

**RELIABILITY AND STATE MACHINES IN AN
ADVANCED NETWORK TESTBED**

by

Mac G. Newbold

Printed 1/2/2004

A thesis submitted to the faculty of
The University of Utah
in partial fulfillment of the requirements for the degree of

Master of Science

in

Computer Science

School of Computing

The University of Utah

May 2004

ABSTRACT

Complex systems have many components that fail from time to time, adversely affecting the system's overall reliability. Emulab, an advanced network testbed, is one of these systems, and its design considerations include reliability, scalability, performance, and generality.

This thesis describes one method used to address these issues in Emulab, namely, the use of state machines. State machines, also known as Finite State Machines (FSMs) or Finite State Automata (FSAs), are well known, conceptually simple, and easy to understand and visualize. Our use of state machines contributes to reduced programmer mistakes, better software design and engineering, and provides a powerful and flexible model that can be used in many different and varied ways throughout the system.

State machines are used in Emulab to monitor and control booting testbed nodes, the allocation of testbed nodes to experiments, and the status of experiments as they progress through their life-cycle. In this thesis, each of these uses is described in detail to highlight the contribution of the state machine model to addressing the issues of reliability, scalability, performance, and generality in the testbed.

CONTENTS

ABSTRACT	i
LIST OF FIGURES	iv
CHAPTERS	
1. INTRODUCTION	1
2. ISSUES, CHALLENGES, AND AN APPROACH	3
2.1 Reliability	3
2.2 Scalability	4
2.3 Performance and Efficiency	5
2.4 Generality and Portability	6
2.5 State Machines in Emulab	6
3. EMULAB ARCHITECTURE	10
3.1 Software Architecture	10
3.2 Hardware Components	13
3.3 Central Database	14
3.4 User Interfaces	16
3.5 Access Control	18
3.6 Link Configuration and Control	19
4. STATE MACHINES IN EMULAB	22
4.1 State Machine Representation	22
4.2 How State Machines Interact	24
4.2.1 Direct Interaction	25
4.2.2 Indirect Interaction	26
4.3 Models of State Machine Control	27
4.3.1 Centralized	27
4.3.2 Distributed	29
4.4 Emulab's State Daemon	30
5. NODE CONFIGURATION PROCESS	33
5.1 Node Self-Configuration	33
5.2 The Node Boot Process	34
5.2.1 Variations of the Node Boot Process	35
5.3 The Node Reloading Process	39

6. EXPERIMENT CONFIGURATION PROCESS	42
6.1 Experiment Status State Machine	42
6.2 Node Allocation State Machine	43
7. THESIS SCHEDULE	46
REFERENCES	47

LIST OF FIGURES

2.1 A simple State Machine, or State Transition Diagram.	8
3.1 Emulab system architecture	10
3.2 Emulab Software Architecture	11
3.3 Emulab Hardware Components	13
3.4 A Link with Traffic Shaping	20
4.1 Interactions between three state machines	26
4.2 Three state machines viewed as one larger machine	32
5.1 The typical node boot state machine	34
5.2 WIDEAREA State Machine	36
5.3 MINIMAL State Machine	37
5.4 PCVM State Machine	38
5.5 ALWAYSUP State Machine	39
5.6 The node reloading process	40
6.1 Experiment Status State Machine	43
6.2 Node Allocation State Machine	44

CHAPTER 1

INTRODUCTION

Complex systems have many components that fail from time to time, adversely affecting the system's overall reliability. Emulab, an advanced network testbed, is one of these systems, and its design considerations include reliability, scalability, performance, and generality.

This thesis describes one method used to address these issues in Emulab, namely, the use of state machines. State machines, also known as Finite State Machines (FSMs) or Finite State Automata (FSAs), are well known, conceptually simple, and easy to understand and visualize. Our use of state machines contributes to reduced programmer mistakes, better software design and engineering, and provides a powerful and flexible model that can be used in many different and varied ways throughout the system.

State machines are used in Emulab to monitor and control booting testbed nodes, the allocation of testbed nodes to experiments, and the status of experiments as they progress through their life-cycle. In this thesis, each of these uses is described in detail to highlight the contribution of the state machine model to addressing the issues of reliability, scalability, performance, and generality in the testbed.

Reliability is directly affected by the scalability and performance of the testbed, and while generality does not affect reliability, it does place constraints on any approach that may be used to improve reliability, scalability or performance of the system. As the complexity of a system increases, there are more opportunities for failures, and if left unchecked, increasing complexity will cause reliability to decrease. Similarly, as a system increases in scale, there are more components that can generate failures, as well as increased opportunity for congestion and overload,

and performance tends to decrease as load on non-scaling services increases. In order to maintain or improve performance, increased concurrency and parallelism may be required, yet they also have a cost in terms of complexity and reliability. Because Emulab's generality goals, any method used to address these issues cannot decrease utility, portability, or place unnecessary limitations or constraints on the experiment workload that the testbed can support.

State machines are comprised of states, representing the condition of an entity, and transitions between states, which are associated with an event or action that causes a change in the condition of the entity. They provide several desirable qualities for addressing the four design issues described above. First, they are an explicit model of processes in the testbed, that contributes to error checking and verification. Second, they are generally well known and are simple to understand and visualize, which aids in software design and helps reduce programmer errors. Third, they provide a powerful and flexible model that can accommodate nearly any process, and a wide variety of implementation styles, including event-driven, process-driven, or component-based styles.

There are currently three primary areas in Emulab where a state machine model has been applied, including the node boot process, the node allocation process, and the experiment life-cycle and configuration process. In the node boot process, they contribute to increased reliability and performance by detecting and working to correct problems that may occur as nodes boot, and help deal with the pressures of increased scale. The flexibility of the state machines also allows for the efficient expression of differences between different boot processes while taking advantage of symmetry and similarities between them. During node allocation, a state machine provides control and promotes a modular programming style that is easy to understand. For experiments, state machines provide a simple way to monitor and control configuration changes, while contributing to good error-recovery practices.

CHAPTER 2

ISSUES, CHALLENGES, AND AN APPROACH

There are four primary challenges addressed in Emulab that are primarily relevant to this thesis. They are:

- Reliability
- Scalability
- Performance and Efficiency
- Generality and Portability

This thesis deals with one method used in Emulab to address these issues, namely, the use of state machines.

2.1 Reliability

As complexity of a system increases, the reliability of the system generally decreases, due in part to increased likelihood that at least one component of the complex system will have problems. Because they are complex devices, personal computers (PCs) inherit this unreliability. The vast array of hardware and software components, many of which are themselves very complex, fail from time to time, and often in ways that are unpredictable, and not infrequently, unpreventable.

In an advanced network testbed, there are many opportunities for things to fail, decreasing the testbed's reliability. In the case of a cluster of PCs, these factors include a variety of hardware, software, and network problems. In a cluster, the local-area network (LAN) is usually quite reliable, but like any network, does not

operate perfectly. Many network failures are transient, and can be resolved by retransmission. The PC nodes in the cluster are typically a more common source of reliability problems, many of which have a corresponding automatable recovery process. Information stored on a hard drive may become corrupt, or get changed or erased. Through automatic disk loading procedures, this can be remedied by recopying a disk image onto the node. A node that hangs and becomes unresponsive can be power cycled using remote power control hardware. Some recovery steps require manual intervention, such as the repairs required to replace bad hardware.

In the case of remote nodes that are connected to the testbed via the Internet or other wide-area networks, the situation is more complicated. The network can be a frequent source of problems, effectively disconnecting nodes from the server, and making communication with any other devices at the same location impossible. The nodes themselves are also at least as unreliable as nodes in a cluster would be, and often suffer from more frequent software problems due to the increased difficulty of reloading the hard drive of a remote node. Many remote nodes also have fewer recovery capabilities than the cluster nodes, for instance, they may lack power control hardware and serial consoles. Automated recovery for short network outages can be accomplished through retries. Long outages can be worked around by using a different node instead of the one that is unavailable. Far fewer options are available for recovery from nodes that are hung or cannot be reloaded with new software in the remote case, and bad hardware always requires human intervention.

Another principle that affects reliability is the observation that as the number of unreliable systems increases, the more likely it is that at least one of those systems will be in a failure mode at any given time. As a testbed increases in scale, it tends to become less reliable overall, which increases to the point where the majority of the time, some part of the system is broken.

2.2 Scalability

Almost everything the testbed provides is harder to provide at a larger scale. Supporting a larger testbed and more nodes requires more bandwidth from the

network, more horsepower on the servers, and so forth. As scale increases, the amount of concurrency between different events increases. In order to maintain the same level of service, throughput must be increased. Time to completion of any action will go up if our capacity to perform that action is not correspondingly increased. Either the testbed needs to do things faster, or everything will happen slower as scale increases.

Because of the increased load on the network, servers, and services as the testbed grows, reliability is adversely affected. Higher loads are much more likely to cause congestion or overload conditions in the various parts of the system, which will cause more failures, and sometimes failures of a different nature than were observed before.

2.3 Performance and Efficiency

One of the primary usage models for Emulab calls for users to be able to use the testbed in a very interactive style, starting experiment configuration tasks, then waiting briefly for the testbed to do its work. This means that configuration of experiments, changes to configurations, etc., all must happen in a timeframe on the order of a few minutes or less. This is the only direct performance requirement that is placed on our testbed design.

Indirectly, however, the testbed has much more stringent requirements on its performance and efficiency. The requirements that the testbed be reliable and scale well place high demands on the different parts of the testbed system. Together the performance and efficiency of the system determine in large part the practical limit on Emulab's scalability, and on the maximum useful utilization of resources that can be achieved. As the testbed has higher and higher demands placed on its resources and becomes increasingly busy, the time that elapses between one user giving up a resource and another user being able to acquire and use it becomes very important.

2.4 Generality and Portability

Another goal of Emulab is that it be general enough to be used for a very wide variety of research, teaching, development, and testing activities, as well as being useful and portable in a wide variety of cluster configurations. Everything we do must be designed to work as well as possible in as many different node and cluster configurations as possible.

An important aspect of generality is workload generality, namely that the design or implementation of the system should place as few limitations or constraints as possible on the experiment workload that the testbed can support. This helps ensure that the testbed is as generally applicable as possible, and allows for the widest range of uses. A good indicator of generality in the case of Emulab has been the things that Emulab is used for that were unforeseen by the testbed's design team. There is an important tradeoff between generality and optimization, and often the priorities involved can change over time.

In particular, Emulab must be able to support, at least minimally, a poorly instrumented or uninstrumented system. For example, a custom operating system image that was made by a user, which we cannot change or enhance, may not have any support for being used inside of Emulab. The testbed must be able to gracefully handle this situation, even though the functionality the OS can provide may not be the same as an OS image that has been customized for use within Emulab.

Any special support that Emulab provides or requires on nodes should be easy to port to other operating systems, so that as many systems as possible can take full advantage of the benefits the testbed provides. The Emulab-specific software for the nodes must also be unobtrusive, so that it does not interfere with any experiments that users may want to perform on the testbed.

2.5 State Machines in Emulab

In Emulab, one method used to address these issues and challenges is the use of state machines, also known as finite state machines or finite state automata. A state machine (as applied in Emulab) consists of a finite set of states and a set

of transitions (or edges) between those states. State machines can be graphically represented as directed graphs called state transition diagrams. States in a machine are identified by their unique label, and transitions are identified by their endpoints. Transitions are also often associated with a particular event or action that happens when the transition is taken. The associated event or action may be either a cause or an effect of the transition.

An illustration¹ of a state machine is shown in Figure 2.1. In this example, entities start in the NEW state, then must be verified and approved to enter the READY state. After becoming ready, they may be locked, unlocked, frozen and thawed to move between the FROZEN and LOCKED states. In the case that they need to be reapproved or reverified, they may reenter the UNAPPROVED, UNVERIFIED or NEW states.

There are several qualities of state machines that make them a very appropriate solution to the problems they address in Emulab. First, they are an explicit model of processes occurring in the testbed. Second, the finite state automata model is well known and easy to understand and visualize. Third, state machines provide a powerful and flexible model that can accommodate many implementation styles, including event-driven, process-driven, or state-based components.

An explicit model of testbed processes is valuable to the testbed system both in terms of reliability and in terms of software engineering. The explicit model that a state machine defines contributes to redundancy and allows for runtime checking to ensure correct operation or detect errors. A state machine also provides a degree of formality that allows software developers to reason about the software, and eventually, could aid in using formal verification methods to check or prove aspects of the testbed system.

¹Throughout this thesis the following convention is used for state machine diagrams: States are shown as ellipses, and are labeled with the state name. Transitions or edges are shown as directional arrows between states, and are labeled with the event that is associated with that transition. The majority of the state machine diagrams shown throughout this thesis are generated automatically from their descriptions in Emulab's database, and they are shown without any manual retouching.

The state machine model is also very powerful and flexible in terms of the implementations it can accommodate. One part of Emulab uses state machines in an event-driven model, where a central event handler uses the state and an incoming event to choose the right state transition and perform any associated actions. Another part of Emulab uses a process driven model to move through a series of states in a state machine as part of a process that includes more than one state transition. Another way to use state machines that is not yet implemented in Emulab is for a state-based component model, where each state and transition is associated with a component that performs any necessary actions and causes any appropriate transitions to occur. Such a model would have relatively simple components and a formal definition of their composition, and could be amenable to a form of automated verification of the source code.

CHAPTER 3

EMULAB ARCHITECTURE

Emulab is an advanced network testbed, and falls into the class of testbeds known as Multi-user Experimental Facilities, as defined by the NSF Workshop on Network Research Testbeds[6]. Figure 3.1 roughly outlines the major components of the Emulab system. More detailed information about Emulab is available in several conference publications and technical reports, including [2, 11, 12, 13, 14], and others listed on the Emulab web interface at www.emulab.net on the Internet.

3.1 Software Architecture

Figure 3.2 depicts Emulab’s software architecture. Actions are initiated by users or testbed administrators, shown at the top of the diagram, through one of Emulab’s user interfaces. The primary interface is the web site, www.emulab.net, where users and administrators can log in to view and change the current state of the Emulab system. Secondary interfaces include *ns* scripts and a Java graphical

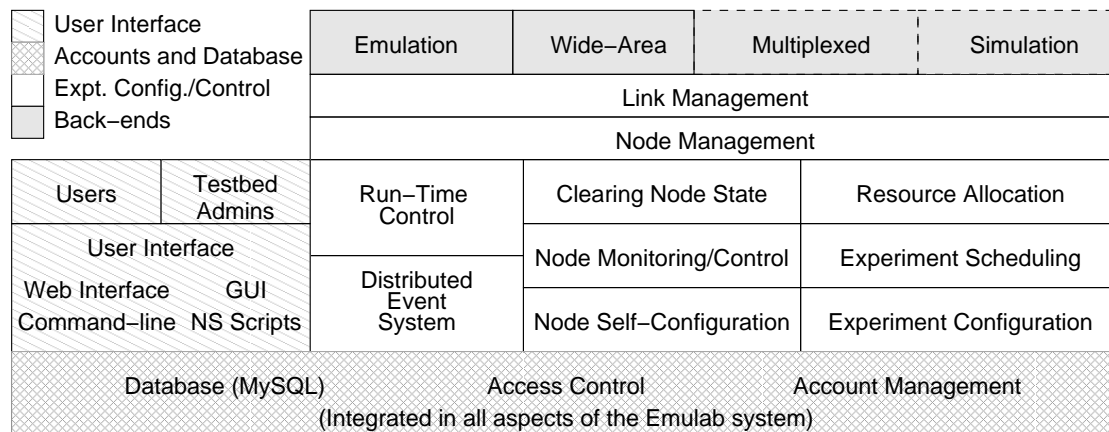


Figure 3.1. Emulab system architecture

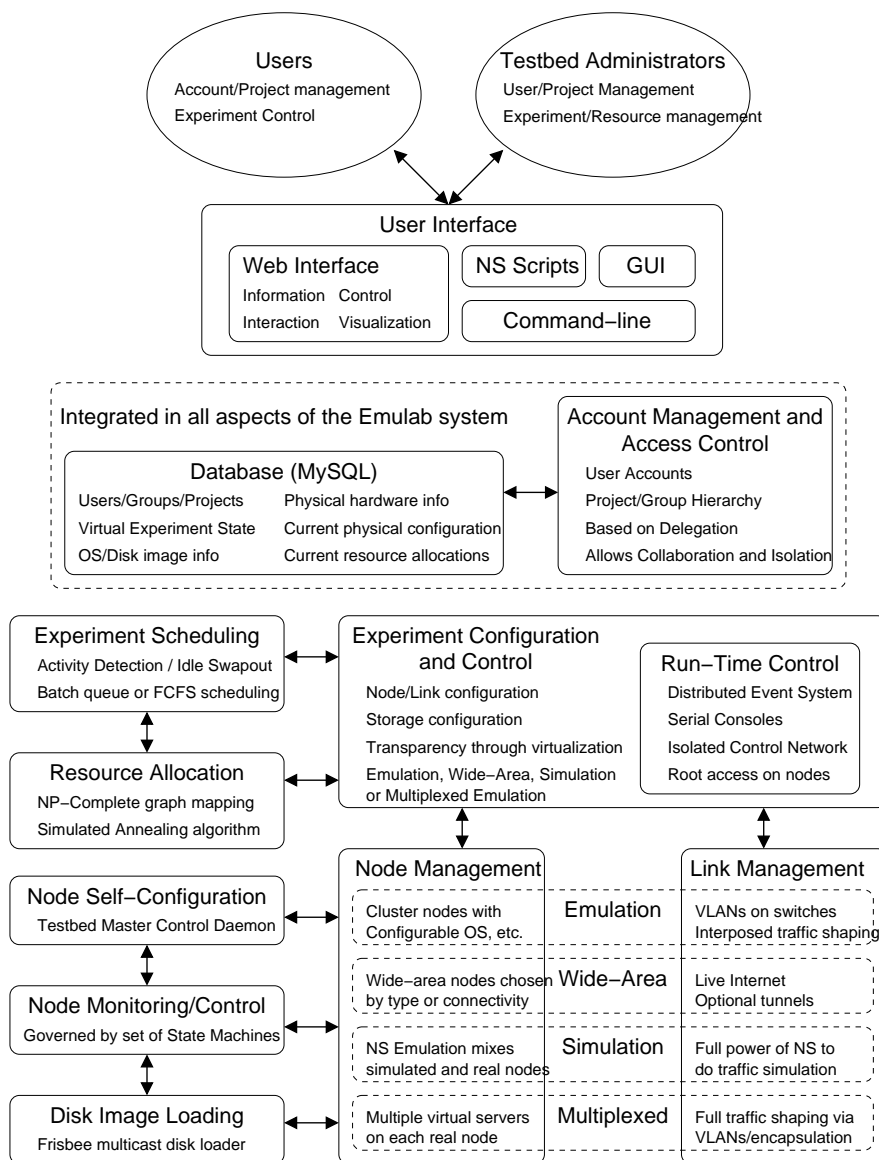


Figure 3.2. Emulab Software Architecture

user interface (GUI) that are used for submitting experiment descriptions, and a command-line interface to many of Emulab’s features. There are also various programmatic interfaces and APIs (not shown) that advanced users may choose to use. The user interfaces operate on Emulab’s central database, populating it with information provided by the user, and retrieving information requested by the user, as well as dispatching actions the users requests.

At the center of the Emulab software is a relational database. It stores information about nearly every aspect of Emulab’s operation. The database is not directly accessible to users, and access to the information it contains is governed by the account management and access control components of the architecture. They limit a user’s visibility to those projects, experiments, and features for which access has been granted. This component functions both in an isolation role as well as a collaboration role.

The core components of Emulab’s software are very closely linked with one another, but can be divided into several large categories. Experiment scheduling determines when an experiment can be instantiated (“swapped in”) on physical resources, and when an experiment should be “swapped out”. The resource allocation component determines if and how an experiment can be instantiated, given the currently available resources in the system. These two components work together with the experiment configuration and control subsystems, which perform the actual work of setting up the hardware and software in the requested configuration for carrying out the experiment. Various methods for run-time control allow the user to interact with the experiment very flexibly.

During experiment configuration and control, the node management and link management subsystems come into use. They configure the individual nodes and links required for the experiment. These two components operate directly on the hardware with one of several “back-end” implementations, similar to a “device driver” for different implementations of nodes and links. The typical cluster model is emulation, where nodes are physical PC nodes and links are made with VLANs (Virtual LANs) on switches and interposed traffic shaping to control latency, bandwidth, and loss rate. The wide-area model uses nodes from around the world as endpoints, and the live internet (and optionally, overlay tunnels) as the links for the experiment’s topology. The simulation model runs instances of the *ns* Emulator, *nse*, where simulated nodes and links can interact with the real network and other types of nodes and links. The multiplexed model uses physical nodes as hosts for

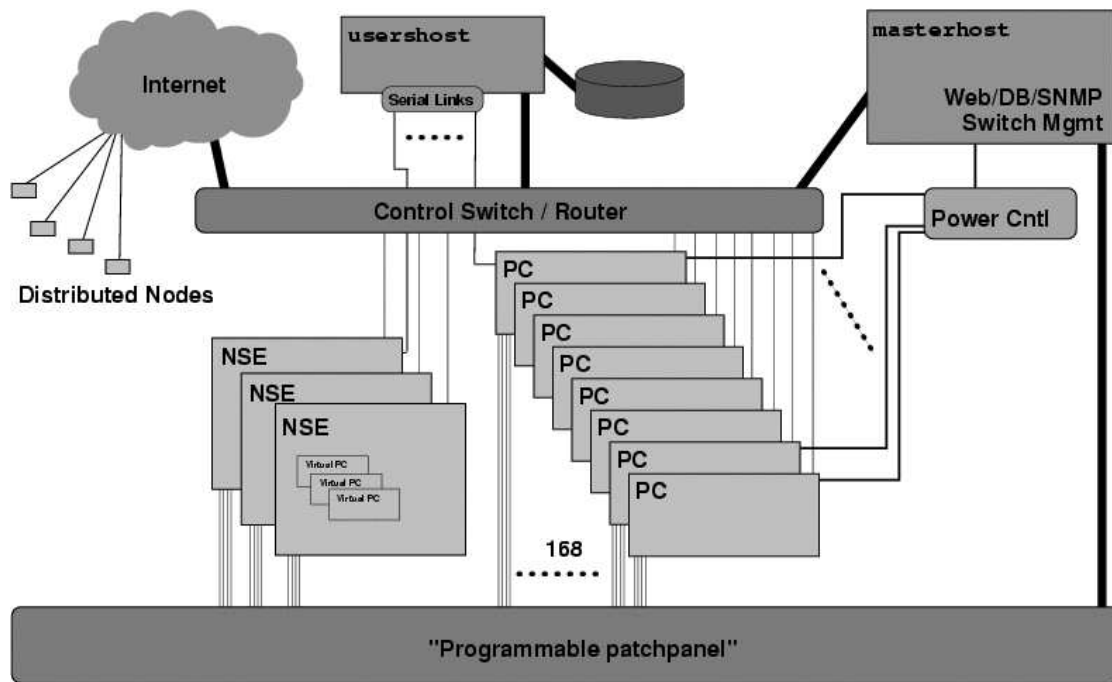


Figure 3.3. Emulab Hardware Components

virtual nodes that are multiplexed onto the physical nodes, and made to appear and behave as near to the behavior of the real physical nodes as possible.

The node management component also works with three other subsystems that help carry out the various node management tasks.

3.2 Hardware Components

Emulab is composed of a wide variety of hardware components, as shown in Figure 3.3. Central to the architecture are the Emulab servers, `usershost` and `masterhost`. `usershost` serves as the file server for the nodes, one of several serial console servers (not shown), and as an interactive server for Emulab users. `masterhost` is the secure server that runs critical parts of the infrastructure, including the web server and the database server, and securely accesses power controllers and switches.

The network infrastructure for Emulab is composed of a control switch/router that serves as gateway and firewall for Emulab, and a set of switches that form the

backplane of the experimental network. The experimental network switches serve as a “programmable patch panel” to dynamically configure the network connections that users need in order to perform their experiments.

The Emulab cluster is currently composed of 168 PentiumIII-class PCs, standard rack-mount servers, configured with 5 network cards each. One network interface card in each node connects to the control router, and the other four connect to the experimental network switches. The nodes also are connected to power controllers, to allow remote power cycling for unresponsive nodes, and have serial console access available for kernel debugging and an interaction path separate from the control network. These nodes can serve directly as physical nodes in experiments, or can host a variety of non-physical node instantiations: “simulated nodes” inside an special instance of the *ns* Network Simulator, “multiplexed nodes” (not shown) implemented via FreeBSD’s `jail`[5] or Linux `vservers`[1]. They can also be used to host special hardware devices like IXP Network Processors[3] (not shown) and provide remote access and other tools for the devices.

Emulab’s control router connects the cluster to the Internet and Internet2[4] and to the Emulab resources available through the wide-area network¹. Emulab provides access to shared machines located around the world that have been dedicated to experimental use. These machines run either FreeBSD or Linux, and can be used in most of the ways that the local cluster nodes can be used. Necessarily there are some differences in security, isolation, sharing, and other aspects. These nodes allow experimenters to use the live Internet to perform measurements or run tests under more realistic conditions than the cluster can offer.

3.3 Central Database

At the core of the Emulab system is a relational database. It brings the full power of relational databases to bear on the information management issues inher-

¹Sometimes the local Emulab cluster is referred to as “Emulab Classic” or simply “Emulab”, while the entire system including the wide-area resources, simulated nodes, and other features is called “Netbed”. For the purposes of this thesis, Emulab will be used to refer to the entire Emulab/Netbed system as well as the local cluster installation unless otherwise noted.

ent in managing a large-scale multi-user testbed. It also functions as a repository for persistent data, and frequently, as a communication channel between different parts of the system. Some of the key information that is stored in the database is described below.

Users, Groups, and Projects: Users can be members of groups, and groups belong to projects. Each user may be a member of many groups and projects, and may have a different level of trust in each of those groups. This information is used for access control for data that may be viewed or changed on the web site, and for any other actions the user may wish to perform in Emulab.

“Virtual” experiment state: When a user loads an experiment configuration into Emulab, either from an *ns* file or through our graphical interface, the information needed to configure the experiment is loaded into the database as abstract “virtual” experiment data that is not tied to any particular hardware. It is preserved while an experiment is inactive for use later to reinstantiate the experiment on physical hardware again.

OS/Disk image information: The database also stores information about each operating system and disk image that is used in Emulab, including the owning user, group and project, operating system features, disk image contents, etc. This information is used by Emulab to determine what image to load when a given OS is requested, as well as what features the OS supports that may be used by Emulab in order to monitor or control nodes using the OS.

Physical hardware configuration: In order to configure experiments, all hardware that is being managed by Emulab or is used in its management is stored in the database. This includes information about nodes, switches, power controllers, and every network cable in the testbed cluster.

Current physical configurations and allocations: Emulab uses the database to allocate nodes and other resources to different experiments that may be running at any given time. Experiments are mapped to currently available

physical hardware, and experiment-specific configuration information is prepared for use by the system in configuring the nodes. Much of this data is later downloaded by the nodes for use in their self-configuration process.

The database is a critical part for Emulab's operation, and is integrated into almost every aspect of the system. Because so much depends on the integrity of the information it contains, the database is served on Emulab's secure server, **masterhost**, and direct access to it is not permitted to users. All changes that a user makes to information in the database are done indirectly through interfaces, like the web pages, that perform access checks as well as validity checks on all submitted data.

3.4 User Interfaces

Emulab provides a wide variety of interfaces to its functionality to meet the widely varied needs of its users. These interfaces include human-centered as well as programmatic methods of control that provide different levels of expressive power, flexibility, convenience, and automation to the user or the user's programs and scripts. These interfaces include a web interface, *ns* scripts, a Graphical User Interface (GUI), command line tools, and other programmatic interfaces.

Web Interface

Emulab's web site is the primary place for viewing and changing testbed data and configurations. It is the principal method for interacting with the system for experiment creation and control. It serves as a front end to the database, both for reading and writing. It works hand in hand with the interfaces provided by *ns* scripts and the GUI, and provides visualization tools to experimenters. It also serves as the central repository and communication method for Emulab users and the general public, providing documentation, publications, software downloads, etc.

ns scripts

Specialized scripts written in TCL and based off of the system used by the *ns*

Network Simulator[10] serve as a primary method for describing experiment configurations. They may be passed to the Emulab system through the web interface or the command line interface. These scripts describe nodes, links, and their respective configuration information, as well as events that should be automatically carried out at specified times during the experiment. In Emulab, standard *ns* syntax has been extended to provide additional data and functionality, including selection of node hardware types, operating systems, software to be installed, and programs to be executed. A compatibility library is also provided to allow these Emulab *ns* scripts to be used unmodified with the *ns* simulator.

Graphical User Interface (GUI)

Because many users of Emulab do not have a background that includes familiarity with *ns* scripts, we also provide a graphical user interface in the form of a portable java applet that is downloaded and executed through our web interface. It provides a simple method to draw a topology of nodes and links, then configure their characteristics, without ever needing to write or read any *ns* files.

Command Line

The command line (i.e. Unix shell) is the most popular way to interact with the nodes in Emulab, but is the least popular way to interact with the main body of the testbed software. We have found that the command line is typically used mostly by programs or scripts written by users, and that users usually favor using the web interface for activities other than direct interaction with an individual node. The command line interface is a critical part of the automatability that Emulab provides, allowing users to fully automate their experiments, from creation to termination.

Part of the command line interface consists of a set of command line tools that are available on the nodes themselves. Others are run on one of the Emulab servers (`usershost`) where users are given full shell access. Tools on the

nodes include a variety of scripts to gather Emulab-specific node configuration information and to provide runtime control for the experiment. Operations that can be performed on the server include rebooting nodes, changing link parameters, and starting and ending experiments.

Programmatic Interfaces (APIs)

Many of Emulab's systems are designed to be easily accessible programmatically for users who desire a higher level of automation in their experiments, or who need to take advantage of special features we provide. In particular, users can interact directly with the distributed event system Emulab uses for control and communication. Users can also directly access `tmcd` servers hosted on the `masterhost` server, which provides controlled access to certain database constructs and information used for node self-configuration.

3.5 Access Control

Permissions and access control in Emulab are based on the principles of hierarchy and delegation, and there are three primary entities involved in the permission system: Users, Groups, and Projects. A user may sign up for an individual account by providing an email address and some personal information. For security, the email address is verified to belong to the user before the account is activated. These individual accounts provide strong accountability, which is critical in a system like Emulab that places a lot of trust in its users and gives them many privileges that they otherwise would likely not have, like “root” system administrator access on nodes in the testbed.

A user's account is not fully active until it has also been approved by a responsible party who knows them personally, who will be in part accountable for their actions on Emulab. This may be an advisor, professor, principal investigator, or senior researcher who has started a project on Emulab which the user applies to join. Qualifying researchers and instructors may apply to create a new project, and are approved directly by Emulab's approval committee. Project heads are delegated authority to approve users in their project, granting them access to the testbed's

resources. The project head decides what level of trust to place in the user, including levels representing normal Unix user-level access, root-level access on nodes in the project, and root-level node access with administrative privileges within the project. These administrative privileges allow the project head to delegate to that user the ability to approve other users in the project.

By default, each project consists of exactly one group, and all the project members belong to that group. When desired, other groups may be configured within the project that are made up of subsets of the project members. This configuration allows for easy collaboration within a project or group, while allowing for isolation between different projects and between different groups inside a project. This provides a good environment for groups that need to share things, yet still maintain a set of data that is private to a single group within the project. One place where this is particularly useful is a class that is using Emulab, because the instructor can provide a set of resources to the whole class, while grouping students in other groups where their work is not accessible to other student groups. It also is being used by a DARPA-funded research program to provide sharing and isolation for the different groups funded under the program.

Throughout the Emulab system, including web pages, nodes, and command line tools, access controls are enforced that ensure that only authorized users have access to private information about a project, group, user, or any data belonging to them, including experiment configurations, OS images, and membership information.

3.6 Link Configuration and Control

Emulab's current cluster hardware includes only 100Mbps LAN links for experimental use. But it can artificially slow down, constrain, or hamper the network connection in a way that lets it accurately and precisely emulate wide-area connections.

When a user specifies a link other than 100Mbps bandwidth with 0ms latency and no fixed packet loss rate, the system determines that the link needs traffic shaping. Normally, this traffic shaping is done by a node that is transparently

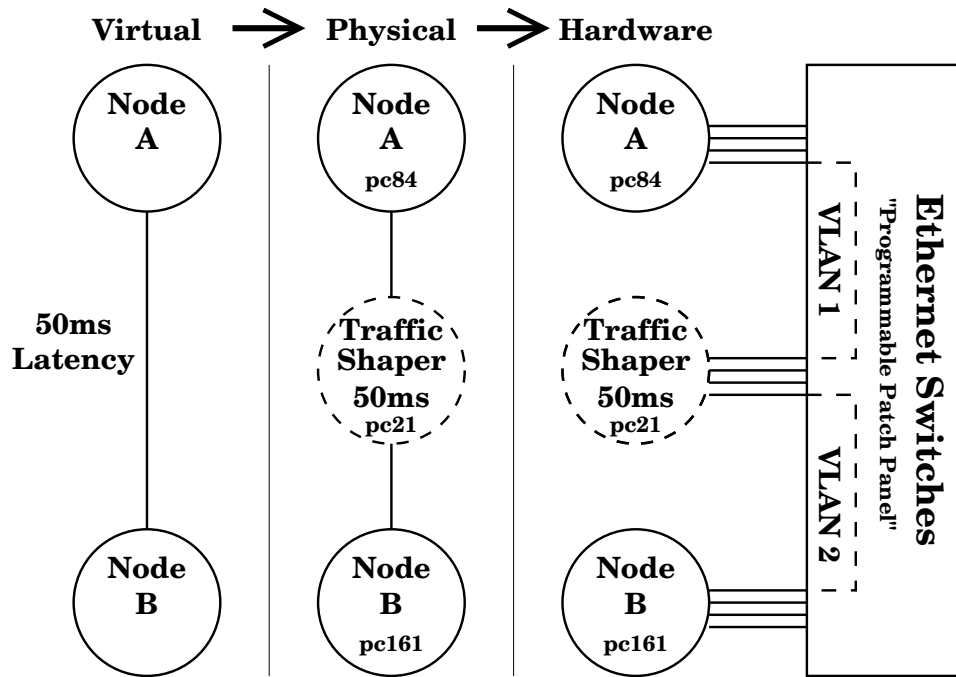


Figure 3.4. A Link with Traffic Shaping

interposed on the link to do the shaping, using FreeBSD and its Dummynet[7, 8, 9] features. A sample of this interposition is shown in Figure 3.4. Alternatively, Emulab also supports traffic shaping by the end-nodes themselves².

When using remote wide-area nodes instead of cluster PCs, typically extra link shaping is not desirable. In most cases, researchers want to see the raw characteristics of the live internet path between the nodes they are using. Emulab also optionally provides overlay tunnels between the nodes, using private IP addresses, so to increase the amount of transparency between cluster and wide-area nodes and ease in transitioning between node types.

Using Emulab's simulation features based on *nse* network emulation, traffic shaping can be controlled using the full power of *ns*. Packets can be subjected to simulated cross-traffic as they pass through the simulated network, and the different statistical models that are available in *nscan* be used for traffic shaping. Packets

²Currently end-node traffic shaping is supported only on FreeBSD nodes, using a kernel with Dummynet support. Support for Linux is currently being developed.

can also be routed through large networks simulated in *ns* by a single PC without using a physical PC to emulate every node in the network.

Emulab's support for multiplexed "virtual" nodes uses end-node shaping techniques in combination with the node multiplexing, by doing traffic shaping on the physical host node for links to or from any "virtual" nodes being hosted on that PC.

CHAPTER 4

STATE MACHINES IN EMULAB

This chapter discusses the uses and implementation of state machines in Emulab. The rationale for state machines and the problems they help address are discussed in Section 2.5 on page 6.

4.1 State Machine Representation

Conceptually, state machines are directed graphs, with nodes in the graph representing states, and unidirectional edges representing transitions from one state into another. In the purest sense, this directed graph is all that is required for a state machine; however, within Emulab, some additional conventions are used. Each state machine is labeled with a name, and each state in a machine is given a unique identifier as the name of the state. Transitions are identified primarily by their two end points, as no two transitions may have the same endpoints in the same order, and optionally have a label that describes the event associated with it by being either the cause or the effect of the transition.

In contrast to most systems, which typically use a single state machine to describe a process, our use of state machines allows for multiple state machines that govern the same entity, and for transitions¹ between the different machines. We have added to this the concept that state machines have an inherent type, and within that type, are mutually exclusive, meaning that an entity (e.g. a node) is in one state in exactly one machine of a given type at any time. The different

¹This method is somewhat related to “scenarios” commonly used with Message Sequence Charts and other Labeled Transition Systems. A set of scenarios are linked together with transitions, and each scenario internally is represented by a message sequence chart. In our system, each state machine is independent, but may have transitions that link a state to a state in a different machine.

state machines within a type are also sometimes called “modes”, and moving from a state in one machine into a state in another machine is sometimes called a “mode transition”, as opposed to normal “state transitions” between two states in the same machine.

While the state machines within a mode could be expressed by a single, large, state machine, using an abstraction that breaks the system down into smaller state machines can make it easier to understand, and can improve our ability to take advantage of similarities between the different machines. Examples of a set of small state machines and their equivalent representation as a single machine are shown, respectively, in Figure 4.1, on page 26, and Figure 4.2, on page 32.

Another unique aspect of state machines as implemented in Emulab is that states may have a “timeout”² associated with them. This timeout establishes a length of time during which it is acceptable to remain in that state. If an entity remains in the state beyond the time limit, a timeout occurs, and the configured “timeout action” is taken. This is typically used for error reporting and recovery.

We also allow actions to be attached to states, such that when an entity enters that state, the configured action, if any, is performed for that entity. These state actions are called “triggers” because of the way they are triggered by entry into the state. They provide a modular and extensible way to attach special functionality to certain states. In practice they are often used for maintaining certain invariants or performing tasks related to a particular state. They are especially useful when an action is associated with a state without being associated to any particular transition into the state. For instance, every time a node finishes its boot cycle, some unique tasks must be performed, without regard to how it arrived in that state. These triggers provide a hook for such tasks that do not depend unnecessarily on how the transition was caused, and can be easily reused for multiple states, and for states in other modes.

²Others have defined “timed automata”, which are state machines that have the concept of a global clock that advances as they execute, which allows reasoning about total time elapsed at any point in execution. In our model, there is simply a per-entity timer that starts when the entity enters a state.

Generally, triggers are set on a per-state basis and are meant to be used with all nodes that arrive in that state, but triggers can also be set for specific nodes. This is used, for example, to delay actions until a particular state transition has occurred, even though the decision to carry out that action was made in an earlier state. As an example, triggers are used when nodes finish reloading their hard drives to cause the node to be placed into the free pool the next time it finishes booting, as well as to cause special handling for determining when a node has finished booting when a node is using an OS image that does not send that notification event.

In the Emulab database, a state machine is represented as a list of transitions. Each transition is a tuple of mode, initial state, final state, and the label for the transition. Transitions between states in different machines (“mode transitions”) are similar, but include both a mode for the initial state and a mode for the final state. Each state may also have a state timeout entry, which lists mode, state, timeout (an integer representing seconds), and a string describing the list of actions to be performed when a timeout occurs (e.g. requesting a notification or a node reboot). Trigger entries consist of node ID, mode, state, and a string describing the trigger actions. Entries may be inserted dynamically for specific nodes. Static entries in the trigger list typically specify the “wildcard” node ID, indicating that the trigger should be executed any time a node reaches that state.

An example of the notation for representing state machines on paper has been shown and explained in Figure 2.1 on page 8. The notation for transitions between modes is shown in Figure 4.1, on page 26.

4.2 How State Machines Interact

As described earlier, state machines in Emulab have an inherent type, and there are currently three types of machines in the system. The first type are node boot state machines, which describe the process that a node follows as it cycles between rebooting and being ready for use, and there are several machines of this type. The second type consists of the node allocation state machine, which manages nodes as they cycle through being allocated, deallocated, and prepared for use in

an experiment. The third type is composed of the experiment state machine, that describes the life-cycle experiments follow as they are created, swapped in or out, modified, or terminated.

Two types of interactions occur between state machines, namely direct and indirect. Direct interaction can only occur between machines of the same type, and used with the node boot state machines. Indirect interaction occurs between machines of different types, due to the relationships between the entities traversing the different machines, and the states of each of the entities in the machines. These interactions occur between all three types of state machines in Emulab.

4.2.1 Direct Interaction

Machines of the same type interact with each directly, because an entity (i.e. a node) can follow a transition from a state in one machine to a state in another machine. Using separate state machines better preserves the simplicity of and symmetry between the individual state machines, while still retaining all the power that could be provided by modeling the set of machines as a single, larger state machine. Figure 4.1 (on page 26) and Figure 4.2 (on page 32) show the same set of states and transitions, both as three separate machines and as a single larger machine, respectively.

This method of changing state machines (also known as “modes”) is used frequently with the node boot state machines. Different operating systems or OS image versions may follow different patterns during their boot process, making it necessary to have a state machine that describes the proper operation of that boot process. When a node changes which operating system or disk image it is running, it may need to change to a different state machine to match the new OS.

For example, whenever a node reloads its hard drive with a new image, it reboots into a special disk reloading operating system image that is loaded over the network. This OS image follows a specialized sequence of boot state transitions and must use a different state machine. When the reloading finishes and the node reboots,

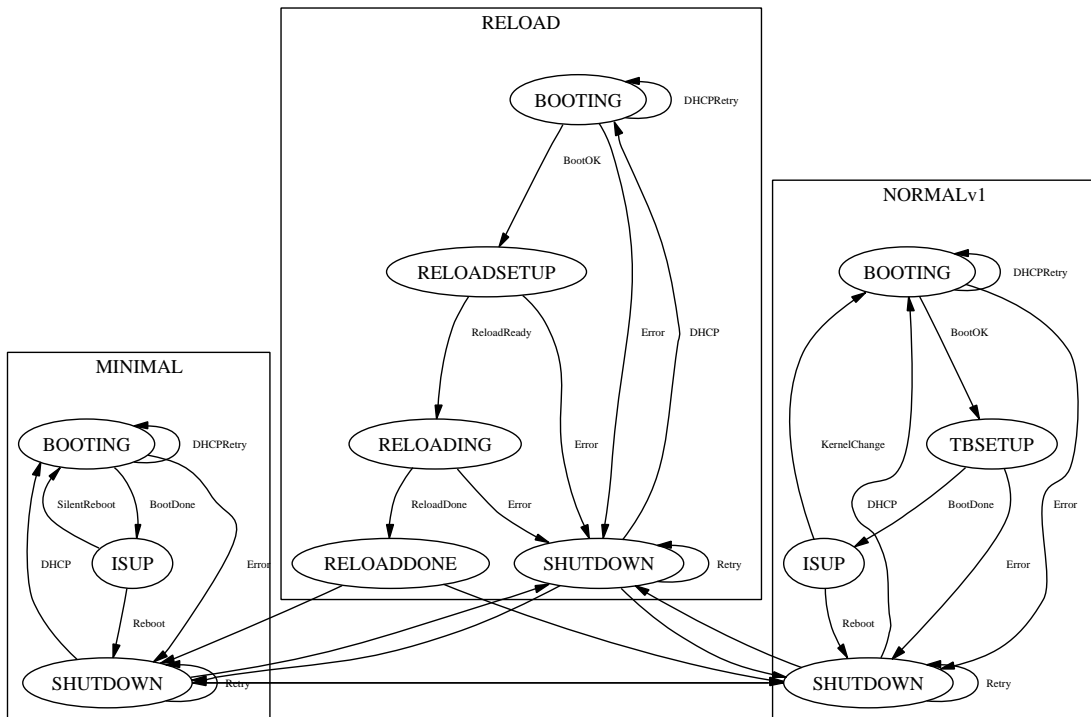


Figure 4.1. Interactions between three state machines

the node changes state machines again to match the new operating system image that it will run.

4.2.2 Indirect Interaction

Indirect interactions occur between state machines of different types, due primarily to the relationships between the entities that each type of machine is tracking. The same entity always has exactly one state in machines of a given type at any time, but the same entity can have a place in multiple types of state machines at once. For instance, there are two types of state machines that deal with nodes as their entity: the node boot state machines, and the node allocation state machine. Every node is in a state in a node boot state machine at the same time that the node also has a state in the node allocation state machine. Other relationships between entities cause indirect interaction as well, like the ownership or allocation

relation that exists between an experiment and the nodes that have currently been assigned to it.

In general, transitions occur independently in each of the different state machines, but there are often correlations between transitions in different machines. A node in the `FREE_CLEAN` state in the node allocation³ state machine should always be in the `ISUP` state in a node boot state machine. When it moves from `FREE_CLEAN` into `RES_INIT_CLEAN`, and then into either `RELOAD_TO_DIRTY` or `RES_CLEAN_REBOOT`, it moves into the `SHUTDOWN` state, and proceeds to cycle through its node boot machine. The node changes state from `RES_WAIT_DIRTY` or `RES_WAIT_CLEAN` to `RES_READY` when it moves into the `ISUP` state in its node boot machine. When all the nodes in an experiment move to `ISUP` in the boot machine and `RES_READY` in the allocation machine, the experiment moves from `ACTIVATING` to `ACTIVE` in the experiment⁴ state machine. Once the experiment is active, the nodes remain in `RES_READY`, but may cycle many times through any of the node boot machines while the experiment is active.

4.3 Models of State Machine Control

Emulab uses two different models for controlling and managing state machines. The node boot state machines use a model of centralized control. A distributed control model is used for the node allocation and experiment status state machines. Each model has different benefits in reliability and robustness, as well as software engineering considerations.

4.3.1 Centralized

The node boot state machines are controlled centrally by a state service running on the masterhost Emulab server. This state service is implemented as a daemon

³A diagram of the node allocation state machine is found in Figure 6.2 on page 44.

⁴A diagram of the experiment state machine is found in Figure 6.1 on page 43.

server called `stated`⁵ and is discussed further in Section 4.4. In the centralized model, state changes are sent to a central server, where the proposed change is checked against the state machine for validity, and saved in the database. These notifications are sent using a publish/subscribe, content-routed, distributed event system. Any invalid transitions cause a notification email to be sent to testbed administrators. Invalid transitions typically indicate an error in testbed software, or a problem with a node's hardware or software. Some errors may be transient, but their patterns and frequency can indicate problems that would otherwise go undetected.

The central also handles any timeouts that may be configured for a particular state. For example, if a node stays in a particular state too long, a timeout occurs, indicating that something has gone wrong, and the service performs the configured action to resolve the situation. In addition to timeouts, actions can also be associated with transitions, and a configured action, or “trigger” can be taken when arriving in a state. Some commonly configured actions include rebooting a node, sending an email notification to an administrator, or initiating another state transition.

Various programs and scripts that are part of the Emulab software also watch for these state change events as they are being sent to the server. For instance, a program may need to wait until a node has successfully booted and is ready for use, and it can do this by requesting to receive any events pertaining to the state of a particular node.

The centralized model for state machine management has many of the same benefits that are typical of centralized systems, as well as similar drawbacks. There is a single program that has a full view of the current state of every entity it manages, and the completeness of this knowledge allows it to take more factors into consideration in its decisions that would otherwise be possible. It also provides a excellent way to monitor and react to state changes in a uniform fashion. It also

⁵The word `stated` is pronounced like “state-dee”, not as the word “stated”, past tense of the verb “state”.

provides a continuously running service that can easily watch for timeouts and other control events to occur. The central model is also easier to implement and maintain than the distributed model. Because every state transition must pass through the central server, it can use caching to improve performance, and only needs to access the database primarily to write changes in the data. It also is better suited to an interrupt-driven style than a distributed model. However, centralized systems have the potential and sometimes tendency to become a bottleneck that hampers the performance of the system. It is not a currently a bottleneck in our system, but the potential is still there. The reliance on a central server also can have a significant effect on the overall reliability of the system, since it cannot function properly when the service is unavailable.

4.3.2 Distributed

The other model of state machine management in use in Emulab is the distributed model, where all the parts of the system that cause transitions in a state machine each take responsibility for part of the management burden. The experiment status and node allocation state machines use the distributed model. Every script or program that wants to check or change the state of an entity directly accesses the database to do so, and performs error checking and validation whenever changes are to be made.

The distributed model does not require a central state management service, which eliminates that service as a source of bottlenecks⁶ in the system. This contributes to better robustness and reliability by not having a dependence on a central state service or the publish/subscribe event distribution system, but also lacks an easy way to watch for timeouts. Each program or script may gain a relatively broad view of the state of all entities, but must do so in a more complex way, and cannot take advantage of the same caching that is possible in

⁶The database server can still become a source of bottlenecks, since every user of the data relies on it. However, solutions exist for configuring distributed database servers and otherwise sufficiently enhancing the performance of the database server to address this issue.

a centralized system. Because changes may come from many different sources, it becomes necessary to use polling to discover changes, which can be problematic at high frequencies when low-latency notification of changes is necessary. It also means that programs must use locking and synchronization with other writers of the data to have the necessary atomicity in their interactions with the data. The distribution of the management burden across many scripts and programs also complicates the software engineering process, although some of this can be alleviated through the use of a software library call to reuse code. Portability also can cause difficulties, since many of the state management actions may need to be performed from scripts and programs written in different languages, which limits code sharing and reuse, and further adds to the software maintenance difficulties, since the code must be written and maintained in multiple languages.

4.4 Emulab's State Daemon

Emulab uses a central state management daemon for controlling the state machines that use the centralized model of management. This daemon is implemented in `stated`, and runs on `masterhost` at all times. The primary architecture of `stated` follows an event-driven model, where the events may be state transitions, state timeouts, or signals sent to the daemon process. At the core of the program is a main loop that continuously alternates between waiting for and processing events. It has been optimized to use a blocking poll whenever possible, and uses a priority queue of upcoming internal events in order to “wake up” as infrequently as possible.

Internally, `stated` uses a cache of state information stored in the database to optimize performance. As the only writer of the state data in the database, cache invalidations are unnecessary, and the data very rarely needs to be read from the database instead of from the cache. It does, however, accept a signal in order to force a reload of the cached data from the database, which can be used when state information is inserted manually into the database by administrators.

The state daemon's event-driven nature helps to make its modular design very natural. Each different kind of event that `stated` handles may have a section

of specialized code to handle the event, if necessary, and it is simple to add new functionality to the daemon due to this design. It contributes to a good balance between database configuration and control for the state machines and special functionality inside the daemon that must be added and maintained as new kinds of events are created.

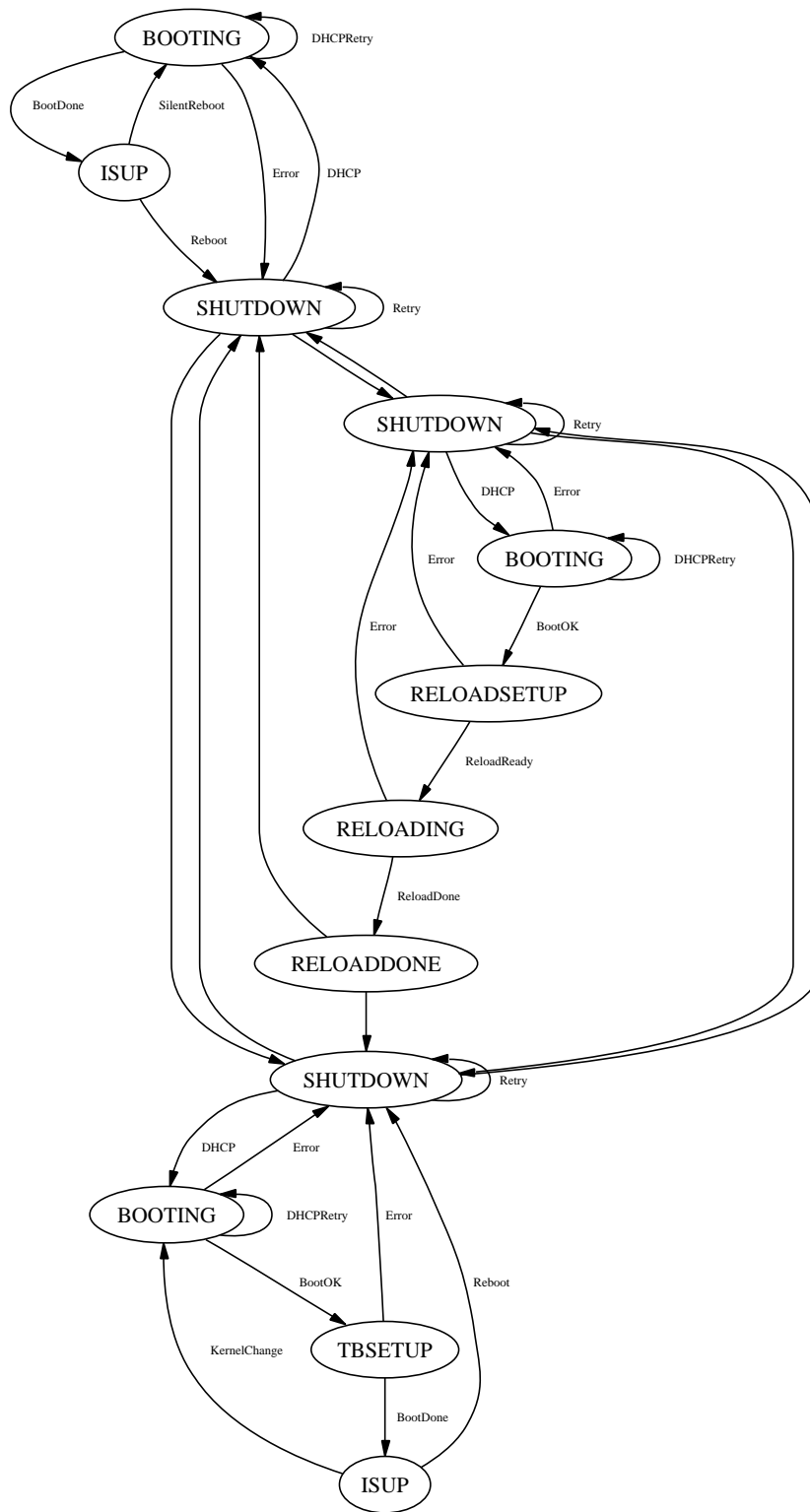


Figure 4.2. Three state machines viewed as one larger machine

CHAPTER 5

NODE CONFIGURATION PROCESS

5.1 Node Self-Configuration

Emulab's node self-configuration process has several benefits over server-driven node configuration. First, because the process is driven by the node, the server doesn't need to keep any extra per-node state while the node is booting. This allows requests from nodes to be served by stateless multithreaded server that gathers the requested information from the database. The node-driven model also contributes to a simple recovery procedure, where nodes can simply be rebooted if they fail. Because node hard drives have no persistent state, disks can also be reloaded as a recovery step. This model is also highly parallel, and contributes to good scalability as the number of nodes booting at once increases.

When nodes begin to boot, they communicate with `masterhost` using PXE, DHCP, and Emulab-specific services to find out what they should do. The answer may be to load a specific kernel or boot image over the network, or to boot from a certain partition on the hard disk. In the case of failure, the default is to boot from a small partition on the disk that causes the node to reboot, forcing a retry of the failed communications with the server. One application of a network loadable boot image is for use with Frisbee[2] to load the hard drive on the node with a disk image.

The normal case is to boot from a partition on the hard drive. In most cases, the partition contains a disk image that has been customized for use in Emulab, although it may be an non-Emulab "custom" OS image that a user has created. Images with Emulab support have hooks in the node boot sequence to perform self-configuration operations. These include configuration of home directories,

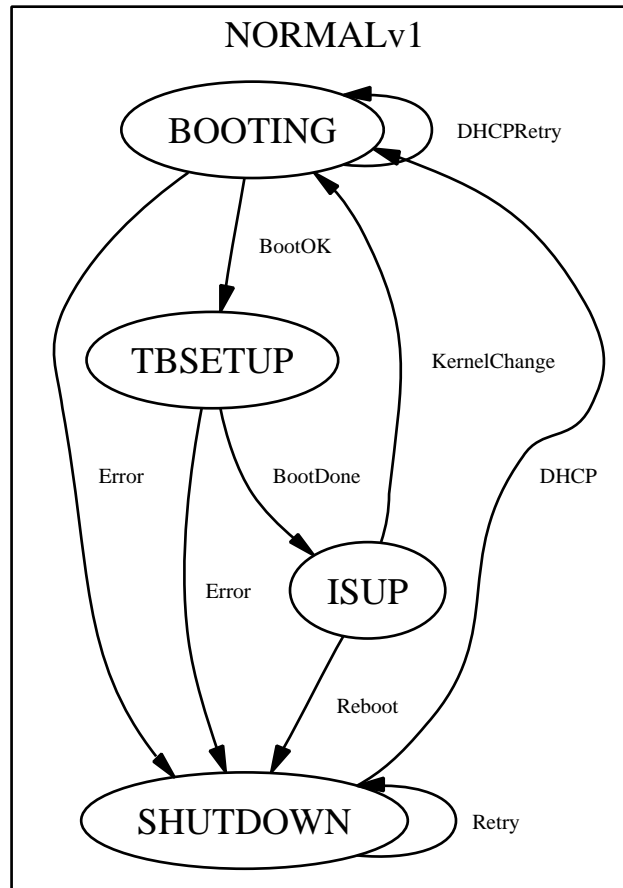


Figure 5.1. The typical node boot state machine

user accounts, network interfaces, experiment-specific host names, routing, traffic generators, RPM or `tar` packages, user programs that should be run on startup, traffic shaping, temperature and activity monitors, and notifying the `masterhost` server that the node has finished booting.

5.2 The Node Boot Process

Figure 5.1 shows the typical boot cycle for OS images that have been customized for Emulab support. In a normal situation, when a node starts rebooting, it transitions to SHUTDOWN. When it contacts `masterhost` for its DHCP information, it enters the BOOTING state. When Emulab-specific node configuration begins,

it moves to TBSETUP. When setup is complete, and the node is ready, it moves to the ISUP state. A failure at any point causes a transition to the SHUTDOWN state again, and booting is retried a small number of times. Note that it is also acceptable for multiple consecutive SHUTDOWN states to be issued. In certain cases it is impossible to make sure that exactly one SHUTDOWN is sent, so the semantics Emulab has chosen to use are that at least one SHUTDOWN must be sent, even though it means that at times two copies may be sent.

There are several ways that a SHUTDOWN transition may be taken. In Emulab images, a transition is sent when a reboot is started (i.e. with the `shutdown` or `reboot` commands). Reboots initiated by Emulab or from the Emulab servers try several methods to reboot the node, and in cases that indicate that the node was unable to send the transition, the transition is sent from the server. If the node reboots without going down cleanly, an invalid transition is detected, notifying testbed administrators that there may be a problem with the hardware or software.

5.2.1 Variations of the Node Boot Process

Different types of nodes, nodes in other roles, and nodes running different OS images often use state machines that are slightly different than the typical node boot state machine. Nodes that are part of the wide-area testbed use one state machine, while “virtual” nodes being multiplexed on physical nodes in the cluster use a different variation. And custom OS images that have reduced Emulab support use a minimal state machine that requires no OS support at all.

The wide-area state machine is shown in Figure 5.2. This state machine is one of the most flexible and permissive, because of the difficulties that arise in managing a diverse and physically distributed set of nodes. Because disks cannot be reloaded as easily in the wide-area, it is not uncommon for a variety of different versions of standard wide-area OS image to be running at any given time on different physical nodes. In some versions, the SHUTDOWN state was known as the REBOOTING state, thus the REBOOTING state has all the same transitions as the SHUTDOWN state. Some versions do not use the BOOTING state, so it is acceptable to go

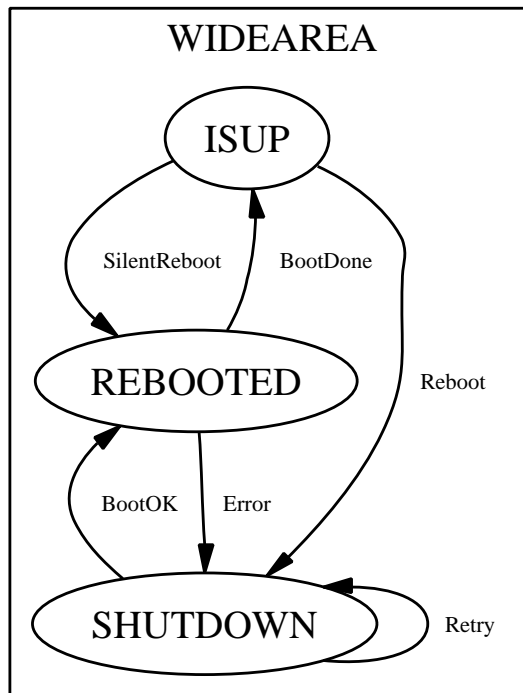


Figure 5.2. WIDEAREA State Machine

from SHUTDOWN directly to the REBOOTED state (the wide-area equivalent of TBSETUP). It is also acceptable to move from ISUP directly to REBOOTED, because at times it is impossible to guarantee that the SHUTDOWN state will be properly sent due to the different software versions on the nodes. The transition from ISUP directly to BOOTING, however, is not allowed.

The “minimal” state machine is shown in Figure 5.3. This state machine is used for OS images that have reduced Emulab support, and may not be capable of giving feedback to the system about their state transitions. This machine does not require any support at all from the node, and adapts to the node’s capabilities to ensure proper operation.

A SHUTDOWN event is sent from the server on any reboot requested from one of the Emulab servers. In MINIMAL mode, a node is permitted to reboot without sending any state change, so a node may also go directly to BOOTING.

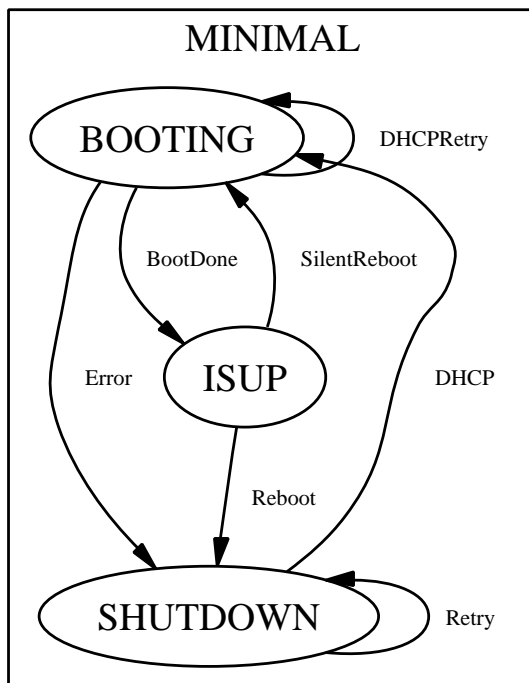


Figure 5.3. MINIMAL State Machine

The BOOTING state is generated from the server when the nodes boot, without regard to what OS they will boot.

The ISUP state can be generated in one of three ways. First, an OS or image may have support for sending the ISUP event itself. Second, an OS that doesn't support ISUP may support ICMP ECHO requests (i.e. `ping`), in which case the server will generate an ISUP after the BOOTING state as soon as the node responds to an echo request. Third, a node that doesn't support any of the above will have an ISUP generated immediately following the BOOTING state. This allows a minimal set of invariants to be maintained for every OS that may be used in Emulab, providing best-effort support for OS images that are not completely Emulab-supported.

The PCVM state machine is shown in Figure 5.4. The PCVM machine's variation is in the TBSETUP state. Because "virtual nodes" do not use DHCP, the BOOTING event must be sent from the virtual node at the point where Emulab-specific setup begins. This is normally the point where TBSETUP would be sent. To

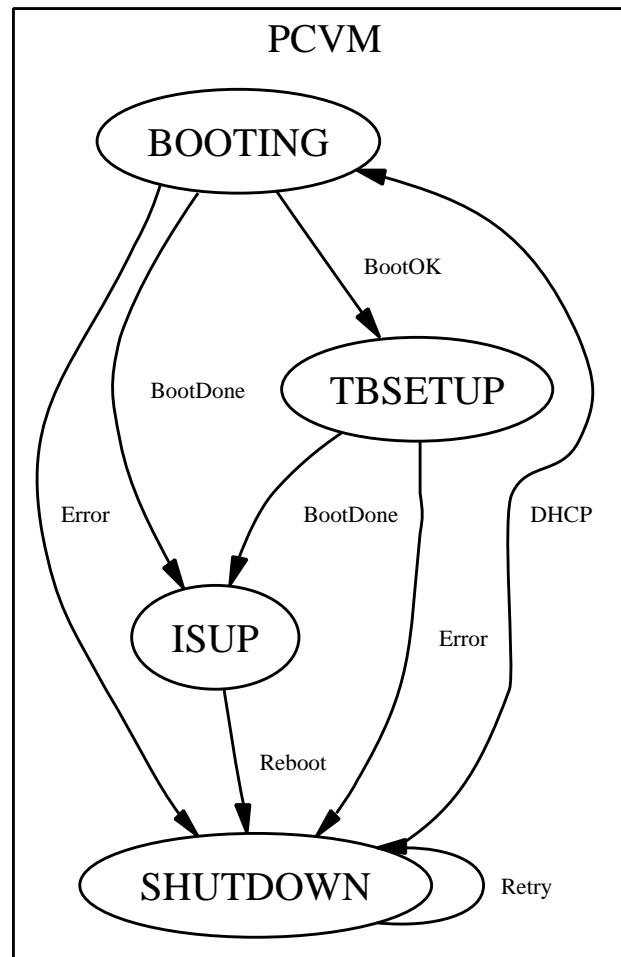


Figure 5.4. PCVM State Machine

eliminate the redundancy, transitions are allowed directly from **BOOTING** to **ISUP**. However, when substantial actions occur between the beginning of Emulab-specific configuration and the time when **TBSETUP** would normally begin, the **TBSETUP** state can be included for better indications of progress from the node.

The **ALWAYSUP** state machine is shown in Figure 5.5. The **ALWAYSUP** machine is used for special nodes that do not have a booting sequence, but instead are simply always ready for use. For consistency with other nodes, a **SHUTDOWN** state is allowed. Any time a **SHUTDOWN** is received by the server, the **ISUP** state is sent

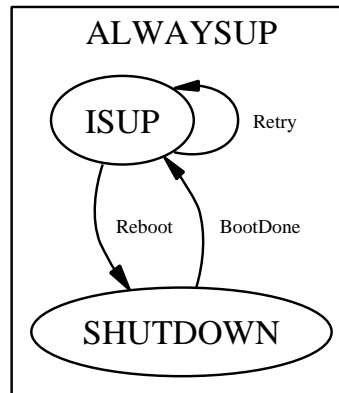


Figure 5.5. ALWAYSUP State Machine

immediately. This allows the Emulab software to treat the node normally, as if it went down, rebooted, and came back up, without requiring special handling.

5.3 The Node Reloading Process

The node reloading process is depicted in Figure 5.6. The process typically starts out with the node node in the ISUP state in one of the node boot state machines. A user, or the Emulab system itself, then requests a reload of the node's disk. On `masterhost`, the preparations are made for the reload, then a reboot is initiated, sending the node to SHUTDOWN. When the node arrives at SHUTDOWN, the centralized state service on `masterhost` makes a check is made for a possible transition between state machines. In this case, it is detected that a change to the reload state machine was requested, and the configured entry point to the reload state machine from the SHUTDOWN state of the current machine is the SHUTDOWN state of the reload machine, and the transition occurs between the two states, moving from one state machine into the other.

As the node starts to boot in the reload state machine, the normal BOOTING transition occurs. When reloading, a small OS image is loaded over the network, which performs a different set of self-configuration instructions. In particular, when it arrives at the self-configuration portion of the booting process, the node sends the

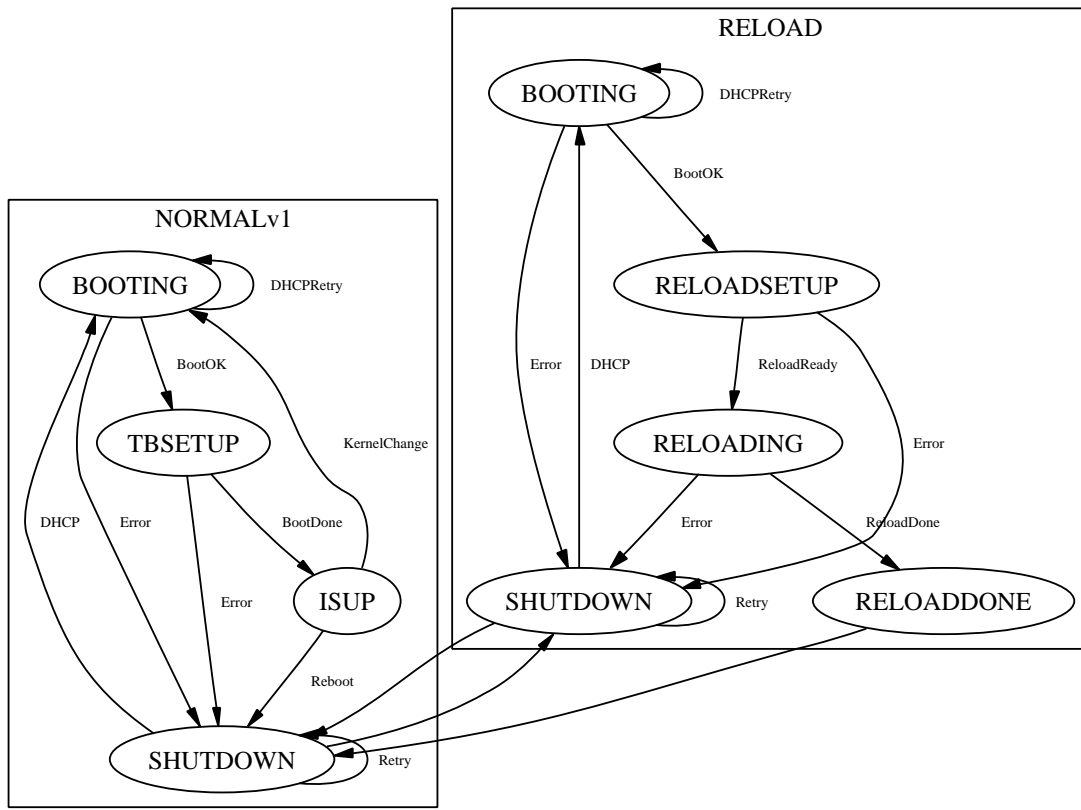


Figure 5.6. The node reloading process

RELOADSETUP state. Then it attempts to connect to the Frisbee[2] disk image server, running on `masterhost`. When the connection is made and the reload is started, the node sends the RELOADING state. The disk image is downloaded as it is uncompressed and written to the hard disk. When the node finishes, it sends the RELOADDONE state, and reboots. At this point, the state service on `masterhost` finds that another mode transition has been requested, and a transition is made from RELOADDONE in the reload state machine into the SHUTDOWN state of the machine that the newly-installed OS uses. The node then continues to boot normally from the newly installed disk image.

One important use of the reloading state machine is to prevent a certain condition that can easily cause nodes to fail unnecessarily. At one time, nodes reloaded after being used in an experiment were released into the free pool as soon as the

RELOADDONE state was reached. This allowed users to allocate those nodes almost immediately, while the operating system on the disk (typically Linux) was booting. During the boot process, the node becomes unreachable by normal means (i.e. ssh and ping), and Emulab detects that a forced power cycle will be necessary to cause the node to reboot. When power was turned off during the Linux boot process, it left the file systems on the disk in an inconsistent and unusable situation, causing the node to fail to boot. This node failure can only be resolved by reloading the disk again. When this race condition was discovered, the state machines were used to ensure that nodes in the free pool were in the ISUP state, and that only nodes in the ISUP state could be allocated to experiments. This has reduced time to swap in an experiment, since it prevents failures that cause long delays in experiment setup, and has also decreased wear on the nodes, since they typically only get reloaded once between experiments, instead of twice or more whenever this failure occurred.

CHAPTER 6

EXPERIMENT CONFIGURATION PROCESS

[Author's note: Due to recent changes in the system, a portion of the content here is outdated in terms of the technical details. It will be updated for the final thesis, but it will remain as it stands for the thesis proposal.]

6.1 Experiment Status State Machine

The experiment status state machine is shown in Figure 6.1. This machine is used to manage the different states of an experiment as it moves through its life-cycle. Experiments start in the `NEW` state when they are initially recorded. They move into the `PRERUN` state while they are being processed, then end up in the `SWAPPED` state. At this point, all the information is present in the database to be able to initiate a swapin to instantiate the experiment on hardware.

When swapin begins, it moves to the `ACTIVATING` state while nodes are allocated and configured, and other experiment setup activities are performed. Then it moves into the `ACTIVE` state. The `TESTING` state is used in the regression testing system during software-only testing, when physical nodes are not allocated or configured. In case of a failure, it may move back to the `SWAPPED` state.

When an active experiment is swapped out, it moves to the `SWAPPING` state, then the `SWAPPED` state as swapout finishes. A swapped experiment may be swapped in again, or may be terminated.

When the experiment is terminated, it moves to the `TERMINATING` state while cleanup proceeds, then to the `TERMINATED` state just before the experiment is deleted.

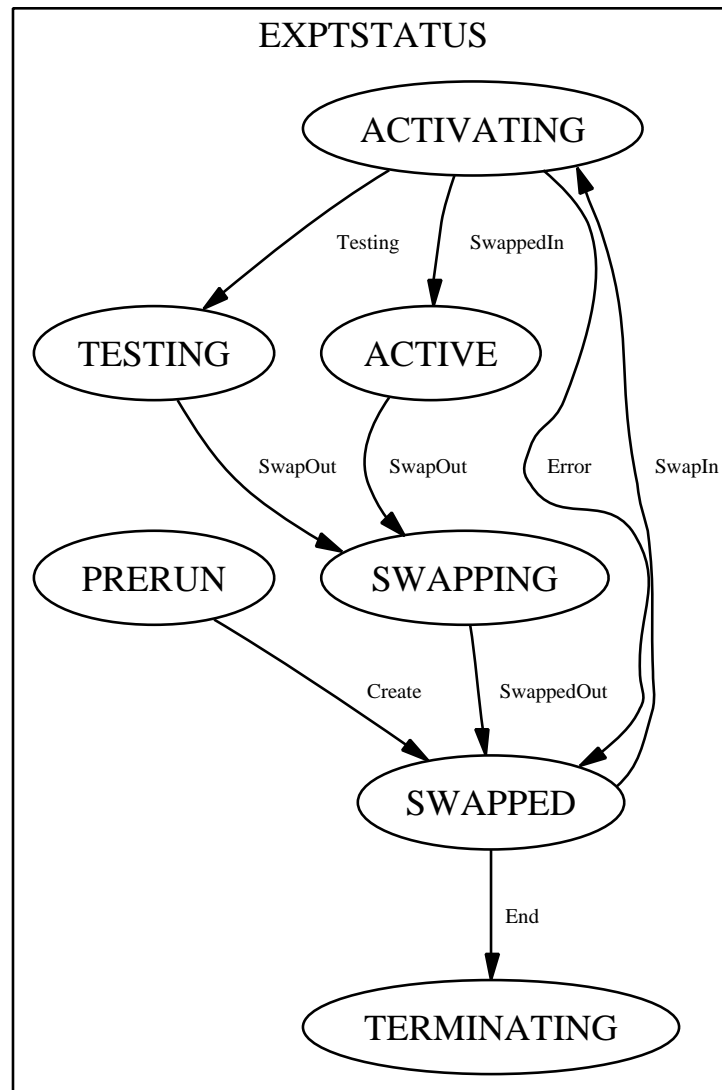


Figure 6.1. Experiment Status State Machine

6.2 Node Allocation State Machine

The node allocation state machine is shown in Figure 6.2. This machine is used to track the progress of a node as it progresses through the allocation/deallocation cycle that occurs with each node as it is moved into and out of experiments. Unlike the node boot state machines, a node's state in this machine is not mutually exclusive of a state in a node boot state machine. Nodes always have a state in each class of state machine.

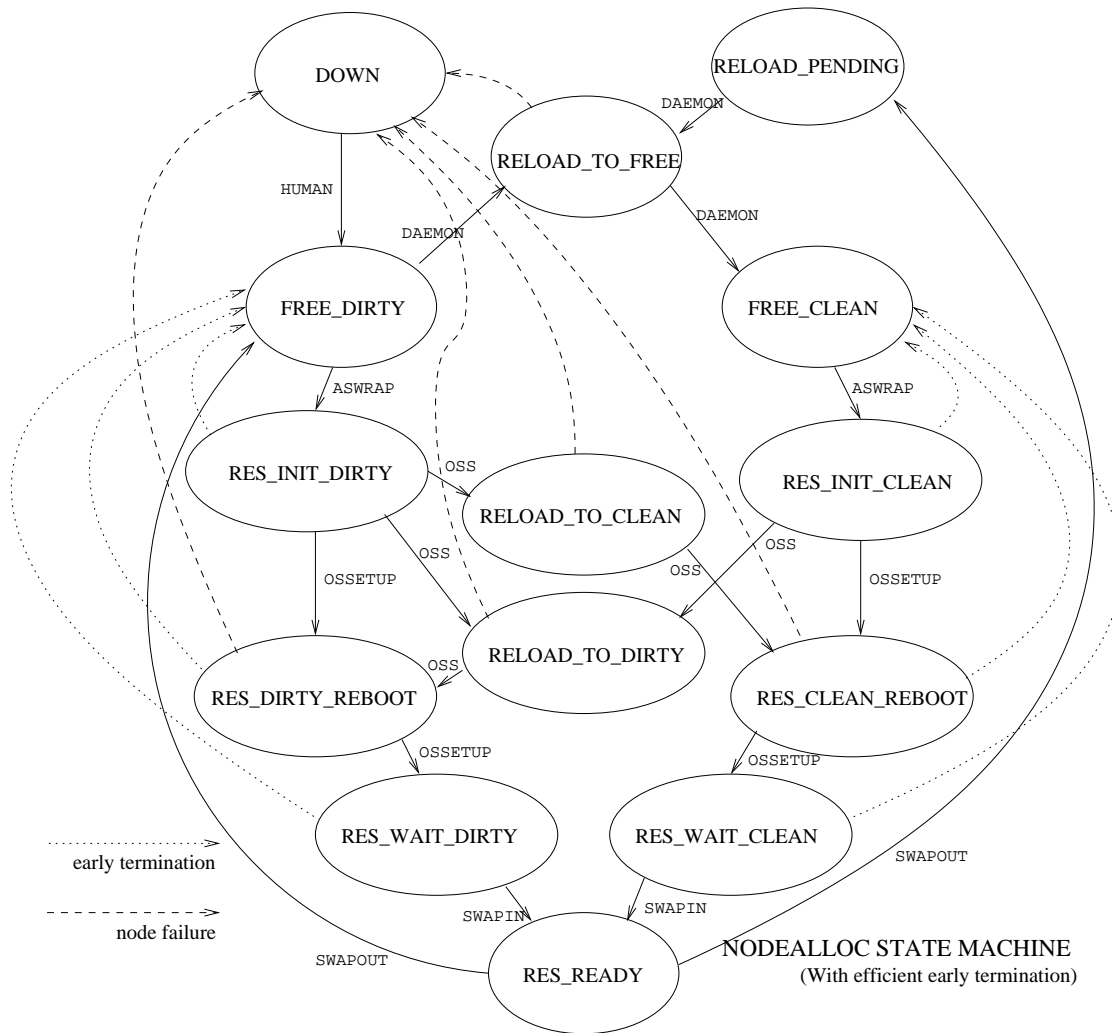


Figure 6.2. Node Allocation State Machine

A node that is currently allocated to an active experiment is in the **RES_READY**¹ state. When it is deallocated, it moves to the **RELOAD_PENDING**² state in the

¹In state names, RES signifies Reserved, meaning the node is allocated to an experiment. FREE signifies the opposite. CLEAN signifies that the node is loaded with a pristine copy of the default operating system image. DIRTY means that the node's disk may have been touched by the user or the system, or may have been loaded with a different image.

²Alternatively, the node could be freed, moving to **FREE_DIRTY**, where it may be reserved, moving to the **RES_INIT_DIRTY** state, or reloaded and moved through the **RELOAD_TO_FREE** and **FREE_CLEAN** states. This path is not currently used in Emulab.

normal case, then as it begins reloading, it moves to `RELOAD_TO_FREE`, and then `FREE_CLEAN` when the reload is complete.

The node then waits in the `FREE_CLEAN` state until it is reserved, where it moves to `RES_INIT_CLEAN`. If the user has chosen one of operating systems included in Emulab's default disk image, it is rebooted and moved to `RES_CLEAN_REBOOT`. If they have chosen a different OS, the node moves into `RELOAD_TO_DIRTY` as it gets reloaded with the right image, and then to `RES_DIRTY_REBOOT` as it boots. When the node finishes booting, it moves from `RES_CLEAN_REBOOT` to `RES_WAIT_CLEAN` (or `RES_DIRTY_REBOOT` to `RES_WAIT_DIRTY`), then finally to `RES_READY`, where it is completely allocated and configured for use in an experiment.

A node has a state in this state machine, as well as a state in a node boot state machine, at the same time that the experiment to which the node may be allocated has a state in the experiment status machine. Transitions and states in each of these machines may or may not be correlated. Most transitions within the node allocation machine occur only during the `ACTIVATING` and `SWAPPING` states of the experiment status machine. When an experiment is `ACTIVE`, its nodes are in the `RES_READY` state in the node allocation machine, but may be in any state in a node boot machine. The some transitions in the node allocation machine correspond to transitions in the node boot state machine, like `RES_INIT_CLEAN` to `RES_CLEAN_REBOOT` corresponds to a transition from `ISUP` to `SHUTDOWN` in the node boot machine.

CHAPTER 7

THESIS SCHEDULE

There are three unfinished major milestones for the completion of this thesis following the approval of the thesis proposal. The first is the addition to and alteration of the thesis proposal, according to the committee's guidelines, to produce a first complete thesis draft, which will then be submitted to the committee for review. The second milestone is committee approval to schedule a thesis defense, based on a revision of the complete thesis draft. The third milestone is the thesis defense, and if necessary, further modification of the thesis, and the final reading approval of the thesis by the committee. After these milestones have been completed, the thesis will be provided to the thesis editor, any format revisions made, and the thesis will be released for publication.

My proposed time-line for the completion of these milestones is as follows. The complete thesis draft should be ready for review by Monday, February 23rd, 2004. The revised and approved final thesis draft should be ready to be defended and a defense held by Monday, March 8th, 2004. The thesis should be submitted to the thesis editor by Monday, March 15th, 2004. This schedule will allow sufficient time for the thesis to be released in a timely manner such that my graduation will occur in May, 2004.

REFERENCES

- [1] J. Gelinas. Virtual private servers and security contexts. Available from http://www.solucorp.qc.ca/miscprj/s_context.hc, 2002.
- [2] M. Hibler, L. Stoller, J. Lepreau, R. Ricci, and C. Barb. Fast, scalable disk imaging with frisbee. In *Proc. of the 2003 USENIX Annual Technical Conf.*, pages 283–296, San Antonio, TX, June 2003. USENIX Association.
- [3] Intel Corp. Ixp1200. <http://www.intel.com/design/network/products/-npfamily/ixp1200.htm>.
- [4] Internet2 Consortium. Internet2. <http://www.internet2.edu/>.
- [5] P.-H. Kamp and R. N. M. Watson. Jails: Confining the omnipotent root. In *Proc. 2nd Intl. SANE Conference*, May 2000.
- [6] NSF Workshop on Network Research Testbeds. Workshop Report. http://-gaia.cs.umass.edu/testbed_workshop/testbed_workshop_report_final.pdf, Nov. 2002.
- [7] L. Rizzo. Dummynet home page. http://www.iet.unipi.it/~luigi/-ip_dummynet/.
- [8] L. Rizzo. Dummynet: a simple approach to the evaluation of network protocols. *Computer Communication Review*, 27(1):31–41, Jan. 1997.
- [9] L. Rizzo. Dummynet and forward error correction. In *Proc. of the 1998 USENIX Annual Technical Conf.*, New Orleans, LA, June 1998. USENIX Association.
- [10] The VINT Project. *The ns Manual*, Apr. 2002. <http://www.isi.edu/nsnam/-ns/ns-documentation.html>.
- [11] B. White, J. Lepreau, and S. Guruprasad. Lowering the barrier to wireless and mobile experimentation. In *Proc. HotNets-I*, Princeton, NJ, Oct. 2002.
- [12] B. White, J. Lepreau, and S. Guruprasad. Wireless and mobile Netscope: An automated, programmable testbed. <http://www.cs.utah.edu/~lepreau/emulab-wireless.ps>. Submitted for publication in Mobicom'02., Mar. 2002.
- [13] B. White, J. Lepreau, L. Stoller, R. Ricci, S. Guruprasad, M. Newbold, M. Hibler, C. Barb, and A. Joglekar. An integrated experimental environment for distributed systems and networks. In *Proc. of the Fifth Symposium on Operating Systems Design and Implementation*, pages 255–270, Boston, MA, Dec. 2002.

- [14] B. White, J. Lepreau, L. Stoller, R. Ricci, S. Guruprasad, M. Newbold, M. Hibler, C. Barb, and A. Joglekar. An integrated experimental environment for distributed systems and networks (full report). <http://www.cs.utah.edu/~flux/papers/emulabtr02.pdf>, May 2002.