver mode. This additional mode causes the volunteer to always execute as a low priority process. This enables the volunteer component to exploit the spare cycles of a computer with much finer granularity than if it were to execute jobs only when the computer was unattended. Responsibility for detecting and enforcing idle operation is again delegated to the Operating System. As a low priority process the host OS will schedule CPU time to the job whenever higher priority processes are not consuming cycles. The job is therefore continuously executing, using whatever scraps of CPU time it can gather. In most circumstances, the user of the workstation will not notice that cycle stealing is occurring. This low priority job approach decreases the frequency of volunteer interruptions because a workstation owner reclaiming the machine is not longer considered to be an interruption and source of volunteer failure. Naturally, the volunteer component has memory and network requirements that may adversely affect other applications.

**Graceful Termination and Linger Longer**

In graceful termination and linger longer schemes [11], a computer is considered to be idle, in the same way that G2 considers a volunteer to be idle. However, the volunteer does not immediately cease execution when the donor returns to their
computer. In a graceful termination scenario, the volunteer is given time to stop execution and transmit a checkpoint to the server or another volunteer. Linger Longer is a scheme where a volunteer immediately ceases execution when the computer is reclaimed, but the job lingers for a brief period of time in the hope that the computer will become idle again. If the donor stops using their computer, then the job is free to resume execution. Otherwise, the job immediately or gracefully terminates. Linger longer is intended to prevent unnecessary terminations of jobs when the workstation is used only for brief periods by its owner. These schemes have been shown to provide better performance to cycle stealing applications, but they can have a noticeable effect on the donor’s use of their computer, for example, they will not release memory when the volunteer is reclaimed, possibly resulting in the poorer performance of local applications.

4.5.2 Minimising the Cost of Remote Job Execution

Earlier, I showed how pipelining can be used to improve performance. Cycle stealing frameworks where volunteers can hold multiple jobs simultaneously, such as Cilk [19] and Jicos [7] have a pipelining capability. Another way to improve performance is to minimise the cost of executing jobs on volunteers.

The cost of jobs can be broken down into three parts: The first is the preparation of the job by the client. The second is the communication of jobs and results, including any overhead imposed by an entity that schedules and assigns job to volunteers and returns results to clients. The last cost is the execution overhead, which is any cost incurred on volunteers associated with the execution of a job. For example in the G2 context, the first cost is the need to decompose an application and serialize a job. The second is the physical communication of jobs and results over the wire and the server, which relays jobs and results to client and volunteers. The third is the serialization and deserialization costs of job execution incurred by volunteers.
Preparation of Jobs

The preparation of jobs covers all of the steps required before jobs can be generated and executed on volunteers. The client must first decompose the problem into a set of jobs, and create jobs in the format recognised by the cycle stealing framework. The work performed by each job depends entirely on the type of application, and is completely at the discretion of the programmer. The format of the job however, is determined by the framework and is influenced by the framework’s communication mechanism.

Job Content

A G2 job consists of an XML serialized string encoding a method name and the parameters of that method. As explained earlier, the job does not contain the actual implementation of the method. This decoupling is similar to the approach taken by application specific cycle stealing frameworks. For example, the volunteer component of SETI@Home is a piece of software that contains the code for the processing of radio telescope data. The jobs themselves consist of the raw telescope data to be analysed.

The other approach is to couple the method implementation with the job itself. This is a common approach taken by many frameworks such as Bayanihan and Parabon. The advantage of this approach is that everything that a volunteer needs to execute a job is encapsulated in the job. There is no need for predeployment of code and other data, and it eliminates potential problems that may arise if volunteers receive a job that it does not have the code to execute. The disadvantage of this approach, and the reason I have not adopted it for G2, is that it can be inefficient, because the same instructions (assemblies) are continually being transmitted to the same volunteers. The G2 approach causes volunteers to lazily download the code as required. Since this code is cached on the volunteer, under normal conditions, it is downloaded only once. By decoupling the job implementation from the job parameters, the overall size of the job payload is decreased. This leads to faster transmission of jobs, from clients to volunteers.
Job Payload

As previously noted, the contents of the job are XML encoded. In reality, there are two phases of XML serialization. The G2 framework performs an inner serialization which serializes the job into a single string. This assists the storage and transmission of jobs because it causes all jobs, regardless of origin, to adopt a uniform structure. As standard XML serialization may not be optimal, the framework provides hooks to which programmers can provide their own serialization schemes. The outer layer of serialization is performed by the web service proxy, which encodes the job using SOAP as required by web services. This outer layer of serialization is quite simple, requiring the SOAP serializer to simply insert string tags around the G2 serialized data. XML serialization can be verbose, but I have chosen to use it for G2 because the .NET framework, under its default security settings, permits XML serialization and deserialization in code received over the Internet. In contrast, binary serialization for example, although more compact, is not permitted by default.

Other systems encapsulate jobs as required by their underlying communication mechanisms. Some are more lightweight than web services, for example HORB [31], as used by Bayanihan [8], Java RMI as used by Knitting Factory, and Jini by CX [28]. These are more efficient than XML web services, but do not have the cross platform abilities of XML web services, and communication via HTTP. At the time of writing, an implementation for HORB for .NET exists but is not publicly available.

Distributing Code

Where the volunteers download method implementations (assemblies in our case), separately from jobs, the files containing the implementation must be made available to the volunteer. Systems such as Jicos rely on the client to serve this data to the volunteers directly. This raises two issues. The first is that the volunteers must be capable of actually connecting to the client, the second is that the client must be capable of handling requests from potentially many volunteers. In
the case of Jicos, the client must be running a web server of some description, since code is distributed via HTTP. G2’s approach is to shift the burden of distributing code to the server, which is typically much more capable of accomplishing this job compared to the client, and eliminates problems if clients are shielded by firewalls.

**Communication of Jobs**

Apart from the throughput and latency of a network, the communication protocols determine the efficiency of job communication. G2 relies on XML web services so incurs some overhead imposed by SOAP and HTTP. Various frameworks use Java RMI (For example, the projects under the MILAN Project (Metacomputing In Large Asynchronous Networks) umbrella, such as Knitting Factory, and Charlotte) which can be more efficient than web services. Bayanihan [8] uses HORB which claims to be lightweight and efficient.

**Server Overhead**

A mechanism that relays jobs from client to volunteers contributes to job overhead. In G2, the server as the intermediary between clients and volunteers, is such an overhead. After a job is submitted to the server, the server deserializes the SOAP encoded job, and then inserts the job into its database along with certain tracking information. When a volunteer tries to fetch a new job, the server searches the database for an available job, encapsulates it within a SOAP envelope and transmits it to the volunteer. The volunteer then remove the SOAP envelope and deserializes the job. All of these steps are part of the communications cost required to send jobs from clients to volunteers. The existence of the intermediate server adds an extra layer of SOAP serialization and deserialization, which would not be necessary if the volunteers connected directly to clients. Other systems, such as Javelin and Bayanihan, also rely on a central server, or broker, to serve jobs to volunteers.
A central server can become a bottleneck and stifle performance by preventing the cycle stealing system from scaling. It is therefore useful to have a distributed means to distribute jobs. One way to achieve this is to have a network of servers. This is an approach taken by Javelin++ [26] among others. In Javelin++ there is a network of brokers which are able to share jobs among themselves. Cilk-NOW and all of Javelin’s successors, Javelin++, Javelin 2.0 [27], CX and Jicos support work stealing, which is a scheme where volunteers maintain large queues of jobs, and when a volunteer exhausts its local job queue, it can directly connect to another volunteer and “steal” unexecuted jobs. For these work stealing systems, as the entity that distributes jobs is not in one location, tracking and scheduling of jobs becomes difficult. A solution to this problem used by Javelin++ is to have the client maintain a heap of job allocations, which is updated whenever results arrive.

**Execution Overhead**

The execution overhead is any of the cost, incurred by a volunteer, associated with the actual execution of a job. In G2, the need to deserialize and serialize job and results is an example of an execution overhead. Existing general purpose cycle systems predominantly use Java, and require users to supply jobs that execute on a JVM. The advantage of this approach is that heterogeneous volunteers are easily supported, and volunteers can be protected by executing jobs within a sandboxed environment. The use of a managed environment can however, constitute an overhead, if the parallel application is being compared to a native code sequential version. G2 makes use of the .NET framework which is a managed environment and has the benefits and drawbacks of the JVM in terms of execution overhead.

The ability to execute native code does not mean there is no execution overhead, because security procedures may have to be performed before code can be executed. XtremWeb [63], for example, has a five part verification process to protect volunteers. First, only trusted clients can submit jobs. Second the jobs are tested
on dedicated volunteers. Third, the jobs are encrypted before transmission to volunteers. Fourth, the job transport procedure uses a private-public key to secure the transaction. Fifth, the worker sends back a checksum of the code to the server which verifies its correctness.

**4.5.3 Broadening the Volunteer Base**

One way to increase performance is to maximise the number of volunteers. The approach I have taken with G2 is to make it a web based system. As long as a computer can connect to the web, (and has the .NET framework installed), it is potentially a G2 volunteer.

Other projects chose Java as the platform for their cycle stealing frameworks. Java has the advantage that implementations of the Java virtual machine exist for many different hardware platforms. In practice, this allows any machine running the JVM to act as a potential volunteer. Apart from Bayanihan Computing .NET [32], which also uses web services, these other frameworks rely on Java RMI, Jini, HORB and other messaging technologies to communicate between framework components. This can often prevent machines behind firewalls from participating.

As many Internet enabled machines are private machines where users have direct control over a firewall (if one exists), this may not a severe limitation on the volunteer base, but I think it is better to provide some support for machines behind firewalls, and is a feature that I wanted for G2.

**4.5.4 Reliability**

Managing the reliability of a G2 server is straightforward, because G2 has a single server design. This is a single logical server, because a G2 server may be powered by more than one computer. A trivial example which applies to the G2 server, is to have one computer running the web service, and another running a database. One of the benefits I derived from the use of commodity components is the capacity to distribute a web server and database over several physical computers,
without having to personally design and implement a mechanism to synchronise
data across those computers. In contrast, where a system is designed to use a net-
work of separate servers, a restart of one server can have consequences for the
other servers in the network. The CX framework and its sibling Jicos coordinate
their networks via entities known as Hosting Service Providers. HSPs manage a
network of Task Servers, to which hosts, or volunteers, connect. A client interacts
only with a HSP which distributes the jobs among Task Servers. CX’s server error
recovery mechanism is well documented. In CX, each Task Server is arranged in a
sibling connected fat tree, where each sibling mirrors the other. A transaction is
not complete until each of the mirrored states is updated. A Task Server failure is
detected by the volunteers that were connected to it, and the HSP is informed.
The HSP will reallocate the most recently added task server to replace the failed
server, and will restore the mirrored state of its new siblings. With this mecha-
nism, the CX “production network”, can recover from a sequence of server fail-
ures although certain simultaneous failures, however unlikely, can still cause data
to be lost. It is the responsibility of a CX host to report task server failures to the
HSP. However, a host cannot distinguish between a server failure and a failed
network connection to that server. To prevent false failure reports, the imple-
menters of CX propose a voting mechanism, whereby the HSP will not recognise
that a server has failed until a quorum of hosts has been reached.

4.5.5 Checkpointing

Checkpointing is the serialization of a job’s state such that it can be resumed from
the checkpoint at a later time. Checkpoints are periodically generated during the
course of job execution and stored. This data is stored on the volunteer’s local disk
(SETI@Home) or it can be transmitted and stored on the client machine, or on
dedicated checkpoint servers (Condor). Checkpointing is a resilient form of error
recovery because a job that periodically checkpoints is resistant to not only ma-
chine reclamation, but other causes of volunteer failure such as machine shut-
down. The disadvantages of checkpointing are that the actual checkpoint process
may be expensive, and the checkpoint data may be substantial. Large checkpoint files take up resources on the volunteer machine and are expensive to transmit. Condor checkpoints can be many megabytes in size and are not stored on the volunteer host machine. Instead they are transmitted and stored either on the client machine, or on dedicated checkpoint servers.

Checkpointing on Condor is automated and occurs without the intervention of the programmer. Other checkpoint systems require the programmer to explicitly create a checkpoint through an API call. In addition, the programmer must decide which data should be checkpointed and create jobs that can continue given the programmer specified checkpoint data. In other words, much of the work required to create checkpoints is manually performed by the programmer. The reason for this complexity, is that checkpoints must store the data state of the job, as well as its current point of execution. Therefore, to automate checkpoints, the volunteer component must be able to store the thread state as well as the memory state of the executing job. This data is often inaccessible by the volunteer component, and even if it were, it may prove to be too large to be useful in practice. One approach is to execute the job on a modified JVM which provides access to the executing state of a job. Unfortunately, this requires all volunteers to be executing the modified JVM which severely limits the number of potential volunteer host computers.

A general purpose cycle stealing system from the SETI@Home developers, called BOINC enables ordinary user to create SETI@Home like cycle stealing applications. BOINC supports checkpointing but it too requires programmers to explicitly store checkpoint data, and to write checkpoint aware jobs.

```c
bool boinc_time_to_checkpoint();
```

If this returns true, the application then calls:

```c
void boinc_checkpoint_completed();
```
Parabon’s Frontier is another example that relies heavily on the programmer to effect checkpoints. In Frontier, programmers write methods that potentially take a checkpoint as an input parameter. The method must be able to make sense of the checkpoint, restore job state and resume.

The Parabon tutorial provides an example of checkpointing excerpted in Listing 4-3.

```java
// Compute the square
// square = intToSquare * intToSquare;

for(int i=startIndex; i<1000; i++) {
    square += intToSquare * intToSquare;

    if(i%100 == 0) {
        map.put("dataID", dataID);
        map.put("square", square);
        startIndex = i;
        map.put("startIndex", startIndex);
        context.logCheckpoint(i/1000, map);
    }
}
```

**Listing 4-3: Parabon Frontier Checkpointing**

A checkpoint is created for every 100 iterations of the loop. The checkpoint is a dictionary called map, and in this example, the checkpoint consists of some identifier called, dataID, the current value of the square, and the value of \(i\), assigned to startIndex, which indicates the execution point of the checkpoint. This method can take this dictionary as an input parameter, extract and assign the values for square and \(i\), and resume the loop.

Systems that support continuation style programming can be said to indirectly support checkpoints. Instead of directly creating a checkpoint, a job may spawn a continuation. An example of such a system is Cilk-NOW [19]. Bayanihan supports bulk synchronous parallel applications, which could support indirect checkpointing by storing their state after the execution of each superstep.

### 4.6 Conclusion

It is one thing to be simple to use, but a cycle stealing system also needs to be reliable, safe and efficient. In this chapter, I have described the techniques that I
have developed or incorporated to meet these needs for a cycle stealing framework, deployable on the Internet using web-based components.

For efficiency, I have designed a hybrid communication mechanism using ordinary polling techniques and asynchronously executing web services, which effectively enables servers to push data to clients over a HTTP, a request/response protocol. This enables the server to relay changes in state, specifically the results of jobs, to the client as soon as the result becomes available. This mechanism minimises G2’s processing time of jobs enabling G2 to support applications where the turn around time of jobs is critical to performance.

The separation of method implementations (assemblies) from the job payload enables the lazy downloading and caching of assemblies by volunteers. Not only does this minimise the continual downloading of the same assemblies by volunteers, it enables scheduling flexibility because volunteers are able to quickly and continually switch between applications if volunteers have cached assemblies.

Other features such as sharing Client Proxy instances among Class Proxies, and the aggregation of messages, ensure communications efficiency. Part of this mechanism is the ability of the server and client to coordinate to optimise the number of results that the server returns to clients using the fetch results mechanism.

Eager scheduling is an algorithm adopted by many cycle stealing frameworks to ensure reliability using a volatile pool of volunteers. However, a naïve implementation of this algorithm can be inefficient. I have identified the conditions where inefficiency can arise and have incorporated a simple modification into the G2 framework, which guards against some causes of inefficiency.

For cycle stealing systems, the ability to checkpoint enables the system to support applications that require jobs to exhibit long execution times in order to achieve acceptable speedup. It also has benefits for the scalability of server based systems by potentially reducing the frequency of volunteer requests for new jobs. Because
the checkpoint created on one machine has the ability to be resumed on another
job migration capabilities are added to the system. This capability ensures that
long running jobs will complete in a reasonable amount of time, and also adds
support for long running non-parallel applications, similar to a batch processing
system. The automatic checkpointing capability that I investigated can provide
benefits to the G2 framework, but its ultimate utility for G2 is unknown due to
the additional communication requirements it imposes on jobs that are check-
cpointed. As such although checkpointing for G2 is available, I do not consider the
checkpoint mechanism that I developed to add blanket support for long running
jobs.
Chapter 5  Performance Modelling

For a cycle stealing system to be used, the benefits must be made obvious to potential users. Successful systems such as SETI@Home are often used as examples to demonstrate the performance benefits of cycle stealing. However, while SETI@Home, and similar projects show that the combined potential of idle machines is enormous, they execute applications that share a set of properties that make them highly suitable for cycle stealing. They are all embarrassingly parallel, and have jobs with modest communication requirements (small in data size), but require large amounts of processing, executing for many hours on an ordinary desktop computer. The difference between communication and computation requirements is expressed as a ratio between computation time and communication time. A large computation to communication time ratio for jobs, compensates the cost of moving jobs and results across a network, and ensures that applications such as SETI@Home benefit from parallel execution.

However, not all parallel applications can be structured to have similarly large ratios between computation and communication time. For users with these non-SETI like applications, determining whether their applications have jobs with sufficient execution time to satisfy their desired level of performance, relies on intuition and guesswork. The resulting uncertainty can discourage potential users from investing the time and effort required to adopt cycle stealing as a means to execute their applications.

To remove this uncertainty, I have devised a performance model of the G2 system, which predicts the performance of user applications. The performance model consists of both a simulator and an analytical model, which can be configured with the characteristics of a user’s G2 network and applications. These tools demonstrate the potential benefits of cycle stealing to users prior to application development and execution, and they can be used to optimise the granularity with which programmers decompose an application into a set of jobs for execution on a G2 system.
5.1 Performance Evaluation Method

The output of the analytical model and simulator is a prediction of job throughput and job latency. A performance model cannot apply perfectly to every type of user application, so I decided to focus on throughput and latency, which are measures that are generally applicable. If a user has an application which is not a basic, single step master-worker application, such as a BSP application, that user can apply these throughput and latency values to produce performance estimates based on the structure and synchronisation of their application.

For throughput estimates, I am interested in the job throughput of a G2 system in a steady state. This requires a constant number of jobs in the system, therefore the behaviour of the client in the simulator and analytical model, is that it submits a new job for each result that it receives. For applications that execute for some time, the steady state throughput that the system can provide will determine the performance of the application. For latency estimates, I am interested in the minimum job latency; that is the time from when the job is created, to when a result is returned.

For long running, or embarrassingly parallel applications, the throughput estimate may provide an accurate prediction of the actual job throughput of a user’s application. For other types of applications, users can use the throughput and latency predictions produced by the performance model, and combined with their knowledge of the level of parallelism and synchronisation in their application, produce an estimate of the performance of their application.

For example, consider the simple case of a master-worker application, where a master submits a number of jobs and then waits for all of the results. Latency information can provide some indication of the startup, or lead-in time of the application, and the lead-out time required for the master to collect the last result. Throughput can give an estimate of the running time of the parts of the application that run in parallel. If the number of total jobs is known, the throughput and
latency predictions provided by the performance model, can be applied to produce an estimate of the running time of the entire application.

To validate the simulator and analytical model, I compared their outputs with the measured job throughput of an application running on a deployed G2 system.

**Performance Profiling**

To make throughput and latency predictions, the performance model requires a profile of the deployed system that characterise its runtime performance. This profile is used by the performance model to calculate the time cost of a particular operation. The profile consists of the timings of the four main web service methods involved in job and result distribution, network latency and throughput, and web method invocation costs incurred by clients and volunteers. This profile is incorporated into the performance model as parameters, listed in Table 5-1.
<table>
<thead>
<tr>
<th>Environment Parameters</th>
<th>System Parameters</th>
</tr>
</thead>
<tbody>
<tr>
<td>• Network throughput</td>
<td>• Client NIC throughput</td>
</tr>
<tr>
<td>• Network latency</td>
<td>• Job serialization time, fixed and per byte costs on the Client</td>
</tr>
<tr>
<td>• The number of volunteers</td>
<td>• Client result deserialization time (per byte)</td>
</tr>
<tr>
<td>• Volunteer Idle Time</td>
<td>• Server request deserialization times; fixed and per byte costs for</td>
</tr>
<tr>
<td></td>
<td>SubmitJob, FetchJob, SubmitResult and FetchJob operations.</td>
</tr>
<tr>
<td></td>
<td>• Server request execution time</td>
</tr>
<tr>
<td></td>
<td>• Server result serialization times</td>
</tr>
<tr>
<td></td>
<td>• Server NIC throughput</td>
</tr>
<tr>
<td></td>
<td>• Volunteer request message serialization times</td>
</tr>
<tr>
<td></td>
<td>• Volunteer reply deserialization times</td>
</tr>
<tr>
<td></td>
<td>• Volunteer performance</td>
</tr>
<tr>
<td></td>
<td>• Volunteer NIC throughput</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Application Parameters</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>• Job execution time</td>
<td></td>
</tr>
<tr>
<td>• Input data size (job size)</td>
<td></td>
</tr>
<tr>
<td>• Output data size (result size)</td>
<td></td>
</tr>
</tbody>
</table>

**Table 5-1: Performance Model Parameters**

I obtained one such profile of a deployed G2 system through performance measurement and analysis, as described further in Section 5.5.

**Performance Model Goals**

The goal for the performance modelling was to produce an analytical model of G2 to provide users with an insight into the relationship between job data size and execution time for the applications that execute on their G2 system. The analytical model would provide users with throughput and latency data and rapidly produce performance predictions over a wide range of possible job size and execution time combinations.

The G2 system consists of a server and a number of connected clients and volunteers. When an application is executed, the client creates new jobs, which are dis-
tributed to volunteers via the server. Volunteers execute those jobs and submit computed results to the server, which are then retrieved by clients. Despite this simplicity, developing an accurate analytical performance model proved to be difficult.

I developed a G2 application and executed it using a variety of job execution times and data sizes, and measured job throughput and latency. This data would be used to produce and assess the accuracy of the analytical model. There were no simple correlations between the range of job configurations and throughput and latency, so I developed a discrete event simulator to have something that was more readily observable than an actual G2 system.

The simulator was able to accurately predict the behaviour of applications, but an analytical model was still the end goal, and it became clear that queuing models should be pursued. However, the simulator had become quite complex and had to be simplified to allow queuing theory techniques to be applied. To determine the extent to which such further simplification affected the accuracy of the predictions, I constructed a simpler simulator that reflected only those aspects of the original simulator that could be modelled using queuing theory techniques.

The simplified simulator produced adequate predictions, and is the basis for an analytical model based on mean value analysis for multi-class closed queuing networks. The performance analysis techniques used for the G2 performance model were analytical modelling, simulation and measurement.

5.2 G2 Simulator

The G2 simulator is a discrete event simulation of the progression of jobs through the system. It includes the interactions between clients and volunteers with the server, job execution on volunteers, and contention for resources. Many aspects of the system that had to be incorporated into the simulator were not exclusive to cycle stealing, so I developed a generic simulation framework for distributed systems, with components to model the interactions between a distributed set of
workstations, including the network connecting them and the processes executing on each workstation. This generic framework can be extended to simulate a particular distributed system, and I made such an extension to model the behaviour of the G2 framework.

The simulation framework models a collection of workstations connected by network links along which packets flow in a FIFO manner. Each workstation contains one or more CPUs which execute a collection of local threads using round robin scheduling. The behaviour of each thread is controlled by a finite state machine. The finite state machine expresses an abstraction of the logic of the application being simulated, and is the non-generic component of the simulation framework. Each thread state determines a task to be performed. Some tasks are purely CPU bound, requiring a certain number of cycles to complete, others are entirely memory or disk bound which, as a simplifying assumption, means that they have no need for the CPU during their execution period. Other tasks deal with peripheral devices, such as sending and receiving messages via a network interface card.

Workstations contain network interface cards which maintain physical I/O buffers and have the ability to send an interrupt to the workstation’s CPU when a new packet arrives. Workstations implement a TCP/IP stack which allows socket connections to be established and maintained with other workstations. The TCP protocol uses special connect, accept and acknowledgement messages in order to control data flow, ensuring that transmission buffers do not overflow. One outcome of this mechanism is that receivers are able to exert backward pressure on senders, preventing them from sending additional packets until the receivers have caught up. This behaviour is incorporated into the simulator.

### 5.2.1 Simulation Components

A resource in the simulation framework is represented as a simulation component, and a real world system is modelled as a set of these components. The use of
a resource is simulated by some functionality encapsulated in a task. Each task has a type, determined by the resources that are used by the task. So a task that uses CPU time is a CPU task, and a task that involves database access is a database task. There are also task types involving network operations such as listening for connection requests and sending and receiving messages. The simulator uses a performance profile to determine the time required to perform each task.

**Workstation**

The workstation component simulates the activity of a single or multi-processor computer and is used to model servers, as well as personal computers. Each workstation has at least one CPU and a NIC (network interface card). To send messages across the network, each workstation also has an IP Address and maintains a TCP stack. The CPU and NIC are themselves simulation components and may be assigned tasks to complete.

![Diagram of Workstation Component](image)

**Figure 5-1: Workstation component shown here with the network.**

**Threads**

The functionality of each workstation is determined by threads. Each workstation hosts one or more threads whose behaviour is determined by a finite state machine. Each state is associated with a task, which is assigned to one of the simulation components within the workstation. A thread will remain in a particular state until the task associated with that state has been completed. The task determines which resources available to a workstation are consumed while the thread is in that state.
CPU
The CPU is a simulation component that models the processing capability of the workstation. One or more threads are bound to a CPU and these threads are given an opportunity to consume CPU time in a round robin fashion. If a thread is in a state that does not require CPU time, it releases the CPU and the next thread is given an opportunity to execute.

Database
The database component, as its name suggests, represents a database in the simulated system. The use of the database is represented by a database task, which consumes both CPU processing time and memory/disk access time.

5.2.1.1 Network Simulation
Communication between workstations occurs via the Network. Each workstation has a Network Interface Card and communication is simulated by interactions between it, the physical Network, TCP stacks and TCP sockets. Messages are modelled as a series of packets.

Packets
A packet in the simulator represents a standard TCP packet. Each packet contains a header plus a portion of a message. Packets can also represent TCP control messages, and may be connection requests or acceptance or acknowledgement notifications.

Network
The network is modelled as a collection of network links and the mechanisms needed to manage them, such as DHCP.

Network Links
Network Links are simulation components used to model the connection from one NIC to another. The data carried by a link is modelled as a queue of packets. Messages are sent along Network Links when a NIC writes a series of packets.
onto the Link. Physical network limits are simulated by each link having latency and maximum throughput, which determines the rate at which packets can progress toward their destination.

**TCP Socket**

The TCP Socket represents the interface to the network. A task responsible for data transmission will either write data or read data from the TCP socket. Each socket is bound to a particular network link, so a new socket will be created whenever a new link is established.

**TCP Stack**

The TCP stack simulates an implementation of TCP. As such it provides a socket whenever a workstation (thread) requires a new connection to a workstation. The stack routes incoming packets from the NIC to the appropriate socket and handles connection requests, and transmission of acknowledgement packets.

**Network Interface Card (NIC)**

The network interface card is a simulation component and is the endpoint of each Network Link. All data coming to and flowing from a workstation goes through a NIC. The NIC incorporates a hardware buffer which is used to model the limited data throughput available to the workstation.

### 5.2.2 Simulator Communication Model

The communication model used by the simulator models TCP, and in particular its connection and flow control mechanisms. It was not my intention to model every aspect of a real system, but to make the simulator detailed only as is necessary to achieve a reasonable level of accuracy. I found that the communication model had to properly simulate the interaction between sender and receiver, especially flow control. Without flow control, senders did not experience any backward pressure, and could push an unlimited number of packets onto the network. This had a cascading effect as senders were not limited by the speed and
throughput of the network, and could perform more tasks than what was observed in reality.

Suppose a workstation is executing a task where it needs to send a message. A new socket is established, and a connection request propagates to the workstation’s NIC, which uses the network’s DHCP service to obtain the location of the intended receiver. The connection request travels via the network to the receiver workstation. When the request is received by the NIC of the receiver, an interrupt is triggered, and if there is a thread on the receiver that is listening for connection requests, a socket will be created and an acceptance packet sent. After the initial connection is made, the sender writes to its socket’s buffer, which writes to the NIC’s buffer, and the message then travels to the receiver along the network, where, on receipt of message packets, another interrupt is triggered and so on. Packets are acknowledged, and if no acknowledgements are forthcoming, then the sender stops sending packets.

The communication model simulates connection request messages, acknowledgements, sockets, the use of hardware and software buffers, interrupts, and the sliding window algorithm for flow control, to provide the simulator with a sufficient level of accuracy to model communication across a network.

5.2.3 G2 Specific Components

In the generic simulation framework, a workstation is a component that does not encapsulate any application logic. It is extended to simulate specific distributed systems. For the G2 simulation, the workstation component was extended to model the server, clients and volunteers. Figure 5-2 is a diagram of the G2 simulator. The server, client and volunteer are extensions of the basic workstation component with threads and corresponding state machines to model their behaviour.
Server

The server is a workstation simulation component that represents a G2 server. In the generic simulation framework, a workstation contains at least one CPU, and one NIC. To represent the G2 server, I added a database component and an additional CPU, to represent the dual processor computer that I used as a G2 server. When a server initializes, it starts one thread which listens for incoming connection requests. This listen thread, in conjunction with a pool of worker threads, is intended to simulate the function of a web server.

Figure 5-3: Listen thread state diagram

When a connection request arrives, the listen thread processes that request resulting in the creation of a new socket through which further communication can occur. The role of the listen thread is to handle incoming connection requests.
After the connection has been established, worker threads take over and receive and process messages. As a web server is able to simultaneously receive multiple messages, the server has a pool of worker threads that it uses to process incoming messages.

The behaviour of the worker threads depends on the type of the incoming message, as is illustrated by the following state diagram. (Figure 5-4)

As with all threads, worker threads are given an opportunity to consume resources in a round robin manner. The resources that they do require will depend on their current state. As the server has two CPUs, two different threads can simultaneously consume CPU time. To reflect the worker thread pool of IIS, there are a finite number of worker threads. If all of the worker threads are busy, any additional incoming messages will not be processed until a worker becomes free.

**Figure 5-4: Worker thread state diagram**
Client

The client is a workstation that makes submit job and fetch result requests to the server. The client has two threads, a main thread which initially connects to the server and submits jobs (Figure 5-5), and a fetch thread which simulates the fetch result mechanism on the client (Figure 5-6).

![Submit Job thread state diagram](image)

**Figure 5-5: Submit Job thread state diagram**
**Volunteers**

A volunteer workstation has three threads, one for fetching jobs (Figure 5-7), one for submitting results (Figure 5-8) and one for executing jobs (Figure 5-9). These simulate the function and the competition for resources of the threads active within an actual G2 volunteer.
Figure 5-7: Fetch Job thread state diagram

Figure 5-8: Execute Job thread state diagram

Figure 5-9: Submit Result thread state diagram
5.2.4 Comparison

The following graphs show the predicted performance of the simulator compared with the measured throughput of a deployed system. I performed comparisons using job execution times of 200ms, one, two and ten seconds. The execution time used for Figure 5-11 and Figure 5-12 is one second and is representative of the results I obtained for the various execution times. I chose relatively short execution times to ensure that the simulated server was under non trivial loads. With each job execution time, I executed a number of tests varying the number of volunteers, and the data size of jobs and results and compared the measured throughput of results to that predicted by the simulator. Within each test, the data size of jobs and results are the same. The setup used to obtain the real world measurements is that described in Section 4.4.
The graphs have the same general shape although the simulator is pessimistic in its prediction of peak performance. The simulated server handles requests sequen-
tially, and there is very little simulated parallel activity between the simulated server components such as the web server, database, and disk and memory access. Their real world counterparts, IIS Server, SQL Server 2000, and Windows 2003, are likely to have highly tuned performance, which is not captured by the simulator. This accounts for the pessimistic performance predictions by the simulator when the server is under heavy load.

Still, the simulator produced results with an acceptable degree of accuracy, but it could take a relatively long time to produce results when a large number of nodes (clients or volunteers) were being simulated. I still preferred an analytical solution, but the simulator was too complicated to provide the basis of a queueing network model. In particular, the simulation of the contention between threads, task interleaving and scheduling, and simulation of TCP messaging, could not be represented.

To produce an analytical model these features of the simulator would have to be simplified. This posed a problem, in that should the resulting analytical model produce greatly different results, I could not be sure whether the flaw was in the simplification process, or whether the problem lay in the process of translating from a simulator with dynamic behaviour into a static analytical model, or whether my analytical approach was otherwise flawed. To handle this uncertainty, I decided to first simplify the existing simulator, so that it could be represented as a queueing network model. If the simplified simulator produced reasonably accurate results, then I would proceed with an analytical model.

5.3 Simplified Simulator

To simplify the simulator, I made the following changes:

- The multithreaded behaviour of the server was simplified, so that incoming web service requests entered a single queue and are processed in a First Come First Served (FCFS) sequential fashion by a single CPU.
The CPU and memory/disk bound phases of server processing were amalgamated in to a single service time.

The time that a request is away from the server, including network transmission time and request generation and response processing times were amalgamated into one value. This value is known as Think Time.

5.3.1 Service Time for web methods

In the complex simulator the execution of the web methods: submit job, fetch job, submit result, fetch results, generally consisted of four steps:

1. The deserialization of the request parameters
2. The actual execution of the request
3. The serialization of the reply
4. The time to write the reply to the network stream

In the simplified simulator, these steps are amalgamated into one web method request service time. The equations below are used to calculate the service time for each class of web method request for use in the simplified simulator. I found that
job, \( i \), and reply, \( o \), measured in bytes. Some requests and replies have a constant size, for example, the submit job reply and the fetch job request, and so some service time components are not dependent on either \( i \), the size of the job, nor \( o \), the size of the result. The fetch results operation can fetch more than one result at a time, which affects the service time of the request. \( E \), is the number of results available for collection when a fetch results operation is executed, and \( j \) is the number of jobs available for execution when a fetch job request arrives. The actual transmission time of the reply is not included because service time is concerned with server side processing time, not network transmission time. The service time equations are:

\[
S_{sj} = S_{sjDeserC} + S_{sjDeserM} \cdot i + S_{sjJobC} + S_{sjJobM} \cdot i + S_{sjSer} \\
S_{fj} = S_{fjDeser} + \begin{cases} 
S_{fjJobC} + S_{fjJobM} \cdot i + S_{fjSerC} + S_{fjSerM} \cdot i & | j > 0 \\
S_{fjJobC} + S_{fjSerC} & | j = 0
\end{cases} \\
S_{sr} = S_{srDeserC} + S_{srDeserM} \cdot o + S_{srJobC} + S_{srJobM} \cdot o + S_{srSer} \\
S_{fr} = S_{frDeser} + S_{frJobC} + S_{frJobM} \cdot o \cdot E + S_{frSerC} + S_{frSerM} \cdot o \cdot E
\]

Each service cost \( S \) is qualified by a subscript identifier where:

- \( sf \) - Request class: submit job
- \( fj \) - Request class: fetch job
- \( sr \) - Request class: submit result
- \( fr \) - Request class: fetch result
- \( cDeserC \) - constant cost required for the server to deserialize the request payload for request class \( c \).
- \( cDeserM \) - data dependent cost for the server to deserialize the request payload from a request of class \( c \).
- \( cJobC \) - constant cost for executing a request of class \( c \).
- \( cJobM \) - data dependent cost for executing a request of class \( c \).
- \( cSerC \) - constant cost for the server to serialize the result of a request of class \( c \).
- \( cSerM \) - data size dependent cost for the server to serialize the result of a request of class \( c \).
Submit Job and Submit Result

\[
S_{sj} = S_{sjDeserC} + S_{sjDeserM} \cdot i + S_{sjJobC} + S_{sjJobM} \cdot i + S_{sjSer}
\]
\[
S_{sr} = S_{srDeserC} + S_{srDeserM} \cdot o + S_{srJobC} + S_{srJobM} \cdot o + S_{srSer}
\]

The service time for the submit job and submit result requests are simply the sum of their constituent steps, and dependent on the size of the job and result respectively.

**Fetch Job**

\[
S_{fj} = S_{fjDeser} + \begin{cases} 
S_{fjJobC} + S_{fjJobM} \cdot i + S_{fjSerC} + S_{fjSerM} \cdot i & |j| > 0 \\
S_{fjJobC} + S_{fjSerC} & |j| = 0
\end{cases}
\]

The fetch job service time depends on whether a job is available or not, hence the two possible costs. If there are no jobs available for execution then the request processing time is constant, and the size of the reply returned to the volunteer is not dependent on the size of jobs, \(i\).

**Fetch Result**

\[
S_{fr} = S_{frDeser} + S_{frJobC} + S_{frJobM} \cdot o \cdot E + S_{frSerC} + S_{frSerM} \cdot o \cdot E
\]

The server can return a number of results to the client as the result of a single fetch result request. Therefore, the service time depends on the size of a result \(o\), and the number of available results, \(E\). The fetch results request is a blocking request and returns only when there are results available. So, the value of \(E\) is at least 1.
5.3.2 Think Times

The time cost of making a request is known as the request’s think time\(^{16}\). In the simplified simulator, the think time consists of:

1. Time to create the request, including serialization of request parameters
2. Time to send the request over the network
3. Time to process the reply, including deserialization of the reply

Note that think time does not include the time to react to a reply. For example, if a volunteer fetches a job, the think time does not include the job execution time. Therefore, think time, as described in this section, is the cost of making a request, it does not indicate the actual interval between requests made by a client or volunteer in the G2 system.

The equations used to calculate think time are provided and explained below; think time is represented by the symbol, \(Z\). The network time to transmit the request and receive the reply is linearly proportional to the size of the message, and is represented by the symbol, \(K\).

**Network Cost of Job Operations: Submit Job, Fetch Job**

\[
K_f = 2 \times \text{network latency} + \frac{\text{request size} + i + \text{response size}}{\text{network throughput}}
\]

**Network Cost of Result Operations: Submit Result, Fetch Results**

\[
K_r = 2 \times \text{network latency} + \frac{\text{request size} + o + \text{response size}}{\text{network throughput}}
\]

The \textit{request size} and \textit{response size} variables represent the size of web service request and response messages, and are usually about 200 bytes in length.

---

\(^{16}\) The classic example of a closed queueing model, where requests spend time away from the server, is that of dumb terminal workloads on a server. The frequency of requests on the server depends on the terminal user’s “think time”.

168
Submit Job and Submit Result

Before a client can submit a job, it has to generate the job then serialize it. The serialization time linearly depends on the size of the job, \( i \), and it has a constant element. \( C \) and \( M \) represent the constant and datasize dependent parameters. The equation is:

\[
Z_{sj} = Z_{sjC} + Z_{sjM} \cdot i + K_j
\]

Similarly for the fetch job and submit result think times with the appropriate \( K \) and data size parameters:

\[
Z_{sr} = Z_{srC} + Z_{srM} \cdot o + K_r
\]

Fetch Job

The think time of the fetch job operation depends on whether a job was available for execution when the fetch job request was made. If no job is available, \( j = 0 \), there is a lesser processing and network transmission cost.

\[
Z_{fj} = \begin{cases} 
Z_{fjc} + Z_{fjm} \cdot i + K_j & | \ j > 0 \\
Z_{fjc} + K_j - \frac{i}{\text{network throughput}} & | \ j = 0
\end{cases}
\]

Fetch Result

For the fetch results request, the average think time depends on the number of results received, \( E \). So,

\[
Z_{fr} = Z_{frc} + Z_{frm} \cdot o \cdot E + K_f
\]
The removal of worker threads, and the sequential processing of web service requests, resulted in simulated performance that is lower than what was actually realised. This result is shown Figure 5-14, when compared to measured performance graphed in Figure 5-11. However, this simplification of the server simulation accurately predicted throughput trends. As with the original simulator, it fails to accurately predict throughput when the server is under load, although this effect is more pronounced. Despite this, I decided that the simplified simulator was accurate enough to serve as the basis of an analytical model.

5.4 Analytical Model

The simplified model of the G2 system consists of a set of clients and volunteers that make requests to a server, which processes those requests on a FCFS basis. This model is represented using a queueing network.

5.4.1 G2 Queueing Network Model

The aim of the analytical model is to provide an estimate of the maximum, sustained, job throughput, and minimum job latency of a G2 system. To maintain a
constant number of jobs in the system to achieve an optimum steady state, I assume that clients submit a new job for each result that they receive. Each client or volunteer can make at most one request of a particular class to the server, for example, a client can attempt to submit a job and fetch a result at the same time, but it cannot simultaneously attempt to submit two jobs. The number of each class of request in the system is fixed and corresponds to the number of volunteers and clients. A request that is not currently outstanding (queued or being processed by the server) is considered to be in existence but delayed at a client or volunteer. This means that there are a constant number of requests in the system, making the G2 queueing network a closed system. To find the job throughput, I based my analytical model on Mean Value Analysis (MVA) [64].

5.4.2 Mean Value Analysis

Mean value analysis is a common way to evaluate closed queueing networks, which is a queueing network with a fixed number of requests. It is based on equations that evaluate average system throughput, $X$, average server queue length, $Q$, and average request response time, $R$.

*Throughput* is the rate at which requests are processed by the system. When a request arrives at the server, it is placed in a queue until the server is able to service the request. The average number of requests in this queue is the *queue length*. The *response time* is the average time that a server takes to respond to a request. It includes the time the request spends in the queue and the request’s *service time*, $S$, which is the time it takes the server to execute the request. In a network with $N$ requests:

$$X_N = \frac{N}{Z + R_N}$$
$$Q_N = X_N R_N$$
$$R_N = S(1 + Q_{N-1})$$
Throughput is based on the time taken for a request to cycle through the network. This cycle time consists of the time needed to create and transmit a request, and receive and process the reply (also known as $Z$, or *think time*), and the response time, $R$.

The average queue length is calculated by considering a request’s response time. During this time, new requests will arrive at a rate equivalent to system throughput, $X$, and be appended to the queue. The average number of requests that arrive during the average response time will give the average length of the queue.

The response time depends on the time a request spends in the queue, which is the time it takes for the server to service all of the requests ahead of it in the queue. The number of requests ahead of it in the queue, is the number of requests already present in the queue when the new request arrives. In MVA, this is assumed to be the average length of the queue in a system with $N$-1 requests. One way to find the average queue length of a system with $N$-1 requests, is through recursive MVA evaluation of a system with $N$-2 requests and so on, until a system with $N$=0 requests is evaluated. Trivially, in a network with no requests, $Q_0 = 0$.

### 5.4.3 Multiclass models

The G2 network model is a multiclass closed queueing network with class dependent service times. For such a system, the response time for a request depends on the number and class of the requests present in the server queue when it arrives at the server. Using an MVA approach to such a network, instead of considering the overall length of the queue, the average number of requests in the server’s queue of each class, $c$, in the network is evaluated, and used to find the average response time.

If there are $k$ classes in the system, let $N_c$ represent the number of requests of class $c$ in the system and let the vector, $[N_1 \ldots N_k]$ describe the numbers of all of the requests in the system. When a request of class $c$ arrives at the server, the assumption is that the queue consists of the average number of requests of all other
classes in the system, and the average number of requests of class \( c \) in the queue in a system with one fewer request of class \( c \). This smaller system is described by the vector \([N_i .. N_{i-1} .. N_k]\) (\( c \) may also be 1 or \( k \)). Therefore:

\[
R_c = \sum_{i=1,i\not=c}^{k} S_i \cdot Q_i^{[N_i .. N_k]} + S_c \left(1 + Q_c^{[N_i .. N_{i-1} .. N_k]}\right)
\]

For each class of request, the link between response time, queue length and throughput is the same as for single class networks.

### 5.4.4 Multiclass MVA extensions for G2

When a request is made, the request enters a queue on the server until it can be processed by a request handler. The G2 server is modelled as a queueing service centre in the G2 network model. There are four different classes of request corresponding to the web methods that are called by clients and volunteers, namely, submit job, fetch job, submit result, and fetch results.

![Simple queueing network with multiple clients and volunteers](image)

**Figure 5-15: Simple queueing network with multiple clients and volunteers**

The different classes of request each have different service times, which make the system a multiclass closed queueing network with class dependent service times. However, there are many dependencies between the different classes of requests.
in G2 that need to be incorporated into the analytical model. For example, the
service time for a particular fetch results request, is dependent on the number of
results available on the server, which in turn depends on the number of submit
results requests processed since the previous fetch results request. This scenario is
more complex than what is handled using existing multiclass MVA techniques, so
I have used a basic multiclass MVA technique and extended it to suit the G2 net-
work model.

To simplify the explanation of the model, the initial description will be of a sys-
tem with reliable volunteers. That is, volunteers that do not disconnect from a G2
network during the course of a computation. Then the model will be expanded to
include volatile volunteers.

5.4.5 Throughput - X

As I described earlier, the throughput, \( X \), of class \( c \), in a closed network with one
service centre is:

\[
X_c = \frac{N_c}{Z_c + R_c}
\]

Where \( N_c \) is the number of requests of class \( c \), \( Z_c \) is the average time a request
spends away from the server, and \( R_c \) is the average response time of a request of
class \( c \). This formula is actually an upper bound on throughput and I modified it
for analysis of the G2 network to account for the relationships between the dif-
ferent classes of request. The requests that I modelled were the submit job, \( sj \),
fetch job, \( fj \), submit result, \( sr \), and fetch result, \( fr \), requests. I am interested in the
throughput of a G2 system in a steady state, so clients submit jobs as results are
retrieved. Therefore, there is a relationship between the throughputs of the dif-
ferent classes of request. The volunteers can’t fetch more jobs than what have
been submitted, volunteers can’t submit more results than have been fetched, cli-
ents can’t fetch more results than what have been submitted, and clients won’t
submit more jobs than results fetched.
Generally, \( X_{sj} \geq X_{sj'} \geq X_{sr} \geq X_{fr} \geq X_{sj} \). This basic relationship illustrated here serves as a useful starting point, and will be expanded upon in the following sections.

**Submit Job**

The throughput of submit job requests is dependent on how quickly requests can be serialized and submitted. As a job is submitted for each result retrieved, the rate of job submission depends on the rate at which results are fetched. This fetch result rate depends on the mean throughput of fetch result requests, as well as the mean number of results fetched per request, \( E \). Naturally, the throughput of submit job requests also depends on how quickly the system can process those requests. Thus,

\[
X_{sj} = \min \left( \frac{n_c}{Z_{sj} + R_{sj}}, X_{fr} \cdot E \right)
\]

\( \frac{n_c}{Z_{sj} + R_{sj}} \) is the maximum rate of submit job request that the clients can make, where \( n_c \) is the number of clients and \( Z_{sj} + R_{sj} \) is the total round trip time of a submit job request. In the case where the throughput of submit job requests is determined by the round trip time of submit job requests (not the rate at which results are fetched) the number of submit job requests in the system, \( N_{sj} \), is equivalent to the number of clients, \( n_c \): A client can’t have more than one request outstanding, but a client will always have one request outstanding. \( X_{fr} \cdot E \) is the rate at which a client can recover results. (\( X_{fr} \) is the throughput of fetch result requests. The evaluation of \( E \) is described in Section 5.4.6.)

**Submit Result**

The throughput of the submit result operation is bounded by the rate at which a volunteer could submit results to the server, and depends on the rate at which that volunteer fetches jobs to execute.
The rate at which a volunteer could submit results, assuming there is always a result ready to be submitted, is given by: 

\[
\frac{n_v}{Z_{sr} + R_{sr}},
\]

where \(n_v\) is the number of volunteers and \(Z_{sr} + R_{sr}\) is round trip time for the submit result request. A volunteer cannot submit more than one result at once, so \(N_{sr} = n_v\).

Volunteers cannot submit results faster than they fetch jobs, so the throughput of submit result operations is bounded by the throughput of successful fetch job requests. A successful fetch job request is one that results in a volunteer receiving a job. A fetch job request *fails* if there is no job available when the server services the fetch job request. The probability of a fetch job request returning a job is \(r\), so the throughput of submit result requests is: (The calculation of \(r\) is explained in the next section)

\[
X_{sr} = \min\left(\frac{n_v}{Z_{sr} + R_{sr}}, r \cdot X_{fj}\right)
\]

**Fetch Job**

The formula for calculating the throughput of the fetch job operation is slightly different. An explanation follows:

\[
X_{fj} = \frac{n_v}{(1-r)(Z_{sleep} + R_{fj}) + r \cdot \max(Z_{fj} + R_{fj}, Z_{execute}, Z_{sr} + R_{sr})}
\]

The total throughput of the fetch job operation, \(X_{fj}\), depends on the number of volunteers, \(n_v\). The throughput of fetch job requests by one volunteer is the inverse of the average time between its successive fetch job requests. This average time is the weighted average of the period between fetch jobs requests, based on the two possible outcomes of a fetch job request. A fetch job request, will either be successful and yield a job, which will be executed, or it will fail and not yield a job, in which case the volunteer will sleep for a period before retrying. The probability of fetching a job is \(r\), therefore the probability of not fetching a job is \((1-r)\).
When the volunteer receives the reply of the fetch job request, its *think time* depends on the contents of the reply. If the volunteer receives a job, then let the think time be represented by $A$. If the fetch job request fails, and a job is not received, let the think time in that case be represented by $B$. Therefore, the mean interval between requests can then be given by: $\frac{1}{r A + (1-r) B}$.

![Figure 5-16: Fetch Job probability graph with stable volunteers](image)

To evaluate the value of $A$, the bounded job input and result output queues maintained by volunteers is considered. A volunteer attempts to fetch a new job only if there is space in its job input queue. Jobs are taken from the input queue, executed and the result is added to the result output queue. Therefore, the interval between fetch job operations depends on the status of the input and output queue, and the job execution time. In other words, if a volunteer successfully fetches a job, the average period until it makes the next fetch job request depends on:

1. The time it takes to make a fetch job request.
2. The time it takes to execute a job.
3. The time it takes to submit a result.

Each of these three factors may be the bottleneck to the fetch job process. For example, if a volunteer can fetch a job relatively quickly, but these jobs take a long time to execute, the execution time will dictate the frequency at which the volunteer makes fetch job requests. The average period between fetch job requests when a job is successfully fetched is therefore:
\[ A = \max \left( Z_{\text{fetch}} + R_{\text{fetch}}, Z_{\text{execute}}, Z_{\text{sr}} + R_{\text{sr}} \right) \] where,

\( Z_{\text{fetch}} + R_{\text{fetch}} \), is the minimum interval between fetch job requests. The response time for the fetch job request is relevant because a volunteer only makes one fetch job request at a time.

\( Z_{\text{execute}} \), is the mean execution time for jobs.

\( Z_{\text{sr}} + R_{\text{sr}} \), is the minimum possible interval between submit result requests.

Next, consider the period \( B \). If the fetch operation fails, then the volunteer sleeps for a period before trying again. The interval between a failed fetch and a new fetch is \( Z_{\text{sleep}} + R_{\text{fetch}} \), where \( Z_{\text{sleep}} \) is the time required to prepare and send a fetch job request, and to receive and process the reply.

\[ Z_{\text{sleep}} = \text{sleep time} + Z_{\text{fetch}} + 2 \times \text{network latency} + \frac{\text{request size} + \text{response size}}{\text{network throughput}} \]

Substituting for \( A \) and \( B \), the mean interval between fetch job requests is given by:

\[ rA + (1-r)B = r \cdot \max \left( Z_{\text{fetch}} + R_{\text{fetch}}, Z_{\text{execute}}, Z_{\text{sr}} + R_{\text{sr}} \right) + (1-r) \left( Z_{\text{sleep}} + R_{\text{fetch}} \right) \]

**Fetch Job Success Probability**

The probability of a job being available when a fetch job request is made, depends on the rate at which new jobs arrive on the server, \( X_{\text{sr}} \), and the rate at which volunteers attempt to fetch jobs, \( X_{\text{fetch}} \). Therefore, the probability that a job is available is the ratio between the throughput of submit job requests and fetch job requests. It is given by:

\[ r = \frac{X_{\text{sr}}}{X_{\text{fetch}}} \]
Due to the relationship between the submit job and fetch job operations, which are enforced in this set of throughput equations, the throughput of submit job requests, $X_{sj}$, will not be higher than the throughput of fetch results requests, $X_{fr}$, and so the value of $r$ will not be greater than one.

**Fetch Results**

A client can only have one fetch results request outstanding, and provided there are results outstanding, it will make a new fetch result request as soon as the previous request has returned. The server handles fetch results requests asynchronously, returning only when results are available. As a result, the throughput of the fetch results operation is limited by both the rate at which results are submitted by volunteers, and the rate at which a client can submit fetch results requests. Therefore, the mean throughput of fetch results operations is given by:

$$X_{fr} = \min \left( \frac{n_c}{Z_{fr} + R_{fr}}, X_{sr} \right)$$

The highest possible rate of fetch results requests by the clients in the system, (ignoring the frequency of job submission, and the fact that the submit job and fetch result operations are linked) is given by, $\frac{n_c}{Z_{fr} + R_{fr}}$. Again, $n_c$ is the number of clients, and $Z_{fr} + R_{fr}$ is the time taken to complete a fetch results operation. The relationship between the throughputs of the fetch results request, and submit job and fetch job requests, is captured indirectly through the dependency on the throughput of the submit result operation.

**Network Congestion**

Having obtained the throughput of each of the web service requests, the network requirements of the server, clients and volunteers can be calculated. As all requests flow through the server, it is possible that the combined throughput of web service requests exceeds the network capabilities of the server. The transmis-
sion times of each web service request and reply, as incorporated in the sleep or \( Z \) times, is increased based on the throughput of requests, so that network throughput is reduced to within theoretical limits, and as a simple mechanism to introduce the effects of network congestion into the analytical model.

### 5.4.6 Service Time - \( S \)

The service time for each class of request in the analytical model is calculated using the equations for service time as used in the simplified simulator\(^{17}\) (Section 5.3.1). There are however some modifications required for the analytical model.

#### Fetch Job

In the simplified simulator, the service time depends on whether a job is available for execution when a fetch job request is serviced, hence there are two possible service times given by:

\[
S_{fJ} = S_{fJDeser} + \begin{cases} 
S_{fJJobC} + S_{fJJobM} \cdot i + S_{fJSerC} + S_{fJSerM} \cdot i & | j > 0 \\
S_{fJJobC} + S_{fJSerC} & | j = 0 
\end{cases}
\]

In the analytical model, \( S_{fJ} \) is the mean service time of a fetch job request, so the weighted mean of these two possible service times is required. The probability that a job is available for execution when a fetch results request is serviced is given by, \( r \), which represents the state where \( j > 0 \). The probability of there being no jobs, \( j = 0 \), is \( 1 - r \), therefore in the analytical model:

\[
S_{fJ} = S_{fJDeser} + r \cdot \left( S_{fJJobC} + S_{fJJobM} \cdot i + S_{fJSerC} + S_{fJSerM} \cdot i \right) + (1 - r) \left( S_{fJJobC} + S_{fJSerC} \right)
\]

\[
= S_{fJDeser} + S_{fJJobC} + S_{fJSerC} + r \cdot i \left( S_{fJJobC} + S_{fJSerC} \right)
\]

\(^{17}\) Standard MVA considers the demand, \( D \), that a request makes on the server as it moves through the queueing network. It is calculated as \( D = V \times S \), where \( V \) is the number of times a customer, or a request in our case, will visit the server as it circulates through the network. \( S \) is the service time of the request. In the G2 model, there is only one server which services every type of request, and once a server processes a request, it returns back to the client or volunteer. Therefore, in the G2 model, the value of \( V \) is one, and therefore all reference to \( D \) can be omitted and only the service time, \( S \) is relevant.
**Fetch Results**

The service time of the fetch result request depends on the number of results available for collection when the request is serviced. The number of available results is given by, \( E \), hence the fetch results service time is calculated by:

\[
S_{fr} = S_{frDeser} + S_{frJobC} + S_{frJobM} \cdot o \cdot E + S_{frSerC} + S_{frSerM} \cdot o \cdot E
\]

In the simplified simulator, the value of \( E \) depends on the state of the server when the execution of the fetch results request is simulated. In the analytical model, \( E \) is the mean number of results. The value of \( E \) is evaluated by considering the mean number of results that are submitted between two successive fetch results requests. This is equivalent to the ratio between the throughput of the submit result and fetch results operations. The other consideration is that the fetch results operation is asynchronously handled on the server side, meaning that it will only be executed if there is a result available. Therefore, \( E \) cannot be less than one and is given by:

\[
E = \max \left( 1, \frac{X_{sr}}{X_{fr}} \right)
\]

The service time equations for the analytical model are:

\[
\begin{align*}
S_{sj} &= S_{sjDeser} + S_{sjDeserM} \cdot i + S_{sjJobC} + S_{sjJobM} \cdot i + S_{sjSer} \\
S_{sj} &= S_{sjDeser} + r \cdot \left( S_{sjJobC} + S_{sjJobM} \cdot i + S_{sjSerC} + S_{sjSerM} \cdot i \right) + \left( 1-r \right) \left( S_{sjJobC} + S_{sjSerC} \right) \\
S_{sr} &= S_{srDeser} + S_{srDeserM} \cdot o + S_{srJobC} + S_{srJobM} \cdot o + S_{srSer} \\
S_{fr} &= S_{frDeser} + S_{frJobC} + S_{frJobM} \cdot o \cdot E + S_{frSerC} + S_{frSerM} \cdot o \cdot E
\end{align*}
\]

**5.4.7 Response Time - R**

The response time, is the mean time from when a request arrives at the server, to when it is completely executed. When a request arrives at the server, it enters a queue and is serviced in a FCFS manner. So, the response time, is the sum of the service times of all of the preceding requests in the queue and the service time of
the newly arrived request. To calculate the response time, the composition of the queue is needed. From this, and knowing the service time for each class of request, the time needed to service all preceding requests in the queue can be calculated. The composition of the queue is determined through the average number of each class of request in the queue. For each class of request, this average is denoted by \( Q_c \), where \( c \) denotes some request class. In the G2 model, \( c = \{sj, fj, sr, fr\} \).

Suppose a request of class \( y \), \( y \in c \) arrives at the server. To calculate \( R_y \), the composition of the queue at the arrival instant of the request is required. However, \( Q_y \) is the mean number of requests of class \( y \) in the queue in the current system. It does not indicate the number of requests of class \( y \) in the queue at the arrival instant of a new class \( y \) request. Therefore, while the \( Q_{cy} \) values of the other classes can be used to determine the composition of the queue, \( Q_y \) itself cannot be used.

The standard MVA technique described earlier relies on the assumption that the number of requests of class \( y \) in the queue at the arrival instant of a class \( y \) request is the average number of requests of class \( y \) in the queue, in a system with one fewer requests of class \( y \).

However, obtaining a solution in this recursive manner, apart from having exponential time complexity, is not possible in the G2 model, due to the dependencies within the system between request classes. For example, changing the number of submit jobs requests will change the success rate of fetch jobs requests, reducing the throughput of results produced and submitted, and so on, ultimately altering the total composition of the queue.

In other words, changing the number of one type of request affects the entire network to such an extent that it is not valid to base the queue lengths of a system with \( N_c \) requests, on a system with \( N_c -1 \) requests, invalidating the basic assumption used in the recursive evaluation technique.
Fortunately, there is a well known approximation technique, known as Schweitzer’s algorithm [65], for approximating the number of requests in the queue when a new request arrives. Applying Schweitzer’s approximation, the number of requests of class \( c \), in the queue when a new request of class \( c \) arrives is:

\[
\frac{N_c - 1}{N_c} Q_c
\]

Using this approximation algorithm, the mean number of requests of a particular class in a queue, in a system with \( N_c - 1 \) requests of that type, is proportional to the mean number of requests in the queue in a system with \( N \) requests of that class.

Earlier, I provided a generic formula for evaluating the mean response time for a request of class \( c \) as:

\[
R_c = \sum_{i=1}^{k} S_i \cdot Q_i^{[N_i]} + S_c \left( 1 + Q_c^{[N_c - 1 - N_c]} \right)
\]

Using Schweitzer’s approximation algorithm, this formula becomes:

\[
R_c = \sum_{i=1}^{k} S_i \cdot Q_i^{[N_i]} + S_c \left( 1 + \frac{N_c - 1}{N_c} Q_c^{[N_c]} \right)
\]

As G2 only has four classes of request, it is feasible to completely expand this formula, and as a system with fewer requests is not considered, the vector qualifying the number of requests of each class is not required and can be omitted.

**Submit Job**

As each client can have at most one submit job request outstanding, the number of submit job requests in the system is equal to the number of clients, \( n_c \). Applying and expanding the formula above, the mean response time of a submit job request is:
The part in square brackets evaluates the service time for all of the requests in the queue that are not submit job requests. The other part evaluates the service time of the newly arrived submit job request, and the service times for the preceding submit job requests in the queue.

I have assumed that the clients are of the same type. That is, the characteristics of the jobs submitted by clients are homogeneous with respect to job size, result size and execution time, such that they will have equivalent service times.

**Fetch Job and Submit Result**

As with clients and their submit job requests, a volunteer can only have one fetch job request outstanding at any one time. Similarly a volunteer can only have one submit result request outstanding. The number of volunteers in the system is \( n_v \), so the mean response time of a fetch job request is:

\[
R_{fj} = \left[ S_{fj} \cdot Q_{fj} + S_{sr} \cdot Q_{sr} + S_{fr} \cdot Q_{fr} \right] + S_{fj} \left( 1 + \frac{n_v - 1}{n_v} Q_{fj} \right)
\]

The mean response time of a submit result request is:

\[
R_{sr} = \left[ S_{sj} \cdot Q_{sj} + S_{fr} \cdot Q_{fr} + S_{fr} \cdot Q_{fr} \right] + S_{sr} \left( 1 + \frac{n_v - 1}{n_v} Q_{sr} \right)
\]

**Fetch Result**

The response time for the fetch result request is slightly different. First the equation:

\[
R_{fr} = \left[ S_{fj} \cdot Q_{fj} + S_{fj} \cdot Q_{fj} + S_{fr} \cdot Q_{fr} \right] + S_{fr} \left( 1 + \frac{n_v - 1}{n_v} Q_{fr} \right) + W_{fr}
\]

The difference arises from the fact that the fetch results operation is executed asynchronously on the server side. When a fetch results request is serviced and there are no available results, the request will block and wait until a result is
submitted by a volunteer. In the calculation of the fetch results response time, any such wait time must be considered, and the mean wait time is represented by $W_{fr}$. This average wait time depends on the probability that a result is available to be fetched when the fetch results request is initially serviced, and it also depends on the mean time between submit result requests. Thus:

$$W_{fr} = M_{fr} \times P_{fr}$$

Where, $M_{fr}$ is the average time interval between submit result requests. It calculated from the throughput of the submit results request and is given by

$$M_{fr} = \frac{1}{X_{sr}} \text{ where } X_{sr} \text{ is the throughput of submit result requests.}$$

$P_{fr}$ is the probability that a there will be no result available when a fetch results request is serviced, and is the ratio between throughputs of the fetch result and submit result operations.

Therefore, the average wait time is the average interval between submit result requests, scaled by the probability of having to wait, thus:

$$W_{fr} = M_{fr} \cdot P_{fr} = \frac{1}{X_{sr}} \cdot \frac{X_{fr}}{X_{sr}}$$

**Think Times**

The evaluation of the think, or $Z$ times used in the analytical model, is identical to the evaluation of the think times in the simplified simulator, described earlier in Section 5.3.2.

**5.4.8 Queue Lengths - Q**

The average number of requests in the queue depends on the time required for the server to process requests, and the rate at which requests arrive in the queue. These are given, respectively by the average response time, $R$ and the average throughput, $X$, of requests. Thus,
The mean number of requests of a particular class in the server’s queue is given by:

\[
Q = X \cdot R
\]

\[
Q_{ij} = X_{ij} \cdot R_{ij}
\]

\[
Q_{ji} = X_{ji} \cdot R_{ji}
\]

\[
Q_{sr} = X_{sr} \cdot R_{sr}
\]

\[
Q_{f} = X_{f} \cdot R_{f}
\]

### 5.4.9 Volunteer Volatility

The model, as described so far assumes a fixed and reliable set of volunteers. That is, once a volunteer joins the computation, it stays until application execution has concluded. The reality however, is that volunteers will come and go during the course of a computation. At different points in time there will be a different number of volunteers connected to the network. Instead of trying to model the variation in volunteer numbers over time, I consider the average number of volunteers connected to the network, and model the effects on the cycle stealing system as volunteers connect and disconnect from the network. Thus, the volatile nature of volunteers is incorporated into the model as a fixed number of volunteers with periodic failure and immediate replacement. That is, if a volunteer leaves, another immediately arrives to take its place, so that the overall number of volunteers remains the same. A volunteer *fails* if it disconnects from the network, and I assume that the cause of failure is that a volunteer is reclaimed by its owner and ceases to be idle. I have assumed that volunteer reclamation can be modelled as a Poisson process. At least one study of the idle times of workstations supports this assumption [1].

The throughput equations described earlier, model the interactions between requests, and the interval between successive requests made by a volunteer. Therefore, incorporating volunteer volatility required changes only to the throughput formulae.
Throughput Formulae Modifications

Fetch Job

In the stable volunteer model, the throughput of fetch jobs is based on the average interval between fetch job operations. Each fetch job attempt had two possible outcomes:

1. The fetch job operations fails because no job is available
2. The fetch job operation succeeds

The probability of a successful fetch job operation is denoted by \( r \), and a failure, \( 1-r \). A volunteer reacts according to the success or failure of an attempt to fetch a job, and so the average interval between fetch job operations with stable volunteers was described earlier as:

\[
(1-r)(Z_{\text{sleep}} + R_f) + r \cdot \max(Z_f + R_f, Z_{\text{execute}}, Z_{\text{sleep}} + R_f)
\]

In the stable volunteer model, I have ignored volunteer startup costs, because I am interested in steady state throughput and startup time is not a factor. This is not the case when volunteer volatility is introduced. As volunteers come and go, the startup cost becomes relevant and has to be factored into the model.

My approach to evaluate the mean throughput of fetch job requests, is to first consider the number of fetch job attempts that a volunteer will make during its lifetime (the time it is connected to a cycle stealing system), and the interval between those fetch job attempts. By considering all possible idle times and the probability that a volunteer will be idle for those times, the average number of fetch job attempts made by volunteers, and the average interval between those fetch job attempts can be determined. With this average interval and knowing the average number of volunteers, the throughput of fetch job requests is then calculated.
When its host machine enters the idle state, the volunteer starts up and connects to the cycle stealing system. The time taken is denoted by the symbol, \( s \). After the initial startup time, the volunteer attempts to fetch a job. The time required is \( c_{\beta} + \) :

\[
c_{\beta} = Z_{\beta} + \text{network latency} + \frac{\text{request size}}{\text{network throughput}}
\]

\( c_{\beta} \) is purely the time needed for the volunteer to prepare, and for the server to receive a fetch job request. It does not include time for the server to process the request and for the volunteer to receive a reply. If a volunteer’s idle time, \( t \), exceeds \( s + c_{\beta} \), then at least one fetch job request is made to the server.

After the initial fetch job attempt, the average interval between further attempts, represented by the symbol, \( k \), is the same as for stable volunteers:

\[
k = (1-r)(Z_{\text{sleep}} + R_{\beta}) + r \cdot \max\left( Z_{\beta} + R_{\beta}, Z_{\text{execute}}, Z_{sr} + R_{sr} \right)
\]

The number of subsequent fetch job requests that a volunteer, active for time, \( t \), will make is:

\[
(t - s - c_{\beta}) / k
\]

Let the function \( f(t) \), represent the number of fetch job attempts that a volunteer will make in its lifetime.

\[
f(t) = \begin{cases} 
0 & | t < s + c_{\beta} \\
1 + (t - s - c_{\beta}) / k & | t \geq s + c_{\beta}
\end{cases}
\]

If \( \Pr(t) \) is the probability that a volunteer will be active for time, \( t \), the average number of fetch job requests that a volunteer will make in its lifetime is:

\[
\int_0^\infty \Pr(t) \cdot f(t)dt
\]
Hence the average fetch job throughput for a volunteer over its lifetime is designated \( \mu_{\bar{\beta}} \):

\[
\mu_{\bar{\beta}} = \int_0^\infty \frac{\Pr(t) \cdot f(t)}{t} \, dt
\]

As the function \( f(t) \) has a transition point at \( t = s + c \), \( \mu_{\bar{\beta}} \) is evaluated by summing the fetch job throughputs for volunteers with idle times shorter and longer than this transition period. Hence:

\[
\mu_{\bar{\beta}} = \int_0^{s+c} \frac{\Pr(t) \cdot f(t)}{t} \, dt + \int_{s+c}^{\infty} \frac{\Pr(t) \cdot f(t)}{t} \, dt
\]

By definition, volunteers with idle times shorter than the transition time do not make any fetch job requests, \( \int_0^{s+c} \Pr(t) \cdot f(t) \, dt = 0 \), so:

\[
\mu_{\bar{\beta}} = \int_{s+c}^{\infty} \frac{\Pr(t) \cdot f(t)}{t} \, dt
\]

Recall that volatility is modelled as a Poisson process so that:

\[
\Pr(t) = \frac{1}{\beta} e^{-\beta t},
\]

Hence:

\[
\mu_{\bar{\beta}} = \int_{s+c}^{\infty} \frac{\Pr(t) \cdot f(t)}{t} \, dt = \int_{s+c}^{\infty} \frac{1}{\beta} e^{-\beta t} \cdot \left(1 + \frac{(t - s - c)}{k}\right) \, dt
\]

\[
= \left[ \operatorname{Ei} \left(-\frac{(s + c)}{\beta}\right) \times (s + c - k) + \exp \left(-\frac{(s + c)}{\beta}\right) \times \beta \right] / k \beta
\]

Where \( \operatorname{Ei}(x) = -\int_x^\infty \frac{e^{-t}}{t} \, dt \) is the Exponential integral.

The average fetch job throughput with volatile volunteers becomes:

\[
X_{\bar{\beta}} = n_v \cdot \mu_{\bar{\beta}}
\]
Submit Results

As with the fetch job operation, the throughput of submit result requests is evaluated by finding the average number of submit result requests made by volunteers.

A volunteer can only submit results after it has executed a job, which occurs only after it successfully fetches a job. Therefore, the number of result submissions depends on the time it takes for a volunteer to successfully fetch a job. One way to approach this problem is to consider the average number of failed fetch job attempts a volunteer makes before successfully fetching a job. The mean number of failures is represented by the symbol, $M$, and depends on the probability of successfully fetching a job. $M$ evaluates to

$$M = \frac{1-r}{r}$$

An explanation is provided in Appendix D.

The time it takes for a volunteer to fail to fetch a job and sleep is, $Z_{sleep}$, hence the time required for a volunteer to successfully fetch a job, represented by, $l$, is:

$$l = M \left(Z_{sleep}\right) + Z_{ji} + R_{ji}$$

Therefore, for a newly connected volunteer, the time taken to submit the first result, denoted by $d$, is:

$$d = s + l + Z_{execute} + Z_{sr} + R_{sr}$$

Hence, a volunteer with an idle time, $t$, which exceeds $d$, will submit at least one result. After the first result has been submitted, as job execution on volunteers is pipelined, the interval between subsequent submit result requests made by the volunteer, is the slowest of the successful fetch job time, $l$, the job execution time, $Z_{execute}$, and the result submission time, $Z_{sr} + R_{sr}$. If $j$ is the interval between submit job requests:
\[ j = \max \left( l, Z_{execute}, Z_{sr} + R_{sr} \right) \]

Let the function \( s(t) \), represent the number of submit result request that a volunteer will make in its lifetime.

\[
s(t) = \begin{cases} 
0 & | \ t < d \\
1 + (t - d)/j & | \ t \geq d
\end{cases}
\]

As with fetch job requests, the mean number of submit result requests \( \mu_{sr} \) is:

\[
\mu_{sr} = \int_{0}^{\infty} \frac{\Pr(t) \cdot s(t)}{t} dt = \int_{0}^{d} \frac{\Pr(t) \cdot s(t)}{t} dt + \int_{d}^{\infty} \frac{\Pr(t) \cdot s(t)}{t} dt
\]

\[
= 0 + \int_{d}^{\infty} \Pr(t) \cdot s(t) dt
\]

\[
= \int_{d}^{\infty} \frac{1}{\beta e^{-\frac{t}{\beta}}} \cdot \left(1 + \frac{(t - d)}{j}\right) dt
\]

\[
= \left[ \text{Ei} \left( \frac{-d}{\beta} \right) \times (d - j) + \exp \left( \frac{-d}{\beta} \right) \times \beta \right] / j \beta
\]

The average fetch job throughput with volatile volunteers becomes:

\[ X_{sr} = n_v \cdot \mu_{sr} \]
5.4.10 G2 MVA Equations

The set of equations for volatile volunteers becomes:

\[ S_{sj} = S_{sjDeserC} + S_{sjDeserM} \cdot i + S_{sjJobC} \cdot i + S_{sjSer} \]
\[ S_{jf} = S_{jfDeser} + r \left( S_{jfJobC} + S_{jfJobM} \cdot i + S_{jfSerC} + S_{jfSerM} \cdot i \right) + \left( 1 - r \right) \left( S_{jfJobC} + S_{jfSerC} \right) \]
\[ S_{sr} = S_{srDeser} + S_{srDeserM} \cdot o + S_{srJobC} + S_{srJobM} \cdot o + S_{srSer} \]
\[ S_{fr} = S_{frDeser} + S_{frJobC} + S_{frJobM} \cdot o \cdot E + S_{frSerC} + S_{frSerM} \cdot o \cdot E \]

\[ R_{sj} = \left[ S_{sj} \cdot Q_{jf} + S_{sr} \cdot Q_{sr} + S_{fr} \cdot Q_{fr} \right] + S_{sj} \left( \frac{n_r - 1}{n_v} Q_{sj} \right) \]
\[ R_{jf} = \left[ S_{jf} \cdot Q_{jf} + S_{sr} \cdot Q_{sr} + S_{fr} \cdot Q_{fr} \right] + S_{jf} \left( \frac{n_r - 1}{n_v} Q_{jf} \right) \]
\[ R_{sr} = \left[ S_{sj} \cdot Q_{sj} + S_{jf} \cdot Q_{jf} + S_{fr} \cdot Q_{fr} \right] + S_{sr} \left( \frac{n_r - 1}{n_v} Q_{sr} \right) \]
\[ R_{fr} = \left[ S_{sj} \cdot Q_{sj} + S_{jf} \cdot Q_{jf} + S_{sr} \cdot Q_{sr} \right] + S_{fr} \left( \frac{n_r - 1}{n_v} Q_{fr} \right) + W_{fr} \]

\[ Z_{sj} = Z_{sjC} + Z_{sjM} \cdot i + K_s \]
\[ Z_{jf} = Z_{jfC} + Z_{jfM} \cdot i + K_f \]
\[ Z_{sr} = Z_{srC} + Z_{srM} \cdot o + K_s \]
\[ Z_{fr} = Z_{frC} + Z_{frM} \cdot o \cdot E + K_f \]

\[ X_{sj} = \min \left( \frac{n_v}{Z_{sj} + R_{sj}}, X_{sr} \right) \]
\[ X_{sr} = n_v \cdot \mu_{sr} \]
\[ X_{jf} = n_v \cdot \mu_{jf} \]
\[ X_{fr} = \min \left( X_{fr}, \frac{n_v}{Z_{fr} + R_{fr}} \right) \]

\[ Q_{sj} = X_{sj} \cdot R_{sj} \]
\[ Q_{jf} = X_{jf} \cdot R_{jf} \]
\[ Q_{sr} = X_{sr} \cdot R_{sr} \]
\[ Q_{fr} = X_{fr} \cdot R_{fr} \]
The modified G2 MVA equations are highly coupled which unfortunately precludes direct evaluation. So I use an iterative approach similar to that used for Schweitzer’s approximation algorithm, where the values in the model are initially seeded, and the equations iteratively evaluated until the new and old values converge to within an acceptable tolerance.

5.4.11 Job Throughput

Given the throughput of each type of request, job throughput can be easily calculated. It depends on the mean throughput of the fetch results request, $X_{fr}$, and the mean number of results received with each request, $E$.

Job Throughput = $X_{fr} \times E$

5.4.12 Comparison

Using the same application parameters used to compare measured throughput from a deployed G2 system to the simulator and simplified simulator, the analytical model predicted the result throughput as plotted below.
Like the simplified simulator, the predicted performance is lower than what was actually obtained using up to twenty volunteers, but the analytical model correctly predicted trends as job and result data sizes were varied, and as volunteer numbers were varied. The execution time of jobs for this comparison was one second, which is fast and causes sufficient load on the server for testing purposes. I have detailed the validation of the accuracy of the simulator and analytical model using more realistic application scenarios in Section 5.6.

5.4.13 Job Latency

To execute a job, it has to be submitted, fetched, executed and the result submitted and the result fetched. Here I consider the best possible job latency, (turn around time), assuming relevant requests from clients and volunteers arrive at ideal times in the ideal order.
Submit job time $T_j$

This is the time it takes to submit a job. That is, to serialize the job and to send it over the network. The time taken to receive a reply to a submit job request is irrelevant. The Submit job time is expressed as:

$$T_j = Z_{jC} + Z_{jM} \cdot i + \text{network latency} + i/\text{network throughput} \quad (A)$$

Submit job service time

Assuming the best case, the server’s request queue is empty when the submit job request arrives, so the processing time on the server is simply the average service time of submit job requests, $S_{sj}$. \hspace{1cm} (B)

Fetch job service time

Sometime after the submit job request arrives at the server, but before it has been processed, a fetch job request arrives, so it is serviced immediately after the submit job request. This service time is $S_{fj}$. \hspace{1cm} (C)

Fetch job response transmission time $T_f$

Once the server processes the request, the job needs to be transmitted and deserialized on the volunteer.

$$T_f = Z_{fC} + Z_{fM} \cdot i + \text{network latency} + i/\text{network throughput} \quad (D)$$

Time to execute the job

The time needed to execute the job is, $Z_{execute}$. \hspace{1cm} (E)

Time to submit result $T_{sr}$

After a volunteer has executed a result, it needs to serialize and submit the result, which is transmitted over the network. The time required is:

$$T_{sr} = Z_{srC} + Z_{srM} \cdot o + \text{network latency} + o/\text{network throughput} + S_{sr} \quad (F)$$

Submit Result and Fetch Result Service Times

After a client submits a job, it will attempt to fetch a result. It is possible for the fetch result request from the client, or the submit result request from the volunteer to arrive in any order. Because the fetch result request is handled asynchr-
nously, it is able to block until a result is submitted. Therefore, only one fetch results request is required, and the total time that the server spends handling fetch result request and the submit result request is independent of their arrival order and is:

\[ S_{sr} + S_{fr} \]  \hspace{1cm} (G)

**Fetch results difference \( D_{fr} \)**

If it takes the client a relatively long period of time to transmit a fetch results request, it is possible that a volunteer submits a result before a fetch result request arrives is processed on the server. The interval between result arrival on the server, and the arrival of a fetch results request adds to the latency of the job. This interval will be the difference between the time it takes for a client to make a fetch results request, and the time required for a volunteer to fetch, receive and execute a job, and to submit the reply. Thus, the interval is:

\[ D_{fr} = \left( Z_{frC} + \text{network latency} + \frac{\text{request size}}{\text{network throughput}} \right) - \left( S_{fr} + Z_{execute} + T_{sr} \right) \]

The time required for the client to fetch results may be less than the time required for the volunteer to fetch, execute and submit a result, so, \( D_{fr} \) contributes to latency only if it is greater than 0. That is, \( \max(0, D_{fr}) \) \hspace{1cm} (H)

**Time for the result (fetch results reply) to arrive on client \( T_{fr} \)**

This is the time taken for the fetch results result to move over the network and be deserialised by the client, which is:

\[ T_{fr} = Z_{frC} + Z_{frM} \cdot o \cdot E + \text{network latency} + o \times E/\text{network throughput} \] \hspace{1cm} (I)

**Total Job Latency**

The best case latency is the sum of these parts, labelled (A) to (I) above:

\[ T_{dj} + S_{sr} + S_{fr} + T_{fr} + Z_{execute} + T_{sr} + S_{sr} + S_{fr} + \max(0, D_{fr}) + T_{fr} \]
5.5 Profiling Method

To derive a performance profile needed by the performance model, as set out in Table 5-1, I measured the performance of the server, clients and volunteers.

Clients and Volunteers

Clients and volunteers invoke a succession of web methods during operation. The client side costs of these invocations need to be calculated to determine the rate of method requests made to the server. Prior to making a Submit type request, the client or volunteer XML serializes the data to be uploaded. A Fetch type request requires the response from the server to be XML deserialized. These operations consume resources on the client or volunteer, and the times taken to complete these operations were determined through the use of a profiling tool, Rational Quantify. The timing data obtained from the profiler did not significantly differ from data obtained through timing the invocation of the .NET framework’s Soap serialization and deserialization methods, using profiling methods built into the .NET framework. As such, these built in profiling methods can be used to compile a performance profile if a specialised profiler is not available.

Server

Conceptually, the simulator breaks down each web request into a web server component and a database access component; these are aggregated to calculate the server resources and time needed to service the incoming request. Timing information for these operations must therefore be available to the simulator before a meaningful result can be computed. This data was obtained through measurements taken on a deployed G2 server.

Database Access Cost

The Submit Job, Fetch Job, Submit Result and Fetch Result methods were profiled by executing them without invoking them though the web server. This approach isolates method execution from any web server related overheads. The function of these web methods is essentially to provide a web interface to database opera-
tions on the G2 server, so the timings obtained from these experiments were deemed to be the cost of database access

**Web Server Overheads**

The web service cost of each operation was calculated by invoking web methods on the server, via a client residing on the server machine. Although there will be some error introduced by running the benchmarking client on the server, I have assumed this to be negligible and preferable over executing tests requiring the server to read data off the network. The execution time of each operation was timed and this total time is assumed to be the sum of:

1. Client invocation cost
2. Web server cost
3. Database access cost

The network cost is assumed to be zero because the client and server are on the same machine. As the client cost, and database access costs are known, simple subtraction can be used to determine the web server cost.

By varying the size of each request, a profile of each web method can be developed providing enough data for the analysis tools.

**5.6 Validation**

I evaluated the accuracy of the simulator and analytical model by comparing their predicted performance of a test application, to actual performance measured using up to 32 volunteers. The test application uses a genetic algorithm to evaluate a travelling salesman problem spanning 1000 cities. A genetic algorithm integrates a possible solution into a data structure known as a genome, and a set of genomes is called a population. Populations are evolved, creating a new generation of the population, and each generation should contain genomes that are closer to the problem solution than that last. I ran a sequential version of the test application
and the number of generations per second that it was able to produce is used as the baseline for determining performance, indicated by speedup.

The following graphs show measured speedup, and predicted speedup of the TSP application. I configured the average volunteer idle time to be eight minutes, which I representative of the time that a workstation may lie idle during business hours. Eight minutes is also sufficiently short to affect at least some of the experiments that I intended to perform.

![Graph showing measured and predicted speedup varying job execution time](image)

**Figure 5-18: Measured and predicted speedup, varying job execution time**

The results shown in Figure 5-18 compare the predicted and measured performance of the TSP application using jobs and results, both of 127165 bytes. The simulator and analytical model produce reasonably accurate predictions of the performance of the actual application. Both the simulator and analytical model overestimate the performance when the job execution times are relatively short, at around 10 to 20 seconds. As job execution times increase, there is an increase in the probability that a volunteer will terminate before completing a job. In that case, the job is re-executed from its beginning on another volunteer. This caused
the decline in speedup exhibited by the actual application and is reflected by both the simulator and analytical model.

Figure 5-19 shows a comparison between measured and predicted speedup, using jobs with an execution time of 20 seconds and the size of jobs and results is 127165 bytes. This combination of job execution time and job and result data sizes exceeds some network threshold when the number of volunteers is greater than 20, causing performance to decline. This phenomenon is not accurately modelled in the simulator or analytical model. However, when it occurs, application performance is poor, given the number of volunteers used, and this is predicted by the simulator and analytical model.
Figure 5-20 shows predicted and measured speedup with a job execution time of one minute. The simulator and analytical model provide reasonable predictions; while not precisely predicting speedup, they reflect the trend in performance as the data requirements for jobs and results are increased.

The accuracy of the simulator and analytical model where jobs execution times are very short is plotted in Figure 5-21.
Unlike the previous graphs, result throughput, not speedup is plotted. As expected, throughput declines as job execution time increases. At very short execution times, the shortest here being 200 milliseconds, the analytical model can produce predictions that significantly differ from what is observed in reality. However, with such short job execution times, the speedup of the application is so poor that it is not worth executing in parallel using G2. The analytical model accurately predicts this poor performance, but not the degree of poor performance.

5.7 Usage Scenarios

The simulator and analytical model can be used to improve the performance of applications, and to provide users with an insight into the parallel execution of their applications and the relationship between job execution time, data size, the number of volunteers, and the average availability of volunteers. The parameters needed by the performance model are listed in Table 5-1. The system parameters and the values I measured are listed in Appendix E. Although I had to measure these manually, a self benchmarking feature could be built into G2 in future.
Once these parameters are obtained, the only parameters that need to be supplied by users are the average job input and output data size, average job execution time, average volunteer idle time, and number of volunteers. (Although average volunteer idle time and average number of volunteers are two parameters that a G2 server could obtain over time.) Some uses of the performance model are illustrated in the following scenarios. Based on their results, a user could optimise their job characteristics to achieve the best possible performance for their application and cycle stealing network.

**Scenario 1**

The travelling salesman application using a genetic algorithm is an example of an application where the execution time of a job is independent of its data size. The data associated with a job is a population representing a number of possible tours of the cities, which is fixed, but the number of times the population is evolved per job, and hence the execution time of the job, is variable. A user with such an application might want to know the job execution time that yields the highest number of populations evolved per second.

In this scenario, each job and result is 127165 bytes in size, and up to 65 volunteers are available to execute the application. The average time that a volunteer remains connected to the system is 8 minutes.
Figure 5-22: TSP speedup with up to 65 volunteers

Figure 5-22 is the plotted output of the analytical model. The effect of volunteer volatility is evident as speedup declines as the job execution time becomes relatively long.

Figure 5-23 shows a cross section using various numbers of volunteers.
At the shorter job execution times, the communication time is not sufficiently counterbalanced by the computation time, resulting in poor performance. The effects of having a large number of volunteers can also be observed, as communication times are increased due to longer request queues on the server and network congestion. Hence, when job execution times are short, a G2 system with 65 volunteers does not perform as well as a system with a fewer number of volunteers, and derives less relative benefit as job execution times are increased. However, a system with 65 volunteers provides the greatest potential speedup peaking at around 50 times, compared to a sequential version of this application. The effect of volunteer volatility is evident; performance decreases as execution times become too large and jobs fail to complete. If only five volunteers are available, their load on the server is relatively small, resulting in fast communication times that are not significant at the smallest job execution time shown here (ten seconds).
Scenario 2

Consider an application where the execution time of a job is proportional to the size of the input data. In this scenario, the application requires ten seconds to process each 32K block of data, and a user has a choice as to the number of blocks to incorporate into each job. The performance models can be used to ascertain the decomposition of the application into a set of jobs of sizes that are likely to yield the best results. In this scenario, up to 32 volunteers are available and the average availability of a volunteer is eight minutes.

![Linked Execution Time and Job/Result sizes](image)

**Figure 5-24: Predicted performance with up to 32 volunteers. Although not shown, job and result data sizes increase proportionally with job execution time.**

As shown by Figure 5-24 speedup increases with the number of volunteers, but there is a small bias toward smaller execution times and data sizes. However, performance will decline with short execution times and a large number of volunteers, as the response times of each web service request increase, due to simulta-
neous requests from numerous volunteers resulting in longer communication times.

![Variable Execution Time](image)

**Figure 5-25: Smaller jobs can overwhelm the server if there are too many volunteers**

A smaller execution time means that the rate at which volunteers make requests to the server increases, which can have a detrimental affect on speedup. This detrimental effect, due to congestion on the server, explains the poor speedup obtained when job sizes are small and there are a large number of volunteers. Figure 5-25 shows that such congestion does not occur when there are fewer than 20 volunteers. The decline in speedup as execution time increases is due to volunteer termination. For example, the with an average volunteer idle time of eight minutes as was used, the speedup obtained with 120 second jobs with 10 and 32 volunteers is, 7.72 and 24.46 respectively. Using non-volatile volunteers, the predicted speedups using the analytical model are 9.99 and 31.9. In Figure 5-26, the effect of congestion can be seen, where speedup declines when there are more than twenty volunteers with a job duration of ten seconds. This graph also shows the effect of volunteer termination, as jobs with longer execution times exhibit
less speedup, as can be seen when comparing the 40, 80 and 120 second execution time plots. For this scenario, the number of available volunteers, and their average idle time dictates how large the jobs should be. Generally jobs of smaller execution time and data size are favoured, but care must be taken to avoid overloading the server.

Figure 5-26: The server is overloaded with many volunteers and short execution times

**Scenario 3**

The performance models can be used to determine how efficiently the system is running. That is, how does the speedup achieved compare to linear speedup using a certain number of volunteers? Efficiency is measured as a percentage. As an example, consider the efficiency of the system in executing the TSP application in the first scenario. The efficiency is plotted in Figure 5-27.
In scenario 1, a large number of volunteers coupled with a short execution time resulted in a decrease in performance, which is reflected here. If a user has no control over the number of volunteers, then the performance models can be used to configure a parallel application such that it executes efficiently irrespective of the size of the volunteer pool.
Figure 5-28: Efficiency with 20 volunteers over a range of data sizes and job execution times

Figure 5-28 plots another example using 20 volunteers running an application over a range of possible job execution times and data sizes. As the data size is increased, there comes a point where the communication cost associated with each job and result becomes sufficiently large to cause a decline in efficiency.
Looking at various cross sections at different execution times, longer execution times are able to achieve a high level of efficiency over a larger range of input and output data sizes, as communication costs are mitigated by the execution time.

Figure 5-29: Cross sections of Figure 5-28
Execution time, data sizes and volunteer volatility

**Figure 5-30: Relationship between execution time, data size and volunteer volatility**

The data plotted in Figure 5-30 illustrates the relationship between job execution time, job and result data sizes and volunteer volatility. There is an average of 65 volunteers. As job data sizes are increased, the communication cost becomes significant compared to execution time and speedup decreases. Execution time can be increased to mitigate the communication cost, but volunteer volatility can become significant, limiting the amount of speedup that can be achieved.
The interplay between job execution time, job and result data sizes, the number of volunteers, and the average idle time of volunteers, influences the performance benefit that can be achieved through the parallel execution of applications using a cycle stealing system. The G2 performance models enable users to visualise this interplay, thereby enabling them to decompose their applications to meet their performance goals.

### 5.8 Related Work

In general, there are four main approaches used in performance analysis: analytical modelling, measurement, simulation and statistical prediction, as summarised in [66]. In the performance analysis of G2, I used simulation and analytical modelling. Prior work to quantify the performance of a cycle stealing network of workstations has predominantly been based on simple or hypothetical systems. In contrast, I have conducted performance analysis and modelling of an existing system, G2.
My goal for the performance analysis of G2 was to produce an analytical model, which would produce performance estimates over a range of potential user application characteristics. I preferred an analytical model over simulation, as an analytical model would produce estimates much faster than simulation. However, in order to develop such an analytical model, I needed a representation of the runtime characteristics of a G2 system that is readily observable and could serve as the basis of an analytical model. This representation took the form of a simulator. There are existing grid simulation frameworks, such as Gridsim [67], SimGrid [68] and Bricks [69], which use discrete event simulation to simulate aspects of Grid architectures. These frameworks precisely simulate network communications, and have been used, for example, to evaluate scheduling algorithms for Grid applications. SimGrid in particular, is highly extensible, but at the time, it was not clear whether any existing simulation framework could be used to model G2, especially the functionality of the G2 server, with the level of detail that I required. Hence, I developed a new simulation framework. Principally, I could represent the functionality of the server, volunteers and clients in my simulation framework as state machines, which was useful for me for developing an analytical model. The networking model that I incorporated into my simulation framework is not as sophisticated compared to the simulation frameworks mentioned above, but it was sufficient for my needs.

I chose to base the G2 analytical model on a queueing network model using a variation of Mean Value Analysis (MVA). There are various approaches to supporting multiple request classes using MVA, for example, [70, 71]. However, I had to develop a unique approach for the G2 model, because these other techniques do not support throughput dependencies between request classes in general, and naturally they don’t support the particular interactions that are a feature of G2.

[70] describes the author’s continuing work on Stochastic Rendezvous Networks (SRVNs), which are used to model client-server interactions where each server handles different classes of requests. Unlike queueing networks, nodes in an
SRVN network can act as client and server\textsuperscript{18}. A markov chain model is used to describe the contention on the server by multiple client processes – and an aggregation technique invented by the author called task directed aggregation (TDA) is used to analyse the chain. This yields a set of transition rates that are used in the mean value analysis of the client-server network. The approach seems quite powerful and accurate – decomposition techniques exist to simplify analysis of complex SRVNs. One of the prerequisites of this model is that the clients that access a particular server are identical and randomly select the server they require. As a consequence, while the model supports different classes of request it does not support different classes of client, which is a limitation shared by the G2 analytical model, but not the G2 simulator.

One of the sources of inaccuracy of MVA is the use of mean values to describe system characteristics [72]. The mean does not reflect variations in workload which is addressed by Lüthi et.al. [71]. Their work proposes input values represented as histograms, mapping values to probabilities, to handle values that are uncertain or variable. The output values of their modified MVA technique are also histograms. This scheme may be useful in my case, not to model different classes of request, but to model the variations in response time for a particular request class, due to variations in the size of the input queue when the request is received by the server. However, I did not adopt this approach, and chose to determine response times through directly analysis of the composition of the server queue.

MVA is not the only means by which a system can be modelled analytically. A Multi-Class Jackson Network was used in [66], to create a performance model of a cluster of workstations that considered not only CPU time and communication costs, but the effect of disk access, both local on remote, on processing nodes. The

\textsuperscript{18} May be particularly useful in modelling pure P2P networks.
data obtained from their model shows that (with their particular parameters) contention on the central server for remote I/O requests does affect performance but this is masked by the communication cost, which is the real bottleneck. In other circumstances, the disk can be the bottleneck but the authors concede that this depends on the input parameters (for task CPU time, local disk time, and remote disk access time) which have to be measured from a test system and test application, if the model is to be used to predict performance on a production system. As there has been no validation on an existing system, the effectiveness and accuracy of the model is uncertain.

In [73], Varki attempts to “derive a simple approximation for mean response time of a parallel system.” It is a basic fork-join model, where a job arriving at a parallel system is split into $K$ tasks that are assigned to $K$ homogeneous task servers. Using this model, a G2 server and volunteers would be amalgamated and modelled as task servers, as the volunteers are the parallel component in the G2 system. As there are different classes of jobs or requests in the G2 model, and as volunteers can come and go, I did not adopt this approach. The fork-join model is a non-product form model and therefore differs from the queuing model used for my analysis – hence MVA, and its variations do not apply.

To perform performance analysis of G2, I used a separate simulator and analytical model. However, a hybrid approach is used in [74]. The authors identify the need to provide accurate predictions in runtime performance for non-dedicated networks of workstations (NoWs) for parallel computing. G2 is such a system. The authors develop a model, which consists of an analytical model of the network and execute nodes, called processing elements. The analytical model is then incorporated into a simulator which is used to predict overall performance. Performance prediction consists of three phases; data obtained via tests on the target system and used as parameters for an analytical model. This model is incorporated into parts of a simulator that models communication contention. This model assumes heterogeneous compute nodes (in terms of performance) and job execution
time is predicted by applying a performance rating on each node relative to a standard node. Pre-emption of processor nodes is incorporated into the model. When a node is pre-empted, that is, it is reclaimed by its owner, the node does not terminate as in the G2 model. Instead, the node enters a low priority mode, and executes in job in the background. The authors validated their model by building a test NoW. Although the authors report an error of less than 10% in most cases, and less than 15% in the worst case, the validity of the model is questioned in [66] due to the use of the deterministic distribution in modelling service time, and the analysis of the contention on the communication channels.

5.9 Conclusion

It is well known that a prerequisite to performance for a parallel application is that it decomposes into a set of jobs with sufficient execution time to offset their communication costs. However, there is currently very little, if any, support for users to determine the computation time needed for efficient execution of their applications with their job sizes on a cycle stealing system. If the effects of volunteer volatility are considered, the idle times of volunteers becomes another factor, and for the users of cycle stealing systems, judging an appropriate computation time to communication time ratio was an exercise in guesswork and intuition.

With the analytical model and simulator described in this chapter, users of G2 can observe the interaction between job computation time, job communication time, volunteer idle time, and the number of volunteers, and they can see how these factors have an effect on their application. This removes much of the uncertainty associated with cycle stealing by providing users with analysis of the performance of their applications. This analysis can be performed prior to the actual implementation of a parallel application, provided the job characteristics of the proposed application are available. Performance models enhance the usefulness of a cycle stealing framework, and I have shown how a performance model of framework may be developed.
The simulator that I have developed is a discrete event simulation, which provides a reasonably accurate model of the G2 system. It models the execution of an application using a simulation of a G2 system with a configurable number of volatile volunteers, and is able to simulate contention for those volunteers among multiple clients. The implementation of a real world G2 server uses highly tuned web server and database components, which interact to handle user requests made to the G2 web service. The interaction between these server components is not perfectly modelled, resulting in inaccurate performance simulations of a server under heavy load. This inaccuracy is not fatal however, as under such circumstances, application characteristics are such that the computation time to communication time ratio is not sufficient to justify execution of the application using G2. In other words, the G2 simulator is able to accurately inform users that application execution is not worthwhile using G2, but fails to precisely quantify how poor the performance will be.

The analytical model performs performance analysis much faster than the simulator. Therefore it is able to produce results over a large range of job parameters in a relatively short period of time. For example, the results that I produced to perform comparisons with the simulator and the real world system were generated almost instantaneously. In contrast, the simulator required over an hour to generate its predictions for some of the scenarios.

The multiple request classes of G2, and the unique dependencies between classes necessitated a unique form of Mean Value Analysis. Using Schweitzer's approximation technique [65], it uses iterative evaluation for the convergence of response time, throughput and queue length values, which are produced by the model. Although the analytical model does not capture a G2 server's ability to execute requests in parallel, this may be remedied in future by modifying the current single server queueing model into a multi server one, where each server represents a resource on the G2 server, such as the web server and database. Of course, such a queuing model would greatly complicate the construction of an analytical model.
Another future modification may be the use of a different, more recently developed approximation technique, co-developed by Schweitzer, which is shown in to be more accurate then other known methods and is introduced in [75].
Chapter 6 Conclusion

6.1 Summary

In this thesis I have demonstrated how the use of a cycle stealing framework, in particular application development, can be simplified through a programming model for cycle stealing on the Internet. I have incorporated this programming model, and other ideas for the efficient and reliable execution of flexible parallel applications, into a cycle stealing framework called G2. I have shown how a performance model can be developed and through performance analysis I have produced a simulator and analytical model for the G2 framework. I have shown how these performance models can be used to assess the value of parallel execution through cycle stealing, optimise the decomposition of applications, and to gain an insight into the interplay between the many factors affecting application performance.

The architecture of G2 is described in Chapter 2, a design defined by the need to operate on the Internet, enabling any computer with web access to make use of, and contribute to a cycle stealing system. I showed how commodity components can be leveraged to assemble the cycle stealing infrastructure, and unlike previous web based systems, volunteer components hosted in browsers that are capable of seamlessly switching between applications.

In Chapter 3, I detailed the ease of use of G2. The deployment of a G2 server simply consists of running a database script and installing a web service onto a computer running a web server. To simplify application development, I developed a programming model specifically for cycle stealing applications for the Internet. It is characterised by a high level of abstraction, making parallel applications resemble ordinary sequential applications that consume web services. Making application development easier has many advantages, but in particular:

1. Simple application development means that people who are not necessarily expert parallel programmers, will be able to easily write parallel appli-
cations. This, combined with the low financial cost of cycle stealing, puts high performance computing within the reach of a wide range of users, many of whom may not have had the access, or the expertise, to use a high performance computing system in the past.

2. Simple application development enables existing applications to be ported easily, and means that programmers should be able to write correct applications faster, giving users the flexibility to experiment with their applications, develop multiple prototypes, and use a rapid application development approach to solve their problems. The consequences of application design errors can become less significant, if the incorrect application can be easily refactored.

The G2 model shows how a high level programming model, that of ASP.NET web services, can be used for parallel programming. This model is flexible enough to enable data flow style programming, and supports further abstractions to simplify different styles of parallel programming. I provide an example for data parallel applications, which also includes a demonstration of the possibility of RMI on distributed objects.

Starting a parallel execution is simplified through the automatic deployment of user code across a cycle stealing network, with support for versioning and unique names. This mechanism operates purely over HTTP, remaining faithful to the web based approach of G2.

A cycle stealing framework is required to provide high performance, reliability and safety. The mechanisms I had incorporated into G2 to meet these requirements are detailed in Chapter 4, and shows the feasibility of web based approaches to cycle stealing. I have considered the execution of applications that have multiple stages, where some jobs must be completed before other jobs can commence. In such applications, minimising the turnaround time of jobs, from job submission to result retrieval, is crucial to performance. I have developed
ways to minimise the turnaround time, and improve the efficiency of communications.

I investigated the widely used eager scheduling algorithm, and devised modifications to make it more efficient and described some circumstances where it can actually be detrimental to performance. I used an existing mobilization framework to demonstrate the possibility of automatic checkpointing for G2. I showed how a managed execution environment provides safety and that that running native code does not come without penalty. I showed how the combination of ease of application development, automatic deployment, and efficient execution enables G2 to support rapid application development and applications for cycle stealing environments that encompass fast executing, throw-away applications, as well as long running grand challenge like applications. By executing applications on a pool of 60 volunteers, I validated G2’s performance and reliability features by achieving near linear speedup with applications with realistic job execution and data requirements.

In Chapter 5 I provided a solution to the uncertainty surrounding application execution using cycle stealing through an analytical model and a simulator that are capable of modelling the execution of a parallel application on G2. I showed how these can be used to optimise applications through a series of usage scenarios.

6.2 Future Work

G2 as a Cycle Stealing Research Platform

Apart from being a vehicle for high performance computing, G2, with improved componentisation of especially the server, could be used as a platform to experiment with different aspects of cycle stealing. For example, I have used web services, hosted by a web server, as the interface to the G2 database, but any number of other options exists. The server is a clearinghouse for jobs and results and I implemented using a database, but alternative technologies could be used and tested,
to store jobs and results. Then there are the algorithms that dictate the behaviour of the server. One example is the scheduler controlling the allocation of jobs to volunteers. Perhaps the hybrid Eager scheduler and volunteer pulse mechanism that I use could be improved, or different scheduling regimes are better for different scenarios.

**Standardised Profiling and Modelling**

I had to perform extensive performance modelling to obtain the measurements that were used in the analytical model and simulator. Measurements such as the time it takes for the database to service queries with different input and output data sizes, and the time it takes for IIS to service a web service request. It would have been useful to build profiling mechanisms directly into the G2 framework. In addition, it may be useful to create a standard profiling system such that if a G2 server component were replaced with something completely different, such as the G2 web service, then only the new component would have to be profiled. The G2 performance model could be modified to support this standardized profiling system, so the data from the new component could be easily integrated.

**Scalability**

The G2 system used a single server, but there are approaches to distributing this load such as the broker architecture of Javelin [26], and work stealing of Cilk [18]. I would prefer a network of servers approach, which is able to distribute load in two ways. Firstly, clients and volunteers can be redirected to other servers. Secondly, servers are able to act as “volunteers” and fetch work from another server. As an example of this option, suppose a server, A, has volunteers, but no connected clients. Server B, has volunteers and many jobs available for execution. Because server B is already working at near maximum capacity, simply redirecting A’s volunteers to B is not an option. In that case, server A acts as a volunteer, and fetches jobs from server B. To server B, server A is nothing more than another volunteer. Now that A has fetched some jobs, it is free to assign them for
execution to its volunteers. When results are computed, server A submits the results to server B, as a normal volunteer.

**Ignoring Firewalls and P2P networks.**

I wanted to make G2 more inclusive and be able to operate in the presence of firewalls. However, there are many large P2P networks in existence, so perhaps large P2P cycle stealing systems are viable. There are many advantages of P2P, such as removing the bottleneck of servers, and enabling direct communication between clients and volunteers, including direct communication between volunteers, and many parallel applications will probably map more naturally to this type of network over G2. Of course, a P2P cycle stealing system that resides wholly behind a single company firewall will also be possible. However, cycle stealing on P2P networks introduces many new challenges to overcome [76].

**Effects of Deploying a Cycle Stealing System on Networks**

One of the potentially negative side effects of deploying a cycle stealing system is the resulting increase in network traffic. Such an increase may lead to a decrease in the overall responsiveness of the network; this means an increase in latency, and less bandwidth for non-cycle stealing related traffic. The SETI@Home project, for example, has significant network related financial overheads. That project has thousands of participants all around the world, and is of a very large scale, but it shows that cycles stealing does have infrastructure costs. The effect of deploying a cycle stealing system on a network has been largely ignored and should be investigated. It may turn out that the effect is insignificant and can be ignored. Or, it may turn out that the network burden limits the number of potential of clients and volunteers that can be supported without upgrading network infrastructure.
**Cycle Markets**

The idea of creating markets for trading cycles is interesting and there has been some research in this field. [43] is an example. How could G2 be integrated into such markets? The G2 server is able to track how much CPU time has been consumed and provided by individual users, (if they choose to register with the server), so some of the infrastructure required to support such markets is already in place. The server could also note the *quality* of the donated time, that is, the power of the volunteer, to assess how much work was contributed by users. Naturally, this type of information could also be used to set up user profiles, which could also be used for better scheduling techniques.

**Data Mining**

The pattern of cycle stealing, in its present form, is that volunteers download jobs including the data needed to process that job, and the data persists on the volunteer only for the duration of job execution. The use of code caching in G2 is a minor deviation from this pattern, but size of the cached code is still small. There are a class of applications, such as data mining, that is suitable for parallelization, but are heavily data dependent. For cycle stealing to be applicable to these types of applications, the present pattern of cycle stealing of not persisting data on volunteers, will have to be changed. Volunteers downloading large amount of data to perform a single search for example, is likely to be not feasible due to communication costs. If however, volunteers were capable of persisting such data, then the initial communication costs will be amortised by a number of operations.

When a volunteer disconnects from the network, the data on that volunteer also becomes unavailable. So, supporting data dependent applications in a cycle stealing environment will require the management of distributed data, and mechanisms to ensure that most, if not all of the needed data is available on a cycle stealing network at any one time.
Data dependent applications, such as data mining, require a different model of cycle stealing than that of existing cycle stealing frameworks, which is a potential avenue for future work.
Appendix

A  G2 Infrastructure Components

A.1  Server

The basic function of the server is to act as an intermediary between clients and volunteers. It receives jobs from the client and assigns them to volunteers. It receives results from volunteers and holds them until retrieved by an appropriate client. As such, the server becomes the manager of an executing application and is responsible for scheduling the execution of jobs, and ensuring applications execute in a timely manner with hopefully the correct results. The server must also be resilient to failures, its own or those occurring on volunteers. It should not lose jobs or results under any circumstances.

To ensure the timely execution of applications, the server needs to keep track of the status of volunteers, and react appropriately when a volunteer becomes unavailable. If a volunteer fails, it is advantageous for the running application if the server detects this as quickly as possible since an assigned, but non-executing job prolongs the total execution time of the application. The volunteer failure detection mechanism cannot simply rely on volunteers to inform the server that they are shutting down, as they may withdraw from a G2 system unexpectedly. Therefore an active detection mechanism is required. This functionality had to be implemented using the request-response model of communication to the G2 server. Ideally, the server would periodically query a volunteer to check if it were still alive based on some heuristic, but the request-response model dictates that the volunteers initiate any communication.

The method chosen for G2 is to have each volunteer periodically send a pulse to the G2 server while it is active. The server records the time a pulse was received for each volunteer and uses this data to determine which volunteers are no longer available. It is possible that a volunteer is still active and executing a job, but communication links to the server has been severed. From the server’s point of
view, this is akin to a failed volunteer. When a volunteer is deemed to have left the G2 system, any jobs assigned to those volunteers are marked for reassignment. Jobs in the G2 system are independent units of execution that execute to completion without reliance on other jobs. They do not need to communicate with other jobs, and they only need to communicate their results to the server once complete. Therefore, on volunteer failure, the server need do nothing more than reassign the job, as jobs are idempotent. If this were not the case, simple re-execution could affect the result of an application. The system would require mechanisms to rollback an application to a safe state, much like a database server rolls back a transaction. This would involve the server creating application checkpoints, and knowing when it is appropriate to create them, and when it is safe to revert back to them if required, since other jobs deployed after the creation of the checkpoint may still be executing. This is difficult to do, is not efficient, and potentially too expensive for a cycle stealing system where volunteer failures are common.

The alternative to the continuous monitoring of volunteers is to implement Eager scheduling. Eager scheduling was first introduced in the Charlotte project. Recognising the difficulties associated with maintaining the status of volunteers, the authors of Charlotte created a scheduling scheme where incomplete jobs are simply redundantly assigned to available volunteers. This redundant scheduling of jobs to multiple volunteers not only had the advantage of foregoing the requirement to monitor job execution on volunteers, but it prevented slow volunteers from slowing down application execution.

Optimally scheduling jobs is difficult for cycle stealing systems due to the incomplete information available to them. By using a more sophisticated volunteer component, systems such as United Devices [34] can gather data on the capabilities and resources available to a volunteer. Even so, the scheduler has no way of knowing what the runtime characteristics of a particular job will be, and it cannot know which volunteers will be available in the future. As a result, current systems, including G2, adopt a first come first served policy, treating all jobs and
volunteers equally. With such a policy, there will be situations where complex jobs will be assigned to inappropriate volunteers. By redundantly allocating jobs, eager scheduling can eliminate the effects of these types of poor scheduling outcomes provided enough volunteers are available.

A.1.1 Server Implementation

The general design and implementation principle of G2 is to use existing parts and protocols where appropriate. This has the advantages of simplifying system implementation, and using components that are readily available, trusted, and familiar to users.

To broaden the potential volunteer base to include workstations behind (most) firewalls, SOAP web services which operate over HTTP was chosen as the communication protocol for G2. The advantages of this protocol is that it operates over a well known port (typically 80) and the server does not initiate requests, which makes it compatible with most firewalls, often with no configuration changes required.

This choice necessitated G2 to expose a web service, hosted on a web server. IIS and .NET web services were chosen as the underlying platform since they offered a simple and well documented solution. The requirement of the server to persist data and recover from failures implied a database which would serve as the primary data store for G2, including jobs and results. Again an existing platform, SQL Server was used. The simplicity and integration between SQL server and IIS made it the obvious choice although other, free and capable database options also exist.

All communication occurs through the web services interface exposed by G2. Clients and volunteers invoke web methods to communicate with the server. These web methods allow clients and volunteers to join and leave a G2 network. Once initial registration has occurred, clients use the methods, SubmitJob and FetchResult, to submit and retrieve jobs and results respectively. Volunteers invoke the
FetchJob and SubmitResult methods to receive new jobs and submit computed results. All data required by these methods are XML encoded by clients and volunteers and passed to and from the server as strings. So job parameters are XML encoded as well as the results.

**RegisterClient**

When a client first connects to the server, it invokes this method. The server records the existence of a new client and gives it a class identifier. This identifier is used to return results back to the correct client as well as recording how many jobs are submitted, and how much volunteer time is consumed, by a particular client.

**RegisterVolunteer**

This method is first invoked by volunteers when they connect to the G2 server. The volunteer is returned an identifier, which it quotes whenever it interacts with the server. The server uses this identifier as a simple security mechanism to ensure that a volunteer returning a result was the same volunteer to which the job was assigned. The server will not assign a job to an unregistered volunteer. The identifier is also used to maintain the status of volunteers. If a volunteer disconnects from the server, the identifier is used to quickly identify those jobs that need to be reassigned.

**RegisterClass**

G2 exposes a functional, not an object oriented programming model, but it does assume that these “functions” are implemented in an OO language and are therefore members of a particular class. Each job is actually a method invocation of a particular user defined class. A client must register each class with the server. During this process, the server determines whether the assembly containing the class exists on the server. If not, the client required to upload the assembly onto the server. The upload process occurs via a web method (UploadAssembly) which takes the assembly as an array of bytes. A particular assembly need only be up-
loaded once, since it is cached on the server provided sufficient disk space is available. The server will periodically flush unused assemblies from its cache. If these assemblies are required, then the client will have to re-upload them. An assembly will never be deleted if it is required by a currently running application.

**SubmitJob**

This method is called by clients to submit a new job onto the server. The parameters to this call are the client’s identifier (obtained when the client first registers), the class identifier of the job (obtained when the client registers a new user class), the job method name and the parameters.

When invoked, the server will insert a new job entry into its database and the job will be tagged for execution. The client will receive a job identifier which it will use internally to trigger a response when the result for that job is eventually retrieved.

**FetchJob**

This method is invoked by volunteers to retrieve jobs for execution. The server uses the supplied volunteer identifier to tag the job as assigned to that particular volunteer. If the server determines that a volunteer has failed, it will reassign that job at the next opportunity. The server also records the number of times the job has been assigned for eager scheduling purposes. The incomplete job with the least number of assignments is typically the one chosen for eager rescheduling.

**SubmitResult**

When the volunteer has completed execution of a job, it submits the result to the server. The server records the result, marks the job as complete, and updates the computation time consumed by the client according to the job execution time reported by the volunteer.

**FetchResults**

This is the method called by clients to retrieve results and is handled asynchronously on the server side. This asynchronous behaviour has several advantages.
The restriction that the server cannot initiate communication means that the server cannot notify clients when results are available, or push results unprompted back to clients. A client has no way of knowing when a result becomes available, so without asynchronously handled web methods, the client will have to constantly poll for results. Without this asynchronous behaviour, the server will have to deal with the load of each client continually polling for results instead of asynchronously handling one fetch results call per client. Another advantage over polling is that there can be a delay of up to the poll period between the time when a result becomes available and when the result is retrieved. This poll period could be configured to be quite long to decrease the load on the server. Asynchronous web methods on the server side allow the server to return a result to the client as soon as one becomes available. The functionality mimics that of the server pushing results to clients unprompted, through a client pull mechanism.

There is a standard way to implement asynchronous ASP.NET web services which requires the programmer to implement two methods. Suppose the programmer wants a method called Foo to be handled asynchronously on the server side. In that case, the programmer is implements two methods, BeginFoo and EndFoo. Each of these methods has a standard signature which is used by the .NET runtime to identify an asynchronous method implementation. The last two parameters for a BeginFoo method must be an AsyncCallback and an Object, used to store state information. The EndFoo method must take an IAsyncResult object as its only parameter. This asynchronous implementation is opaque to the caller of the method. The caller just calls Foo with the parameters of BeginFoo minus the AsyncCallback and the state object. IIS will handle the method asynchronously since methods with the prefix “Begin” and “End” exist for Foo with the correct signatures. If an asynchronous BeginFoo and EndFoo implementation exists, a synchronous method named Foo is not allowed. ASP.NET web services does not allow overloading.
The fetchresults method will return all available results belonging to that client as the reply to the fetchresults request. This is irrespective of the class to which each job belongs.

**Asynchronous Fetch Results Mechanism**

In accordance to the asynchronous web method pattern, the Fetch Results web method is implemented in two parts. The Begin Part of the method takes an ASP.NET generated callback to the End part of the method as an input parameter. If a result is available, the callback is triggered. The End Part causes a result or results for the client to be extracted from the database, and returned. If there are no results, the callback is stored until a result becomes available.

The callback is stored in memory, in the G2 web service’s Application State. This is a feature of all ASP.NET web services, and enables a web service to share data between web method calls in the form of a shared dictionary. Whenever a result is submitted, the server checks whether a callback for a fetch results operation exists in the Application state. The client Id is used as the key, so callbacks from other uninvolved clients are ignored. If a callback exists, it is removed from the Application State and triggered, causing the execution of the End part of the Fetch Results method.

The use of the Application state does raise some issues however. The first is that it is not intended to be a means by which web services become stateful. IIS can be configured to periodically restart any web application, including a web service. This is to prevent misbehaving applications from affecting the performance of the entire server. One of the side effects of such a restart is that the Application state is destroyed, and any stored callbacks are lost. I have assumed that the fetch results mechanism is not reliable; if no callback is triggered on the server side, the fetch results request will eventually time out on the client and it will issue a new fetch results request. This functionality combines the benefits of minimal server load of a polling mechanism with a long period, with the immediacy of a short polling period, without their drawbacks.
The other consequence of relying on the Application state affects techniques that can be employed to scale the server. One the major benefit of using commodity components is that it enables users to leverage the work that has been carried out to improve that component. In the case of G2, use of IIS and Sql Server 2000 running on Windows Server 2003 allows for use of the clustering technology offered by that platform. This clustering technology allows administrators to simply combine servers together to create a web farm, which scales the web site, or web service. This technology can be used to easily scale a G2 server, but there is one undesirable consequence. The Application state is bound to a particular machine, regardless of whether it is part of a cluster (web farm). As a consequence, if the G2 server is deployed as a web farm, if Server A handles the fetch result operation, and Server B handles a subsequent submit result operation, since server B does not have access to server A’s Application state, the callback will not be triggered. The client will not actually receive the result until the first fetch results call times out and it tries again.

Callbacks are bound to a particular computer, and can only be stored in the Application State. The callback supplied to the Begin method of an asynchronous web method is not a direct callback to the End method. It is actually a callback to a method called Callback which is a member of the System.Web.Services.Protocols.AsyncSessionlessHandler class. This class contains all the information regarding the request, which cannot be migrated because it is bound to a particular TCP connection. For an optimised multi-server solution, some mechanism is needed that triggers the correct callback on the appropriate machine when a result is received. A custom scaling technique, not based on standard Windows clustering technology is needed to do this properly; otherwise the asynchronous fetchResults mechanism becomes a defacto polling mechanism with the call timeout period as the poll interval.
Due to the imperfect nature of the Asynchronous mechanism for G2, and the need to store callbacks for each request, I decided to limit asynchronous execution to only the fetch results mechanism.

**ClientPulse and VolunteerPulse**

In addition to these methods, there is a pulse method invoked by clients and volunteers that inform the server that they are still active. This method provides an opportunity for future expansion since it is a periodic poll from the nodes to the server. The server could return data about the status of the server or the current application for example. Currently, the server determines whether a volunteer is still active by the existence of a pulse. If there is no pulse for a preset period of time, the server will mark that volunteer as having left and will reassign jobs. If a client ceases the pulse the server will assume that the client has failed and will no longer assign jobs from that client to volunteers. This behaviour is not ideal because it is incompatible with long running clients, or clients that may wish to disconnect periodically. However, it is adequate for the initial version of G2.

**A.2 The Client**

The client submits jobs for eventual execution on volunteers. The functionality of the client is incorporated into proxies which register the client, submit jobs and retrieve results. The client component of G2 is actually implemented as two proxies. The first is the class proxy, which submits jobs onto the client. The other is the client proxy which is responsible for initially registering the client, retrieving results and sending pulses to the server. The client proxy operates independently; it’s operations are not directly triggered by programmers.

These proxies are autogenerated for each class that the user supplies. Programmers are required to instantiate an instance of a class proxy with a string representing the URL of the G2 server to be used. Once a proxy is created, it will detect or create an instance of the G2 class proxy. In this way, there is one G2 class proxy per client.
proxy per application domain on the client machine. This assists in cutting down
the levels of communication between the client and the server. When a job is
submitted, the server returns a job identifier. The client stores this identifier
along with an instance of a G2AsyncResult in a hashtable. A G2AsyncResult is
simply an implementation of an IAsyncResult and contains a waithandle on
which a user can block while waiting for the results, or a callback method that
the user supplies to be triggered when the result returns. Results are retrieved
from the server in a data structure that contains the job identifier, as well as the
XML encoded result. The client proxy retrieves the correct G2AsyncResult object
from the hashtable, inserts the result into its results field, and either sets the
waithandle or invokes the callback.

The fetchResults operation itself is handled by a separate thread on the client,
which comes into existence and stays in existence if there is an outstanding re-
sult. That is, the client has submitted a job to which no result has yet been re-
turned.

There is yet another thread for the pulse operation. It is responsible for periodi-
cally sending a pulse to the server to inform the server that the client is still op-
erational and interested in the execution of its submitted jobs.

A.3 The Volunteer Host

The volunteer host is that component that executes the job on behalf of the cli-
ents. When a workstation becomes idle, the workstation connects to the G2
server and registers itself as an available volunteer. This is a two part process. To
expedite software delivery onto an available volunteer workstation, the work-
station opens up a web browser and connects to an ASP.NET web page on the
server, which has embedded within it the G2Volunteer Component. This has the
advantages of requiring no pre-installation, and forces the G2Volunteer compo-
nent to operate within the sandboxed Internet Zone environment of the .NET
runtime. The code within the web page forces the volunteer machine to create a
new instance of the G2Volunteer component with the URL of the server as a pa-
rameter. If this is not the first time that a workstation has volunteered, the em-
bedded component should already exist in the download cache of the volunteer,
so in that case, no downloading is required. On initialisation, the G2Volunteer
registers itself and calls the FetchClass operation. This operation returns the class
of any job that is ready for assignment on the server. The fetchclass operation also
returns the name of the assembly that implements the class and where the volu-
teer can obtain it on the server. The volunteer then tries to load the assembly
from this location. If the assembly does not exist on the volunteer side, then it
will download it from the server. Otherwise, the assembly will exist in the
download cache of the volunteer.

After the assembly is loaded, the volunteer creates and instance of the class, and
then contacts the server for any jobs to execute using methods of that class. If no
jobs exist, which could occur if some other volunteer is assigned the available jobs
that were present when the present volunteer requested a class, the volunteer
will simply reconnect to the server and ask for another class with available jobs.
If no such class exists, the volunteer will sleep for a period and try again. The vol-
unteer will continually increase this sleep time up to a specified maximum time if
it continually fails to retrieve a class.

If there are jobs available, the volunteer will extract the method to execute from
the job, as well as the XML serialized parameters, and invoke the appropriate
method on the class instance with the specified parameters. When the job is
complete, the volunteer will take the return value, XML serialize the result and
return the result to the server. It will then attempt to retrieve a new job, for that
class from the server.

Once a volunteer retrieves a class, it will attempt to execute jobs exclusively for
that class until all jobs become unavailable. (Either jobs are complete or assigned.)
This is to minimise the downloading of assemblies by a volunteer, as a new as-
sembly is typically required for each class.
B  **Queueing Theory**

A queueing network model is a method of computer system modelling, where the system is represented as a network of queues. This model is capable of being evaluated analytically. A queueing network model consists of a series of queues each representing a system resource. Customers, representing transactions with system resources, flow throughout the network. The route that a customer takes through the network represents a particular usage scenario of the computer system.

There are two types of service centres in a queueing network, representing system resources that sequentially handle requests, and those that can handle requests in parallel. Resources that sequentially handle request are known as queueing centres, because they have to maintain a work queue, and any customer interacting with the resource must first enter the queue. Resources that can handle multiple requests in parallel are known as delay centres and do not maintain a queue. They are termed delay centres, because the resource always takes the same amount of time to handle a transaction with the customer, irrespective of any other customers using the resource. The delay centre is a fixed “delay” that the customer endures before continuing through the network.

In queueing network terms, the server represents a service centre, and each job is a customer that moves through the network. Each web method invocation is an interaction with the server and represents a service request by a customer. The request enters a queue maintained by the server and is handled in a FCFS manner.

The aim of the model is to provide an accurate estimate of the throughput of a G2 system. Therefore, after seeding the system with a user specified number of jobs, the client will submit a new job via a submit job request, for each result that it receives. While the reality is that new requests are being submitted, conceptually this is identical to requests continually circulating through the system. This en-
sures that the system is never starved of requests, limits the total number of re-
quests and makes the system a closed queueing system.

B.1 Mean Value Analysis

The benefit of a queueing network model is that it can easily be analytically
evaluated. For closed queueing networks, the standard technique is called mean
value analysis, MVA. It is based on three equations that evaluate system through-
put, \(X\), average queue length, \(Q\), at each queueing centre \(K\), \(Q_K\), and Response
time \(R\), at each queueing centre \(K\), \(R_K\).

**Throughput**

The throughput, \(X\), of a system with \(N\) customers is the inverse of the time that a
customer spends in the system (throughput of one customer), multiplied by the
number of customers.

\[
X(N) = \frac{N}{Z + \sum_{K=1}^{K} R_K(N)}
\]

Where, \(N\) is the number of customers in the system and \(Z\) is the time a customer
spends not using system resources, for example, the average time that the cus-
tomer waits for user IO. A popular example of closed queueing networks is a
model consisting of terminals and their transactions with a central server. The
model is closed because there are a fixed number of terminals, but these terminals
often do nothing because they are waiting for some action by their user. It is this
wait that is represented by \(Z\), and is commonly known as think time.

A time taken for a customer to walk through the network is the sum of the think
time, and the time it spends at each resource, given by \(Z + \sum_{K=1}^{K} R_K(N)\).
Queue Length

The average length of the queue at a queueing centre $K$ is determined by the rate at which new customers arrive at $K$, and the average time $R$, that a customer spends at service centre $K$. It is given by:

$$Q_k(N) = X(N)R_k(N)$$

Where $X(N)$, the throughput of the system, is the rate at which new customers arrive at $K$. The response time is the total time that a customer spends at a service centre, including the time spent waiting in the queue.

Response Time

The response time is the actual service time at a service centre, and the time spend waiting in the queue. When a customer arrives at the back of the queue, its response time is its own service time, plus the service times of all of the customers already in the queue.

$$R_k(N) = \begin{cases} D_k & \text{(Delaying Centres)} \\ D_k \left[1 + A_k(N)\right] & \text{(Queueing Centres)} \end{cases}$$

Where $A_k(N)$ is the number of customers already present at queueing centre when a new customer arrives. $D_k$ is the demand that on a particular service centre placed by a customer. $D_k$ computed by:

$$D_k = V_kS_k$$

Where $V_k$ is the number of times a customer uses the resource and, $S_k$ is the service time, or the time a service centre takes to transact with the customer. (In the simple case, $V_k$ is one.) The MVA formulae shown here are for queueing networks with only one type of customer. In this case, the response time at queueing centre $K$ is the time taken to handle the incoming customer, $D_k$, and the time taken to handle all prior customers in the queue, $A_k(N)$.
The key to solving this network is to determine the value of $A_k(N)$, or the number of prior customers in the queue, because with a value for $R$, evaluation of $X$ and $Q$ follow.

**B.2 Multiclass models**

A limitation of the above MVA technique is that it assumes only one type of customer. That is, each customer instance takes the same route through the network as all other customers, and the transactions with each resource take the same amount of time. In G2, clients make different requests to the server, each using different resources and requiring different service times. The presence of multiple classes of requests requires evaluation using multiclass MVA.

Analysing closed multiclass queueing networks follows the basic pattern of single class MVA, that is, $R \rightarrow X \rightarrow Q$.

**Throughput**

$$X_c(\tilde{N}) = \frac{\tilde{N}_c}{Z_c + \sum_{k=1}^{K} R_{c,k}(\tilde{N})}$$

The general equation is similar to single class MVA except throughputs, think times and response times are evaluated with respect to a particular class of customer, $C$. Hence, $R_{c,k}$ is the response time for a customer of class $C$ at service centre $K$.

$\tilde{N}$ is a vector representing the number of customers of each class present in the system. For example if there are 2 customers of class A, and one of class B, $\tilde{N}$ is (2,1).

**Queue Length**

$$Q_{c,k}(\tilde{N}) = X_c(\tilde{N})R_{c,k}(\tilde{N})$$

Queue lengths are evaluated for each class of customer for each service centre.
Response Times

\[ R_{C,K}(\tilde{N}) = \begin{cases} 
D_{C,K} & (\text{Delaying Centres}) \\
D_K \left[ 1 + A_{C,K}(\tilde{N}) \right] & (\text{Queueing Centres}) 
\end{cases} \]

Where \( A_{C,K}(N) \) is the length of the queue at queueing centre K when a customer of class C arrives.
C G2 MVA Equation Coupling

In Chapter 5 I mentioned that the close coupling between request classes necessitated the evaluation of the G2 queueing system using approximation techniques. The following graphs illustrate the dependencies between MVA variables due to the dependencies between job classes. Here, I have used the Equation to calculate the response time of the Submit Job, to show how evaluation is not possible.

![Diagram](image)

Figure C-1: Direct Dependencies of the Submit Job Response time equation

This graph displays the direct, or first generation dependencies between the equations, also displaying the dependency on the number of volunteers, V, and the number of clients, C. The arrows indicate the direction of the dependency, so to evaluate the throughput of the submit job operation, (X Submit Result), the re-
response time of the Submit Result operation, \((R \text{ Submit Result})\) must be known. So far there are no cycles.

However, each of the equations that appear in the first generation, are dependent on other equations for their evaluation. If these second generation equations are also included, the directed graph becomes:

**Figure C-2: Second Generation Dependencies**

Already there is a problem. The throughput of the Submit Job operation \((X \text{ Submit Job})\) depends on the value of the response time of the Submit Result Operation \((R \text{ Submit Response})\). In turn, this is dependent on the number of Submit Job requests in the queue \((Q \text{ Submit Job})\) when a volunteer makes the request to the
server. But, the value of $Q$ Submit Job depends on the throughput of the Submit Job operation, creating a circular dependency.

The full dependency graph across all generations shows many such circular dependencies.

![Full Dependency Graph](image)

**Figure C-3: Full Dependency Graph**

The presence of these circular dependencies precludes evaluation of the G2 queueing network in one pass. Therefore, an iterative solution is used where initial values of $R$, $X$ and $Q$ are chosen for each customer class. This is not unlike approximate MVA techniques, for example using Schweitzer’s approximation. These solutions are refined through each evaluation of the formulae until the values stabilize.
Value Refinement

Each time that the equations are re-evaluated, the existing value is altered to be the mean of the existing value, \(X\) and the new calculated value, \(X'\).

\[
X = \frac{X + X'}{2}
\]

For example, the value of the Response time for a submit job operation, given by

\[
R_{sj} = S_{jy} \cdot Q_{jy} + S_{y} \cdot Q_{y} + S_{fr} \cdot Q_{fr} + S_{yj},
\]

would be re-evaluated as:

\[
R_{sj} = \frac{R_{sj} + S_{jy} \cdot Q_{jy} + S_{y} \cdot Q_{y} + S_{fr} \cdot Q_{fr} + S_{yj}}{2}
\]

The use of the mean can slow down the rate of refinement, but it prevents anomalous values from affecting the final outcome.

The equations are evaluated until the differences between the existing \(Q\) value and the newly calculated \(Q\) value falls within an acceptable tolerance, \(\tau\).

\[
\tau = \frac{|Q_{sj} - (X_{sj} \cdot R_{sj})|}{\max(Q_{sj}, X_{sj} \cdot R_{sj})} + \frac{|Q_{jy} - (X_{jy} \cdot R_{jy})|}{\max(Q_{jy}, X_{jy} \cdot R_{jy})} + \frac{|Q_{fr} - (X_{fr} \cdot R_{fr})|}{\max(Q_{fr}, X_{fr} \cdot R_{fr})}
\]

The probability of a successful fetch job

In the stable volunteer model, the probability that a job would be available for execution when a fetch job request was processed was the ratio between the throughput of job arrival, given by \(X_{sj}\), and the throughput at which jobs were requested, \(X_{jy}\). The equation for \(r\), the probability that a job is available when a fetch job request is processed was:

\[
r = \frac{X_{sj}}{X_{jy}}
\]

In the presence of volatile volunteers, the frequency at which jobs become available to volunteers is not determined solely by the frequency of submit job requests. Jobs also become available when a volunteer, that had previously been assigned a job, terminates during its execution. These previously assigned jobs are
then available to be reassigned to other volunteers to ensure their execution. So, I had to factor in this alternate source of jobs, when computing the value of \( r \).

If a volunteer successfully fetches a job, the probability, \( p \), that it will be executed and submitted is:

\[
p = e^{\frac{-Z_{\text{success}} + Z_{\text{fail}} + R}{p}}
\]

The rate of failed executions is the overall rate of successful fetch job requests which resulted in failed job executions and result submissions. The overall rate at which jobs become available is the sum of the job submission rate and the failed execution rate, or:

\[
X_{ij} + r(1 - p)X_{\bar{g}}.
\]

The probability of a successful fetch job operation with volatile volunteers can now be described as:

\[
r = \min \left( 1, \frac{X_{ij} + r(1 - p)X_{\bar{g}}}{X_{\bar{g}}} \right) = \min \left( 1, \frac{X_{ij}}{X_{\bar{g}}} / p \right)
\]

As this is a probability value, it is bounded by 1. This bounding is performed to prevent the initial guessed values for \( R \), \( X \), and \( Q \) from creating impossible probability values.
D Mean number of failed fetch job attempts

The probability of successfully fetching a job is, \( r \), and the probability of failing to fetch a job is simply, \( 1-r \).

Let \( x \) be the number of failed fetch attempts before a successful fetch. Considering the probability, \( P \), for various values of \( x \):

\[
P(x = 0) = r \\
P(x = 1) = (1-r)^1 r \\
P(x = 2) = (1-r)^2 r \\
\ldots \\
P(x = n) = (1-r)^n r
\]

The probability distribution of \( x \), \( f(x) \) is:

\[
f(x) = (1-r)^x r
\]

The mean, or expected value of \( x \) is:

\[
\mu = E(x) = \sum x(1-r)^x r
\]

\( x \) is geometrically distributed with probability of failure, \( 1-r \), hence the mean,

\[
\mu = \frac{(1-r)}{r}
\]
### Performance Model Parameter Table

The simulator and analytical model require input parameters of a G2 system and its network environment. Using a commercial profiler, and custom profiling tools, I obtained the value of these parameters on a particular G2 system. The parameters and the values that I obtained are listed below. Note that these values are the average measured values accurate for the G2 system that I measured, but there is an inflated level of precision.

#### Client Parameters

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Job serialization time constant:</td>
<td>0.4E+00 (ms)</td>
</tr>
<tr>
<td>Job serialization time variable:</td>
<td>3.0E-04 (ms per Byte)</td>
</tr>
<tr>
<td>Result deserialization time constant:</td>
<td>6.59574E+00 (ms)</td>
</tr>
<tr>
<td>Result deserialization time variable:</td>
<td>2.13737E-04 (ms per Byte)</td>
</tr>
</tbody>
</table>

#### Server Parameters

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>General Request deserialization time constant:</td>
<td>1.14322E+01 (ms)</td>
</tr>
<tr>
<td>General Request deserialization time variable:</td>
<td>2.37293E-04 (ms per Byte)</td>
</tr>
<tr>
<td>General Response serialization time constant:</td>
<td>3.19231E+00 (ms)</td>
</tr>
<tr>
<td>General Response serialization time variable:</td>
<td>1.63642E-04 (ms per Byte)</td>
</tr>
<tr>
<td>Submit Job execution CPU time constant:</td>
<td>1.24796E-02 (ms)</td>
</tr>
<tr>
<td>Submit Job execution CPU time variable:</td>
<td>6.14928E-07 (ms per Byte)</td>
</tr>
<tr>
<td>Submit Job overall execution time constant:</td>
<td>9.66993E+00 (ms)</td>
</tr>
<tr>
<td>Submit Job overall execution time variable:</td>
<td>2.01569E-04 (ms per Byte)</td>
</tr>
<tr>
<td>Submit Job request deserialization time constant:</td>
<td>9.83135E+00 (ms)</td>
</tr>
<tr>
<td>Submit Job request deserialization time constant:</td>
<td>2.29849E-04 (ms per Byte)</td>
</tr>
<tr>
<td>Fetch Job execution CPU time constant:</td>
<td>1.42713E-02 (ms)</td>
</tr>
<tr>
<td>Fetch Job execution CPU time variable:</td>
<td>4.80647E-07 (ms per Byte)</td>
</tr>
<tr>
<td>Fetch Job overall execution time constant:</td>
<td>9.83466E+00 (ms)</td>
</tr>
<tr>
<td>Fetch Job overall execution time variable:</td>
<td>7.98776E-05 (ms per Byte)</td>
</tr>
</tbody>
</table>
Fetch Job response serialization time constant: $7.42552 \times 10^{-1}$ (ms)
Fetch Job response serialization time variable: $1.87036 \times 10^{-4}$ (ms per Byte)
Submit Result execution CPU time constant: $1.71481 \times 10^{-2}$ (ms)
Submit Result execution CPU time variable: $6.33322 \times 10^{-7}$ (ms per Byte)
Submit Result overall execution time constant: $9.58187 \times 10^{0}$ (ms)
Submit Result overall execution time variable: $2.81291 \times 10^{-4}$ (ms per Byte)
Submit Result request deserialization time constant: $8.56806 \times 10^{0}$ (ms)
Submit Result request deserialization time variable: $2.49575 \times 10^{-4}$ (ms per Byte)
Fetch Result execution CPU time constant: $1.65194 \times 10^{-2}$ (ms)
Fetch Result execution CPU time variable: $4.69515 \times 10^{-7}$ (ms per Byte)
Fetch Result overall execution time constant: $9.8687 \times 10^{0}$ (ms)
Fetch Result overall execution time variable: $8.2965 \times 10^{-5}$ (ms per Byte)
Fetch Result response serialization time constant: $7.12717 \times 10^{0}$ (ms)
Fetch Result response serialization time variable: $1.40248 \times 10^{-4}$ (ms per Byte)

Volunteer Parameters
Volunteer Job deserialization time constant: $7.31175 \times 10^{0}$ (ms)
Volunteer Job deserialization time variable: $1.93871 \times 10^{-4}$ (ms per Byte)
Volunteer Result serialization time constant: $0.4 \times 10^{0}$ (ms)
Volunteer Result serialization time variable: $3.0 \times 10^{-4}$ (ms)

Environment Parameters
Network Latency: 1 (ms)
Network Throughput: 8000 (Bytes per ms)
F Parabon Example Code

The following code is from the Parabon Tutorial [77]. Note that some frameworks, including Parabon, use a similar naming convention where a Job is an application submitted for execution. A Job is decomposed into a series of executable units, called a task. It is the tasks that are executed on volunteer nodes.

Create a session manager.

manager = new RemoteSessionManager(url, TWO_MINUTES_IN_MILLIS);

Name the Job.

// Name for the job
String jobName = "ExampleApp";

Assign attributes to the Job by creating and populating a parameter map.

NamedParameterMap jobAttributeMap = new NamedParameterMap();
jobAttributeMap.put("JobName", jobName);

Associate the code with the task. In Parabon, this is known as an “Executable Element”.

File exampleTaskFile = new File(taskFileName);

ExecutableElementID eeID = null;
try {
    // Add the element to the job
    eeID = exampleAppJob.addJarFileExecutableElement(egTaskFile);
} catch(java.io.FileNotFoundException e) {
    System.err.println("Error: Simple task jar file (" + egTaskFile + ") not found");
    System.exit(1);
}

Create a task. First create a task spec, which is a description of the task.

TaskSpec exampleTaskSpec = new TaskSpec();

Associate the executable element, and the class in the executable element with the task.

// Identify the code for the task
exampleTaskSpec.addRequiredExecutableElement(eeID);

// Identify the name of the class to be created on
// the provider machine; ExampleTask.run()
// will be called to run the task
exampleTaskSpec.setRunnableClass(ExampleTask.class);

Add attributes to the task. These are used by the programmer to identify the task.

// Create a new parameter map to hold the task attributes
NamedParameterMap taskAttributes = new NamedParameterMap();
taskAttributes.put("TaskName", "ExampleTask");
taskAttributes.put("TaskID", 1);
Add parameters to the task.

```java
exampleTaskSpec.putParam("input", (int) 5);
exampleTaskSpec.putParam("square", (int) 0);
```

Add the task to the Job. The task spec and task attributes have to be associated with the task.

```java
// Add the task to the job
System.out.println("Adding task to ExampleApp job");
TaskProxy proxy = exampleAppJob.addTask(exampleTaskSpec,
                                       taskAttributes);
```

A client must “listen” for the results of the job. This is done by creating a listener on the client. The listener implements various interfaces, depending on the job events that it wishes to handle.

```java
class ExampleAppJobListener implements
    TaskProgressListener,
    TaskResultListener,
    TaskExceptionListener {
    SessionManager manager;
    Job job;

    // Constructor
    public ExampleAppJobListener(SessionManager manager_, Job job_) {      
        manager = manager_;
        job = job_;
    }

    // Listener for task progress events
    public void progressReported(TaskProgressEvent event) {
        ...
    }

    // Method to handle TaskResultEvents
    public void resultsPosted(TaskResultEvent event) {
        ...
    }
}
```

Instantiate the Listener and associate it with the Job.

```java
ExampleAppJobListener jobListener = new ExampleAppJobListener(manager, exampleAppJob);
// Add the listener to the job
exampleAppJob.addListener(jobListener);
```
**Parabon Tasks**

Each task implements the Task interface, which mandates a `run` method.

```java
public class ExampleTask implements Task {
    private TaskContext context;                   // task context
    private boolean runtimeWantsMeToStop = false;

    public ExampleTask(TaskContext context) {
        this.context = context;
        runtimeWantsMeToStop = false;
    }

    // Task interface method to start the task
    public void run() throws TaskStoppedException {
        DoExampleTask();
        catch(TaskStoppedException e) {
            throw(e);
        }
    }

    // Task interface method to stop the task
    public void stop() {
        runtimeWantsMeToStop = true;
    }

    ...}
```

**Required Mutators to set parameters called by the Runtime.**

```java
private int input;         // task parameters
private int square;
public void setInput(int val) { input = val; }
public void setSquare(int val) { square = val; }
```

**The actual work.**

```java
private void DoExampleTask() throws TaskStoppedException {
    double progress = 0.0;
    NamedParameterMap map = new NamedParameterMap();

    if(runtimeWantsMeToStop) {
        throw new TaskStoppedException();
    }

    // Compute the square
    square = input * input;

    // Return the final status (with the results)
    map.put("square", square);
    context.postResults(1.0, map);   // 1.0 denotes 100% completion
}
Bibliography


[34] United Devices, http://www.ud.com


[50] Condor Version 6.7.7 Manual Section 2.9


