AN IMPLEMENTATION AND ANALYSIS
OF A
RANDOMIZED DISTRIBUTED STACK

A Senior Honors Thesis

by

DUSTIN CHARLES KIRKLAND

Submitted to the Office of Honors Programs
& Academic Scholarships
Texas A&M University
In partial fulfillment of the requirements of the

UNIVERSITY UNDERGRADUATE
RESEARCH FELLOWS

April 2001

Group: Computer Science

AN IMPLEMENTATION AND ANALYSIS OF A RANDOMIZED DISTRIBUTED STACK

A Senior Honors Thesis

By

DUSTIN CHARLES KIRKLAND

Submitted to the Office of Honors Programs
& Academic Scholarships
Texas A&M University
In partial fulfillment for the designation of

UNIVERSITY UNDERGRADUATE
RESEARCH FELLOW

Approved as to style and content by:

Jennifer L. Welch
Jennifer Welch
(Fellows Advisor)

Edward A. Funkhouser
Edward A. Funkhouser
(Executive Director)

April 2001

Group: Computer Science

ABSTRACT

An Implementation and Analysis

of a

Randomized Distributed Stack. (April 2001)

Dustin Charles Kirkland
Department of Computer Science
Texas A&M University

Fellows Advisor: Dr. Jennifer Welch
Department of Computer Science

This thesis presents an algorithm for a randomized distributed stack, a coded simulator for examining its behavior, and an analysis of data collected from simulations configured to investigate its performance in particular situations. This randomized distributed stack represents an experimental extension of the probabilistic quorum algorithm of Malki et al. [5,4] and the random regular register of Welch and Lee [3]. Employing the probabilistic quorum algorithm in the same manner as the random regular register, the randomized distributed stack stands to positively affect the load and availability of a system. Popping this randomized distributed stack, however, sometimes returns incorrect values. Analysis of the data assembled reveals two interesting conclusions: 1) as the number of uninterrupted pops increases, the variance of the pop success percentage increases, and 2) for a fixed quorum size percentage, a larger system of data servers yields a higher pop success percentage. Further research remains to fully characterize and generalize the behavior of the randomized distributed stack.

Dedicated to my loving parents,

Allen and Donna Kirkland

ACKNOWLEDGEMENTS

In addition to Dr. Welch's counsel, I would like to acknowledge her graduate student, Hyunyoung Lee. Her suggestions and ideas are much appreciated.

TABLE OF CONTENTS

LIST OF FIGURES

I. INTRODUCTION

A *data structure* is a method of organizing information and its storage allocation in a computer [2]. Conventional examples include lists, stacks, queues, trees, graphs, and hash tables. Perhaps the simplest data structure is a single, simple variable called a *register* that can be read and written. In a typical, sequential, deterministic environment, these data structures behave very reliably and predictably.

Distributed computing describes the concept of multiple components working together in parallel, rather than sequentially, to solve a problem [1]. These systems accomplish parallelism through shared data structures (distributed data structures) on several different levels. Algorithms that optimize the advantages of parallel processing using shared data structures can be complex and expensive. Improving techniques, simplifying the algorithms, and refining approaches associated with distributed data structures is important.

Randomization occurs when an algorithm performs different actions depending on the values of randomly selected numbers, sometimes producing completely different output [3]. Randomization can simplify algorithms, improve efficiency over non-randomized counterparts, and sometimes allow the existence of a solution when there is no deterministic solution. The tradeoff is that randomization cannot guarantee absolute correctness in every situation and sometimes, unpredictable behavior is not tolerable. It is important to realize when randomization is a viable and desirable solution.

N.B. This thesis uses the *IEEE Transactions on Computers* journal as a style guide.

A *randomized distributed data structure* is a concept in computer science that combines the three terms described above. One application of randomized distributed data structures is implementing shared variables in a distributed system, shown in Figure 1. In this sort of system, applications invoke and receive responses from *distributed shared memory* (DSM), which consists of high-level modules that hide the implementations of the randomized distributed data structures within. The intent is for applications to use the distributed shared memory as they would use conventional data structures. The distributed shared memory, then, employs special algorithms and communicates with the data servers over the network by sending and receiving messages to access the data.

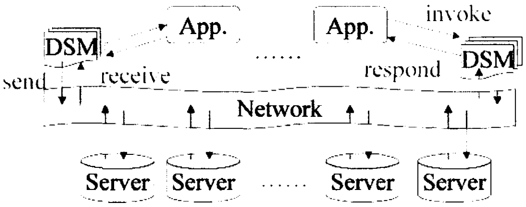


Figure 1: Distributed Shared Memory

There are several ways of implementing the distributed shared memory. In one model, there is one master server that maintains the data. The problem with this design is that the entire system is dependent on the *availability* of the master server. A second model improves this availability weakness of the first model using redundancy. Here, each server maintains an updated replica of the data. In some systems, this can overwhelm the *load* of the busiest replica server with update messages constantly being passed.

There are some applications that need reliable availability but also want to keep the load at a suitable minimum. This can be accomplished with each server keeping a local copy of the data, but not publishing updates to every server on the network. The basis for my research lies in a refinement of this idea by Malkhi, Reiter, and Wright, researchers at AT&T Labs.

Malkhi et al. propose a Probabilistic Quorum Algorithm (PQA) [5,4] to implement shared variables that optimizes load and availability. This algorithm implements a simple read-write variable in a message-passing system. To read this variable, messages are passed to all members of a quorum (a selected subset of all available data servers). The latest value present in the quorum is chosen and returned to the reading process. Similarly, a request to write a variable is passed as messages to a quorum and the latest value is stored. These quorums are chosen randomly. As the quorum size increases, more messages are passed and the network traffic increases, but subsequent quorums more often overlap and the probability of returning an out-of-date value decreases. The quorum size should be carefully chosen to balance the network load with the accuracy requirements of the application operating on the data.

The advantageous performance as analyzed in [5,4] seemed to Welch and Lee as an interesting and potentially valuable distributed tool. Here, Welch and Lee defined more complete semantics for the random regular register, suggested techniques for programming effectively with them, and identified classes of applications that can tolerate the more tenuous operation of a non-deterministic variable while profiting from the improved load and availability in the system [3].

Many distributed algorithms use more complicated data structures than simple read-write registers. Yelick et al. have studied parallel realizations of these data types, helpful in symbolic algebra, phylogeny trees, and eigenvalue computations [8,7]. Perhaps some of these algorithms need sustained availability yet desire a controlled load on the system. These cases may well benefit from randomized distributed implementations of these advanced data structures.

Randomization can be a powerful tool in simplifying coded algorithms and in providing rather simple solutions to some problems that are difficult in a deterministic environment. When the resources are available, distributed computing can solve complex problems better than sequential or concentrated computing. Distributed systems share data when working together. This can be a difficult problem when trying to provide good load and availability while keeping the system simple. The research herein described attempts to extend randomized distributed data structures beyond the random regular register so as to provide solutions to this problem.

II. ALGORITHM

In reality, programmers often use more complicated data structures than simple read-write registers to more conveniently accomplish certain tasks. One such data structure is the stack. A stack is a growable/shrinkable data structure wherein write operations "push" new values to the top of the stack and read operations "pop" the most recently pushed value from the top of the stack. The stack uses what is also known as a "last in, first out" (LIFO) discipline [2].

In the research I performed, I sought to define, characterize, and analyze the performance of a stack data structure operating in a randomized distributed environment according to the Probabilistic Quorum Algorithm. Below are the specifications for a base case algorithm. In this simplest scenario, there are multiple data servers, each server storing a replica, or local copy, of the stack. Each server also associates an assigned timestamp to its replica. There exists a single application that pushes and pops. To push or pop the randomized distributed stack:

1. Choose a quorum of a given size at random
2. Examine the timestamp of each stack selected in the quorum
3. Select the stack with the greatest timestamp
4. Perform the desired operation (push or pop) on the selected stack
5. Update the selected stack's timestamp
6. Publish the modified stack and the new timestamp to every server in the quorum

A more realistic system utilizing a randomized distributed stack may necessitate multiple-popping applications and/or multiple-pushing applications. The multiple-popper and multiple-pusher situations presented a more complicated problems than I was able to consider without first examining the most basic case. For these more complicated cases, additional rules are needed to manage overlapping pops and pushes and to coordinate the synchronicity of the timestamps. Therefore it is my intuition that the base case behaves most favorably and most simply as compared to the others. Analysis thereof provides an upper bound best-case performance of the randomized distributed stack.

III. ANALYSIS

Parameters

There are four basic parameters configurable for the test simulations. These are described below.

Number of Servers: This variable represents the integral number of data servers available to the system. Each server stores a replica of the stack and a timestamp associated with that stack.

Quorum Size: This variable represents the integral quorum size the system will select each time a push or pop is requested.

Number of Operations: This variable represents the integral total pushes and pops the simulation will perform before terminating.

Pattern: This is a selectable variable that determines the pattern at which pushes and pops are interleaved. This simulation can test the following regularly interleaved pop patterns: push/pop, push/push/pop, push/push/push/pop. It can also test a random pattern, where pushes and pops are equally likely to occur. Finally, the simulator can test a pattern that continuously pushes 1000 items onto the stack, and then continuously pops 1000 off of the stack.

Simulation

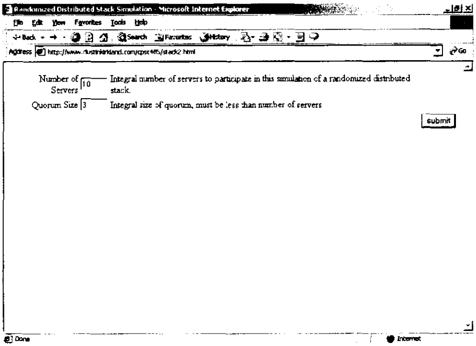
I designed and coded a software simulation that implements the randomized distributed stack according to the above specifications. Note that the objective of this simulation is not to benchmark the performance of the randomized distributed stack in a particular existing application. Rather, the objective of this simulation is to collect data and better describe the behavior of the randomized distributed stack by extrapolating from the tested situations.

I used Perl, CGI (Common Gateway Interface), and HTML (Hyper Text Markup Language) in coding three versions of the simulator. Perl is a freely available, interpreted programming language. The Perl interpreter takes Perl code and generates relatively fast and efficient C code. The true power of Perl is the ease and speed at which a knowledgeable developer is able to solve sophisticated problems. CGI is an indispensable tool for web developers designing interactive, input-driven browser applications [6]. I used Perl, CGI, and HTML to rapidly develop a powerful application with which I could interact through the convenience of a web browser.

I developed three simulators that implemented the randomized distributed stack. Though there are only slight variations among the three, each is tailored for a specific examination of the performance of the stack. Descriptions of each of the simulators and screen captures of the simulations in execution follow immediately. The source code is found in the attached appendices.

Simulation 1: This simulation initiates when the user submits an HTML form specifying the user's desired number of servers and quorum size. The user is then able to very meticulously and surgically push and pop the randomized distributed stack according to any particular push-pop pattern desired by subsequently choosing a push or pop button. This simulation is not designed for collecting large sums of data due to its reliance on user input. However, it is handy for precisely examining a given situation.

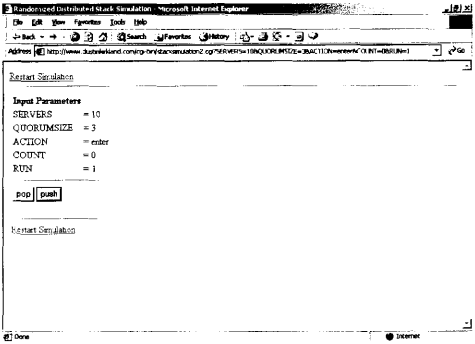
Figures 2-5 are screen shots demonstrating *Simulation 1* in execution. Figure 2 shows the HTML form in which the user configures the parameters (the number of data servers and the quorum size). Figure 3 reflects the input parameters and allows the user to select a push or a pop operation. Figure 4 demonstrates the results of a push operation, including the selected quorum, their timestamps, the selected server, and the value to push. Figure 5 shows the results of a pop operation, including the selected quorum, their timestamps, the selected server, the value that was popped, the ideal value that should have been popped, and the status of the latter two matching. See Appendix A for the documented source code.


Randomized Distributed Stack Simulation - Microsoft Internet Explorer
File Edit View Favorites Tools Help
Back Search Favorites History
Address http://www.dustinkirkland.com/cpsc485/stack2.html
Go
submit
Done
Internet

Number of Servers 10 Integral number of servers to participate in this simulation of a randomized distributed stack.

Quorum Size 3 Integral size of quorum, must be less than number of servers

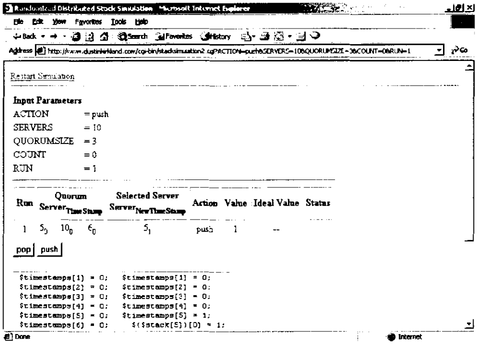
Figure 2: *Simulation 1* Sample Parameter Entry


Randomized Distributed Stack Simulation - Microsoft Internet Explorer
File Edit View Favorites Tools Help
Back Search Favorites History
Address http://www.dustinkirkland.com/cgi-bin/stacksimulation2.cgi?SERVERS=10&QUORUMSIZE=3&ACTION=enter&COUNT=0&RUN=1
Go
Restart Simulation
pop
push
Restart Simulation
Done
Internet

Input Parameters

SERVERS	= 10
QUORUMSIZE	= 3
ACTION	= enter
COUNT	= 0
RUN	= 1

Figure 3: *Simulation 1* Sample Initialized Empty Stack


Randomized Distributed Stack Simulation - Microsoft Internet Explorer
File Edit View Favorites Tools Help
Back Search Favorites History
Address
Go
Restart Simulation
pop
push
Done
Internet

http://www.dustinkirkland.com/cgi-bin/stacksimulation2.cgi?ACTION=push&SERVERS=10&QUORUMSIZE=3&COUNT=0&RUN=1

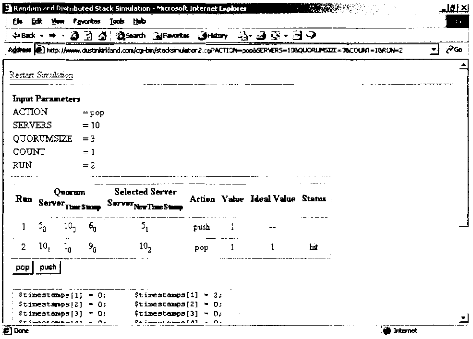
Input Parameters

ACTION	= push
SERVERS	= 10
QUORUMSIZE	= 3
COUNT	= 0
RUN	= 1

Run	Quorum Server$_{\text{TimeStamp}}$			Selected Server Server$_{\text{NewTimeStamp}}$	Action	Value	Ideal Value	Status
1	5_0	10_0	6_0	5_1	push	1	--	

```
$timestamps[1] = 0;   $timestamps[1] = 0;
$timestamps[2] = 0;   $timestamps[2] = 0;
$timestamps[3] = 0;   $timestamps[3] = 0;
$timestamps[4] = 0;   $timestamps[4] = 0;
$timestamps[5] = 0;   $timestamps[5] = 1;
$timestamps[6] = 0;     ${$stack[5]}[0] = 1;
```

Figure 4: *Simulation 1* Sample Push Performed


Randomized Distributed Stack Simulation - Microsoft Internet Explorer
File Edit View Favorites Tools Help
Back Search Favorites History
Address
Go
Restart Simulation
pop
push
Done
Internet

http://www.dustinkirkland.com/cgi-bin/stacksimulation2.cgi?ACTION=pop&SERVERS=10&QUORUMSIZE=3&COUNT=1&RUN=2

Input Parameters

ACTION	= pop
SERVERS	= 10
QUORUMSIZE	= 3
COUNT	= 1
RUN	= 2

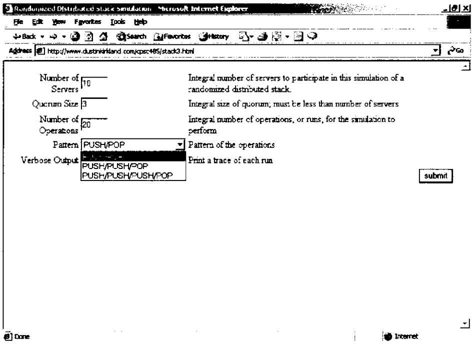
Run	Quorum Server$_{\text{TimeStamp}}$			Selected Server Server$_{\text{NewTimeStamp}}$	Action	Value	Ideal Value	Status
1	5_0	10_0	6_0	5_1	push	1	--	
2	10_1	[illegible]	9_0	10_2	pop	1	1	hit

```
$timestamps[1] = 0;        $timestamps[1] = 2;
$timestamps[2] = 0;        $timestamps[2] = 0;
$timestamps[3] = 0;        $timestamps[3] = 0;
```

Figure 5: *Simulation 1* Sample Pop Performed

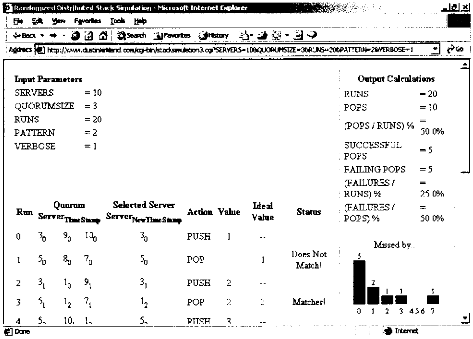
Simulation 2: This is an extension of the first simulation described. The previous simulation performs one operation per form submission. This simulation can perform any number of operations at submission according to one of the input parameters. Here, one uses a form to submit the number of servers, the quorum size, the number of operations, and the pattern to execute. This simulation can perform hundreds, thousands, or millions of push-pop operations according to the selected push-pop pattern. The form returns a dynamically generated HTML page that describes each individual operation. For each operation, the selected quorum and each server's timestamp is display, the selected server from the quorum and the updated timestamp, the desired operation (push or pop), to value pushed or popped, as well as the ideal value as would be expected from a deterministic stack. In addition, the simulation also displays calculations and statistics according to the current execution, including the pop success percentage. Finally, a histogram is displayed showing the number timestamps off of the optimal value for each failing pop. This simulation was designed for examining the performance of the randomized distributed stack over the course of thousands or millions of operations.

Figures 6 and 7 demonstrate the execution of *Simulation 2*. Figure 6 shows the HTML form in which the user configures the parameters (the number of data servers, the quorum size, the number of operations, and the pattern). Figure 7 illustrates the part of the entire results of the simulation, including a trace of each operation, output calculations, and histogram showing the distance each popped value was from its ideal value. See Appendix B for the documented source code.


Randomized Distributed Stack Simulation - Microsoft Internet Explorer
File Edit View Favorites Tools Help
Back Search Favorites History
Address http://www.dustinkirkland.com/cpsc485/stack3.html
Go
submit
Done
Internet

Number of Servers	10	Integral number of servers to participate in this simulation of a randomized distributed stack.
Quorum Size	3	Integral size of quorum; must be less than number of servers
Number of Operations	20	Integral number of operations, or runs, for the simulation to perform
Pattern	PUSH/POP	Pattern of the operations
Verbose Output	PUSH/PUSH/POP PUSH/PUSH/PUSH/POP	Print a trace of each run

Figure 6: *Simulation 2* Sample Parameter Entry


Randomized Distributed Stack Simulation - Microsoft Internet Explorer
File Edit View Favorites Tools Help
Back Search Favorites History
Address http://www.dustinkirkland.com/cgi-bin/stacksimulation3.cgi?SERVERS=10&QUORUMSIZE=3&RUNS=20&PATTERN=2&VERBOSE=1
Go
Missed by..
5
2
1
1
1
0 1 2 3 4 5 6 7
Done
Internet

Input Parameters

```
SERVERS       = 10
QUORUMSIZE    = 3
RUNS          = 20
PATTERN       = 2
VERBOSE       = 1
```

Output Calculations

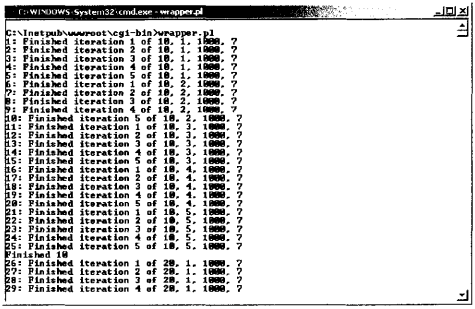
```
RUNS                = 20
POPS                = 10
(POPS / RUNS) %     = 50 0%
SUCCESSFUL POPS     = 5
FAILING POPS        = 5
(FAILURES / RUNS) % = 25 0%
(FAILURES / POPS) % = 50 0%
```

Run	Quorum $\text{Server}_{\text{TimeStamp}}$			Selected Server $\text{Server}_{\text{NewTimeStamp}}$	Action	Value	Ideal Value	Status
0	3_0	9_0	10_0	3_0	PUSH	1	--	
1	5_0	8_0	7_0	5_0	POP		1	Does Not Match!
2	3_1	1_0	9_1	3_1	PUSH	2	--	
3	5_1	1_2	7_1	1_2	POP	2	2	Matches!
4	[illegible]	[illegible]	[illegible]	[illegible]	PUSH	3	--	

Figure 7: *Simulation 2* Sample Results

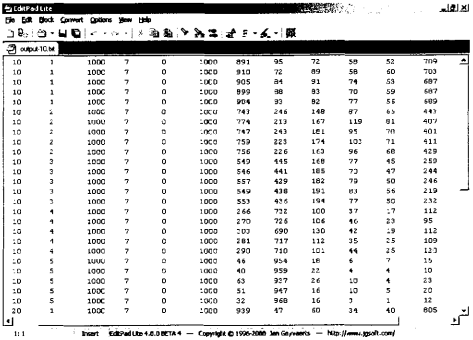
Simulation 3: This simulation is a slight modification of *Simulation 2* in conjunction with a wrapper script. In addition to using a web browser to send and receive input and output, Perl programs can receive input parameters through a command line interface and output data to files. Rather than manually entering every combination of parameters and manually collecting the results of my test runs, I slightly modified the simulator code. In this version, the simulator takes each of its input variables as parameters on the command line program call. A very brief wrapper script contains a series of nested loops that calls the simulator program with each combination of the varying parameters. After each execution, it appends a tab-delimited list of output calculations to the specified output file. This file is easily and immediately loaded into a spreadsheet for analysis and graphing.

Figures 8 and 9 demonstrate the execution of *Simulation 3*. Figure 8 shows part of the command line execution of the wrapper script that repeatedly feeds the simulator different combinations of parameters. Figure 9 illustrates part of the entire results of the simulation as was written to a text file in tab-delimited format. See Appendix C for the documented source code.



```
C:\Inetpub\wwwroot\cgi-bin>wrapper.pl
1: Finished iteration 1 of 10, 1, 1000, 7
2: Finished iteration 2 of 10, 1, 1000, 7
3: Finished iteration 3 of 10, 1, 1000, 7
4: Finished iteration 4 of 10, 1, 1000, 7
5: Finished iteration 5 of 10, 1, 1000, 7
6: Finished iteration 1 of 10, 2, 1000, 7
7: Finished iteration 2 of 10, 2, 1000, 7
8: Finished iteration 3 of 10, 2, 1000, 7
9: Finished iteration 4 of 10, 2, 1000, 7
10: Finished iteration 5 of 10, 2, 1000, 7
11: Finished iteration 1 of 10, 3, 1000, 7
12: Finished iteration 2 of 10, 3, 1000, 7
13: Finished iteration 3 of 10, 3, 1000, 7
14: Finished iteration 4 of 10, 3, 1000, 7
15: Finished iteration 5 of 10, 3, 1000, 7
16: Finished iteration 1 of 10, 4, 1000, 7
17: Finished iteration 2 of 10, 4, 1000, 7
18: Finished iteration 3 of 10, 4, 1000, 7
19: Finished iteration 4 of 10, 4, 1000, 7
20: Finished iteration 5 of 10, 4, 1000, 7
21: Finished iteration 1 of 10, 5, 1000, 7
22: Finished iteration 2 of 10, 5, 1000, 7
23: Finished iteration 3 of 10, 5, 1000, 7
24: Finished iteration 4 of 10, 5, 1000, 7
25: Finished iteration 5 of 10, 5, 1000, 7
Finished 10
26: Finished iteration 1 of 20, 1, 1000, 7
27: Finished iteration 2 of 20, 1, 1000, 7
28: Finished iteration 3 of 20, 1, 1000, 7
29: Finished iteration 4 of 20, 1, 1000, 7
```

Figure 8: *Simulation 3* Sample Automated Command Line Execution



```
10   1   1000   7   0   1000   891   95    72    58    52   709
10   1   1000   7   0   1000   910   72    89    58    60   703
10   1   1000   7   0   1000   905   84    91    74    53   687
10   1   1000   7   0   1000   899   88    83    70    59   687
10   1   1000   7   0   1000   904   83    82    77    56   689
10   2   1000   7   0   1000   743   246   148   87    65   443
10   2   1000   7   0   1000   774   213   167   119   81   407
10   2   1000   7   0   1000   747   243   181   95    70   401
10   2   1000   7   0   1000   759   223   174   103   71   411
10   2   1000   7   0   1000   756   226   163   96    68   429
10   3   1000   7   0   1000   549   445   168   77    45   259
10   3   1000   7   0   1000   546   441   185   73    47   244
10   3   1000   7   0   1000   557   429   182   79    50   246
10   3   1000   7   0   1000   549   438   191   83    56   219
10   3   1000   7   0   1000   553   436   194   77    50   232
10   4   1000   7   0   1000   266   732   100   37    17   112
10   4   1000   7   0   1000   270   726   106   46    23   95
10   4   1000   7   0   1000   300   690   130   42    19   112
10   4   1000   7   0   1000   281   717   112   35    25   109
10   4   1000   7   0   1000   290   710   101   44    25   120
10   5   1000   7   0   1000   46    954   18    6     7    15
10   5   1000   7   0   1000   40    959   22    4     4    10
10   5   1000   7   0   1000   63    937   26    10    4    23
10   5   1000   7   0   1000   51    947   16    10    5    20
10   5   1000   7   0   1000   32    968   16    3     1    12
20   1   1000   7   0   1000   939   47    60    34    40   805
```

1: 1
Insert
EditPad Lite 4.0.0 BETA 4 — Copyright © 1996-2000 Jan Goyvaerts — http://www.jgsoft.com/

Figure 9: *Simulation 3* Sample Output Text File

Results

The following graphs are most representative of the results of the simulation executions.

All of the graphs have identical x- and y-axes. The y-axes represent the percentage of pops performed by the system where the randomized distributed stack returned the same value as a deterministic stack would have. The x-axes represent varying quorum sizes as a percentage of the total number of servers in the system. Using these percentages allows for simulations of different server sizes to be displayed on the same graph. Each simulation tested systems with 10, 20, 50, and 100 data servers. For each level of data servers, every integral quorum size from 1 to half of the total number of servers was tested*. For each of these quorum sizes, 1000 pops were performed, 5 separate times. Finally, 3 different push/pop patterns were applied to each of these systems.

* If the quorum size is over half of the total number of servers, each subsequent operation is guaranteed to overlap quorums. This is not consistent with the probabilistic quorum model and the load of the system suffers with the increased quorum size.

Push/Pop Pattern: This pattern consists of alternating pushes and pops. As mentioned above, five identical runs were recorded for each set of parameters. Note the low variance of the five identical runs for each quorum size with the data points tightly clustered (Figure 10). The performance of the randomized distributed stack is therefore rather predictable for this pattern. Also, notice that as total number of data servers available to the system increases, the quorum size percentage needed to achieve pop success at 100% decreases. The results of push/push/pop and push/push/push/pop patterns were virtually identical to this graph.

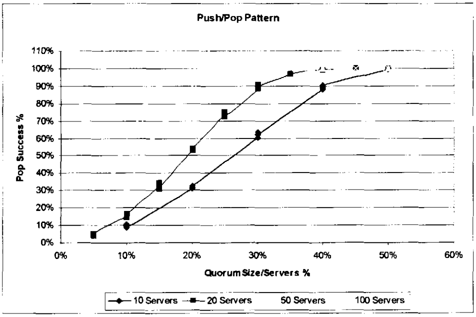


Figure 10: Results of the *Push/Pop Pattern* Simulation

Random Pattern: This pattern randomly chooses to push or pop with equal probability each time an operation is to be performed. Later, I generated a single, random pattern of pushes and pops and used this pattern for all of the test runs. Both forms of random patterns returned practically identical results. The variance of the repeated executions is still relatively low and the results predictable. The curves closely match those of the push/pop pattern shifted slightly right, indicating slightly worse performance (Figure 11).

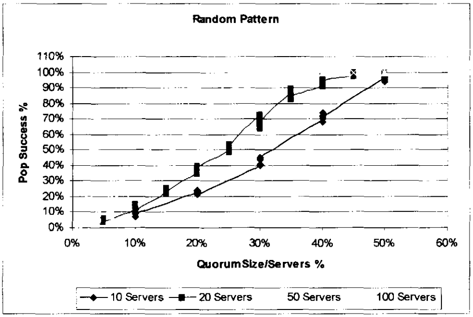


Figure 11: Results of the *Random Pattern* Simulation

1000-Push/1000-Pop Pattern: This pattern represents a common use of a stack, where many values are continuously pushed onto a stack and later continuously popped from the stack. Programmers expect the values to come off of the stack in precisely the opposite order in which they went onto the stack. This pattern suggests a weakness of this randomized distributed stack. The variance of the five identical runs for each quorum size is far higher than in the push/pop and random patterns (Figure 12). However, the averages of the five identical runs nearly follow the same functions in the previous two cases (Figure 13).

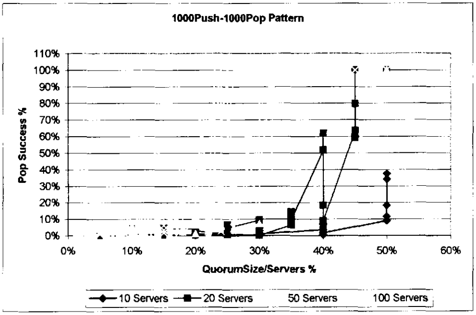


Figure 12: Results of the *1000-Push/1000-Pop Pattern* Simulation

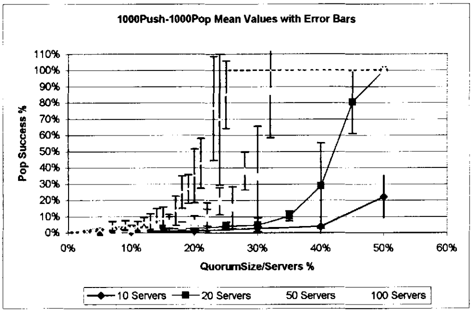


Figure 13: Averages of the Results of the *1000-Push/1000-Pop Pattern* Simulation with Standard Deviation Error Bars

IV. FUTURE RESEARCH

Measurements

While this research concentrated on the absolute accuracy of the randomized distributed pops matching the ideal, deterministic stack, there are other measures by which one might gauge the usefulness of the randomized distributed stack. Perhaps future research on the topic should include tests on the values that are popped more than once or never popped at all. The metric used in this research is particularly harsh toward patterns involving repeated pops. If at some point several quorums are chosen poorly and a single value is lost on all replicas, the system quorums can perfectly overlap for the rest of the simulation, yet every value popped will be off by one and considered a failure. Other forms of measurement may bring to light different strengths and weaknesses of randomized distributed stacks.

Applications

For what kinds of distributed applications are these randomized distributed stacks advantageous to use? How beneficial can it be? These are the most important questions this research leaves unanswered. Answering these questions will be key in motivating further investments of time and resources in additional research.

Other Algorithms

The randomized distributed stack algorithm used in this research is but one of several that I considered. These are several tangents from this algorithm that may merit further study.

- Future research should certainly consider algorithms that allow for multiple-pushing and multiple-popping applications on the network.
- In current simulations, the quorum size remains constant for the length of the simulation. Perhaps additional research could include dynamic quorums that tradeoff accuracy with network load during runtime.
- In this algorithm, both push and pop quorums are the same size. Particularly in multiple-pushing or multiple-popping scenarios, there may be accuracy/performance advantages to having different sizes for push and pop quorums.
- The current algorithm moves entire stacks around during pushes and pops. This could easily overwhelm a network with traffic when dealing with large stacks, while usually only the top few values are of importance. A more efficient algorithm may exist where a finite number of the topmost values on the stack are passed across the network rather than the entire contents of the stack.

V. CONCLUSION

This research presents a single implementation of a randomized distributed stack in the style the probabilistic quorum algorithm of Malkhi et al. [5,4] and the random regular register of Welch and Lee [3]. In general, the stack is a far more complicated data structure than a simple register. These complications are multiplied in the randomized and distributed settings. More research remains to fully characterize and generalize the behavior of the randomized distributed stack.

This research does demonstrate some particular strengths and weaknesses in the performance of the randomized distributed stack for particular cases. Research within the cases examined yields the following two conclusions.

1. As the number of uninterrupted pops increases, the variance of the pop success percentage increases. The success of a single push operation is independent of the success of any previous operation. When a pop immediately follows a push, the pop operation's success depends only on whether its quorum overlaps with the previous push operation's quorum. But when a pop follows one or more pops, this operation's chances for success are reduced since they depend on the success of each of the previous pops.
2. For a fixed quorum percentage, a larger system of data servers yields a higher pop success percentage. The shapes of the curves yielded by the test simulations probably relate to an equation by Malkhi et al. [5] that

defines the probability of two quorums intersecting (Figure 13). They prove that the probability of two randomly chosen quorums intersecting is at least $1-e^{-\ell^2}$, where the quorum size is $\ell\sqrt{totalservers}$ (Figure 14). The exact mathematical relationship between this formula and the pop success rate remains to be revealed.

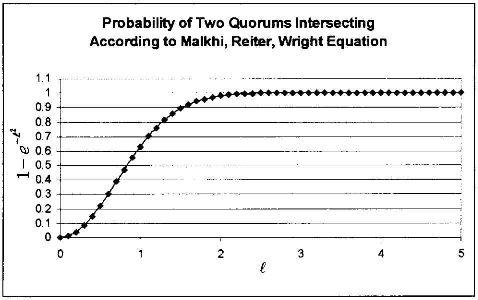


Figure 14: Graph of $1-e^{-\ell^2}$ versus ℓ

REFERENCES

[1] Burghart, T., "Distributed Computing Overview," July 1998, Available from http://www.quoininc.com/quoininc/dist_comp.html#Distributed Computing Overview

[2] Cormen, T., C. Leiserson and R. Rivest, Introduction to Algorithms, Cambridge, MA: The MIT Press, pp. 197-296, 1998.

[3] Lee, H. & J. L. Welch, "Specification, Implementation and Application of Randomized Regular Registers," *Proc. 21st International Conference on Distributed Computing Systems*, 2001.

[4] Malkhi, D. and M. Reiter, "Byzantine Quorum Systems," *Proc. 29th ACM Symposium on Theory of Computing*, pp. 569-578, May 1997.

[5] Malkhi, D. M Reiter and R. Wright, "Probabilistic Quorum Systems," *Proc. 16th ACM Symp. Principles of Distributed Computing*, pp. 267-273, 1997.

[6] Seiver, E., S. Spainhour and N. Patwardhan, Perl in a Nutshell, Sebastopol, CA: O'Reilley & Associates, Inc., pp. 3-6 & 321, 1999.

[7] Yelick, K. *et al.*, "Parallel Data Structures for Symbolic Computation," Workshop on Parallel Symbolic Languages and Systems, Oct. 1995. Also available from http://www.cs.berkeley.edu/projects/parallel/castle/multipol/papers.html

[8] Yelick, K. *et al.*, "Data Structures for Irregular Applications," DIMACS Workshop on Parallel Algorithms for Unstructured and Dynamic Problems, June 1993. Also available from http://www.cs.berkeley.edu/projects/parallel/castle/multipol/papers.html

```
<html>
        <head>
                <META HTTP-EQUIV="Expires" CONTENT="0">
                <META HTTP-EQUIV="Pragma" CONTENT="no-cache">
                <META HTTP-EQUIV="Cache-Control" CONTENT="no-cache">
                <title>Randomized Distributed Stack Simulation</title>
        </head>

        <body>
                <form method="get" action="../cgi-bin/stacksimulation2.cgi">
                        <table>
                                <tr>
                                        <td align="right">Number of Servers</td>
                                        <td><input type="text" size=4 name="SERVERS"></td>
                                        <td>Integral number of servers to participate in
this simulation of a randomized distributed stack.</td>
                                </tr>
                                <tr>
                                        <td align="right">Quorum Size</td>
                                        <td><input type="text" size=4
name="QUORUMSIZE"></td>
                                        <td>Integral size of quorum; must be less than
number of servers</td>
                                </tr>
                                <tr>
                                        <td colspan=3 align="center"> </td>
                                        <td>
                                                <input type="hidden" name="ACTION"
value="enter">
                                                <input type="hidden" name="COUNT" value=0>
                                                <input type="hidden" name="RUN" value=1>
                                                <input type="submit" value="submit">
                                        </td>
                                </tr>
                        </table>
                </form>
        </body>
</html>
```

```
#!/usr/local/bin/perl


# Global variables passed through the URL:
#       $SERVERS        - Number of servers in simulation
#       $QUORUMSIZE     - Size of quorum to select
#       $RUNS           - Number of runs to perform
#       $DISTRIBUTION   - Distribution of pops/pushes


sub printstack (@) {
#>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>
# Function      : printstack (STACK)
# Arguments     : STACK - Array of stack to be printed
# Return        : SUM   - Checksum of all elements in the stack
#                       This function will print the contents of a stack stored
#                       in an array to an HTML table
# Error Codes   : None
#<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<

# Locally scoped variables
        my (@s, $i, $sum);
# Input argument
        @s = @_;
        print ("<table><tr colspan=2><th>Dump of
Stack</th></tr><tr><th>Index</th><th>Value</th></tr>\n");
        for ($i=0; $i<scalar(@s); $i++) {
                print ("<tr align=center><td>$i</td><td>$s[$i]</td></tr>\n");
                $sum = $sum + $s[$i]
        }
        print ("<tr><td>&nbsp;</td><td>&nbsp;</td><td>$sum</td></tr></table>");
        return ($sum);
}



sub selectquorum () {
#>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>
# Function      : selectquorum ()
# Arguments     : None
#                       This function acts on the globally scoped variables
#                       $QUORUMSIZE and $SERVERS
# Return        : QUORUM - Array of indices to randomly selected group of
#                       servers
# Error Codes   : None
#<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<

# Locally scoped variables
        my (@quorum, @chosen, $server);
        $i = 0;
        while ($i<$QUORUMSIZE) {
                $server = int(rand($SERVERS)) + 1;
                if (!$chosen[$server]) {
                        push (@quorum, $server);
                        $chosen[$server] = 1;
                        $i++;
                }
        }
        return (@quorum);
}


sub selectserver (\@\@) {
#>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>
# Function      : selectserver (QUORUM, TIMESTAMPS)
# Arguments     : QUORUM - Array of indices to the chosen quorum of servers
#                TIMESTAMPS - Array of timestamps for each server in the set
# Return       : SELECTEDSERVER - Index of the most currently timestamped
#                       server
# Error Codes  : None
#<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<

# Locally scoped variables
        my (@quorum, @timestamps, $server, $newesttimestamp, $selectedserver);
        @quorum = @{$_[0]};
        @timestamps = @{$_[1]};
        $newesttimestamp = -99999;
        foreach $server (@quorum) {
                if ($timestamps[$server] > $newesttimestamp) {
                        $newesttimestamp = $timestamps[$server];
                        $selectedserver = $server;
                }
        }
        return ($selectedserver);
}



sub updatequorum (\@\@$) {
#>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>
# Function     : updatequorum (QUORUM, NEWSTACK, NEWTIMESTAMP)
# Arguments    : QUORUM - Array of indices to the chosen quorum of servers
#                NEWSTACK - Array of the newly pushed/popped stack
#                NEWTIMESTAMP - Timestamp for all of the newly updated servers
# Return       : 0 on completion
# Error Codes  : None
#<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<

# Locally scoped variables
        my (@quorum, @newstack, $server, $newtimestamp);
        @quorum = @{$_[0]};
        @newstack = @{$_[1]};
        $newtimestamp = $_[2];
        foreach $server (@quorum) {
                @{$stack[$server]} = @newstack;
                $timestamps[$server] = $newtimestamp;
        }
        return 0;
}



sub loaddata ($) {
#>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>
# Function     : loaddata (FILENAME)
# Arguments    : FILENAME - String name of file to load
# Return       : 0 on successful file load
#                       This function reads a file and evaluates the contents.
#                       In the context of this simulation, this function will
#                       load the values of the all variables used in the last
#                       run of the simulation.
# Error Codes  : 1 if file does not exist
#<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<

# Locally scoped variables
        if (-e "$filename") {
                my ($filename, @lines, $line);
                $filename = $_[0];
                open (FH, $filename);
                @lines = <FH>;
                close (FH);
                foreach $line (@lines) {
# Evaluate each line as a legal Perl expression
```

```
                        eval ($line);
                }
                return 0;
        }
        else {
                return 1;
        }
}


sub storedata ($) {
#>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>
# Function      : storedata (FILENAME)
# Arguments     : FILENAME - String name of file to which to write
# Return        : 0 on completion
#                       This function will dump the values of each stack, the
#                       ideal stack, and the timestamps to a file.  The file
#                       will consist of legal Perl assignments of variables
#                       so that the entire contents of the file can be loaded
#                       as legal Perl expressions.
# Error Codes   : None
#<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<

# Locally scoped variables
        my ($filename, $i, );
        $filename = $_[0];
        open (FH, ">$filename");
        for ($i=1; $i<=$SERVERS; $i++) {
                print FH ("\$timestamps[$i] = $timestamps[$i];\n");
                for ($j=0; $j<scalar(@{$stack[$i]}); $j++) {
                        print FH ("  \${\$stack[$i]}[$j] = ${$stack[$i]}[$j];\n");
                }
        }
        print ("\n\n");
        for ($i=0; $i<scalar(@thestack); $i++) {
                print FH ("\$thestack[$i] = $thestack[$i];\n");
        }
        close(FH);
}



####################################################################################
# Main Body of Simulator
####################################################################################

# HTML Headers, etc.
         print "Content-type:text/html\n\n";
         print "<html><head><title>Randomized Distributed Stack
Simulation</title></head>\n";
         print "<body>\n";


# Parse the input parameters out of the query string and evaluate them into
# the globally scoped variables, and then print

        print ("<a href=/cpsc485/stack2.html>Restart Simulation</a><hr>");
         $str = $ENV{'QUERY_STRING'};
         @params = split (/&/, $str);
         print ("<table><tr><td valign=top>\n");
         print ("<table><tr><th>Input Parameters</th></tr>");
         foreach $param (@params) {
                @temp = split (/=/, $param);
                eval ("\$$temp[0] = '$temp[1]'");
                print ("<tr><td>$temp[0] </td><td>= $temp[1]</td></tr>");
         }
         print ("</table><hr>");
```

```
# Data dump file name
	$filename = "temp.data";
# HTML dump file name
	$htmlfilename = "temp.html";

# Store previous contents of file
	open (FH, $filename);
	@before = <FH>;
	close (FH);

	if ($ACTION eq "enter") {
# If the simulator is now being entered, delete all old data store in the files
		if (-e $filename) {
			unlink ($filename);
		}
		if (-e $htmlfilename) {
			unlink ($htmlfilename);
		}
# Initialize the timestamps to zero
		open (FH, ">$filename");
		for ($i=1; $i<=$SERVERS; $i++) {
			print FH ("\$timestamps[$i] = 0;\n");
		}
		close (FH);
	}
	else {
# Otherwise, act as if a push or pop will happen

# Load the existing variables from file
		loaddata ("$filename");
# Choose a quorum
		@quorum = selectquorum();
# Print the table header
		 print ("<table cellpadding=3 cellspacing=3 border=1><tr><th>Run</th><th
colspan=$QUORUMSIZE>Quorum<br>Server<sub>TimeStamp</sub></th><th>Selected
Server<br>Server<sub>NewTimeStamp</sub></th><th>Action</th><th>Value</th><th>Ideal
Value</th><th>Status</th></tr><tr>");
# Load the existing HTML
		open (HTML, "$htmlfilename");
		@html = <HTML>;
		close (HTML);
# Append to the HTML file
		open (HTML, ">>$htmlfilename");
		print HTML ("<tr><td align=center>$RUN</td>\n");
		print ("@html");
		print ("<td align=center><a name=latest></a>$RUN</td>\n");
# Print the quorum
		foreach $server (@quorum) {
			print ("<td>$server<sub>$timestamps[$server]</sub></td>");
			print HTML ("<td>$server<sub>$timestamps[$server]</sub></td>\n");
		}
# Choose the best server from the quorum
		$selectedserver = selectserver (@quorum, @timestamps);
		$newtimestamp = $timestamps[$selectedserver] + 1;

		if ($ACTION eq "push") {
			$COUNT++;
			$value = $COUNT;
			$ideal = "--";
			$status = "\&nbsp;";
# Push onto the ideal stack, and the chosen stack
				push (@thestack, $value);
				push (@[$stack[$selectedserver]}, $value);
		}
		elsif ($ACTION eq "pop") {
# Pop from the ideal stack and the chosen stack
```

```
                        $ideal = pop (@thestack);
                        $value = pop (@{$stack{$selectedserver}});
                        if ($ideal == $value) {
                                $status = "hit";
                        }
                        else {
                                $status = "miss";
                        }
                }
# Update the entire quorum with the new stack
                updatequorum (@quorum, @{$stack{$selectedserver}}, $newtimestamp);
# Store all of the data to file
                storedata ("$filename");
                $RUN++;

# Print the results to the current HTML page, and store it to the HTML file
                print ("<td
align=center>$selectedserver<sub>$timestamps{$selectedserver}</sub></td>\n");
                print ("<td align=center>$ACTION</td>\n");
                print ("<td align=center>$value</td>\n");
                print ("<td align=center>$ideal</td>\n");
                print ("<td align=center>$status</td></tr>\n");
                print HTML ("<td
align=center>$selectedserver<sub>$timestamps{$selectedserver}</sub></td>\n");
                print HTML ("<td align=center>$ACTION</td>\n");
                print HTML ("<td align=center>$value</td>\n");
                print HTML ("<td align=center>$ideal</td>\n");
                print HTML ("<td align=center>$status</td></tr>\n");
                print ("</table>\n");
                close (HTML);

        }

# Create the push and pop buttons
        print ("<table><tr><td><form method=get action=stacksimulation2.cgi><input
type=submit name=ACTION value=pop>");
        print ("<input type=hidden name=SERVERS value=$SERVERS><input type=hidden
name=QUORUMSIZE value=$QUORUMSIZE><input type=hidden name=COUNT value=$COUNT><input
type=hidden name=RUN value=$RUN></form></td>");
        print ("<td><form method=get action=stacksimulation2.cgi><input type=submit
name=ACTION value=push>");
        print ("<input type=hidden name=SERVERS value=$SERVERS><input type=hidden
name=QUORUMSIZE value=$QUORUMSIZE><input type=hidden name=COUNT value=$COUNT><input
type=hidden name=RUN value=$RUN></form></td></tr></table>\n");

        if ($ACTION ne "enter") {
                open (FH, $filename);
                @after = <FH>;
                close (FH);
                print ("<table border=1><tr><td valign=top><pre> @before</pre></td><td
valign=top><pre> @after</pre></td></tr></table>\n");
        }

        print ("<hr><a href=/cpsc485/stack2.html>Restart Simulation</a>");
        print ("</body></html>\n");
```

APPENDIX B

- HTML Source Code for Submission Form
- Documented Perl Source Code for *Simulator 2*

```
<html>
        <head>
                <META HTTP-EQUIV="Expires" CONTENT="0">
                <META HTTP-EQUIV="Pragma" CONTENT="no-cache">
                <META HTTP-EQUIV="Cache-Control" CONTENT="no-cache">
                <title>Randomized Distributed Stack Simulation</title>
        </head>

        <body>
                <form method="get" action="../cgi-bin/stacksimulation3.cgi">
                        <table>
                                <tr>
                                        <td align="right">Number of Servers</td>
                                        <td><input type="text" size=4 name="SERVERS"></td>
                                        <td>Integral number of servers to participate in
this simulation of a randomized distributed stack.</td>
                                </tr>
                                <tr>
                                        <td align="right">Quorum Size</td>
                                        <td><input type="text" size=4
name="QUORUMSIZE"></td>
                                        <td>Integral size of quorum; must be less than
number of servers</td>
                                </tr>
                                <tr>
                                        <td align="right">Number of Operations</td>
                                        <td><input type="text" size=4 name="RUNS"></td>
                                        <td>Integral number of operations, or runs, for the
simulation to perform</td>
                                </tr>
                                <tr>
                                        <td align="right">Pattern</td>
                                        <td>
                                                <select name="PATTERN">
                                                        <option selected
value=2>PUSH/POP</option>
                                                        <option
value=3>PUSH/PUSH/POP</option>
                                                        <option
value=4>PUSH/PUSH/PUSH/POP</option>
                                                </select>
                                        </td>
                                        <td>Pattern of the operations</td>
                                <tr>
                                        <td align="right">Verbose Output</td>
                                        <td>
                                                <select name="VERBOSE">
                                                        <option selected
value=1>YES</option>
                                                        <option value=0>NO</option>
                                                </select>
                                        </td>
                                        <td>Print a trace of each run</td>
                                <tr>
                                        <td colspan=3 align="center"> </td>
                                        <td><input type="submit" value="submit"></td>
                                </tr>
                                </tr>

                        </table>
                </form>
        </body>
</html>
```

```
#!/usr/local/bin/perl


# Global variables passed through the URL:
#       $SERVERS        - Number of servers in simulation
#       $QUORUMSIZE     - Size of quorum to select
#       $RUNS           - Number of runs to perform
#       $PATTERN        - Pattern of pops/pushes
#       $VERBOSE        - Print a trace of each run



sub printstack (@) {
#>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>
# Function      : printstack (STACK)
# Arguments     : STACK - Array of stack to be printed
# Return        : SUM   - Checksum of all elements in the stack
#                       This function will print the contents of a stack stored
#                       in an array to an HTML table
# Error Codes   : None
#<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<

# Locally scoped variables
        my (@s, $i, $sum);
# Input argument
        @s = @_;
        print ("<table><tr colspan=2><th>Dump of
Stack</th></tr><tr><th>Index</th><th>Value</th></tr>\n");
        for ($i=0; $i<scalar(@s); $i++) {
                print ("<tr align=center><td>$i</td><td>$s[$i]</td></tr>\n");
                $sum = $sum + $s[$i]
        }
        print ("<tr><td>&nbsp;</td><td>&nbsp;</td><td>$sum</td></tr></table>");
        return ($sum);
}



sub selectquorum () {
#>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>
# Function      : selectquorum ()
# Arguments     : None
#                       This function acts on the globally scoped variables
#                       $QUORUMSIZE and $SERVERS
# Return        : QUORUM - Array of indices to randomly selected group of
#                       servers
# Error Codes   : None
#<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<

# Locally scoped variables
        my (@quorum, @chosen, $server);
        $i = 0;
        while ($i<$QUORUMSIZE) {
                $server = int(rand($SERVERS)) + 1;
                if (!$chosen[$server]) {
                        push (@quorum, $server);
                        $chosen[$server] = 1;
                        $i++;
                }
        }
        return (@quorum);
}



sub selectserver (\@\@) {
#>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>
```

```
# Function      : selectserver (QUORUM, TIMESTAMPS)
# Arguments     : QUORUM - Array of indices to the chosen quorum of servers
#                 TIMESTAMPS - Array of timestamps for each server in the set
# Return        : SELECTEDSERVER - Index of the most currently timestamped
#                        server
# Error Codes   : None
#<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<

# Locally scoped variables
        my (@quorum, @timestamps, $server, $newesttimestamp, $selectedserver);
        @quorum = @{$_[0]};
        @timestamps = @{$_[1]};
        $newesttimestamp = -99999;
        foreach $server (@quorum) {
                if ($timestamps[$server] > $newesttimestamp) {
                        $newesttimestamp = $timestamps[$server];
                        $selectedserver = $server;
                }
        }
        return ($selectedserver);
}


sub updatequorum (\@\@$) {
#>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>
# Function      : updatequorum (QUORUM, NEWSTACK, NEWTIMESTAMP)
# Arguments     : QUORUM - Array of indices to the chosen quorum of servers
#                 NEWSTACK - Array of the newly pushed/popped stack
#                 NEWTIMESTAMP - Timestamp for all of the newly updated servers
# Return        : 0 on completion
# Error Codes   : None
#<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<

# Locally scoped variables
        my (@quorum, @newstack, $server, $newtimestamp);
        @quorum = @{$_[0]};
        @newstack = @{$_[1]};
        $newtimestamp = $_[2];
        foreach $server (@quorum) {
                @{$stack[$server]} = @newstack;
                $timestamps[$server] = $newtimestamp;
        }
        return 0;
}


sub printcolumn ($;$) {
#>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>
# Function      : printcolumn (VALUE, COLOR)
# Arguments     : VALUE - Integer value for the height of the column
#                 COLOR - HTML legal text for color of column
# Return        : None
# Error Codes   : None
#<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<

        my ($value, $i, $color);

        $value = $_[0];
        $color = $_[1];
        if (!$color) {
                $color = "red";
        }
        print ("<table cellspacing=0 cellpadding=0 border=0>");
        print ("<tr><td align=center><small>$value</small></td></tr>");
        for ($i=0;$i<$value;$i++) {
                print ("<tr><td bgcolor=$color width=20
height=2><small>&nbsp;</small></td></tr>");
        }
        print ("</table>");
}

sub printhistogram (@) {
        my (@data, $datum, $i);

        @data = @_;
        print ("<table><tr>\n");
        print ("<td valign=bottom align=center>");
        printcolumn ($data[$i], "blue");
        print ("</td>\n");
        for ($i=1; $i<scalar(@data); $i++) {
                print ("<td valign=bottom align=center>");
                printcolumn ($data[$i]);
                print ("</td>\n");
        }
        print ("</tr><tr>");
        for ($i=0; $i<scalar(@data); $i++) {
                print ("<td align=center><small>$i</small></td>");
        }
        print ("</tr></table>");
}

sub printv ($) {
        if ($VERBOSE) {
                print ("$_[0]");
        }
}


################################################################################
# Main Body of Simulator
################################################################################

# HTML Headers, etc.
        print "Content-type:text/html\n\n";
        print "<html><head><title>Randomized Distributed Stack
Simulation</title></head>\n";
        print "<body>\n";


# Parse the input parameters out of the query string and evaluate them into
# the globally scoped variables, and then print

        $str = $ENV{'QUERY_STRING'};
        @params = split (/&/, $str);
        print ("<table border=1><tr><td valign=top>\n");
        print ("<table><tr><th>Input Parameters</th></tr>");
        foreach $param (@params) {
                @temp = split (/=/, $param);
                eval ("\$$temp[0] = $temp[1]");
                print ("<tr><td>$temp[0] </td><td>= $temp[1]</td></tr>");
        }
        print ("</table><br><br><br><hr>");


# Locally scoped variable
        my ($i);
        $start = (times)[0];
# Seed the random number generator, initialize counters
        srand;
        $count = 0;
        $failures = 0;
        $successes = 0;
        $pops = 0;
        for ($i=1;$i<=$SERVERS;$i++) {
               $timestamps[$i] = 0;
        }

# Print HTML table header
        printv ("<table cellpadding=3 cellspacing=3 border=0><tr><th>Run</th><th
colspan=$QUORUMSIZE>Quorum<br>Server<sub>TimeStamp</sub></th><th>Selected
Server<br>Server<sub>NewTimeStamp</sub></th><th>Action</th><th>Value</th><th>Ideal
Value</th><th>Status</th></tr>");

# Main loop of simulation runs
        for ($i=0;$i<$RUNS;$i++) {
# Choose an action
               $action = int(rand (10));
               if ((($i + 1) % $PATTERN) != 0) {
                      $actionverb = "PUSH";
               }
               else {
                      $actionverb = "POP";
               }
               printv ("<tr><td>$i</td>\n");
# Choose the quorum and print the timestamps of each
               @quorum = selectquorum();
                      foreach $server (@quorum) {
                             printv ("<td>$server<sub>$timestamps[$server]</sub></td>");
               }
# Choose the most recently stamped server and calculate the new timestamp, and print
               $selectedserver = selectserver (@quorum, @timestamps);
               $newtimestamp = $timestamps[$selectedserver] + 1;
               printv ("</td><td
align=center>$selectedserver<sub>$timestamps[$selectedserver]</sub></td>\n");
               if ((($i + 1) % $PATTERN) != 0) {
# Perform a push
                      $count++;
                      $value = $count;
                      $idealvalue = "--";
# Push onto the ideal stack
                      push (@thestack, $value);
# Push onto the chosen stack
                      push (@{$stack[$selectedserver]}, $value);
# Publish to everyone on the quorum and update the timestamps
                      updatequorum (@quorum, @{$stack[$selectedserver]}, $newtimestamp);
                      $status = "\&nbsp;";
               }
               else {
# Perform a pop
                      $pops++;
# Pop the ideal stack
                      $idealvalue = pop (@thestack);
# Pop the chosen stack
                      $value = pop (@{$stack[$selectedserver]});
# Publish to everyone on the quorum and update the timestamps
                      updatequorum (@quorum, @{$stack[$selectedserver]}, $newtimestamp);
               }
               printv ("<td>$actionverb</td>\n");
               if ($value == $idealvalue) {
                      $status = "Matches!";
                      $successes++;
                      $missedby[$idealvalue - $value]++;
                      printv ("<td align=center><font color=red>$value</font></td><td
align=center><font color=red>$idealvalue</font></td><td
align=center>$status</td></tr>\n");
               }
               else {
                      if ($idealvalue == "--") {
                             $status = "\&nbsp;";
```

```
                        }
                        else {
                               $status = "Does Not Match!";
                               $failures++;
                               $missedby[$idealvalue - $value]++;
                        }
                        printv ("<td align=center>$value</td><td
align=center>$idealvalue</td><td align=center>$status</td></tr>\n");
                }
        }

# Calculate and print statistics of the simulation
        print ("</table></td><td valign=top align=right><table><tr><th>Output
Calculations</th></tr>");
        $failurepercent = 100 * $failures / $RUNS;
        $poppercent = 100 * $pops / $RUNS;
        $popfailurepercent = 100 * ($failures / $pops);
        print ("<tr><td><table><tr><td>RUNS</td><td>= $RUNS</td></tr>\n");
        print ("<tr><td>POPS</td><td>= $pops</td></tr>\n");
        printf ("<tr><td>(POPS / RUNS) % </td><td>= %.1f\%</td></tr>\n",$poppercent);
        print ("<tr><td>SUCCESSFUL POPS</td><td>= $successes</td></tr>\n");
        print ("<tr><td>FAILING POPS</td><td>= $failures</td></tr>\n");
        printf ("<tr><td>(FAILURES / RUNS) % </td><td>= %.1f\%</td></tr>\n",
$failurepercent);
        printf ("<tr><td>(FAILURES / POPS) % </td><td>=
%.1f\%</td></tr></table></td></tr>\n", $popfailurepercent);

        print ("<tr valign=bottom align=center><td><br>Missed by...");
        printhistogram (@missedby);
        print ("</tr></td>");
        print ("</table></td></tr></table>\n");
        $end = (times)[0];
        $runtime = $end - $start;
        print ("Simulation run time on CPU: $runtime");

        print "</body></html>\n";
```

APPENDIX C

- Documented Perl Source Code for Wrapper Script
- Documented Perl Source Code for *Simulator 3*

```
# wrapper.pl

# Wrapper script which repeatedly calls the main simulator
# with each combination of parameters

# Modify this file to configure combinations of automated
# test runs

# Create array with each parameter value to use
@servers = (10, 20, 50, 100);
# @quorumratios = (.5, .4, .3, .2, .1);
@pops = (1000);
@patterns = (7);

$count = 1;

# Nested loops to generate unique and ordered combinations
foreach $s (@servers) {
        @quorumratios = ();
        for ($i=1;$i<=$s/2;$i++) {
                push (@quorumratios, $i/$s);
        }
        foreach $q (@quorumratios) {
                foreach $ps (@pops) {
                        foreach $p (@patterns) {
                                $qs = $q * $s;
#                               $r = $ps * $p;
                                $r = $ps;
                                $iter = $count % 5;
# This loop performs the five identical test runs for
# calculating the variance of the system
                                while ($iter <= 5) {
# The following line calls the simulation program according
# to the current combination of parameters and chooses the
# output text file.  Modify this line to output to a
# different file.
                                         `stacksimulation3.pl $s $qs $r $p output-10.txt`;
                                        print ("$count: Finished iteration ", $iter, " of
$s, $qs, $r, $p\n");
                                        
                                        $iter++;
                                        $count++;
                                }
                        }
                }
        }
        print ("Finished $s\n");
}
```

```
#!/usr/local/bin/perl


# Global variables passed through the URL:
#       $SERVERS        - Number of servers in simulation
#       $QUORUMSIZE     - Size of quorum to select
#       $RUNS           - Number of runs to perform
#       $PATTERN        - Pattern of pops/pushes
#       $VERBOSE        - Print a trace of each run



sub printstack (@) {
#>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>
# Function      : printstack (STACK)
# Arguments     : STACK - Array of stack to be printed
# Return        : SUM   - Checksum of all elements in the stack
#                       This function will print the contents of a stack stored
#                       in an array to an HTML table
# Error Codes   : None
#<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<

# Locally scoped variables
        my (@s, $i, $sum);
# Input argument
        @s = @_;
        printhtml ("<table><tr colspan=2><th>Dump of
Stack</th></tr><tr><th>Index</th><th>Value</th></tr>\n");
        for ($i=0; $i<scalar(@s); $i++) {
                printhtml ("<tr align=center><td>$i</td><td>$s[$i]</td></tr>\n");
                $sum = $sum + $s[$i]
        }
        printhtml ("<tr><td>&nbsp;</td><td>&nbsp;</td><td>$sum</td></tr></table>");
        return ($sum);
}



sub selectquorum () {
#>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>
# Function      : selectquorum ()
# Arguments     : None
#                       This function acts on the globally scoped variables
#                       $QUORUMSIZE and $SERVERS
# Return        : QUORUM - Array of indices to randomly selected group of
#                       servers
# Error Codes   : None
#<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<

# Locally scoped variables
        my (@quorum, @chosen, $server);
        $i = 0;
        while ($i<$QUORUMSIZE) {
                $server = int(rand($SERVERS)) + 1;
                if (!$chosen[$server]) {
                        push (@quorum, $server);
                        $chosen[$server] = 1;
                        $i++;
                }
        }
        return (@quorum);
}



sub selectserver (\@\@) {
#>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>
```

```
# Function      : selectserver (QUORUM, TIMESTAMPS)
# Arguments     : QUORUM - Array of indices to the chosen quorum of servers
#                 TIMESTAMPS - Array of timestamps for each server in the set
# Return        : SELECTEDSERVER - Index of the most currently timestamped
#                       server
# Error Codes   : None
#<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<

# Locally scoped variables
        my (@quorum, @timestamps, $server, $newesttimestamp, $selectedserver);
        @quorum = @{$_[0]};
        @timestamps = @{$_[1]};
        $newesttimestamp = -99999;
        foreach $server (@quorum) {
                if ($timestamps[$server] > $newesttimestamp) {
                        $newesttimestamp = $timestamps[$server];
                        $selectedserver = $server;
                }
        }
        return ($selectedserver);
}



sub updatequorum (\@\@$) {
#>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>
# Function      : updatequorum (QUORUM, NEWSTACK, NEWTIMESTAMP)
# Arguments     : QUORUM - Array of indices to the chosen quorum of servers
#                 NEWSTACK - Array of the newly pushed/popped stack
#                 NEWTIMESTAMP - Timestamp for all of the newly updated servers
# Return        : 0 on completion; This function publishes the new stack to
#                     each server in the quorum and updates their timestamps
#
# Error Codes   : None
#<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<

# Locally scoped variables
        my (@quorum, @newstack, $server, $newtimestamp);
        @quorum = @{$_[0]};
        @newstack = @{$_[1]};
        $newtimestamp = $_[2];
        foreach $server (@quorum) {
                @{$stack[$server]} = @newstack;
                $timestamps[$server] = $newtimestamp;
        }
        return 0;
}


sub printcolumn ($;$) {
#>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>
# Function      : printcolumn (VALUE, COLOR)
# Arguments     : VALUE - Integer value for the height of the column
#                 COLOR - HTML legal text for color of column
# Return        : None; This function generates and prints the HTML table for
#                     the column of a histogram
# Error Codes   : None
#<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<

        my ($value, $i, $color);

        $value = $_[0];
        $color = $_[1];
        if (!$color) {
                $color = "red";
        }
        printhtml ("<table cellspacing=0 cellpadding=0 border=0>");
        printhtml ("<tr><td align=center><small>$value</small></td></tr>");
        for ($i=0;$i<$value;$i++) {
                printhtml ("<tr><td bgcolor=$color width=20
height=2><small>&nbsp;</small></td></tr>");
        }
        printhtml ("</table>");
}

sub printhistogram (@) {
#>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>
# Function      : printhistogram (DATA)
# Arguments     : DATA - Array of the data to display in the histogram format
# Return        : None; This function generates and prints the HTML for a
#                    statistical histogram by repeatedly calling the
#                    printcolumn() function
# Error Codes   : None
#<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<

        my (@data, $datum, $i);

        @data = @_;
        printhtml ("<table><tr>\n");
        printhtml ("<td valign=bottom align=center>");
# Print the first column individually
        printcolumn ($data[$i], "blue");
        printhtml ("</td>\n");
# Print each subsequent column individually
        for ($i=1; $i<scalar(@data); $i++) {
                printhtml ("<td valign=bottom align=center>");
                printcolumn ($data[$i]);
                printhtml ("</td>\n");
        }
        printhtml ("</tr><tr>");
        for ($i=0; $i<scalar(@data); $i++) {
                printhtml ("<td align=center><small>$i</small></td>");
        }
        printhtml ("</tr></table>");
}

sub printv ($) {
#>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>
# Function      : printv (STRING)
# Arguments     : STRING - String to be printed
#                   VERBOSE - This function also acts on the globally defined
#                     value of VERBOSE
# Return        : None; This function prints the STRING argument if the VERBOSE
#                     argument is asserted
# Error Codes   : None
#<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<

        if ($VERBOSE) {
                printhtml ("$_[0]");
        }
}

sub printhtml ($) {
#>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>
# Function      : printv (STRING)
# Arguments     : STRING - String to be printed
#                   HTML - This function also acts on the globally defined
#                     value of HTML
# Return        : None; This function prints the STRING argument if the HTML
#                     argument is asserted
# Error Codes   : None
#<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<

        if ($HTML) {
                print ("$_[0]");
        }
}

sub sum (@) {
#>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>
# Function      : sum (DATA)
# Arguments     : DATA - Array of numbers
# Return        : This function returns the sum of an array of numbers
# Error Codes   : None
#<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<

        my (@array, $sum, $i);
        @array = @_;
        foreach $i (@array) {
                $sum += $i;
        }
        return ($sum);
}

################################################################################
# Main Body of Simulator
################################################################################

# HTML Headers, etc.
if ($HTML) {
        printhtml "Content-type:text/html\n\n";
        printhtml "<html><head><title>Randomized Distributed Stack
Simulation</title></head>\n";
        printhtml "<body>\n";


# Parse the input parameters out of the query string and evaluate them into
# the globally scoped variables, and then print

        $str = $ENV{'QUERY_STRING'};
        @params = split (/&/, $str);
        printhtml ("<table border=1><tr><td valign=top>\n");
        printhtml ("<table><tr><th>Input Parameters</th></tr>");
        foreach $param (@params) {
                @temp = split (/=/, $param);
                eval ("\$$temp[0] = $temp[1]");
                printhtml ("<tr><td>$temp[0] </td><td>= $temp[1]</td></tr>");
        }
        printhtml ("</table><br><br><br><hr>");
}
else {
        $SERVERS = shift (@ARGV);
        $QUORUMSIZE = shift (@ARGV);
        $RUNS = shift (@ARGV);
        $PATTERN = shift (@ARGV);
        $OUTFILE = shift (@ARGV);
        $VERBOSE = 0;
        $programstart = (times)[0];
        open (FH, ">>$OUTFILE");
        require ("pattern.txt");

}

# Locally scoped variable
        my ($i);
        $start = (times)[0];
# Seed the random number generator, initialize counters
        srand;
        $count = 0;
        $failures = 0;
        $successes = 0;
        $pops = 0;
        $do = 0;
        for ($i=1;$i<=$SERVERS;$i++) {
                $timestamps[$i] = 0;
        }

# Print HTML table header
        printv ("<table cellpadding=3 cellspacing=3 border=0><tr><th>Run</th><th
colspan=$QUORUMSIZE>Quorum<br>Server<sub>TimeStamp</sub></th><th>Selected
Server<br>Server<sub>NewTimeStamp</sub></th><th>Action</th><th>Value</th><th>Ideal
Value</th><th>Status</th></tr>");

        $i = 0;
# Main loop of simulation runs
#        for ($i=0;$i<$RUNS;$i++) {
#         while ($pops < $RUNS) {
        while ($do < scalar (@pattern)) {

# Choose an action
                $action = int(rand (10));
#               if ((($i + 1) % $PATTERN) != 0) {
                if ($pattern[$do] == 0) {
                        $actionverb = "PUSH";
                }
                else {
                        $actionverb = "POP";
                }
                print ("$actionverb\n");
                printv ("<tr><td>$i</td>\n");
# Choose the quorum and print the timestamps of each
                @quorum = selectquorum();
                        foreach $server (@quorum) {
                                printv ("<td>$server<sub>$timestamps[$server]</sub></td>");
                }
# Choose the most recently stamped server and calculate the new timestamp, and print
                $selectedserver = selectserver (@quorum, @timestamps);
                $newtimestamp = $timestamps[$selectedserver] + 1;
                printv ("</td><td
align=center>$selectedserver<sub>$timestamps[$selectedserver]</sub></td>\n");
#               if ((($i + 1) % $PATTERN) != 0) {
                if ($pattern[$do] == 0) {
# Perform a push
                        $count++;
                        $value = $count;
                        $idealvalue = "--";
# Push onto the ideal stack
                        push (@thestack, $value);
# Push onto the chosen stack
                        push (@{$stack[$selectedserver]}, $value);
# Publish to everyone on the quorum and update the timestamps
                        updatequorum (@quorum, @{$stack[$selectedserver]}, $newtimestamp);
                        $status = "\&nbsp;";
                }
                else {
# Perform a pop
                        $pops++;
# Pop the ideal stack
                        $idealvalue = pop (@thestack);
# Pop the chosen stack
                        $value = pop (@{$stack[$selectedserver]});
                        if (!$value) {
                                $nullpops++;
                        }
# Publish to everyone on the quorum and update the timestamps
                        updatequorum (@quorum, @{$stack[$selectedserver]}, $newtimestamp);
                }
                $do++;
                printv ("<td>$actionverb</td>\n");
                if ($value == $idealvalue) {
                        $status = "Matches!";
                        $successes++;
                        $missedby[abs($idealvalue - $value)]++;
                        printv ("<td align=center><font color=red>$value</font></td><td
align=center><font color=red>$idealvalue</font></td><td
align=center>$status</td></tr>\n");
                }
                else {
                        if ($idealvalue == "--") {
                                $status = "\&nbsp;";
                        }
                        else {
                                $status = "Does Not Match!";
                                $failures++;
                                $missedby[abs($idealvalue - $value)]++;
                        }
                        printv ("<td align=center>$value</td><td
align=center>$idealvalue</td><td align=center>$status</td></tr>\n");
                }
        }

        @printmissedby = @missedby;
# Calculate and print statistics of the simulation
        printhtml ("</table></td><td valign=top align=right><table><tr><th>Output
Calculations</th></tr>");
        $failurepercent = 100 * $failures / $RUNS;
        $poppercent = 100 * $pops / $RUNS;
        $popfailurepercent = 100 * ($failures / $pops);
        printhtml ("<tr><td><table><tr><td>RUNS</td><td>= $RUNS</td></tr>\n");
        printhtml ("<tr><td>POPS</td><td>= $pops</td></tr>\n");
#        printf ("<tr><td>(POPS / RUNS) % </td><td>= %.1f\%</td></tr>\n",$poppercent);
#         printf ("<tr><td>(FAILURES / RUNS) % </td><td>= %.1f\%</td></tr>\n",
$failurepercent);
        printhtml ("<tr><td>SUCCESSFUL POPS</td><td>= $successes</td></tr>\n");
        printhtml ("<tr><td>FAILING POPS</td><td>= $failures</td></tr>\n");
        $temp = shift(@printmissedby);
        printhtml ("<tr><td>POPS off by 0</td><td>= $temp</td></tr>\n");
        $temp = shift(@printmissedby);
        printhtml ("<tr><td>POPS off by 1</td><td>= $temp</td></tr>\n");
        $temp = shift(@printmissedby);
        printhtml ("<tr><td>POPS off by 2</td><td>= $temp</td></tr>\n");
        $temp = shift(@printmissedby);
        printhtml ("<tr><td>POPS off by 3</td><td>= $temp</td></tr>\n");
        $missedbymore = sum(@printmissedby);
        printhtml ("<tr><td>POPS off by more</td><td>= $missedbymore</td></tr>\n");
        printhtml ("<tr><td>Null POPS</td><td>= $nullpops</td></tr>\n");
        printhtml ("<tr><td>(FAILURES / POPS) % </td><td>=
$popfailurepercent\%</td></tr>\n");
        $instancetime = (times)[0]-$start;
        printhtml ("<tr><td>Sim run time</td><td>= $temp</td></tr></table></td></tr>\n");

        printhtml ("<tr valign=bottom align=center><td><br>Missed by...");
        printhistogram (@missedby);
        printhtml ("</tr></td>");
         printhtml ("</table></td></tr></table>\n");

         printhtml "</body></html>\n";
        for ($i=0;$i<=3;$i++) {        if (!$missedby[$i]) { $missedby[$i] = 0; } }
        if (!$missedbymore) {$missedbymore = 0; }
        if (!$nullpops) { $nullpops = 0; }
        if (!$instancetime) {$instancetime = 0; }
        print (FH "$INSTANCE   $SERVERS        $QUORUMSIZE     $RUNS   $PATTERN
        $VERBOSE        $pops   $failures       $missedby[0]    $missedby[1]    $missedby[2]
        $missedby[3]    $missedbymore   $nullpops       $instancetime\n");
        close (FH);
```

VITA

Dustin Charles Kirkland is the oldest son of Allen and Donna Kirkland of 5213 Eudora Dr. Addis, Louisiana 70710. He achieved National Merit Commended Student status and graduated Valedictorian of Catholic High School in Baton Rouge, Louisiana in 1997. Dustin was awarded a President's Endowed Scholarship, a Carolyn Lipscomb Web Opportunity Award, and a Robert C. Byrd Scholarship. He anticipates graduation from Texas A&M University in College Station, Texas in May of 2001 Magna Cum Laude with University Honors, a Bachelor of Science in Computer Engineering, and a minor in Mathematics. Dustin has accepted a position with Tivoli Systems, Inc. of IBM in Austin, Texas where he will work as a Software Engineer in Build Automation.