A JOB SERVER FOR PARALLEL AND CONCURRENT EXECUTION OF MÖBIUS SIMULATORS

BY

RONALD JOSEPH WRIGHT

THESIS

Submitted in partial fulfillment of the requirements
for the degree of Master of Science in Electrical and Computer Engineering
in the Graduate College of the
University of Illinois at Urbana-Champaign, 2014

Urbana, Illinois

Adviser:

Professor William H. Sanders

ABSTRACT

Möbius is an extensible modeling environment used to validate the reliability, availability, security, and performance of computer systems. It provides graphical editors through which users can construct system models or compositions of several models, and it provides interfaces for finding the metrics of the modeled systems. It supports a large set of modeling formalisms through an abstract functional interface, which allows it to support new formalisms without requiring modification of existing formalisms. It offers several solvers that work with any modeling formalism; one of those solvers is a tool for executing simulations over a network. However, that tool does not resemble a service, because the Möbius graphical user interface launches and collects data from the simulators directly, and simulators cannot be executed remotely if they exist on different subnets.

This thesis presents the Remote Job Server (RJS), an extension to the Möbius modeling environment that supports the remote execution of simulators. It provides more flexibility in the networking topology, and it features a tree hierarchy in which each node in the tree represents an RJS instance that serves a specific purpose. One or more client nodes, which serve as leaves in the tree, connect to a manager node, the root of the tree. Part of the manager's role is to route commands through zero or more forward nodes to the worker nodes, which also serve as leaves in the tree. The manager also dispatches simulation jobs, aggregates simulation data from the worker nodes, and reports simulation results to the client. The workers execute the jobs assigned by the manager; that typically involves compiling of Möbius projects and running of simulators on the compiled project.

The client is implemented as a graphical user interface that is integrated with Möbius so that the user can seamlessly develop the experiment studies and run the simulations from a single interface.

To my mother, for her love and support.

ACKNOWLEDGMENTS

I thank my advisor, Professor William H. Sanders, for his advice on setting up timelines for accomplishing implementation goals for RJS and thesis-writing goals. I also thank the lead developer of Möbius and research scientist, Kenneth Keefe, for his advice on the next steps in RJS and his insights on the thesis. Additionally, I thank Brett Feddersen, a developer of Möbius and other software projects distributed by the PERFORM research group, for contributing to the implementation of RJS, helping me develop RJS more efficiently, and giving thoughtful feedback on the thesis. Finally, I thank the Department of Electrical and Computer Engineering and Jenny Applequist for their editorial comments.

The development of RJS was started by Quincy Mitchell, a former member of the PERFORM group who was an undergraduate at the time of RJS's early development. He laid out the basic foundation of RJS, which originally supported Linux systems and already had the fully-implemented network hierarchy formalism and an extensive set of commands. Among other improvements, Brett Feddersen helped RJS support Macintosh OS X and Windows, and he also helped update the programming language of RJS from C++98 to C++11. More importantly, he created a C++ test client for RJS, which made RJS easier to test, which ultimately helped in setting up timing experiments to assess the performance and scalability of RJS.

This material is based upon work supported by the Army Research Office under Award Nos. W911NF-09-1-0273 and W911NF-13-1-0086. Any opinions, findings, and conclusions or recommendations expressed in this publication are those of the author(s) and do not necessarily reflect the views of the Army Research Office. This material is based on research sponsored by the Department of Homeland Security (DHS) Science and Technology Directorate, Homeland Security Advanced Research Projects Agency (HSARPA), Cyber

Security Division (DHS S&T/HSARPA/CSD), BAA 11-02 via contract number HSHQDC-13-C-B0014. This material is based upon work supported by the Maryland Procurement Office under Contract No. H98230-14-C-0141. Any opinions, findings and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of the Maryland Procurement Office.

TABLE OF CONTENTS

LIST OF ABBREVIATIONS

CCS	Concurrent and comparative simulation
CIS	Critical Utility Infrastructural Resilience Information Switch
CPU	Central processing unit
DDR3	Double data rate type 3
DOM	Document object model
FIFO	First-in, first-out
GB	Gigabyte(s)
HT	Hierarchical troika
ID	Identifier
LTS	Long-term support
MR	Majority rule
NAT	Network address translation
PDES	Parallel discrete event simulation
QEMU	Quick EMUlator
RAM	Random access memory
RJS	Remote job server
RT	Random troika
SAN	Stochastic activity network
SHA	Secure hash algorithm
TCP	Transmission Control Protocol
XML	Extensible Markup Language

LIST OF SYMBOLS

$\mathcal{T}$	RJS tree
$\mathcal{V}$	Set of vertices in the RJS tree
$\mathcal{E}$	Set of edges in the RJS tree
V_{m}	Manager node
V_{c}	Client node
$\mathcal{V}_{\mathrm{f}}$	Set of forward nodes
V_{f}	Single forward node
$\mathcal{V}_{\mathrm{w}}$	Set of worker nodes
V_{w}	Single worker node
$\mathcal{E}_{\mathrm{mf}}$	Set of manager-forward edges
$\mathcal{E}_{\mathrm{mw}}$	Set of manager-worker edges
$\mathcal{E}_{\mathrm{ff}}$	Set of forward-forward edges
$\mathcal{E}_{\mathrm{fw}}$	Set of forward-worker edges
$\mathcal{E}_{\mathrm{mc}}$	Set of manager-client edges
$\mathcal{E}_{\mathrm{fc}}$	Set of forward-client edges
client-forward	Client forward annotation
worker-forward	Worker forward annotation
$\alpha_{\mathrm{f}} \colon \mathcal{V}_{\mathrm{f}} \mapsto \{\text{client-forward}, \text{worker-forward}\}$	Forward annotation function
$\mathcal{C}$	Set of all serializable classes
CI	Confidence interval

CL	Confidence level
$t_{n,(1-CL)/2}$	Critical value of the Student t-distribution with n degrees of freedom
$\Phi^{-1}\colon \mathbb{R} \mapsto \mathbb{R}$	Quantile function
σ^2	Sample variance
$\overline{x}$	Sample mean
$\mathrm{Var}\,(X)$	Variance of a random variable X
$\sigma^2_{\text{histogram}}$	Variance of a bin in a histogram
h_0	Mean frequency of a bin in a histogram
p_0	Mean relative frequency of a bin in a histogram
p	Relative frequency of a bin in a histogram
M_1	Fault model in which an attacker sabotages a majority vote with some probability
M_2	Fault model in which an attacker always sabotages a majority vote

CHAPTER 1

INTRODUCTION

1.1 Motivation for Efficient Simulation

Möbius [1], [2], [3], [4] is a tool that provides an environment for validating system reliability, availability, security, and performance. It provides editors for designing models that describe the configuration of a system. Each functional component in the model represents an event that is assumed to occur at deterministic or probabilistic (e.g., exponentially distributed) rates so that the model can be analyzed quantitatively. Möbius also allows the user to replicate a submodel to save on computation time and merge (or join) multiple submodels together into a single composed model to keep the design modular.

The user can specify reward variables so that he or she can study particular states of the system; the Möbius tool incorporates these reward variables into the model and, if an analytic solution is desired, converts the model into a Markov state space. However, even with replicated models, the state space tends to grow exponentially as the number of states increases. That phenomenon is known as *state-space explosion*, which is an inherent side effect of computing probabilities in a Markov state space with many combinations of states. So for large models, scalability becomes an issue.

To alleviate the issue of state-space explosion, Möbius provides a feature that allows the user to simulate his or her model rather than generate a state space for the model to be solved analytically. It even offers a network utility that parallelizes the simulation of system state by running computations corresponding to multiple independent experiments on multiple machines. Those "machines" stay resident so that the workload is shared among multiple experiments. However, the tool as it stands in Möbius 2.4 does not support large-scale clusters well, as the aggregation of data is performed by the client. Furthermore, the

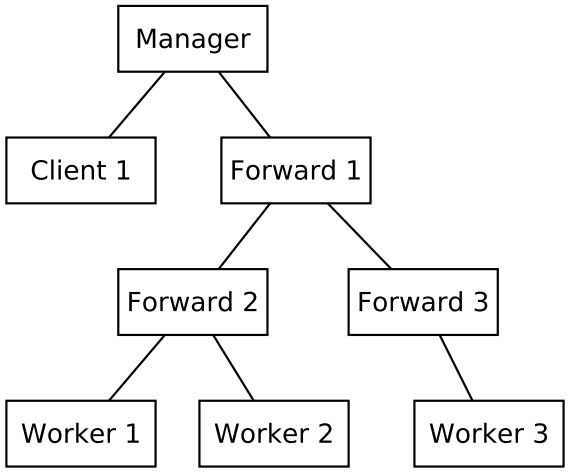


Figure 1.1: An example RJS network topology.

availability of the host that manages the distributed simulations largely depends on the tolerance of the complex graphical user interface to faults, which means that the simulation will stop unexpectedly if any other component of Möbius raises an uncaught exception. Therefore, there is a need to implement a dedicated simulation manager that separates the role of receiving simulation data from the role of managing the data.

1.2 Overview of Remote Job Server

Remote Job Server (RJS) is an extension of Möbius, written in C++, that replaces the existing network feature by providing a dedicated facility for running multiple simulation experiments simultaneously. RJS consists of four types of nodes: client, manager, forward, and worker. The client node is responsible for initiating the simulation; the manager is responsible for assigning workers to run experiments; the forward is responsible for bypassing firewall restrictions by relaying packets from one network to another network; and the worker is responsible for launching simulations and gathering data from the simulations.

Figure 1.1 shows an example of an RJS network topology. The manager is always the root vertex in all RJS topologies, and clients and workers connect to it through zero or more forward nodes. As a result, the manager serves as a centralized server in the RJS network.

1.3 Thesis Statement

The RJS framework is more than just a dedicated facility for simulation. It is our thesis that RJS provides a service that enables flexible deployment and allows the user to make sound decisions on specific network configurations.

RJS offers the following three main contributions:

- complete separation of the user interface from the core RJS functionality,
- the ability to use any mixture of operating systems in the RJS network, and
- the ability to use RJS across different networks.

1.4 Thesis Organization

The remainder of the thesis discusses the structure and behavior of RJS and the testing framework, and describes several testbed experiments that examined the performance and scalability of RJS.

Chapter 2 describes work related to the distributed simulation framework of RJS. Chapter 3 describes the structure of the RJS, and Chapter 4 describes the behavior of the RJS. Chapters 5 and 6 describe experimental setups and results, respectively, and Chapter 7 concludes the thesis.

CHAPTER 2

BACKGROUND AND RELATED WORK

2.1 Background

RJS is a tool that has the capability of making simulations run in parallel, so it is classified as a parallel discrete event simulation (PDES) tool. An overview of PDES is given in [5], which begins by discussing various related efforts that dealt with different ways of parallelizing a discrete event simulation. In particular, the paper mentions the "replicated trials" approach, in which multiple sequential simulations are run concurrently. RJS and the pre-existing network simulation feature in Möbius both use the replicated trials approach, according to which they try to run as many experiments simultaneously as possible. Next, [5] describes simulation sequencing constraints that make PDES hard in general, the events and logical processes that characterize PDES, and optimistic and conservative mechanisms of PDES. Logical processes are components in a distributed system that execute parts of a simulation task and work together to execute the entire task, and events are messages containing simulation state that are required to ensure sequential consistency. Optimistic and pessimistic approaches differ in their attempts to work around causality errors in distributed simulation, which occur when logical processes receive events relating to an earlier point in time in the simulation. Optimistic approaches handle causality errors by utilizing a recovery mechanism known as *rollback*, which restores the state of all logical processes with causality errors to an earlier one from immediately before causality was violated. Conservative approaches avoid causality errors altogether. RJS's approach is considered conservative, since each experiment is run from the beginning to the end sequentially, and the simulation processes do not communicate with each other. More specifically, for those reasons, RJS does not require the use of messages between experiments.

On the other hand, the ideas in [6] suggest that RJS is more accurately classified as a concurrent and comparative simulation (CCS) tool rather than a PDES tool, because just like the pre-existing network simulation feature, RJS is too conservative to be considered a parallel simulation framework, as it makes no effort to make the simulations themselves parallel. That is, each experiment, which represents a serial simulation, runs independently of other experiments, possibly in parallel with the other experiments. Still, each of the experiments has a chance of running on only one node, unlike most PDES approaches, according to which every experiment constantly runs on multiple nodes throughout the duration of the run. Moreover, the experimental setup in Möbius typically involves changing input values for some set of variables in each experiment, and as a result, the experiments are designed so that the user can compare the results of one experiment with the results of another experiment.

2.2 Related Work

2.2.1 *EcliPSe*

The authors of [7] and [8] describe a framework for concurrent execution of stochastic simulations known as *EcliPSe*, which is designed to run specific types of stochastic simulation problems such as Monte Carlo and discrete-event simulations. Its key objectives were to require minimal involvement on the user's end (i.e., the user only needs to write sequential simulations) and to provide transparency (i.e., the user does not need to know the structure of the underlying network). Simulations are translated by *EcliPSe* to run on concurrent architectures and are then executed repeatedly until the observed output samples aggregated by a central monitor process or a distributed monitor system converge within a confidence interval, or until an abnormal termination occurs. The user specifies the names of the routines for aggregating the observation data and testing whether the aggregated samples have converged. The framework was tested on simulations such as multidimensional integration involving the sample mean method, discrete-time Markov chains for calculating hitting times, Dijkstra's algorithm for finding all the smallest costs to get from one vertex

to another vertex in a graph, and simulation of a token ring network. The benchmarks were run on homogeneous supercomputer clusters and homogeneous and heterogeneous networks of workstations. The authors found that for the majority of the tested simulations, the run time on a single processor was on the order of hours, whereas the run time on supercomputer clusters was on the order of minutes. The speedups on workstation networks, on the other hand, were not as large as the speedups on supercomputer clusters.

Like RJS, *EcliPSe* supports the simulation of many types of models, and it offers multiple ways of aggregating the observation data. However, *EcliPSe* distributes the simulation across multiple hosts and runs the simulation in parallel, in contrast to RJS, which takes multiple experiments belonging to each simulation and runs the serial simulations simultaneously across multiple CPU cores and multiple machines. Moreover, the authors of [7] and [8] did not clearly mention whether *EcliPSe* supports communication between nodes that lie in completely different networks, which can be a problem when incoming packets between two nodes are blocked because the subnets differ between the two nodes. That is one of the reasons why forwarding nodes that propagate packets across different networks exist in RJS. Also, the *EcliPSe* framework only supports the parallel execution of single simulations, meaning that only one simulation runs at a time, and it does not provide support for varying input variables for effective comparisons between runs. Further, deterministic applications can cause *EcliPSe* to incur substantial communication overheads over stochastic simulations [7]. RJS, on the other hand, does not make any assumptions or guarantees concerning the overheads of such applications, which justifies its approach for concurrent execution of experiments. Indeed, stochastic models such as stochastic activity networks (SANs) [9] can be designed to run deterministically (e.g., with fixed activity firing rates), but they would have no direct influence on the communication overhead, since experiments run independently of each other, and the worker nodes therefore do not need to exchange simulation state information with each other.

2.2.2 Akaroa2

The authors of [10] describe a distributed simulation framework known as Akaroa2, which runs the same model on multiple hosts independently. It shares with RJS the key objective of integrating a pre-existing simulation program to run under the framework with minimal modification. The statistics are aggregated by a central *akmaster* process, which, like the manager process in RJS, is responsible for starting new simulations and determining when to stop the running simulations. An *akrun* process, like the client process in RJS, connects to the *akmaster* process and initiates a simulation, providing arguments such as the project filename to the *akmaster* process, which associates unique ID numbers with those arguments. Much like the worker processes in RJS, *akslave* processes run on separate hosts and are responsible for launching simulation engine processes for the simulation. Each simulation engine process receives the host name and port of the *akmaster* process from the *akslave* process and connects to the *akmaster* process. Once it receives the data to run the simulation, it starts running the simulation while sending observation statistics to the *akmaster* process. The *akmaster* process includes a "local data analyzer," which analyzes the simulation in two stages: the transient stage and the estimation stage. In the transient stage, the simulation runs until the local analyzer determines that the simulation has entered steady state. In the estimation stage, the local analyzer computes the values corresponding to the output variables of the simulation and the statistical precision of each value.

The network structure and objective of Akaroa2 are remarkably similar to those of RJS, but the framework has several shortcomings that RJS does not. First, the framework only supports Unix systems, so, for example, the framework cannot be run on Windows systems. Second, the framework does not utilize convergence criteria to determine whether the statistical precision is sufficiently small. Sometimes, in order to obtain accurate output data, many runs are required for the confidence interval to fall within some upper bound, which is especially the case for large simulation models. Moreover, Akaroa2 supports only steady-state simulations, unlike RJS, which can also handle transient simulations. Like *EcliPSe*, Akaroa is not known to contain forwarding nodes that allow communication between two hosts on two different networks.

CHAPTER 3

RJS STRUCTURE

3.1 Tree Formalism

The RJS model consists of a series of nodes that send commands to each other. As a result, the set of nodes and the relationships between them can be modeled as a tree. Figure 3.1 shows the tree structure of the RJS network. Note that it is a doubly chained tree, which allows a node to access its parent (if it exists) or its children (if they exist). The tree, $\mathcal{T} = (\mathcal{V}, \mathcal{E})$, is defined by the following:

1. $\mathcal{V}$ is a set of vertices.
2. $V \in \mathcal{V}$ is a vertex.
3. $\mathcal{E}$ is a set of edges.
4. $(V_p, V_c) \in \mathcal{E}$ is an edge, where $V_p \in \mathcal{V}$ is a parent vertex and $V_c \in \mathcal{V}$ is a child vertex.
5. There exists no edge $(V, V) \in \mathcal{E}$ where $V \in \mathcal{V}$.
6. There exist no two edges $(V_{p1}, V_c) \in \mathcal{E}$ and $(V_{p2}, V_c) \in \mathcal{E}$, where $\{V_{p1}, V_{p2}, V_c\} \subseteq \mathcal{V}$ and $V_{p1} \neq V_{p2}$.
7. There exists only one vertex $V \in \mathcal{V}$ such that $((\mathcal{V} \setminus \{V\}) \times \{V\}) \cap \mathcal{E} = \emptyset$.

Properties 5, 6, and 7 define the structure of the tree. Property 5 states that a node's child cannot be the node itself. Property 6 states that a node cannot have more than one parent. Property 7 states that there must exist exactly one root node (i.e., a node without any parents).

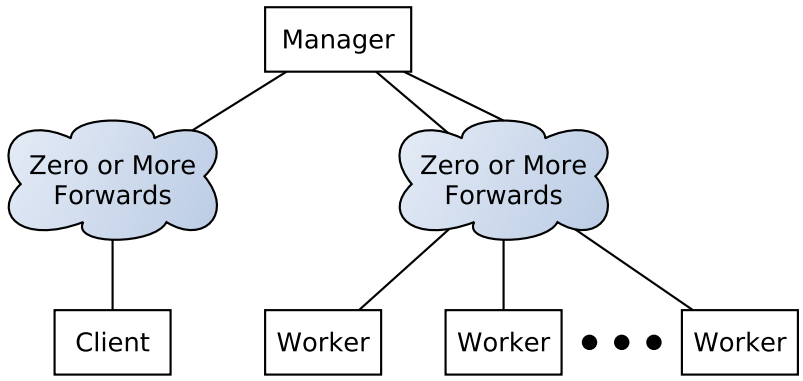


Figure 3.1: The RJS network structure.

That leads to the definition of the RJS tree. Suppose $V_{\mathrm{m}} \in \mathcal{V}$ is a vertex representing the manager node; $V_{\mathrm{c}} \in \mathcal{V}$ is a vertex representing the client node; $\mathcal{V}_{\mathrm{f}} \subseteq \mathcal{V}$ is a set of vertices representing the forward nodes; and $\mathcal{V}_{\mathrm{w}} \subseteq \mathcal{V}$ is a set of vertices representing the worker nodes. Also, let $\mathcal{E}'_{\mathrm{mf}} = \{V_{\mathrm{m}}\} \times \mathcal{V}_{\mathrm{f}}$, $\mathcal{E}'_{\mathrm{mw}} = \{V_{\mathrm{m}}\} \times \mathcal{V}_{\mathrm{w}}$, $\mathcal{E}'_{\mathrm{ff}} = \mathcal{V}_{\mathrm{f}} \times \mathcal{V}_{\mathrm{f}}$, $\mathcal{E}'_{\mathrm{fw}} = \mathcal{V}_{\mathrm{f}} \times \mathcal{V}_{\mathrm{w}}$, $\mathcal{E}'_{\mathrm{mc}} = \{(V_{\mathrm{m}}, V_{\mathrm{c}})\}$, and $\mathcal{E}'_{\mathrm{fc}} = \mathcal{V}_{\mathrm{f}} \times \{V_{\mathrm{c}}\}$. Then, all of the following properties must hold true:

1. $\{V_{\mathrm{m}}\}$, $\{V_{\mathrm{c}}\}$, $\mathcal{V}_{\mathrm{f}}$, and $\mathcal{V}_{\mathrm{w}}$ are mutually disjoint sets.

2. $\{V_{\mathrm{m}}\} \cup \{V_{\mathrm{c}}\} \cup \mathcal{V}_{\mathrm{f}} \cup \mathcal{V}_{\mathrm{w}} = \mathcal{V}$.

3. $\mathcal{E} \subseteq \mathcal{E}'_{\mathrm{mf}} \cup \mathcal{E}'_{\mathrm{mw}} \cup \mathcal{E}'_{\mathrm{ff}} \cup \mathcal{E}'_{\mathrm{fw}} \cup \mathcal{E}'_{\mathrm{mc}} \cup \mathcal{E}'_{\mathrm{fc}}$.

Properties 1 and 2 specify a partition of $\mathcal{V}$. That is, the set of vertices is partitioned into the manager and client nodes and the sets of forward and worker nodes. Property 3 reinforces the definition of edges with respect to the partitioning properties. That property also allows the partitioning of edges so that edge groups are formed:

- $\mathcal{E}_{\mathrm{mf}} = \mathcal{E} \cap \mathcal{E}'_{\mathrm{mf}}$.
- $\mathcal{E}_{\mathrm{mw}} = \mathcal{E} \cap \mathcal{E}'_{\mathrm{mw}}$.
- $\mathcal{E}_{\mathrm{ff}} = \mathcal{E} \cap \mathcal{E}'_{\mathrm{ff}}$
- $\mathcal{E}_{\mathrm{fw}} = \mathcal{E} \cap \mathcal{E}'_{\mathrm{fw}}$.

- $\mathcal{E}_{\text{mc}} = \mathcal{E} \cap \mathcal{E}'_{\text{mc}}$.
- $\mathcal{E}_{\text{fc}} = \mathcal{E} \cap \mathcal{E}'_{\text{fc}}$.

In other words, the set of edges is partitioned into the sets of manager-forward edges, manager-worker edges, forward-worker edges, manager-client edges, and forward-client edges. The following property for the partitioning of edges must hold true, as it does for the partitioning of vertices:

$$\mathcal{E}_{\text{mf}} \cup \mathcal{E}_{\text{mw}} \cup \mathcal{E}_{\text{ff}} \cup \mathcal{E}_{\text{fw}} \cup \mathcal{E}_{\text{mc}} \cup \mathcal{E}_{\text{fc}} = \mathcal{E}.$$

That property guarantees that no forbidden edges exist in the RJS tree. Specifically, it guarantees that the manager node is the root and that all client and worker nodes are leaves.

However, the definition of the RJS tree is still incomplete, because it does not control the separation of specific forward nodes, and it does not ensure that the client and worker nodes are attached to the correct forward nodes. To address that issue, let $\alpha_{\text{f}} \colon \mathcal{V}_{\text{f}} \mapsto \{\text{client-forward}, \text{worker-forward}\}$ denote a forward annotation function that determines the role a particular forward plays. Consider a forward node $V_{\text{f}} \in \mathcal{V}_{\text{f}}$. Then, if $\alpha_{\text{f}}(V_{\text{f}})$ is a client-forward, then the corresponding forward is responsible for routing information between the manager and the reachable client and is thus called a client-forward. On the other hand, if $\alpha_{\text{f}}(V_{\text{f}})$ is a worker-forward, then the corresponding forward is responsible for routing information between the manager and each reachable worker and is thus called a worker-forward.

The following properties introduce the necessary constraints to complete the definition of the RJS tree:

1. For all $(V_{\text{f}_1}, V_{\text{f}_2}) \in \mathcal{E}_{\text{ff}}$ and $\{V_{\text{f}_1}, V_{\text{f}_2}\} \subseteq \mathcal{V}_{\text{f}}$, $\alpha_{\text{f}}(V_{\text{f}_1}) = \alpha_{\text{f}}(V_{\text{f}_2})$.
2. For all $(V_{\text{f}}, V) \in \mathcal{E}_{\text{fw}} \cup \mathcal{E}_{\text{fc}}$, $V = V_{\text{c}}$ if and only if $\alpha_{\text{f}}(V_{\text{f}}) = \text{client-forward}$, and $V \in \mathcal{V}_{\text{w}}$ if and only if $\alpha_{\text{f}}(V_{\text{f}})$ is a worker-forward.

Constraint 1 separates the client-forwards from the worker-forwards. Constraint 2 prevents workers from attaching to client-forwards and clients from attaching to worker-forwards.

3.2 Thread Model

Each RJS node has five different types of threads: receiver threads, which are responsible for receiving incoming commands; sender threads, which are responsible for sending outgoing commands; the main job server thread, which is primarily responsible for running incoming commands; command threads, which are threads that were spawned by commands; and the main program thread, which simply spawns the main job server thread and waits for it to terminate. Each node that communicates with another node has a sender thread and a corresponding receiver thread. Each worker node also consists of zero or more job threads dedicated to receiving results from each simulation. That means that for a node with n communication links, there are n receiver threads, n sender threads, one job server thread, zero or more command threads, and one main program thread, which amounts to at least $2n + 2$ threads.

To ensure that no race conditions occur, the outgoing messages are sequenced in queues for sending command packets to their destinations in a first-in-first-out manner. Command scheduler queues are used to schedule incoming commands in a first-in-first-out manner. Figure 3.2 shows the graphical representation of the thread model. In each receiver thread, incoming commands are enqueued on the scheduler queue so that the main job server thread eventually dequeues the command from the scheduler queue and runs it. A command may contain another command to run locally or remotely, and/or it may spawn a new thread to run separately from the main job server thread. If the other command in the former case is to run locally, it is placed in the scheduler queue. If the command is to run remotely, it is placed in the appropriate send queue. The command is picked up by the corresponding sender thread, and the message is eventually sent out to the appropriate recipient.

3.3 Queuing Model

Figure 3.3 shows the queue model representation of Figure 3.2. In Figure 3.3, λ is the arrival rate of incoming commands; μ_{r_k} is the rate of commands that are enqueued on communication link k; and μ_{s_k} is the rate of commands that are dequeued on communication

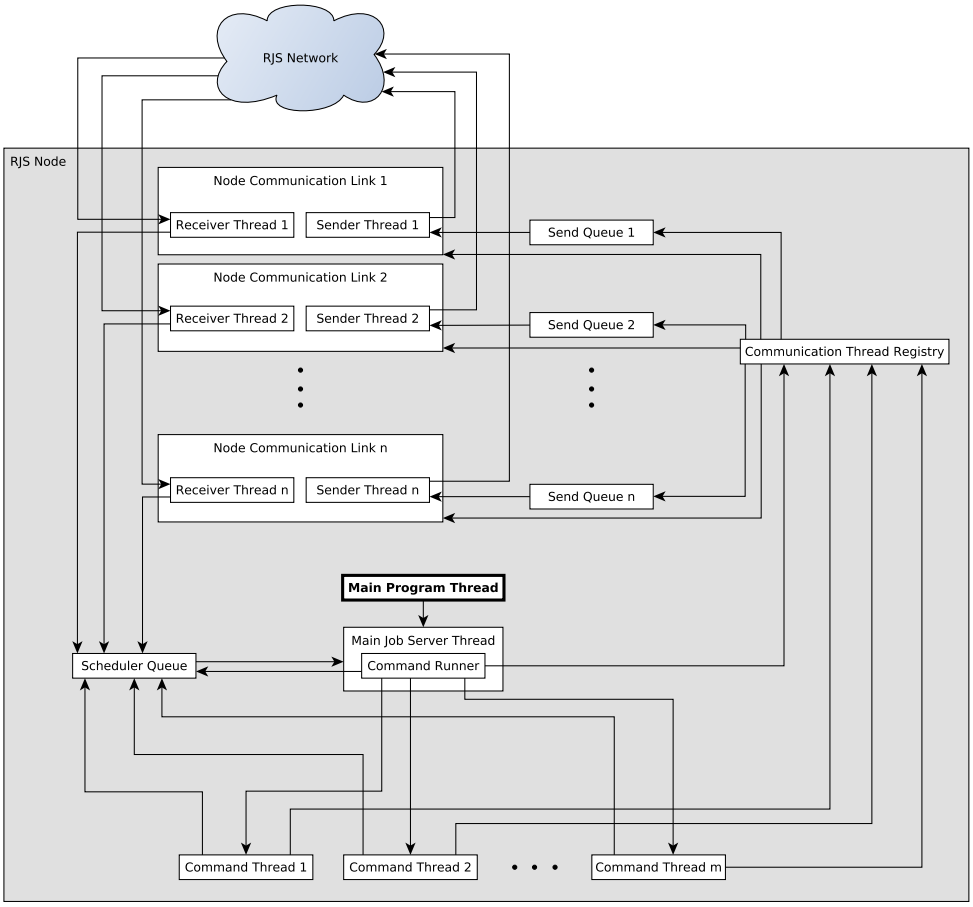


Figure 3.2: The RJS thread model.

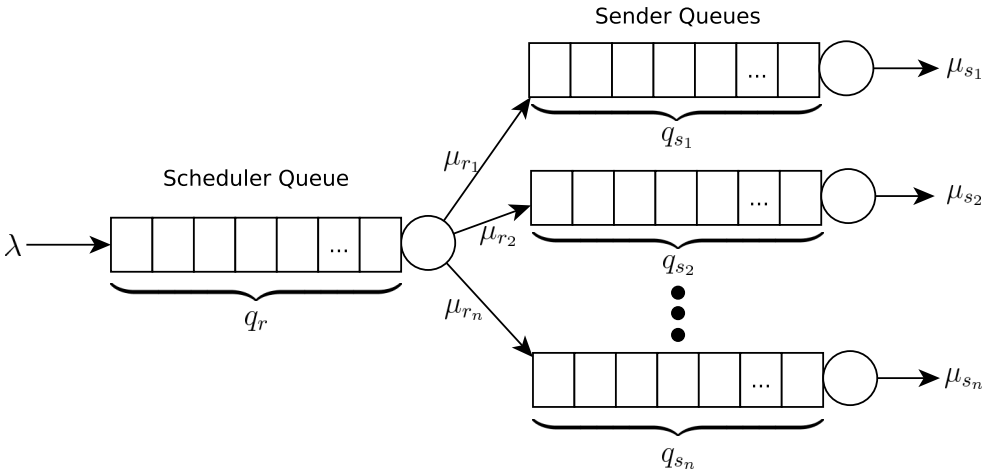


Figure 3.3: The RJS queue model.

link k. Also, q_r is the queue length of the receiver queue, and q_{s_k} is the queue length of the sender queue on communication link k. The model makes it clear that there is a single thread per communication link for receiving commands, a single thread for running commands, and one thread per communication link for sending commands. The advantage of that approach is that there are fewer race conditions. The disadvantage, though, is that the queue length of each sender queue tends to be linear in the number of communication links, which makes the main job server thread a central bottleneck. The problem is especially bad in the case of the manager node, which is a centralized server that handles commands from every client and worker node in the RJS tree. Nevertheless, this model is suitable in most cases, since most RJS networks are expected to contain a small number of client and worker nodes.

CHAPTER 4

RJS BEHAVIOR

4.1 RJS Lookups

Each RJS network must have an identifier to ensure that clients do not inadvertently connect to the wrong RJS network and try to use a non-matching RJS network topology. Each time a new RJS network is created, a 68-character identifier is generated, which always begins with `net_`. The rest of the string is made up of 64 random alphanumeric characters. The probability that two RJS networks have the same identifier, assuming a uniform distribution, is one in 62^{64}, or approximately 1.9×10^{-115}. Because RJS is expected to represent a small user base, two RJS networks are unlikely to have the same identifier.

In order to differentiate among various hosts in the RJS network, each host must be identified by some unique node identifier. Obvious choices include using the IP address and port number or the hostname. The problems with the first choice are that key comparison is too complicated and differentiation of hosts that share the same external IP address (e.g., on a network with network address translation (NAT)) is difficult. The problem with the second choice is that two hosts may have the same hostname, which results in a conflict. A solution that leverages those problems is to assign unique identifiers in the form of numbers to each host in the RJS network; the identifiers are relative to each RJS network. The manager node always has a unique identifier of one, and the unique identifier of zero is reserved for an "anonymous" host. All other hosts are assigned the next available identifier, starting from two.

```
net_[64-character network ID][4-byte base-10 zero-padded XML length string
    ]0000000000000000000000000000
<commandPacket originID="nodeID" destination="nodeID" datasize="unsigned 32-bit integer">
  <route>
    <hop nodeID="nodeID"/>
    <hop nodeID="nodeID"/>
    ...
  </route>
  <commandXML commandType="commandID">
    serialized command data
  </command>
</commandPacket>
```

Figure 4.1: The XML packet format.

4.2 RJS Protocol

To ensure the reliability of data delivered from node to node and first-in, first-out (FIFO) semantics of message delivery, RJS relies on a reliable transfer protocol known as the *Transmission Control Protocol*, or *TCP*. TCP offers several advantages: for example, messages are delivered in order, and any packets that are lost are automatically re-sent. The in-order delivery is especially important, not only because of the message delivery semantics, but also because of the queues, which further enforce FIFO ordering of sending and receiving of messages. The disadvantage of TCP is that the reliability mechanisms introduce overhead. However, because the majority of the time is spent running simulations, the advantages outweigh the disadvantage.

4.2.1 Packet Format

The base unit of message exchange in an RJS network is known as a *packet*. Sharing of data is trivial in single programs that rely on shared memory, as long as it implements proper mutual exclusion. However, sharing of data across programs in a distributed system is less trivial. Communication of data between two nodes requires conversion of command state into a packet representation so that the receiving end extracts the data from the packet to match the command state at the sender. Therefore, data are copied from node to node in RJS.

Packet data are represented in a format known as *Extensible Markup Language*, or *XML*.

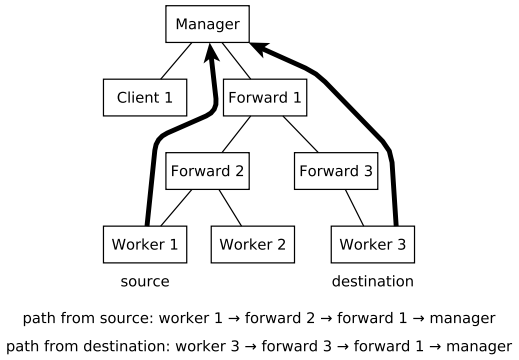
Figure 4.1 shows the format of a packet. Every packet consists of a 68-character network identifier header followed by a four-byte-long base-ten length of the XML data, a padding of 28 zeros reserved for future use, the actual XML data, and optional binary data. The first XML element is commandPacket, which contains attributes such as the origin identifier and the destination identifier. The last attribute of a commandPacket is the size of the optional binary data, whose value is zero if there are no data.

The commandPacket element contains two child elements: route and commandXML. The route element contains child hop elements, which are elements corresponding to nodes that relay information from the source node to the destination node through zero or more nodes. Information is sent from the origin to the first hop to the second hop to the third hop, and so on, until the information is delivered at the destination. The commandXML element consists of an attribute named *commandType*, whose value is set to the unique identifier of the command. The content of the commandXML consists of the serialized representation of the command state at the sender, which is to be unserialized into the same command state at the receiver.

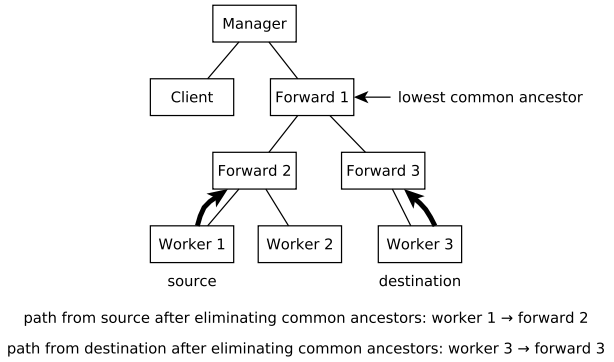
4.2.2 Routing

To establish a route, the source node must look in the tree for itself and the specified destination node. Afterwards, a path in the tree is traced from each node to the root of the tree. A common ancestor is then found, which is the first node that does not match from the root to the node preceding it. Thus, the path of delivery from the source node to the destination node is a stack containing the upward path trace from the source to the common ancestor, followed by the downward path trace from the common ancestor to the destination node.

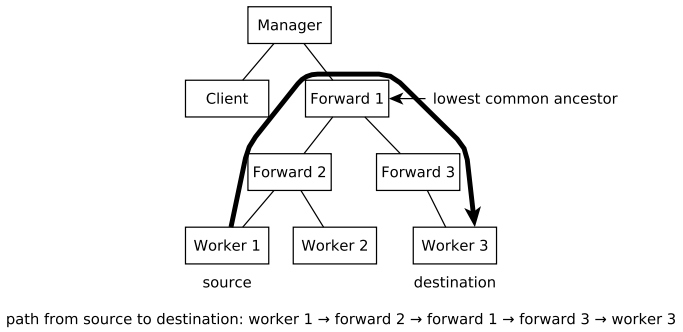
Figure 4.2 illustrates an example of that routing mechanism. In this example, a single client is connected directly to the manager, and the manager is connected to a forward (Forward 1), which is connected to two other forwards (Forwards 2 and 3). Forward 2 is connected to two workers (Workers 1 and 2), and Forward 3 is connected to another worker (Worker 3). In Figures 4.2a, 4.2b, and 4.2c, Worker 1 is attempting to send data to



(a) Step 1. Two stacks are created with the manager at the top of each stack and the source or destination at the bottom of the stacks.



(b) Step 2. All items that are found in both stacks are removed.



(c) Step 3. The lowest common ancestor followed by all the items in the source stack (except the one corresponding to the source), starting from the top, are pushed to the destination stack. The second stack represents a route in which the next hop is at the top of the stack.

Figure 4.2: The steps for establishing a route.

Worker 3. (Note that that would rarely happen in practice; this example is shown here only for illustrative purposes.) To establish a route for sending the data, Worker 1 needs to find the most efficient path in the RJS tree. To do so, the source node performs the following steps. First, it traces a path from itself to the manager node and places the nodes in a stack, and then it traces a path from the destination (Worker 3) to the manager node and places the nodes in another stack (see Figure 4.2a). Next, it prunes the stacks by popping the top of each stack until the tops of the stacks do not match (see Figure 4.2b). Finally, it adds the lowest common ancestor (i.e., the most recently popped item from the stacks) and appends all items in the first stack, except the one corresponding to the source, to the second stack (see Figure 4.2c). Since the top of the second stack represents the next hop, each host can simply pop the top item from the stack when the packet arrives.

4.2.3 Packet Serialization and Deserialization

As stated before, command packets are the messages sent between RJS nodes. Command packets must contain enough data so that the command state at the receiving end is the same as the original command state at the sending end. The conversion of state into its data representation is known as *serialization*, and RJS uses the Apache Xerces C++ library [11] for reading and writing the state in the XML format. The opposite conversion of the state's data representation into its actual state is known as *deserialization*.

Each class for a particular command is derived from a base command class. While a command packet does share ownership of the command, the command packet also holds the routing information for the command. The base command, command packet, and network classes are derived from a serialization class, which is responsible for holding the output data from Xerces C++ library API functions to serialize and unserialize the data.

Figure 4.3 illustrates that idea in the form of pseudocode. Here, the cross-platform representation of the data in XML format is known as the *Document Object Model* (*DOM*); *node* is a reference to the Xerces DOM node state. Figure 4.3a shows the pseudocode for serialization of data in a given class, and Figure 4.3b shows the pseudocode for unserializing of data. Examples of Xerces functions that write to the DOM node that are commonly used

```
function class::SetupDOMNode(node)
    if super(class)::SetupDOMNode is not purely virtual then
        super(class)::SetupDOMNode(node)
    end if
    Serialize the class by calling Xerces functions that write to node
end function
```

(a) Serialization setup pseudocode.

```
function class::InitFromDOM(node)
    if super(class)::InitFromDOM is not purely virtual then
        super(class)::InitFromDOM(node)
    end if
    Unserialize the class by calling Xerces functions that read from node
end function
```

(b) Unserialization setup pseudocode.

Figure 4.3: Pseudocode that sets the stage for serialization and unserialization. $super : \mathcal{C} \mapsto \mathcal{C}$, where $\mathcal{C}$ denotes the set of all serializable classes, is a mapping function that symbolizes the parent class of the given class.

by RJS include `setAttribute`, which adds an XML attribute to the node, and `appendChild`, which adds a child node to the node. Examples of Xerces functions that read from the DOM node that are commonly used by RJS include `getAttribute`, which retrieves the given XML attribute from the node, and `getChildNodes`, which retrieves the child nodes of the node.

Figure 4.4 shows how the base serializable class calls the setup functions in the actual serialization and unserialization routines. The serialization routine in Figure 4.4a is invoked when the command data are to be sent, and the unserialization routine in Figure 4.4b is invoked when command data are received. Note that the setup functions in the base serialization class are purely virtual, meaning that the class actually represents a derived class such as a command class or a network class. As a result, the actual functions called are the ones in the derived class.

function SERIALIZABLE::SERIALIZE
Create DOM document, and store reference to document element to *element*
SERIALIZABLE::SETUPDOMNODE(*element*)
Create serializer, call its serialization function, and store resulting buffer into *retval*
return *retval*
end function

(a) Serialization pseudocode.

function SERIALIZABLE::UNSERIALIZE(*serialized*)
Create DOM parser for *serialized*, call its parse function, and store reference to resulting document element into *element*
SERIALIZABLE::INITFROMDOM(*element*)
end function

(b) Unserialization pseudocode.

Figure 4.4: Pseudocode for serialization and unserialization.

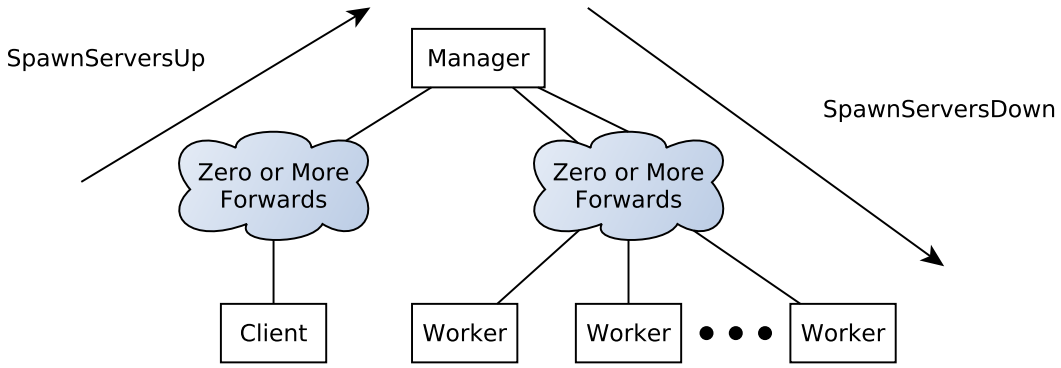


Figure 4.5: An illustration of how nodes are spawned in an RJS network.

4.3 RJS Network Initialization and Shutdown

4.3.1 RJS Network Initialization

Network initialization occurs when an authorized host sends a network initialization request to the parent host in the RJS tree. An *authorized host* is a host in the RJS network that is designated to initialize, change, and tear down the network; and the *parent host* is a host that represents the parent of the node in the RJS tree corresponding to the current host. Figure 4.5 shows how nodes are created. The creation of nodes requires upward spawn (SpawnServersUp) and downward spawn (SpawnServersDown) commands. The SpawnServersUp command is called when a node is spawning parent nodes (e.g., used for spawning a manager node from a host designated as a client); and the SpawnServersDown command is called when a node is spawning child nodes (e.g., used for spawning worker nodes from the host designated as a manager).

An RJS network is initialized as follows. The client checks whether its parent in the corresponding RJS model is online. If it is, the client initiates a handshake sequence with the parent. If it is not, the client remotely executes a new RJS process as a daemon and checks whether the process has been successfully created. If it has not been successfully created, the client returns a failure. Otherwise, the client initiates a handshake sequence with the newly created parent (e.g., the manager node or a forward node leading to the manager node). Each ancestor node recursively does the same until the node creation request reaches the manager. The manager then checks whether all of its children are online. For each child that is online, it initiates a handshake sequence with it. For each child that is not online, it remotely executes a new RJS process as a daemon and checks whether the process has been successfully created. If it has not been successfully created, the manager returns a failure, which is reported to the client. Otherwise, the manager initiates a handshake sequence with the newly created child. Each descendant node recursively does the same until the node creation request reaches the workers.

```
timeout ← Implementation-defined timeout
numTimeouts ← Implementation-defined number of timeouts
shutdownStateCode ← 0
node ← GetCurrentNode
node.waitSet ← children
for all descendant ∈ node.GetAllDescendants do
    descendant.waitSet ← descendant.children
end for

function ShutdownCommand::OnReceive(clientID)
    shutdownStateCode ← 1
    k ← 0
    done ← ⊥
    source ← GetNodeByID(nodeID)
    while (k < numTimeouts) ∧ (done = ⊥) do
        numChildren ← node.children.count
        for all child ∈ node.children do
            if child.id ≠ nodeID ∧ ¬(∃source ∈ child.GetAllDescendants) then
                cmd ← CreateCommand(ShutdownNodeRequest)
                cmd.originID ← node.id
                SendCommandTo(cmd,child.id)
            end if
        end for
        Wait until waitSet = ∅ or timeout of timeout is reached
        if last operation timed out then
            k ← k + 1
        else
            done = ⊤
        end if
    end while
    if (done = ⊥) ∧ (clientID ≠ 0) then
        cmd ← CreateCommand(NodeShutdownFailure)
        cmd.waitSet ← node.waitSet
        SendCommandTo(cmd,clientID)
    end if
    Shutdown
end function
```

Figure 4.6: Pseudocode for node shutdown.

```
function ShutdownNodeRequest::OnReceive(originID)
    shutdownStateCode ← 2
    if node.IsWorker then
        Shutdown
    else
        for all child ∈ children do
            cmd ← CreateCommand(ShutdownNodeRequest)
            cmd.originID ← node.id
            SendCommandTo(cmd,child.id)
        end for
    end if
end function

function ReaderThread::OnDisconnect(nodeID)
    waitSet ← waitSet\{GetNodeByID(nodeID)}
    if waitSet = ∅ then
        Shutdown
    else if (∃node.parent) ∧ (nodeID = node.parent.id) then
        cmd ← CreateCommand(ShutdownCommand)
        cmd.clientID = 0
        ScheduleCommand(cmd)
    else if shutdownStateCode = 2 then
        cmd ← CreateCommand(NodeShutdownStatus)
        cmd.nodeID ← node.id
        cmd.childWaitSet ← node.waitSet
        SendCommandTo(cmd,node.parent.id)
    end if
end function

function NodeShutdownStatus::OnReceive(nodeID,childWaitSet)
    GetNodeByID(nodeID).waitSet = childWaitSet
    if shutdownStateCode = 2 then
        cmd ← CreateCommand(NodeShutdownStatus)
        cmd.nodeID ← node.id
        cmd.waitSet ← node.waitSet
        SendCommandTo(cmd,node.parent.id)
    end if
end function
```

Figure 4.6: Pseudocode for node shutdown (cont.).

4.3.2 RJS Network Shutdown

RJS network shutdown occurs when an authorized host sends a network shutdown request to the parent host in the RJS tree, or a node has detected that its parent node is disconnected. A clean shutdown is required in order to free all memory and cancel any jobs without leaving simulations running in the absence of workers. The pseudocode for the shutdown procedure is shown in Figure 4.6. For the sake of simplicity, it is assumed that the node to be shut down is the manager, a forward between the manager and a worker, or a worker. The timeout and number of timeouts are set to the quantities defined by the RJS implementation, which were arbitrarily chosen to be two seconds and three, respectively. The shutdown state takes on three values: zero, one, and two. A value of zero means that RJS is not in the shutdown state. A value of one means that the RJS node has received a request to shut down from the client and is currently conducting a shutdown. A value of two means that the RJS node has received a shutdown request. A wait set is the set of all nodes that have yet to shut down. Each node carries its own wait set.

When a shutdown command from the client is received (see ShutdownCommand::OnReceive in Figure 4.6), the node conducts a shutdown procedure and sets the shutdown state to one. Then, it sends shutdown requests to its children until all children have been shut down or the timeout occurs. If the timeout occurs, it tries again, or if the node has exceeded its number of tries, it sends the results to the client (if the client is not anonymous) and simply shuts down.

When a node receives a shutdown request (see ShutdownNodeRequest::OnReceive in Figure 4.6), the node simply shuts down if it is a worker or relays the request to its children otherwise.

When a node detects that one of its links has disconnected (see ReaderThread::OnDisconnect in Figure 4.6), it removes the node from the wait set (if it exists in the wait set). If there are no more items in the wait set, the node shuts down. If the node ID refers to the parent, the node starts a shutdown sequence on itself with an anonymous client ID. If the node received a shutdown request (i.e. its shutdown state is set to two), then it sends a shutdown status report with its ID and its wait set to its parent.

When a node receives a shutdown status report (see NodeShutdownStatus::OnReceive in Figure 4.6), it updates the wait set of its child node and relays the shutdown status report to its parent.

4.4 RJS Simulation Framework

RJS is an extension to Möbius's simulation framework, where each simulation consists of one or more experiments. Möbius, an RJS client, allows the user to set up simulation parameters such as the maximum number of batches, the number of batches to store in the simulator before sending them to the worker, the random number generator, the experiment numbers to run, the random number generator and seed, and whether the simulation is terminating or steady-state. Each simulation run is identified by a job number, starting from zero. The workers, on the other hand, receive project archives, compile the files required for simulation, and run the simulation. It is especially important that no nodes other than the worker nodes compile the simulations, since that allows any operating system—Windows, Macintosh OS X, or Linux—to function as a worker, which also allows the RJS network to be heterogeneous.

4.4.1 Job Requests

To initiate the job, the manager must know the file name, the integrity of the file, the name of the solver, and the set of experiments to run. The file name is used to determine whether the manager has a copy of the file. The integrity of the file is determined by comparing the SHA-512 hash of the file computed from the client and the SHA-512 hash of the file computed from the stored copy at the manager node, and it is used to verify whether the manager has a valid copy of the file. The solver name is needed to determine the appropriate simulation to run. The set of experiments to run allows the manager to allocate an initial number of workers ahead of time.

The client first requests a job ID by sending the filename, the SHA-512 hash of the file, the solver name, and the set of active experiments to run. The manager then 1) selects at most n of the most-available workers and sets the stage of the workers to "not started,"

where n is the number of active experiments to run; 2) stores the assigned workers, project name, solver name, and active experiments in a job model; and 3) replies to the client with the job ID and a Boolean flag indicating whether or not to send the file. (For example, the file would not be sent if it does not exist on the manager or if the SHA-512 hash from the client does not match the computed hash on the manager node.) If the client needs to send the file, it starts a new thread that computes the SHA-512 hash of the file and sends the file in small chunks to the manager in the form of file part transfer commands. The transfer command includes information such as the filename, the job ID, the data, the SHA-512 hash, and a flag determining whether the end of the file has been reached. As soon as the client finishes reading the end of the file, the manager responds back with a file transfer completion command with the job ID and a Boolean flag indicating whether the file was successfully transferred. The manager can determine whether a file transfer was successful by comparing the given SHA-512 hash with the computed SHA-512 hash of the manager's copy. Once the manager acquires a valid copy of the file, or determines that it already has a valid copy of the file, the simulation is ready to begin.

4.4.2 Simulation Setup

Before the simulation job can begin, the manager must use the same file transfer protocol as described above. In other words, it must ensure that the project files of the assigned workers are the same as the copies stored on the manager node. Notice that a thread is used for file transfer to prevent a backlog of commands at the RJS node. The project file could have been transferred directly to the worker nodes; however, the manager must serve the client quickly, even when a job must wait until a worker becomes available, if no workers were assigned initially. Storing a copy of the file on the manager node improves the speed with which the manager submits a job for the client, because the client does not need to wait until a worker becomes available, and that allows the manager to assign jobs to new workers that become available at any time.

The manager sets the stage of the workers to "receiving file," asks them whether they have copies of the file, and proceeds to send its copy to any worker that does not have a valid copy.

Once all the assigned workers have the file, the manager schedules a build request to itself, which is distributed to all the assigned workers. Before the build request is distributed to a worker, the manager sets the stage of the worker to "building." The worker runs the build request sent by the manager; it involves decompression of the project files from the given archive and compiling of the project in the solver directory. If the compilation succeeds, the worker sends a request for a set of experiments to run. Otherwise, the worker notifies the manager of a build failure. The request is run in a separate thread, since a backlog of worker commands may occur if the request is run from the main job server thread.

4.4.3 Simulation Scheduling

After a worker successfully builds the project, it sends a command to request experiments to run. Before the worker requests an experiment, it reads the appropriate project files and finds the appropriate files describing the corresponding study and performance variable metrics associated with the simulation. The worker extracts the maximum number of batches, the number of batches to store in the simulator before the simulator sends them to the worker, the experiment numbers to run, the random number generator and seed, and whether the simulation is terminating or steady-state; that information is forwarded to the manager to be stored in the job model. The worker also sends the contents of the relevant files to the manager so that the manager can read additional information, such as the names of the performance variables. As soon as the manager receives the relevant information, it sets the stage of the worker in the job to "ready."

After a worker is set to "ready" in the job, the manager directs the worker to run experiments based on the schedule it creates for the workers in the RJS network. The pseudocode for the scheduling algorithm used in RJS is shown in Figure 4.7. The algorithm runs in three phases. The first phase is the creation of the job schedule; the second phase is the allocation of tasks to workers, such as building and running of simulations; and the third phase is the actual communication of commands to direct the workers to start the tasks.

The first phase is shown in Figure 4.8, which shows the pseudocode for creation of a schedule map describing which type of task to run and how many tasks to run. The key is

```
function JobScheduleCommand::schedule(w ∈ V_w)
    coresRemaining ← coreCount(w) − numActiveJobs(w)
    if coresRemaining = 0 then
        return
    end if
    jobAssignmentMap ← JobScheduleCommand::createJobScheduleMap(w,
coresRemaining)
    sendMap ← ∅, cancelMap ← ∅, replyMap ← ∅, rescheduleSet ← ∅
    for ⟨kJobId, vAssignData⟩ ∈ jobAssignmentMap do
        jobModel ← Network::GetJob(kJobId)
        if vAssignData.sendFilesToWorker then
            Network::AssignBuildInJob(jobModel,w)
            jobModel.state ← InProgress
            sendMap ← sendMap ∪ {⟨kJobId, jobModel.projectFileName⟩}
        else
            data.projectName ← job.projectName
            data.solver ← job.solver
            data.maxBatches ← job.maxBatches
            data.reportFrequency ← job.reportFrequency
            data.isTerminating ← job.isTerminating
            data.randomNumberGenerator ← job.randomNumberGenerator
            data.expSet ← ∅
            for i ∈ {1, . . . , vAssignData.taskAssignmentCount} do
                expNum ← Network::AssignExperimentInJob(jobModel, w)
                nextSeed ← jobModel.GetNextRandomNumber
                data.expSet ← data.expSet ∪ {⟨expNum, nextSeed⟩}
            end for
            if data.expSet ≠ ∅ then
                JobScheduleCommand::checkAggThread(jobModel,data)
                jobModel.SetStage(w, Running)
                JobScheduleCommand::updateAggThread(kJobId,w.id)
                replyMap ← replyMap ∪ {⟨kJobId, data⟩}
                if job.activeExpsRemaining = ∅ then
                    ⟨s1, s2⟩ ← JobScheduleCommand::CancelBuildingWorkers
                    cancelMap ← cancelMap ∪ s1
                    rescheduleSet ← rescheduleSet ∪ s2
                end if
            end if
        end if
    end for
    JobScheduleCommand::DispatchData(sendMap, cancelMap, replyMap,
rescheduleSet)
end function
```

Figure 4.7: Scheduling pseudocode.

```
function JobScheduleCommand::createJobScheduleMap(w ∈ 𝒱_w,
coresRemaining ∈ ℕ)
    assignMap ← {}, fairCounts ← [], done ← ⊥, rounds ← 0
    jobSet ← w.GetAllWaitingJobs, sorted by job timestamps
    numNewExperimentsInIteration ← k, where k is any integer other than zero
    while ¬done ∧ numNewExperimentsInIteration ≠ 0 do
        k ← 0, ℓ ← 0, numNewExperimentsInIteration ← 0
        for job ∈ jobSet do
            if coresRemaining = 0 then
                done ← ⊤, break
            end if
            if job.GetStage(w) = NotStarted then
                if |assignMap| = k then
                    if job.activeExpsRemaining ≠ ∅ then
                        data.sendFilesToWorker ← ⊤
                        data.taskAssignmentCount ← 1
                        assignMap ← assignMap ∪ {⟨job.id, data⟩}
                        coresRemaining ← coresRemaining − 1, k ← k + 1
                    end if
                end if
            else if rounds ≥ job.CountNumTasksAssignedToWorker(w) then
                if |fairCountVec| = ℓ then
                    count ← JobScheduleCommand::ComputeFairExpCount(job, w)
                    fairCountVec ← fairCountVec ∘ [count]
                end if
                fairAssignCount ← fairCountVec[ℓ]
                if fairAssignCount > 0 then
                    fairCountVec[ℓ] ← fairAssignCount − 1
                    if |assignMap| = k then
                        data.sendFilesToWorker ← ⊥
                        data.taskAssignmentCount ← 0
                        assignMap ← assignMap ∪ {⟨job.id, data⟩}
                    end if
                    coresRemaining ← coresRemaining − 1
                    numNewExpsInIteration ← numNewExpsInIteration + 1
                end if
                ℓ ← ℓ + 1
            end if
        end for
        rounds ← rounds + 1
    end while
    return assignMap
end function
```

Figure 4.8: Pseudocode for schedule map construction.

```
function JobScheduleCommand::ComputeFairExpCount(job, w ∈ 𝒱_w)
    workerSet ← job.GetWorkerOwnerSet
    countVector ← []
    expsRemaining ← 0
    for worker ∈ workerSet do
        numExps ← job.CountNumExperimentsAssignedToWorker(worker)
        coresAvail ← worker.numCores > worker.ComputeNumActiveTasks ≡ ⊤
        expCountDesc.workerID ← worker.id
        expCountDesc.assignCount ← numExps
        expCountDesc.hasCoresAvail ← coresAvail
        countVector ← countVector ∘ [expCountDesc]
        expsRemaining ← expsRemaining + numExps
    end for
    Sort countVector by largest value of assignCount, followed by true values of
hasCoresAvail, followed by workerID matching w.workerID
    surplusCount ← 0
    limit ← |job.activeExpsRemaining|
    workersRemaining ← |countVector|
    expsRemaining ← expsRemaining + limit
    for expCountDesc ∈ countVector do
        expectedFairnessCount ← ⌈expsRemaining / workersRemaining⌉
        numExpsToAssign ← expectedFairnessCount − expCountDesc.assignCount
        if expCountDesc.workerID = w.id then
            numExpsToAssign ← numExpsToAssign + surplusCount
            return min(max(0, numExpsToAssign), limit)
        else if ¬expCountDesc.hasCoresAvail ∧ numExpsToAssign > 0 then
            surplusCount ← surplusCount + numExpsToAssign
        end if
        expsRemaining ← expsRemaining − expectedFairnessCount
        workersRemaining ← workersRemaining − 1
    end for
    return 0
end function
```

Figure 4.9: Pseudocode for counting a fair number of experiments for the given job.

the job ID, and the value is the data descriptor describing whether the task requires transfer of files (i.e., whether it is a build task), how many tasks to assign (which is always one in the case of build tasks), and the number of experiments to run in the case of simulation tasks. The procedure first determines whether there are available cores on the given worker. That is, it determines whether the total number of tasks running on the worker is strictly less than the number of cores available on the worker. If there are no cores available, then the procedure exits. Otherwise, the set of waiting jobs sorted by timestamps is obtained, which is a set of jobs that have not started, are ready to run experiments, or are already running experiments, and have extra cores available to run the tasks. The algorithm runs through the set of experiments and assigns the tasks one by one until there are no more cores available or no more tasks remain. That helps ensure that the earlier jobs run first, which helps maximize the number of jobs running on the worker. Build jobs simply occupy a single processor core on the worker, but the number of simulation jobs to run on the worker depends on the value returned by the procedure in Figure 4.9. In that procedure, the number of experiments running on each worker is computed and then sorted by largest to smallest number. Next, the procedure computes a fair number of jobs to assign, which depends on the number of experiments running on the other workers. A surplus value representing the shortage of experiments on heavily loaded workers is computed and added to the fair number of experiments to assign to a worker. On the other hand, if all workers have processors available to run, then the load becomes balanced, since the algorithm tries to assign the same amount of experiments to each worker. As a result, the algorithm balances the tradeoff between fairness (when the workers have processors available to run experiments) and greediness (when most of the workers are completely loaded, i.e., they have no processors available to run tasks).

The second phase in Figure 4.7 is the allocation of tasks to workers, which involves running through the schedule map returned by the procedure in Figure 4.8. The for loop in the procedure constructs three additional maps: a map for starting a build job and sending files, a map for replying to workers with experiments in the case of simulation jobs, and a map for canceling all build tasks on all active workers when no more experiments in a job remain. All the maps are keyed by the job ID, and each key has a value describing the

information required to dispatch tasks to the given worker. The loop also constructs the set of IDs corresponding to all the workers that require rescheduling due to job cancellation. In other words, it constructs the set of all worker IDs described in the cancellation map. If the schedule map indicates that the worker needs to send files, then the send map is updated with values needed to start the build task. Otherwise, the parameters for starting a simulation are gathered, experiments are assigned to the worker in the job, and the reply map is updated with these parameters.

The third stage in Figure 4.7 involves dispatching of the data to the appropriate workers, which involves running through each item in the three maps and the one set and then sending the appropriate commands. Each item in the send map has data that are used for sending a build task request to the given worker; each item in the reply map has data that are used for sending an experiment task request to the given worker; each item in the cancellation map has data that are used to cancel the jobs on the given workers; and the reschedule set has data that are used to rerun the scheduling procedure on each worker specified.

4.4.4 Simulation Startup

When a worker receives a task request to a given worker, it gathers the information to run the experiments, which includes the set of experiments to run, the maximum number of batches, the number of batches to store in the simulator before it sends them to the worker, the random number generator and seed, and whether the simulation is terminating or steady-state. The worker runs each experiment in a separate thread, and each thread initiates the simulator binary using the information provided by the manager. Therefore, the number of new simulation tasks is the number of experiments given by the manager.

The simulator sends the observed values of the performance variables through TCP sockets to help separate program output from the observation data it generates. When a simulation binary is executed, it tries to find an unused port by starting from the port number it was given and incrementing the number by one until it successfully binds to a port. However, to make the process more efficient, the RJS worker first finds a random port to use by binding to port zero, which is an efficient way for the operating system to find and report an unused

port. That technique of specifying system-allocated (or dynamic) ports is especially useful when a worker with many processor cores is fully loaded with simulation runs. Without taking advantage of system-allocated ports, a block of consecutive ports may be used, which means that the complexity of finding a port can be $O(n)$, where n is the number of workers, rather than amortized constant time, in most cases.

Once the worker obtains an unused port from the operating system, the worker collects the simulator parameters provided by the manager and launches the simulation binary with this information. The worker reads the port number that the simulator will use to send the observation data to the worker and then connects to the simulator binary using the given port. As soon as the simulator establishes the connection, it sends a registration packet to the worker; the packet includes the process ID in case a client sends a cancellation request to the manager, in which case the worker needs to kill the simulation job as soon as the manager forwards the request to it.

4.4.5 Observation Collection

After sending the registration packet to the worker, the simulator starts running the discrete-event simulation. It collects the observed values of all the performance variables and stores them in batches until the threshold number of batches has been reached. The simulator sends the batches of observed values to the worker, and the raw data from the worker is forwarded to the manager.

4.4.6 Data Aggregation

Once the manager receives the raw observation data, it converts the data into a vector of observation data, which is reported to the appropriate data aggregation thread for the job. The data aggregation thread uses the project information that it gathered from the worker and aggregates it with the received observation data to produce statistical results for each performance variable, such as the mean, variance, distribution of observed values, and number of observed values that fall within an interval. If the aggregation thread detects

that a computation of a performance variable has completed before the maximum number of simulation batches has been reached (e.g., as a result of convergence of the mean or variance), it will mark the performance variable as having completed the computation. If all performance variables have converged before the maximum number of simulation batches has been reached, then the aggregation thread will tell the worker to quit the simulation so that another experiment or job can run.

The confidence interval computed from the critical values of the Student t-distribution is used to determine whether the mean or variance value has converged. The degrees of freedom is equal to the number of observations, and the confidence level CL specified by the client is used to determine the largest allowable confidence interval half-width. A sufficiently large number of observations is considered to be equivalent to infinite degrees of freedom, which is equivalent to critical values of the standard Gaussian distribution.

The computation of the mean is trivial: the observation statistics are summed up, and the mean value is determined by taking the sum and dividing it by the number of observations that were added up. The confidence interval is determined by taking the critical value and multiplying it by the square root of the sample variance σ^2 divided by the number of observations n. Therefore,

$$CI = t_{n,(1-CL)/2}\sqrt{\frac{\sigma^2}{n}},$$

where $t_{n,(1-CL)/2}$ represents the inverse cumulative distribution function of the Student t-distribution that satisfies

$$\int_{-\infty}^{t_{n,(1-CL)/2}} f(t)\,dt = 1 - \frac{1-CL}{2},$$

where

$$f(t) \triangleq \frac{\Gamma\left(\frac{n+1}{2}\right)}{\sqrt{n\pi}\,\Gamma\left(\frac{n}{2}\right)}\left(1+\frac{t^2}{n}\right)^{-\frac{n+1}{2}} \tag{4.1}$$

is the probability density function of the Student t-distribution.

For a sufficiently large value of n, the critical values of the Student t-distribution can be approximated by the critical values of the standard normal distribution. Therefore, the

expression of $f(t)$ in Eq. 4.1 is approximated by

$$f(t) \approx \frac{1}{\sqrt{2\pi}} e^{-t^2/2},$$

which means that the critical values are approximated by using

$$t_{n,(1-CL)/2} \approx \Phi^{-1}\left(1 - \frac{1-CL}{2}\right),$$

where $\Phi^{-1}(x)$ is the inverse cumulative distribution function of the standard normal distribution, otherwise known as the *quantile function*. RJS utilizes a lookup table for determining the critical value given n and CL, and the approximation helps to define the limiting n value so the approximation can be used for each confidence level provided. In RJS, the limiting value of n was chosen to be the same as the limiting value of n in the Möbius suite, which is 31.

The sample variance for each performance variable is computed using the following formula:

$$\sigma^2 = \frac{1}{n-1}\left(\sum_{i=1}^{n} {x_i}^2 - \frac{\left(\sum_{i=1}^{n} x_i\right)^2}{n}\right), \tag{4.2}$$

where x_i is the i-th observed performance variable sample. A derivation leading to that formula follows.

The sample variance is defined by

$$\sigma^2 \triangleq \frac{1}{n-1}\sum_{i=1}^{n}\left(x_i - \overline{x}\right)^2, \tag{4.3}$$

where $\overline{x}$ is the sample mean defined by

$$\overline{x} \triangleq \frac{1}{n}\sum_{i=1}^{n} x_i. \tag{4.4}$$

Expanding the summation term of Eq. 4.3,

$$\begin{aligned}\sigma^2 &= \frac{1}{n-1}\sum_{i=1}^{n}\left({x_i}^2 - 2x_i\overline{x} + \overline{x}^2\right) \\ &= \frac{1}{n-1}\left(\sum_{i=1}^{n}{x_i}^2 - 2\left(\sum_{i=1}^{n}x_i\right)\overline{x} + n\overline{x}^2\right).\end{aligned} \tag{4.5}$$

Substituting Eq. 4.4 into Eq. 4.5 and simplifying,

$$\begin{aligned}\sigma^2 &= \frac{1}{n-1}\left(\sum_{i=1}^{n}{x_i}^2 - 2\frac{\left(\sum_{i=1}^{n}x_i\right)^2}{n} + \frac{\left(\sum_{i=1}^{n}x_i\right)^2}{n}\right) \\ &= \frac{1}{n-1}\left(\sum_{i=1}^{n}{x_i}^2 - \frac{\left(\sum_{i=1}^{n}x_i\right)^2}{n}\right).\end{aligned}$$

The computation of the variance and associated confidence interval of the performance variables is rather nontrivial, since it is based on jackknife resampling. The computed value has been determined to be the same as the value of Eq. 4.2, and the confidence interval is computed as

$$CI = t_{n,(1-CL)/2}\sqrt{\frac{{\sigma_z}^2}{n}},$$

where

$${\sigma_z}^2 = \frac{1}{n-1}\left[n\cdot\frac{3{S_2}^2 + nS_4 - 4S_3S_1}{(n-2)^2} - 4\cdot\frac{\left(nS_2 - {S_1}^2\right)^2}{n(n-2)^2}\right], \tag{4.6}$$

and

$$S_j \triangleq \sum_{k=1}^{n}{x_k}^j$$

represents an "unscaled" j-th moment (i.e., the j-th moment multiplied by n). A derivation leading to Eq. 4.6 follows.

The jackknife mean is defined as

$$x_i^J = \frac{1}{n-1}\left[S_1 - x_i\right],$$

and the jackknife variance is defined as

$$
\begin{aligned}
\left(\sigma_i^J\right)^2 &= \frac{1}{n-2}\left[\left(\sum_{k=1}^{n}\left(x_k - x_i^J\right)^2\right) - \left(x_i - x_i^J\right)^2\right] \\
&= \frac{1}{n-2}\left[\left(\sum_{k=1}^{n} {x_k}^2\right) - 2x_i^J\left(\sum_{k=1}^{n} x_k\right) + n\left(x_i^J\right)^2 - \left({x_i}^2 - 2x_i^J x_i + \left(x_i^J\right)^2\right)\right] \\
&= \frac{1}{n-2}\left[\left(S_2 - {x_i}^2\right) - 2x_i^J\left(S_1 - x_i\right) + (n-1)\left(x_i^J\right)^2\right] \\
&= \frac{1}{n-2}\left[\left(S_2 - {x_i}^2\right) - \frac{2\left(S_1 - x_i\right)^2}{n-1} + \frac{\left(S_1 - x_i\right)^2}{n-1}\right] \\
&= \frac{1}{n-2}\left[\left(S_2 - {x_i}^2\right) - \frac{\left(S_1 - x_i\right)^2}{n-1}\right]. \qquad (4.7)
\end{aligned}
$$

Suppose

$$\mu_x = \frac{S_1}{n}$$

is the mean of the n samples $x_1, \ldots, x_n$, and

$$
\begin{aligned}
{\sigma_x}^2 &= \frac{1}{n-1}\sum_{k=1}^{n}\left(x_k - \mu_x\right)^2 \\
&= \frac{1}{n-1}\left[S_2 - \frac{{S_1}^2}{n}\right] \qquad (4.8)
\end{aligned}
$$

is the variance of the n samples $x_1, \ldots, x_n$. The corrected variance of sample i is thus

$$
\begin{aligned}
\left(\sigma_i^{\mathrm{corr}}\right)^2 &= n{\sigma_x}^2 - (n-1)\left(\sigma_i^J\right)^2 \\
&= \frac{nS_2 - {S_1}^2}{n-1} - (n-1)\left(\sigma_i^J\right)^2.
\end{aligned}
$$

The variance in this case is defined as the average value of the corrected variances:

$$
\begin{aligned}
\sigma^2 &= \frac{1}{n}\sum_{i=1}^{n}\left(\sigma_i^{\mathrm{corr}}\right)^2 \\
&= \frac{1}{n}\sum_{i=1}^{n}\left(n{\sigma_x}^2 - (n-1)\left(\sigma_i^J\right)^2\right) \\
&= n{\sigma_x}^2 - \frac{n-1}{n}\left(\sum_{i=1}^{n}\left(\sigma_i^J\right)^2\right),
\end{aligned}
$$

where

$$
\begin{aligned}
\sum_{i=1}^{n}\left(\sigma_i^J\right)^2 &= \sum_{i=1}^{n}\left[\frac{1}{n-2}\left[\left(S_2 - {x_i}^2\right) - \frac{\left(S_1 - x_i\right)^2}{n-1}\right]\right] \\
&= \frac{1}{n-2}\left[\sum_{i=1}^{n}\left(S_2 - {x_i}^2\right) - \frac{\sum_{i=1}^{n}\left(S_1 - x_i\right)^2}{n-1}\right] \\
&= \frac{1}{n-2}\left[\left(nS_2 - \sum_{i=1}^{n}{x_i}^2\right) - \frac{\sum_{i=1}^{n}\left({S_1}^2 - 2x_iS_1 + x_i^2\right)}{n-1}\right] \\
&= \frac{1}{n-2}\left[(n-1)S_2 - \frac{n{S_1}^2 - 2\left(\sum_{i=1}^{n}x_i\right)S_1 + \sum_{i=1}^{n}{x_i}^2}{n-1}\right] \\
&= \frac{1}{n-2}\left[(n-1)S_2 - \frac{(n-2){S_1}^2 + S_2}{n-1}\right] \\
&= \frac{1}{n-2}\left[\frac{((n-1)^2-1)S_2 - (n-2){S_1}^2}{n-1}\right] \\
&= \frac{1}{n-2}\left[\frac{n(n-2)S_2 - (n-2){S_1}^2}{n-1}\right] \\
&= \frac{nS_2 - {S_1}^2}{n-1} = n{\sigma_x}^2.
\end{aligned}
$$

Therefore,

$$
\sigma^2 = n{\sigma_x}^2 - (n-1){\sigma_x}^2 = {\sigma_x}^2,
$$

which verifies that the variance is equal to the sample variance for the confidence interval of

the mean expressed in Eq. 4.2. The confidence interval is defined as

$$CI = t_{n,(1-CL)/2}\sqrt{\frac{\mathrm{Var}\left((\Sigma^{\mathrm{corr}})^2\right)}{n}},$$

where

$$\begin{aligned}\mathrm{Var}\left((\Sigma^{\mathrm{corr}})^2\right) &= \frac{1}{n-1}\sum_{i=1}^{n}\left((\sigma_i^{\mathrm{corr}})^2 - \sigma^2\right)^2 \\ &= \frac{1}{n-1}\left[\left(\sum_{i=1}^{n}\left((\sigma_i^{\mathrm{corr}})^2\right)^2\right) - \frac{\left(\sum_{i=1}^{n}(\sigma_i^{\mathrm{corr}})^2\right)^2}{n}\right], \end{aligned} \tag{4.9}$$

$$\begin{aligned}\sum_{i=1}^{n}\left((\sigma_i^{\mathrm{corr}})^2\right)^2 &= \sum_{i=1}^{n}\left[n{\sigma_x}^2 - (n-1)\left(\sigma_i^J\right)^2\right]^2 \\ &= \sum_{i=1}^{n}\left[n^2{\sigma_x}^4 - 2n(n-1){\sigma_x}^2\left(\sigma_i^J\right)^2 + (n-1)^2\left(\sigma_i^J\right)^4\right] \\ &= n^3{\sigma_x}^4 - 2n(n-1){\sigma_x}^2\left(\sum_{i=1}^{n}\left(\sigma_i^J\right)^2\right) + (n-1)^2\sum_{i=1}^{n}\left(\sigma_i^J\right)^4 \\ &= n^3{\sigma_x}^4 - 2n^2(n-1){\sigma_x}^4 + (n-1)^2\sum_{i=1}^{n}\left(\sigma_i^J\right)^4 \\ &= (n-2(n-1))n^2{\sigma_x}^4 + (n-1)^2\sum_{i=1}^{n}\left(\sigma_i^J\right)^4 \\ &= -(n-2)n^2{\sigma_x}^4 + (n-1)^2\sum_{i=1}^{n}\left(\sigma_i^J\right)^4, \end{aligned} \tag{4.10}$$

and

$$\begin{aligned}\frac{\left(\sum_{i=1}^{n}(\sigma_i^{\mathrm{corr}})^2\right)^2}{n} &= \frac{1}{n}\left[\sum_{i=1}^{n}\left(n{\sigma_x}^2 - (n-1)\left(\sigma_i^J\right)^2\right)\right]^2 \\ &= \frac{1}{n}\left[n^2{\sigma_x}^2 - (n-1)\sum_{i=1}^{n}\left(\sigma_i^J\right)^2\right]^2 \\ &= \frac{1}{n}\left[n^2{\sigma_x}^2 - n(n-1){\sigma_x}^2\right]^2 \\ &= n{\sigma_x}^4. \end{aligned} \tag{4.11}$$

Substituting Eq. 4.10 and Eq. 4.11 into Eq. 4.9,

$$\begin{aligned}
\mathrm{Var}\left((\Sigma^{\mathrm{corr}})^2\right) &= \frac{1}{n-1}\left[-(n-2)n^2{\sigma_x}^4 + (n-1)^2\sum_{i=1}^{n}\left(\sigma_i^J\right)^4 - n{\sigma_x}^4\right] \\
&= \frac{1}{n-1}\left[-\left(1+n(n-2)\right)n{\sigma_x}^4 + (n-1)^2\sum_{i=1}^{n}\left(\sigma_i^J\right)^4\right] \\
&= \frac{1}{n-1}\left[-(n-1)^2 n{\sigma_x}^4 + (n-1)^2\sum_{i=1}^{n}\left(\sigma_i^J\right)^4\right] \\
&= (n-1)\left[\sum_{i=1}^{n}\left(\sigma_i^J\right)^4 - n{\sigma_x}^4\right]. \qquad (4.12)
\end{aligned}$$

Substituting Eq. 4.7 into Eq. 4.12,

$$\begin{aligned}
\sum_{i=1}^{n}\left(\sigma_i^J\right)^4 &= \frac{1}{(n-2)^2}\sum_{i=1}^{n}\left[\left(S_2 - {x_i}^2\right) - \frac{\left(S_1 - x_i\right)^2}{n-1}\right]^2 \\
&= \frac{1}{(n-1)^2(n-2)^2}\sum_{i=1}^{n}\left[(n-1)\left(S_2 - {x_i}^2\right) - \left(S_1 - x_i\right)^2\right]^2. \qquad (4.13)
\end{aligned}$$

Expanding the summation on the right-hand side of Eq. 4.13,

$$\begin{aligned}
&\sum_{i=1}^{n}\left[(n-1)\left(S_2 - {x_i}^2\right) - \left(S_1 - x_i\right)^2\right]^2 \\
=\ &\sum_{i=1}^{n}\left[(n-1)^2\left(S_2 - {x_i}^2\right)^2 - 2(n-1)\left(S_2 - {x_i}^2\right)\left(S_1 - x_i\right)^2 + \left(S_1 - x_i\right)^4\right] \\
=\ &(n-1)^2\sum_{i=1}^{n}\left(S_2 - {x_i}^2\right)^2 - 2(n-1)\sum_{i=1}^{n}\left(S_2 - {x_i}^2\right)\left(S_1 - x_i\right)^2 + \sum_{i=1}^{n}\left(S_1 - x_i\right)^4,
\end{aligned}$$

where

$$\begin{aligned}\sum_{i=1}^{n}\left(S_2-{x_i}^2\right)^2 &= \sum_{i=1}^{n}\left({S_2}^2-2{x_i}^2S_2-{x_i}^4\right)\\ &= n{S_2}^2-2\left(\sum_{i=1}^{n}{x_i}^2\right)S_2-\sum_{i=1}^{n}{x_i}^4\\ &= n{S_2}^2-2{S_2}^2-S_4\\ &= (n-2)\,{S_2}^2-S_4,\end{aligned}$$

$$\begin{aligned}&\sum_{i=1}^{n}\left(S_2-{x_i}^2\right)\left(S_1-x_i\right)^2\\ =&\sum_{i=1}^{n}\left(S_2-{x_i}^2\right)\left({S_1}^2-2x_iS_1+{x_i}^2\right)\\ =&\, nS_2{S_1}^2-2\left(\sum_{i=1}^{n}x_i\right)S_1S_2+\left(\sum_{i=1}^{n}{x_i}^2\right)\left(S_2-{S_1}^2\right)+2\left(\sum_{i=1}^{n}{x_i}^3\right)S_1-\left(\sum_{i=1}^{n}{x_i}^4\right)\\ =&\,(n-2)\,S_2{S_1}^2+S_2\left(S_2-{S_1}^2\right)+2S_3S_1-S_4\\ =&\,(n-3)\,S_2{S_1}^2+{S_2}^2+2S_3S_1-S_4,\end{aligned}$$

and

$$\begin{aligned}\sum_{i=1}^{n}\left(S_1-x_i\right)^4 &= nS_1^4-4\left(\sum_{i=1}^{n}x_i\right){S_1}^3+6\left(\sum_{i=1}^{n}{x_i}^2\right){S_1}^2-4\left(\sum_{i=1}^{n}{x_i}^3\right)S_1+\left(\sum_{i=1}^{n}{x_i}^4\right)\\ &= (n-4)\,{S_1}^4+6S_2{S_1}^2-4S_3S_1+S_4.\end{aligned}$$

Therefore,

$$\begin{aligned}&-2\,(n-1)\sum_{i=1}^{n}\left(S_2-{x_i}^2\right)\left(S_1-x_i\right)^2+\sum_{i=1}^{n}\left(S_1-x_i\right)^4\\ =&-2\,(n-1)\left((n-3)\,S_2{S_1}^2+{S_2}^2+2S_3S_1-S_4\right)+\left((n-4)\,{S_1}^4+6S_2{S_1}^2-4S_3S_1+S_4\right)\\ =&-2n(n-4)S_2{S_1}^2-2(n-1){S_2}^2-4nS_3S_1+(2n-1)S_4+(n-4){S_1}^4,\end{aligned}$$

and

$$
\begin{aligned}
&(n-1)^2 \sum_{i=1}^{n} \left(S_2 - {x_i}^2\right)^2 - 2\,(n-1) \sum_{i=1}^{n} \left(S_2 - {x_i}^2\right)(S_1 - x_i)^2 + \sum_{i=1}^{n} (S_1 - x_i)^4 \\
&= \left(n^2 - 2n + 1\right)\left((n-2)\,{S_2}^2 - S_4\right) - 2n(n-4)S_2{S_1}^2 - 2(n-1){S_2}^2 - 4nS_3S_1 \\
&\qquad\qquad + (2n-1)S_4 + (n-4){S_1}^4 \\
&= n(n-3)(n-1){S_2}^2 + n^2S_4 + (n-4){S_1}^4 - 2n(n-4)S_2{S_1}^2 - 4nS_3S_1 \\
&= n^2(n-4){S_2}^2 - n^2(n-4){S_2}^2 + n(n-3)(n-1){S_2}^2 + n^2S_4 + (n-4){S_1}^4 \\
&\qquad\qquad - 2n(n-4)S_2{S_1}^2 - 4nS_3S_1 \\
&= (n-4)(nS_2 - {S_1}^2)^2 - n^2(n-4){S_2}^2 + n(n-3)(n-1){S_2}^2 + n^2S_4 - 4nS_3S_1 \\
&= (n-4)(nS_2 - {S_1}^2)^2 - (n^3 - 4n^2){S_2}^2 + (n^3 - 4n^2 + 3n){S_2}^2 + n^2S_4 - 4nS_3S_1 \\
&= (n-4)(nS_2 - {S_1}^2)^2 + 3n{S_2}^2 + n^2S_4 - 4nS_3S_1. \qquad (4.14)
\end{aligned}
$$

Substituting Eq. 4.8 and Eq. 4.14 into Eq. 4.12,

$$
\mathrm{Var}\left((\Sigma^{\mathrm{corr}})^2\right) = (n-1)\left[\frac{(n-4)(nS_2 - {S_1}^2)^2 + 3n{S_2}^2 + n^2S_4 - 4nS_3S_1}{(n-1)^2(n-2)^2} - \frac{\left(nS_2 - {S_1}^2\right)^2}{n(n-1)^2}\right],
$$

where

$$
\begin{aligned}
\frac{(n-4)\left(nS_2 - {S_1}^2\right)^2}{(n-1)^2(n-2)^2} - \frac{\left(nS_2 - {S_1}^2\right)^2}{n(n-1)^2} &= \frac{(n(n-4) - (n-2)^2)\left(nS_2 - {S_1}^2\right)^2}{n(n-1)^2(n-2)^2} \\
&= \frac{(n^2 - 4n - (n^2 - 4n + 4))\left(nS_2 - {S_1}^2\right)^2}{n(n-1)^2(n-2)^2} \\
&= -\frac{4\left(nS_2 - {S_1}^2\right)^2}{n(n-1)^2(n-2)^2}.
\end{aligned}
$$

Therefore,

$$
\begin{aligned}
\mathrm{Var}\left((\Sigma^{\mathrm{corr}})^2\right) = {\sigma_z}^2 &= (n-1)\left[\frac{3n{S_2}^2 + n^2S_4 - 4nS_3S_1}{(n-1)^2(n-2)^2} - 4 \cdot \frac{\left(nS_2 - {S_1}^2\right)^2}{n(n-1)^2(n-2)^2}\right] \\
&= \frac{1}{n-1}\left[n \cdot \frac{3{S_2}^2 + nS_4 - 4S_3S_1}{(n-2)^2} - 4 \cdot \frac{\left(nS_2 - {S_1}^2\right)^2}{n(n-2)^2}\right].
\end{aligned}
$$

RJS supports computation of the distribution of observed values and the number of observed values that fall within a interval. The former computation is known as *distribution estimation*, and the latter computation is known as *interval estimation*. Those two types of computations are similar, except that the interval estimation determines the estimated fraction of values out of all observations that lie within a single interval, and the distribution estimation determines the estimated fractions of values out of all observations that lie within a series of intervals and produces a histogram of relative frequencies.

The relative frequencies of the bins in a histogram, $p \in [0, 1]$, are assumed to follow a Bernoulli distribution, which means that the variance of the relative bin frequency is

$$\sigma_{\text{histogram}}{}^2 = p(1-p).$$

The Wilson score method with continuity correction [12] is derived from

$$|p_0 - p| - \frac{1}{2n} \leq \varphi\sqrt{\frac{p(1-p)}{n}},$$

where

$$\varphi = t_{n-1,(1-CL)/2},$$

$$p_0 = \frac{h_0}{n}$$

is the mean relative bin frequency, h_0 is the mean bin frequency, and n is the sum of all bin frequencies in the histogram. That means that the lower bound of the relative bin frequency is given by

$$p \geq p_0^- - \varphi\sqrt{\frac{p(1-p)}{n}}, \tag{4.15}$$

and the upper bound of the relative bin frequency is given by

$$p \leq p_0^+ + \varphi\sqrt{\frac{p(1-p)}{n}}, \tag{4.16}$$

where

$$p_0^- = p_0 - \frac{1}{2n} = \frac{h_0 - \frac{1}{2}}{n},$$

and

$$p_0^+ = p_0 + \frac{1}{2n} = \frac{h_0 + \frac{1}{2}}{n}.$$

The goal is to solve for the smallest p in Eq. 4.15 to obtain the smallest possible lower bound and to solve for the largest p in Eq. 4.16 to obtain the largest possible upper bound.

Before solving for p in Eq. 4.15, we first rewrite the equation as follows:

$$p_0^- - p \leq \varphi \sqrt{\frac{p(1-p)}{n}}.$$

Squaring both sides, we get

$$\left(p_0^- - p\right)^2 = \left(p - p_0^-\right)^2 \leq \varphi^2 \cdot \frac{p(1-p)}{n}.$$

Expanding both sides, we get

$$p^2 - 2pp_0^- + {p_0^-}^2 \leq \frac{\varphi^2}{n} p - \frac{\varphi^2}{n} p^2.$$

Moving all terms from the right-hand side to the left-hand side,

$$\frac{n + \varphi^2}{n} p^2 - 2 \cdot \frac{np_0^- - \frac{\varphi^2}{2}}{n} p + {p_0^-}^2 \leq 0.$$

Multiplying by n,

$$\left(n + \varphi^2\right) p^2 - 2 \left(np_0^- + \frac{\varphi^2}{2}\right) p + n{p_0^-}^2 \leq 0.$$

Using the quadratic formula,

$$p \lesseqgtr \frac{2\left(np_0^- + \frac{\varphi^2}{2}\right) \pm \sqrt{4\left(np_0^- + \frac{\varphi^2}{2}\right)^2 - 4\left(n + \varphi^2\right) \cdot n{p_0^-}^2}}{2\left(n + \varphi^2\right)}.$$

Eliminating the common constant in the numerator and denominator,

$$p \lesseqgtr \frac{np_0^- + \frac{\varphi^2}{2} \pm \sqrt{\left(np_0^- + \frac{\varphi^2}{2}\right)^2 - \left(n + \varphi^2\right) \cdot n{p_0^-}^2}}{n + \varphi^2}.$$

Expanding the discriminant,

$$p \lesseqgtr \frac{np_0^- + \frac{\varphi^2}{2} \pm \sqrt{np_0^-\varphi^2 + \varphi^2 \cdot \frac{\varphi^2}{4} - \varphi^2 \cdot {np_0^-}^2}}{n + \varphi^2}.$$

Factoring the discriminant,

$$p \lesseqgtr \frac{np_0^- + \frac{\varphi^2}{2} \pm \varphi\sqrt{\frac{\varphi^2}{4} + \frac{np_0^-\left(n - np_0^-\right)}{n}}}{n + \varphi^2}.$$

Since $np_0^- = h_0 - \frac{1}{2}$,

$$p \lesseqgtr \frac{h_0 - \frac{1}{2} + \frac{\varphi^2}{2} \pm \varphi\sqrt{\frac{\varphi^2}{4} + \frac{\left(h_0 - \frac{1}{2}\right)\left(n - h_0 + \frac{1}{2}\right)}{n}}}{n + \varphi^2}.$$

The minus sign in front of the square root of the discriminant gives the smallest possible lower bound:

$$p \geq \frac{h_0 - \frac{1}{2} + \frac{\varphi^2}{2} - \varphi\sqrt{\frac{\varphi^2}{4} + \frac{\left(h_0 - \frac{1}{2}\right)\left(n - h_0 + \frac{1}{2}\right)}{n}}}{n + \varphi^2}.$$

However, the discriminant may be negative when $h_0 = 0$. Luckily, by definition of p, p can never be less than 0, and this leads to a safe definition of the lower bound:

$$p \geq \begin{cases} \dfrac{h_0 - \frac{1}{2} + \frac{\varphi^2}{2} - \varphi\sqrt{\frac{\varphi^2}{4} + \frac{\left(h_0 - \frac{1}{2}\right)\left(n - h_0 + \frac{1}{2}\right)}{n}}}{n + \varphi^2} & \text{if } h_0 > 0 \\ 0 & \text{if } h_0 = 0 \end{cases}.$$

Before solving for p in Eq. 4.16, we first rewrite the equation as follows:

$$p - p_0^+ \leq \varphi\sqrt{\frac{p(1-p)}{n}}.$$

Squaring both sides, we get

$$\left(p - p_0^+\right)^2 \leq \varphi^2 \cdot \frac{p(1-p)}{n}. \tag{4.17}$$

From Eq. 4.17, it can be seen that the solution for p in Eq. 4.16 is identical to the solution

for p in Eq. 4.15:

$$p \lesseqgtr \frac{np_0^+ + \frac{\varphi^2}{2} \pm \varphi\sqrt{\frac{\varphi^2}{4} + \frac{np_0^+\left(n-np_0^+\right)}{n}}}{n+\varphi^2}.$$

Since $np_0^+ = h_0 + \frac{1}{2}$,

$$p \lesseqgtr \frac{h_0 + \frac{1}{2} + \frac{\varphi^2}{2} \pm \varphi\sqrt{\frac{\varphi^2}{4} + \frac{\left(h_0+\frac{1}{2}\right)\left(n-h_0-\frac{1}{2}\right)}{n}}}{n+\varphi^2}.$$

The plus sign in front of the square root of the discriminant gives the largest possible upper bound:

$$p \leq \frac{h_0 + \frac{1}{2} + \frac{\varphi^2}{2} + \varphi\sqrt{\frac{\varphi^2}{4} + \frac{\left(h_0+\frac{1}{2}\right)\left(n-h_0-\frac{1}{2}\right)}{n}}}{n+\varphi^2}.$$

However, the discriminant may be negative when $h_0 = n$. Luckily, by definition of p, p can never be greater than 1, and this leads to a safe definition of the upper bound:

$$p \leq \begin{cases} \dfrac{h_0 + \frac{1}{2} + \frac{\varphi^2}{2} + \varphi\sqrt{\frac{\varphi^2}{4} + \frac{\left(h_0+\frac{1}{2}\right)\left(n-h_0-\frac{1}{2}\right)}{n}}}{n+\varphi^2} & \text{if } h_0 < n \\ 1 & \text{if } h_0 = n \end{cases}.$$

CHAPTER 5

EXPERIMENTAL SETUP

For the testbed setup, two studies were examined. The first study examined the effects of specific network hierarchies on job performance, and the second study examined the scalability of RJS.

5.1 Effects of Network Hierarchy on Job Performance

In a realistic setting, delays due to network latency, context switching, and distribution of experiments can negatively impact the run time of a simulation. Context switching tends to occur more often when the number of processes running on a worker is large, since increasing the number of CPU-intensive processes running on a machine forces an increasing number of background processes to share the same CPU cores as the CPU-intensive processes. Network latency tends to be more pronounced for large models simply because of the high number of observations that each worker must report to the manager. Distribution of experiments can occur for any model and any type of RJS network, since processing times often vary in a system.

To demonstrate that simulation times depend heavily on at least two of those factors (specifically, context switching and distribution of experiments), various benchmarks of simulations with different models and network hierarchies have been run. The benchmark environment consists of three machines. The first machine, primarily used as a manager, is a QEMU virtual machine with two 2.5 GHz virtual CPU cores and 4 GB of virtual RAM; it runs Ubuntu 14.04 LTS and uses Linux kernel 3.13.0. The second machine, primarily used as a worker, is a physical machine with a four-core, 2.80 GHz Intel Core i7-930 CPU with hyperthreading and 9 GB of 1333 MHz DDR3 RAM; it runs Ubuntu 14.04 LTS and

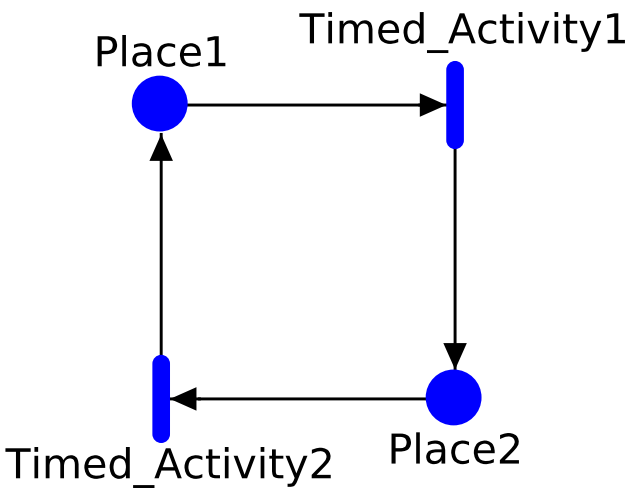


Figure 5.1: The simple example SAN model.

uses Linux kernel 3.13.0. The third machine, also primarily used as a worker, is a physical machine with a four-core, 3.06 GHz Intel Core i7-950 CPU with hyperthreading and 8 GB of 1333 MHz DDR3 RAM; it runs Ubuntu 14.04 LTS and uses Linux kernel 3.13.0. The first set of tests investigated a small simulation, and the second set investigated a large simulation.

5.1.1 Tested Models

To help us understand the relationship between delays in the RJS network and the size of the simulation, two models were used to measure the run time performance of RJS. The first model is a simple model that consists of only one SAN. The second model is a complex firewall model that consists of a composition of several SANs.

Simple Example Model

The first model consists of a single SAN. This simple example model, shown in Figure 5.1, consists of two places and two activities, connected in a ring. *Place1* initially consists of two tokens and feeds into *Timed_Activity1*, which has some arbitrary exponential firing rate. That activity feeds into *Place2*, initially empty, which feeds into *Timed_Activity2* with some arbitrary exponential firing rate that is not necessarily equal to *Timed_Activity1*'s firing rate. *Timed_Activity2* feeds into *Place1*. The range study for the project experiments involved ma-

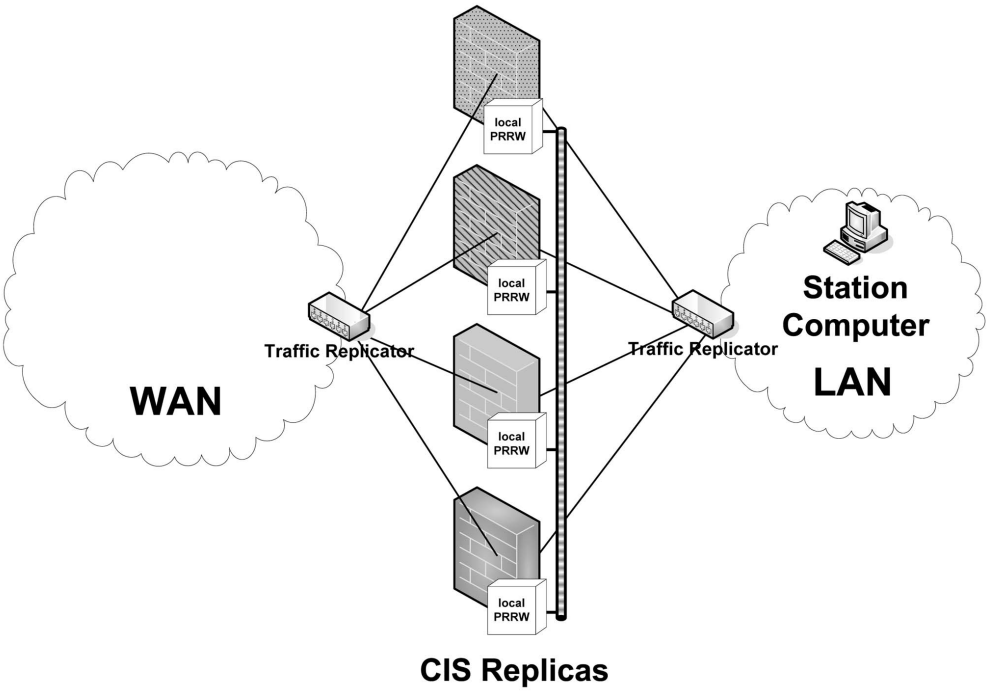


Figure 5.2: CIS architecture as shown in [13].

nipulation of the firing rates of *Timed_Activity1* and *Timed_Activity2*. *Timed_Activity1* was varied three times, and *Timed_Activity2* was varied twice, yielding a total of six experiments.

CIS Model

The second model is a model of a firewall system known as the *Critical Utility Infrastructural Resilience Information Switch* (*CIS*) [13]. Figure 5.2 shows the architecture of the CIS, which consists of multiple nodes that decide whether to accept or drop a packet. The model examines four different voting schemes as described in [14] and [15]: majority rule (MR), random troika (RT), clustering (Array), and hierarchical troika (HT). Figures 5.3, 5.4, and 5.5 show the decision flows for RT, Array, and HT, respectively. In the MR scheme, the decision value that received the greatest number of votes is chosen (with ties broken randomly). In the RT scheme, three nodes are randomly chosen, and the value that receives the greatest number of votes from the three nodes is used. In the Array scheme, groups of odd numbers of hosts are formed; the majority vote of each group is determined; and the

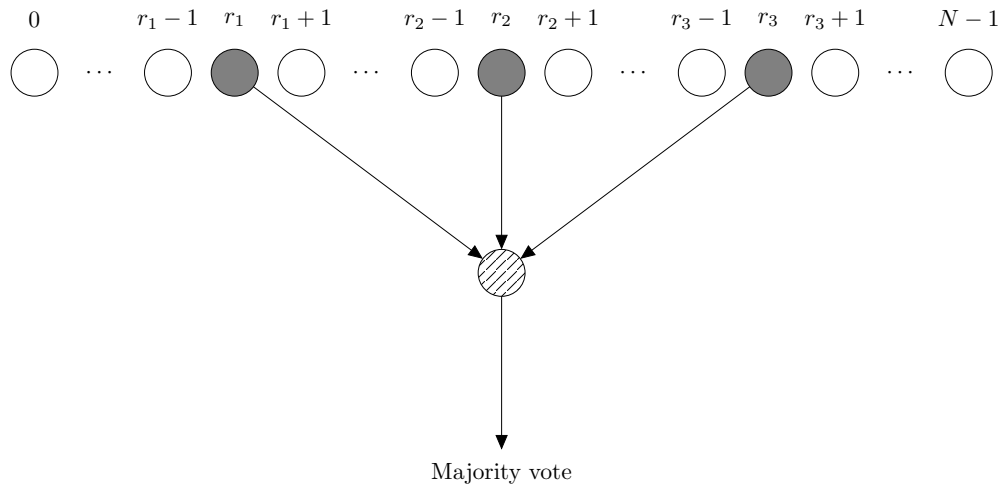


Figure 5.3: Random three-group decision flow.

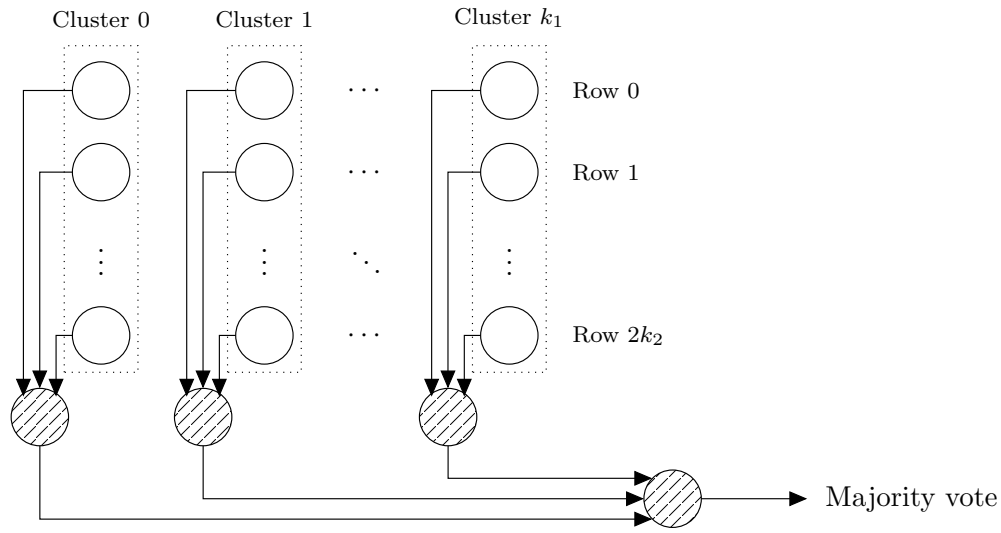


Figure 5.4: Array decision flow.

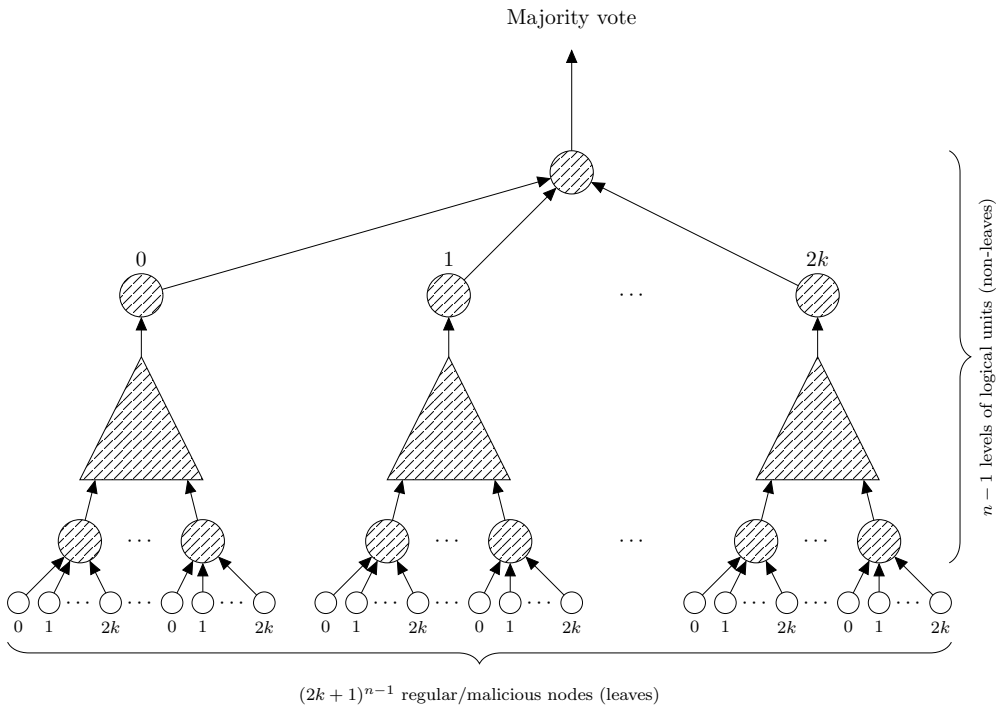


Figure 5.5: Hierarchical troika decision flow.

majority of the majority group votes is used to make the final decision of the cluster. In the HT scheme, $3^{\ell-1}$ hosts are split into groups of three, and the groups of three are recursively subdivided into groups of three until $\ell-1$ levels of decision units are formed. The authors of [15] also provided the motivation for the model, which is the tenacity of the voting scheme, which is a quantitative metric for the smallest percentage of nodes that an attacker must compromise to influence the decisions made by the system. Two fault models from [16], which deal directly with collusion from attackers, were implemented: M_1 and M_2. The first fault model, M_1, assumes that a node will sabotage the majority vote with some probability. The second fault model, M_2, always sabotages the majority vote.

The Möbius model for the CIS consists of four SAN submodels—*System*, *Node*, *Adversary*, and *Recovery*—which are composed into a single Rep/Join model. The *System* and *Node* submodels represent a system with replicated nodes, where the nodes receive workload units and perform voting to obtain an aggregate decision. The *Recovery* submodel detects errors in the system and recovers selected nodes, thus removing the nodes from the decision-making process. Once a node has been recovered, it is placed back into the system. Finally, the

Adversary submodel represents an active adversary that compromises nodes in the system, which causes them to behave in a faulty manner, according to M_1 or M_2.

The model supports two different recovery methods: proactive and reactive recovery. Proactive recovery occurs periodically, whereas reactive recovery occurs only on detection of an error. To test the CIS model, the proactive recovery project study was selected. It examines the effects of two variables: compromised node goal and consensus strategy. The compromised node goal is the percentage of nodes that an attacker is willing to compromise, and the consensus strategy is the voting scheme (i.e., MR, RT, Array, or HT) that is used to decide whether to accept or drop the packet. Three compromised node goal values were used in the project study, yielding a total of 12 experiments.

5.1.2 Network Hierarchy Setup

Several RJS hierarchies were tested on each model; they are shown in Table 5.1. The first test hierarchy is a simple network consisting of a manager and two workers. The virtual machine runs the manager, and the two physical machines run individual workers. Each of the two worker processes is allowed to use three CPU cores. The second test hierarchy is the same simple network used in the first test, except that the first physical machine (with the Core i7-930 CPU) runs a worker that is allowed to use four CPU cores, and the other machine (with the Core i7-950 CPU) runs a worker that is allowed to use two CPU cores. The third test hierarchy is nearly the same as the simple network used in the first test, except that a single forward node is placed between the manager node and each of the worker nodes. The forwards and workers run on "opposite" machines; that is, each forward run by one physical machine is connected to a worker run by the other physical machine.

5.2 Scalability

One important feature of RJS is the ability to scale to networks of computers with large numbers of cores and possibly supercomputer clusters. That is especially important for large simulations with multiple long-running project experiments. Figure 5.6 shows the topology

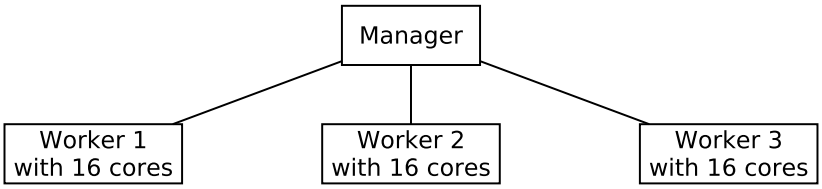


Figure 5.6: The topology used for testing scalability of RJS.

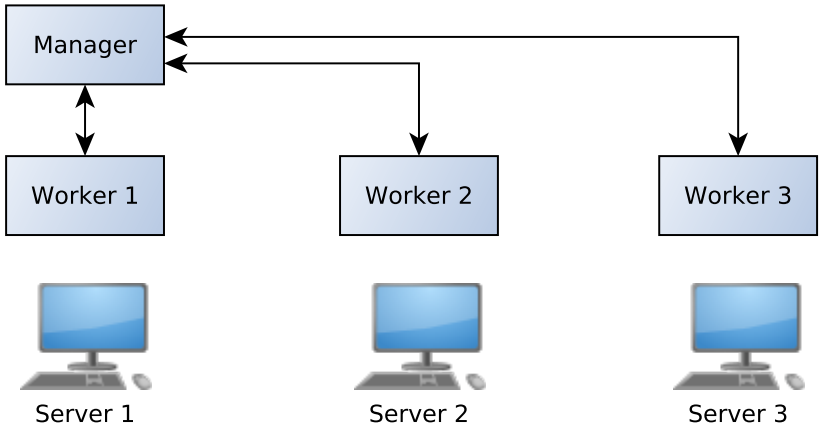
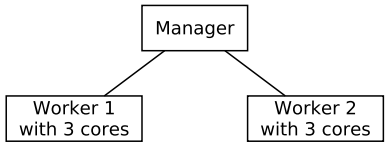
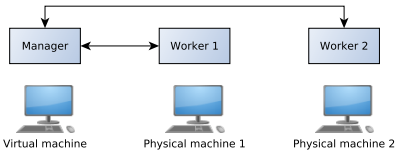
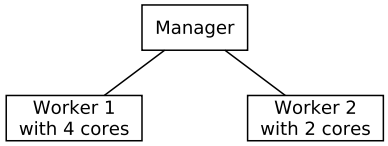
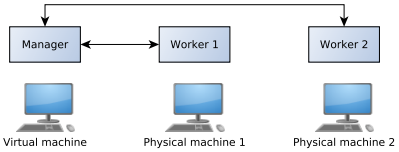
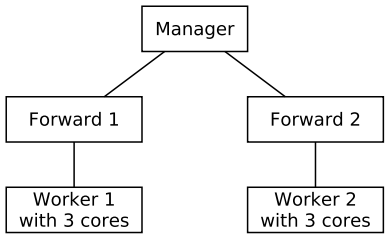
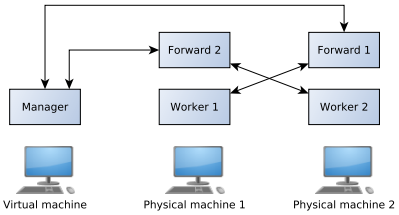


Figure 5.7: The mapping of nodes to machines for testing scalability of RJS.

for scalability testing, and Figure 5.7 shows the corresponding setup. RJS was tested on three Dell PowerEdge R520 servers used as workers. One of the three servers was also used as a manager. Each server consists of two eight-core, 2.50 GHz Intel Xeon E5-2450 v2 CPUs with hyperthreading and 64 GB of 1600 MHz DDR3 RAM, which runs Ubuntu 14.04 LTS and uses Linux kernel 3.13.0. The CIS model was chosen for the scalability test. One important aspect of the model is the attacker's strategy for compromising the nodes. The attacker can compromise the nodes one at a time, or the adversary can gain access to the system, wait for some time, and compromise all of the nodes at once. The consensus project study examined the effects of these two attacker strategies, in addition to examining the four different consensus strategies, the two different fault models, and three different compromised node goal values. In all, the consensus project study contained 48 experiments. As seen in Figure 5.6, the number of cores was split to divide the workload evenly among the four workers, so each worker was allowed to have up to 16 simulation processes. In addition to the consensus project study, the scalability test also included the reactive project study, which examined three different compromised node goals, the four different consensus strategies, the two fault models, and two different recovery strategies, also resulting in a total of 48 experiments.

Table 5.1: The RJS network hierarchy for each test. Each test is used for each model and for each round.

Test	Network hierarchy	Machine mapping
1		
2		
3		

CHAPTER 6

EXPERIMENTAL RESULTS

6.1 Effects of Network Hierarchy on Job Performance

6.1.1 Overall Results

Each test was run on each model repeatedly until all of the overall timings for each test fell within a 10% confidence interval corresponding to a 95% confidence level. The benchmarks required 11 rounds for the results to converge within the 10% confidence interval. Figure 6.1 shows the timing results for the simple example model, which show that Test 1 was the fastest, followed by Test 3. Based on the large confidence interval in Test 2, it is clear that the timing in Test 2 varied substantially, which indicates that at least one worker had run times that exhibited high variance across rounds. Figure 6.2 shows the timing results for the CIS model, which also show that Test 1 was the fastest, followed by Test 3. However, the timing across all tests varied substantially, and the time difference between Tests 2 and 3 was smaller for the CIS model than for the simple example model. That indicates that Test 3 contained a mixture of workers assigned to particular experiments, while in all other tests in all models, workers always received experiments in the same order. Nevertheless, the results indicate that the most basic RJS setup performed best, while more complex RJS setups performed marginally worse.

To gain a better understanding of the relationship between variance and confidence intervals, it suffices to look at the overall timings of each round of runs. Table 6.1 shows the run times in seconds for each test, for each model, for each round. For the simple example model, all of the timings for Tests 1 and 2 were generally consistent, but for Test 2, the timings varied substantially. The smallest run time, of 5.45 seconds, occurred in the

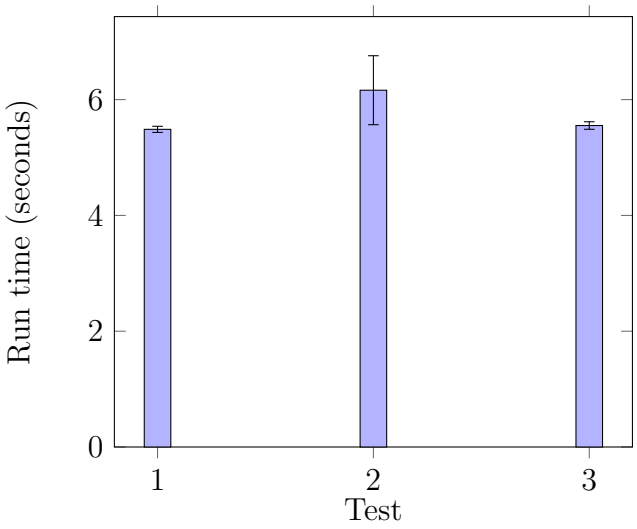


Figure 6.1: The timings of runs for each test, for the simple example model, averaged over the 11 rounds.

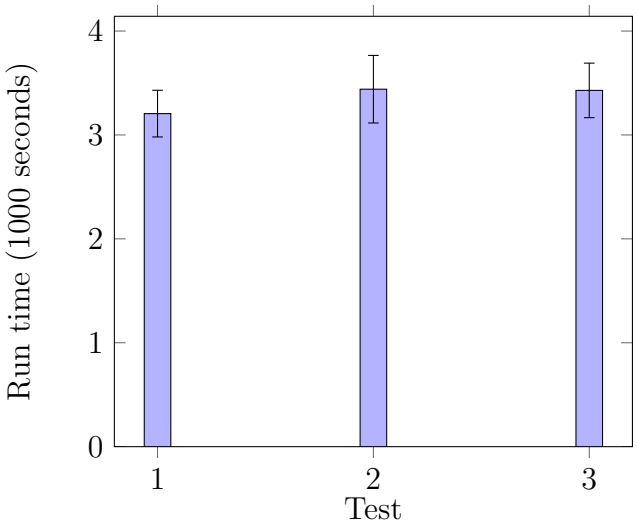


Figure 6.2: The timings of runs for each test, for the CIS model, averaged over the 11 rounds.

Table 6.1: The timings of runs for each test, for each model, for each round, in seconds.

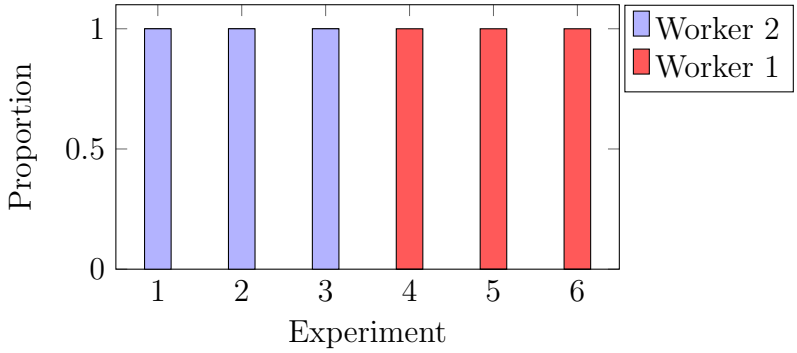
Round	Simple example model			CIS model		
	Test 1	Test 2	Test 3	Test 1	Test 2	Test 3
1	5.66	5.47	5.631	3672.93	3315.23	3629.47
2	5.44	7.051	5.63	3070.7	3008.48	3648.35
3	5.615	5.472	5.486	3486.54	3649.76	2919.93
4	5.436	5.45	5.443	3669.13	2964.17	3950.84
5	5.418	6.861	5.452	2929.32	2943.07	3672.48
6	5.494	5.402	5.465	2931.8	3147.46	3660.65
7	5.485	6.258	5.67	2993.39	4452.46	3010.33
8	5.474	8.048	5.423	2939.45	3507.56	3750.64
9	5.444	5.457	5.621	2978.21	4101.9	3601.6
10	5.448	5.668	5.617	2939.28	3201.73	2934.5
11	5.439	6.649	5.641	3649.94	3552.85	2938.73

fourth round, and the largest run time, of 8.048 seconds, occurred in the eighth round. For the CIS model, the timings varied substantially in all tests. The range of variation was $3672.93 - 2931.8 = 741.13$ seconds in Test 1, $4452.46 - 2964.17 = 1488.29$ seconds in Test 2, and $3950.84 - 2934.5 = 1016.34$ seconds in Test 3. Those results confirm that Test 2 had the largest variation for both models.

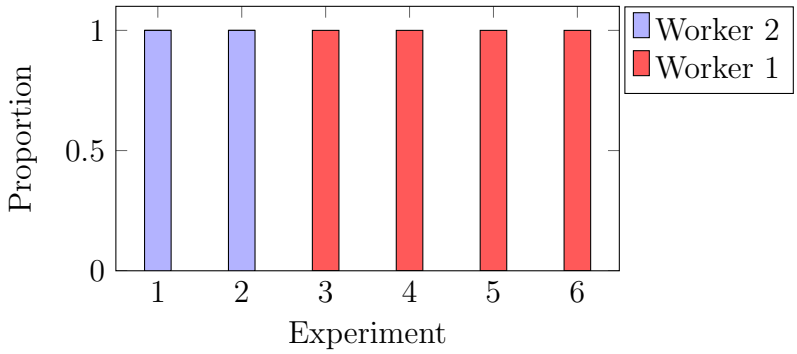
6.1.2 Results by Möbius Experiment

The run times for each experiment confirmed that specific workers and ordering of experiment assignments influenced the overall run times.

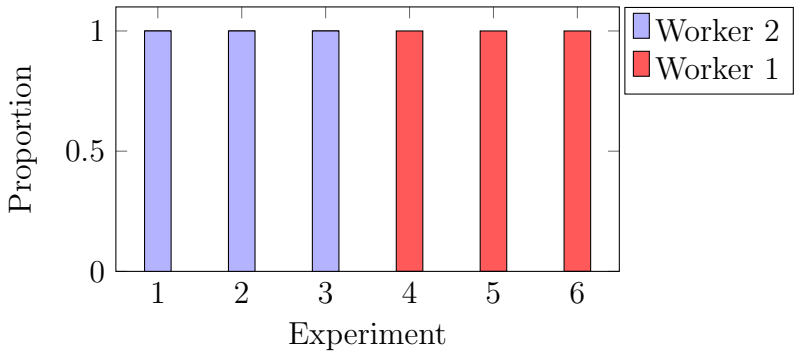
Figures 6.3 and 6.4 show the distribution of workers assigned to each experiment for both models. The y-axes represent the proportion of the total number of rounds that each worker was assigned to a particular experiment. For Tests 1 and 3 in the simple model, Worker 1 was assigned experiments 1, 2, and 3 for all runs, and Worker 2 was assigned experiments 4, 5, and 6 for all runs. For Test 2, Worker 1 was assigned experiments 1 and 2 for all runs, and Worker 2 was assigned experiments 3, 4, 5, and 6 for all runs. That demonstrates that the scheduling algorithm is greedy in that it tries to assign as many experiments as possible to any available worker if no other workers are available. The same



(a) Test 1.

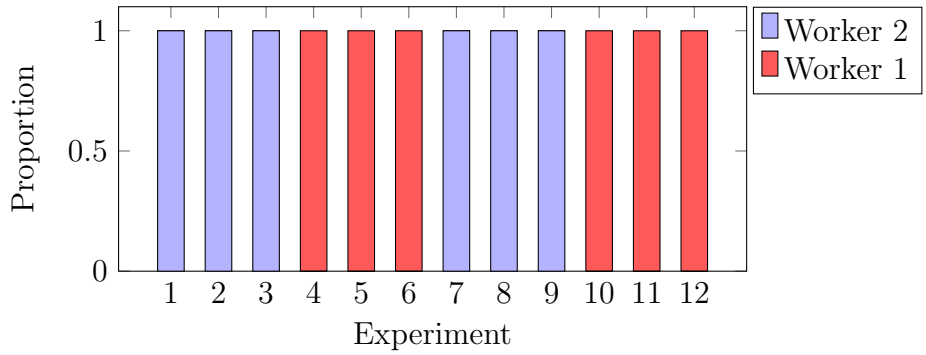


(b) Test 2.

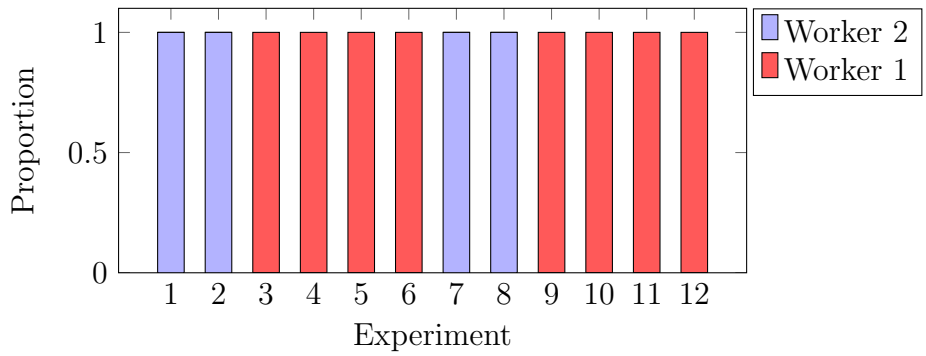


(c) Test 3.

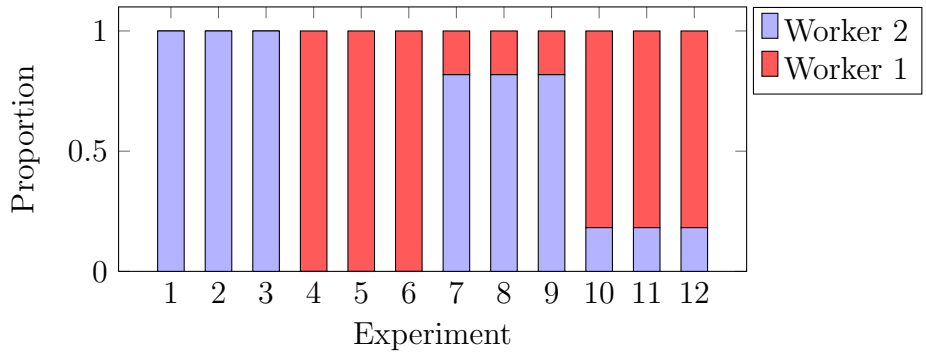
Figure 6.3: Distributions of workers assigned to experiments in the simple model.



(a) Test 1.



(b) Test 2.



(c) Test 3.

Figure 6.4: Distributions of workers assigned to experiments in the CIS model.

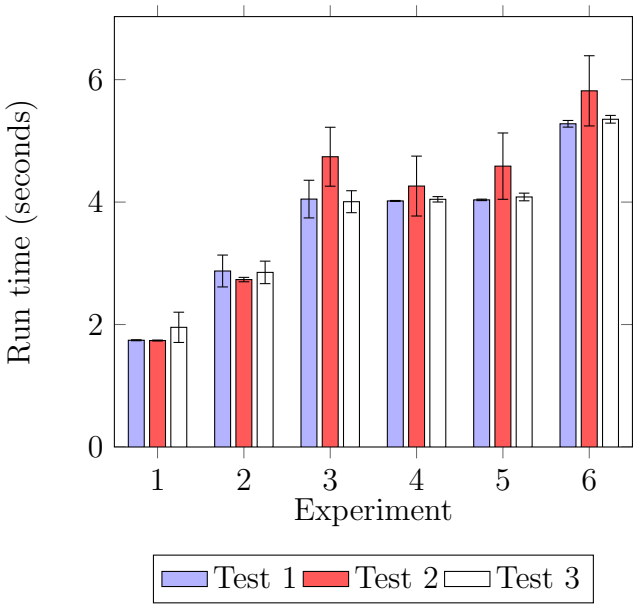


Figure 6.5: The timings of runs for each experiment, for each test, for the simple example model, averaged over the 11 rounds.

type of behavior was seen in Tests 1 and 2 in the CIS model. Notice that there were twice as many experiments as worker cores in the RJS network, which is why the alternating pattern exists in the distribution. In all tests, Worker 2 always started before Worker 1, but for Test 3, the largest proportion of experiments assigned to any particular worker was less than one for experiments 7 through 12. That indicates that the variation in run times caused the termination of experiments on Worker 1 to occur sooner than the termination of experiments on Worker 2. As a result, the ordering of experiment assignments varied in some rounds of runs.

Figure 6.5 shows the run times for each test, for each experiment, for the simple model. Test 2 had the smallest mean run times and confidence intervals for experiments 1 and 2, whereas Test 2 had the largest mean run times and confidence intervals for experiments 3, 4, 5, and 6. The reason is that experiments 3, 4, 5, and 6 ran on the first worker, which had four cores. The use of all four cores forced all background processes to use one of the cores. The variations indicate that the processor usage of background processes varied from time to time. In experiments 1 and 2, three cores were used, which indicates that the Linux operating system often migrated the background processes to run on the unused core. So

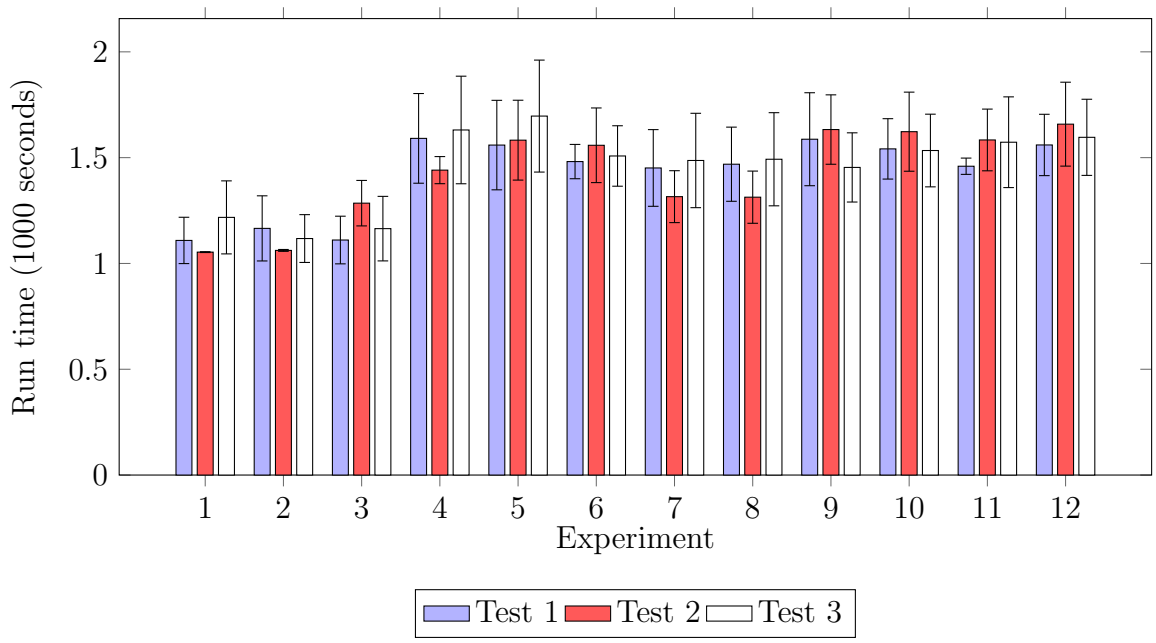


Figure 6.6: The timings of runs for each experiment, for each test, for the CIS model, averaged over the 11 rounds.

for small experiments, to minimize timing discrepancies on workers that have more than one physical CPU core, it is recommended that each worker be assigned at most $N - 1$ experiments, where N is the total number of physical processors available on the worker.

Figure 6.6 shows the run times for each test, for each experiment, for the CIS model. In general, nearly all experiment runs exhibited large amounts of variance in timings, which indicates that during the long simulation runs, either the operating system migrated the simulation processes among the different cores on each worker, or some background processes needed to use the CPU at the same time. However, Test 2 exhibited relatively small variance for experiments 1, 2, and 3, similar to what was seen in Test 2 for the simple example model. Experiments 7 through 12 were not always assigned to the same workers (according to Figure 6.4c), which explains the larger variance for experiments 7, 8, and 9.

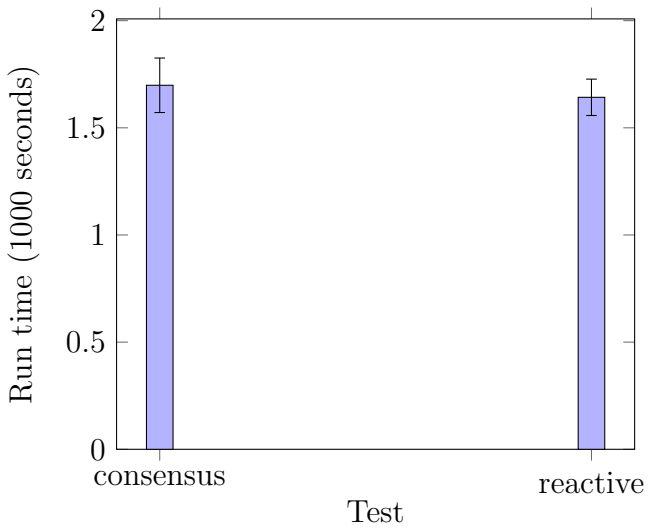


Figure 6.7: The timings of runs for the 48 consensus and reactive recovery experiments in the CIS model, averaged over the 11 rounds.

Table 6.2: The timings of runs for the 48 consensus and reactive recovery experiments in the CIS model in each round, in seconds.

Round	Consensus study	Reactive recovery study
1	1798.35	1690.92
2	1639.83	1574.15
3	1727.49	1626.44
4	1628.49	1677.98

6.2 Scalability

It took only four rounds of simulations until all of the overall timings fell within a 10% confidence interval, corresponding to a 95% confidence level. Figure 6.7 shows the overall timings for the consensus and reactive recovery experiments, and Table 6.2 shows the individual timings for each of the four simulation runs. In the previous set of tests, there was room for an average of three experiments to run on two workers, which meant that only half of all experiments could run at a time. That is, for the 12 experiments in the proactive recovery case, only six of them could run at a time. However, in the scalability test, there was enough room to run all 48 experiments at once, which resulted in substantial time savings. In theory, assuming that roughly all experiments take approximately the same time to run, that means

that it should have taken roughly half as long for all the experiments to complete, and the results in Figure 6.7 and Table 6.2 confirm that this is the case.

CHAPTER 7

CONCLUSION

It has been shown that RJS provides a service that enables flexible deployment and allows the user to make sound decisions on specific network configurations. It addresses the weaknesses that were found in related simulation systems such as *EcliPSe* and Akaroa2 by supporting multiple operating systems and providing a way for RJS to communicate across two different networks. With the differentiation of node types in a tree topology, RJS separates the role of the graphical user interface from the role of handling jobs. It was demonstrated that when a large number of experiments run on a worker, that tends to have a negative impact on overall performance, primarily because of increased context switching; further, the overall performance of any type of network can be negatively impacted by variations in the distribution of experiments. However, it was also shown that RJS scales well to computers with large numbers of CPU cores, which demonstrates its potential to run large numbers of experiments on large supercomputer clusters.

7.1 Future Work

While RJS provides a basic implementation that offers promising separation of the graphical user interface from the core Möbius application space, we can still identify features that would make Remote Job Server a more powerful tool.

7.1.1 Support for Functions Other than Simulation

One way RJS could be extended is by adding support for state-space generators, which are used to convert a model with associated performance variables into an analytical model, or

by adding other solvers, such as the analytical solver, which is used to compute the values of reward values associated with the analytical model. Currently, only the simulator is supported, and supporting the other tools would require modification of those tools.

7.1.2 Automatic Sharing of Worker Cores

Currently, the only way to handle the running of multiple workers on a single machine is to specify the number of cores that each worker uses manually. However, RJS could be modified to allow a group of workers on the same machine to run a consensus algorithm to decide how to split the cores when more than one of them needs to run a job. That would be useful because it would make RJS inherently more resilient to worker failures.

7.1.3 Persistent Mode

Another limitation of RJS is that it allows the execution of jobs only in a non-persistent fashion. That is, it allows network setup only for a single job, and the client may not disconnect from the network at any time. An alternative would be to provide a mode known as the *persistent mode*, which allows jobs to run while the client is disconnected and allows other clients to connect to the manager running as a service. RJS would offer two different login modes: the administrator mode and the user mode. Users could connect to the network and start jobs, while the administrators could modify the network topology. If a client disconnected while a job was running, the manager would need to store the results in a database until the client connected to it again.

7.1.4 Fault Tolerance

As it stands, RJS has not been designed to tolerate failures. One goal is to provide RJS with the capability of dispatching and running jobs even in the presence of faults (e.g., any one of the workers fails to compile the project, or a worker shuts down unexpectedly because of simulation startup failures). A heartbeat mechanism would be required that would allow the

manager to check the liveness of all the forwards and the workers in the tree. The manager could then respawn the appropriate nodes.

7.1.5 Security

RJS does not offer secure communication. That can be a problem in persistent mode when administrator and user logins are required, and it can be a problem in both the persistent and non-persistent modes because the integrity of network IDs is susceptible to compromise due to man-in-the-middle attacks. To prevent problems such as man-in-the-middle attacks, RJS needs security features so that all communications between nodes are encrypted.

REFERENCES

[1] "Möbius website," Mar. 2014. [Online]. Available: https://www.mobius.illinois.edu/

[2] D. D. Deavours, G. Clark, T. Courtney, D. Daly, S. Derisavi, J. M. Doyle, W. H. Sanders, and P. G. Webster, "The Möbius framework and its implementation," *IEEE Transactions on Software Engineering*, vol. 28, no. 10, pp. 956–969, Oct. 2002. [Online]. Available: http://dx.doi.org/10.1109/TSE.2002.1041052

[3] G. Clark, T. Courtney, D. Daly, D. Deavours, S. Derisavi, J. M. Doyle, W. H. Sanders, and P. Webster, "The Möbius modeling tool," in *Proceedings of the 9th International Workshop on Petri Nets and Performance Models, 2001*, 2001, pp. 241–250.

[4] T. Courtney, S. Derisavi, S. Gaonkar, M. Griffith, V. Lam, M. McQuinn, E. Rozier, and W. H. Sanders, "The Möbius modeling environment: Recent extensions –2005," in *Proceedings of the Second International Conference on the Quantitative Evaluation of Systems*, ser. QEST '05. Washington, DC, USA: IEEE Computer Society, 2005. [Online]. Available: http://dx.doi.org/10.1109/QEST.2005.39 pp. 259–260.

[5] R. M. Fujimoto, "Parallel discrete event simulation," *Communications of the ACM*, vol. 33, no. 10, pp. 30–53, Oct. 1990. [Online]. Available: http://doi.acm.org/10.1145/84537.84545

[6] E. G. Ulrich, V. D. Agrawal, and J. Arabian, *Concurrent and Comparative Discrete Event Simulation*. Norwell, MA, USA: Kluwer Academic Publishers, 1994.

[7] V. S. Sunderam and V. J. Rego, "EcliPSe: A system for high performance concurrent simulation," *Software Practice and Experience*, vol. 21, no. 11, pp. 1189–1219, Nov. 1991. [Online]. Available: http://dx.doi.org/10.1002/spe.4380211104

[8] V. J. Rego and V. S. Sunderam, "Experiments in concurrent stochastic simulation: The EcliPSe paradigm," *Journal of Parallel and Distributed Computing*, vol. 14, no. 1, pp. 66–84, Jan. 1992. [Online]. Available: http://dx.doi.org/10.1016/0743-7315(92)90098-8

[9] W. H. Sanders and J. F. Meyer, "Stochastic activity networks: Formal definitions and concepts," in *Lectures on Formal Methods and Performance Analysis*, ser. Lecture Notes in Computer Science, E. Brinksma, H. Hermanns, and J.-P. Katoen, Eds. Springer Berlin Heidelberg, 2001, vol. 2090, pp. 315–343. [Online]. Available: http://dx.doi.org/10.1007/3-540-44667-2_9

[10] G. C. Ewing, K. Pawlikowski, and D. McNickle, "Akaroa2: Exploiting network computing by distributing stochastic simulation," in *Proceedings of the 13th European Simulation Multi-Conference, ESM'99*, 1999, pp. 175–181.

[11] "Xerces-C++ XML parser," Apr. 2010. [Online]. Available: http://xerces.apache.org/xerces-c/

[12] R. G. Newcombe, "Two-sided confidence intervals for the single proportion: Comparison of seven methods," *Statistics in Medicine*, vol. 17, no. 8, pp. 857–872, Apr. 1998.

[13] P. Sousa, A. N. Bessani, M. Correia, N. F. Neves, and P. Verissimo, "Highly available intrusion-tolerant services with proactive-reactive recovery," *IEEE Transactions on Parallel and Distributed Systems*, vol. 21, no. 4, pp. 452–465, 2010.

[14] K. Kwiat, A. Taylor, W. Zwicker, D. Hill, S. Wetzonis, and S. Ren, "Analysis of binary voting algorithms for use in fault-tolerant and secure computing," in *Proceedings of the 2010 International Conference on Computer Engineering and Systems (ICCES)*, Nov. 2010, pp. 269–273.

[15] C. A. Kamhoua, K. A. Kwiat, and J. S. Park, "A binary vote based comparison of simple majority and hierarchical decision for survivable networks," in *Proceedings of the Second International Conference on Computer Science, Engineering & Applications (ICCSEA 2012)*, ser. Advances in Intelligent Systems and Computing, J. Kacprzyk, Ed. New Delhi, India: Springer Berlin Heidelberg, May 2012, vol. 2, pp. 883–896. [Online]. Available: http://dx.doi.org/10.1007/978-3-642-30111-7_85

[16] F. Araujo, J. Farinha, P. Domingues, G. C. Silaghi, and D. Kondo, "A maximum independent set approach for collusion detection in voting pools," *Journal of Parallel and Distributed Computing*, vol. 71, no. 10, pp. 1356–1366, 2011. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S0743731511001316