GODS: Global Observatory for Distributed Systems

Cosmin Arad, Ozair Kafray, Ali Ghodsi and Seif Haridi
cosmin, ozair, ali, seif @ sics.se

SICS Technical Report T2007:10
ISSN 1100-3154
ISRN:SICS-T–2007/10-SE
Revision: 1.00, 2007-08-30

keywords : distributed systems, evaluation framework, deployment test-bed, distributed algorithms debugging, performance tuning, regression testing, bandwidth accounting, automated experiments, benchmark

Abstract

We propose GODS, an ecosystem for the evaluation and study of world-wide distributed and dynamic systems under a realistic emulated network environment. GODS allows the evaluation of a system's actual implementation in reproducible experiments, collecting global knowledge about the system state.

Furthermore, GODS addresses the problems of debugging distributed algorithms, performance tuning, measuring bandwidth consumption, regression testing, and benchmarking similar systems, thus offering a complete evaluation environment for distributed applications.

Our framework uses ModelNet for the network emulation and enhances that by (1) adding dynamism by varying link properties, partitioning the network and emulating churn, (2) offering global knowledge about the observed system by gathering statistics and events and (3) enabling the user to easily deploy, manage and monitor complex, large-scale distributed systems.

Contents

List of Acronyms iv

1 Introduction 2

1.1 Motivation . . . 2

1.2 ModelNet Overview . . . 3

2 Functional Features 5

2.1 Deployment and Management . . . 5

2.2 Monitoring and Control . . . 5

2.3 Tracing and Debugging . . . 6

2.4 Bandwidth Accounting . . . 7

2.5 Automated Experiments . . . 7

2.6 Performance Tuning . . . 7

2.7 Regression Testing . . . 8

2.8 Benchmarking . . . 8

2.9 Byzantine Behaviour Observation . . . 8

3 Architecture 9

3.1 Topology Module . . . 10

3.2 Churn Module . . . 11

3.3 Network Partitioning Module . . . 12

3.4 Statistics Monitoring, Aggregation and Caching Module . . . 12

3.5 Operations Module . . . 13

3.6 Automation Module . . . 14

3.7 Bandwidth Accounting Module . . . 14

4 Statistics and Notifications 16

4.1 Statistics . . . 16

4.2 Notifications . . . 18

5 Use Cases 19

5.1 Interactive Control and Monitoring . . . 19

5.2 Automated Experiments . . . 21

6 Implementation Details 22

6.1 Core Concepts . . . 22

6.1.1 Event . . . 22

6.1.2 Module 22
6.1.3 Event Handler 23
6.1.4 Subscriptions Registry 23
6.1.5 Task 24
6.2 Event Handling 24
6.3 Software Architecture 24
6.3.1 Overview 24
6.3.2 Control Center 25
6.3.3 Agent 26
6.3.4 Application Interface 27
6.4 Extension Mechanisms 28
6.5 Technologies Used 29

7 Contributions **31**
7.1 Controllability and Reproducibility 31
7.2 Emulating Churn 31
7.3 Emulating Network Partitioning 31
7.4 Bandwidth Accounting 31
7.5 Emulating User Behaviour 32

8 Related Work **33**
8.1 Application Control and Monitoring Environment (ACME) 33
8.2 Distributed Automatic Regression Testing (DART) 33
8.3 WiDS 34
8.4 Liblog 35
8.5 Ganglia 36

9 Users Guide **38**
9.1 Preliminaries 38
9.1.1 Enable Password Less Login 38
9.1.2 Setup a Webserver 38
9.2 Configuring GODS 38
9.2.1 GODS config file 38
9.2.2 Deploying Agents 38
9.2.3 Configuring Agents 39
9.2.4 Deploying Application 40
9.3 Running GODS 40
9.3.1 Environment Setup File 40

9.3.2 Running Visualizer 41
9.4 Experiments 41
9.4.1 Generating an Experiment 41
9.4.2 Running an Experiment 44

References **46**

List of Acronyms

ACM Agent Churn Module, 10
AM Automation Module, 13, 14
AOM Agent Operations Module, 13
ASM Agent StatsMon Module, 12, 15–17
ATM Agent Topology Module, 10, 16

BAA Bandwidth Accounting Agent, 14
BCEL Byte Code Engineering Library, 27

CCCM Control Center Churn Module, 10, 11, 14
CCOM Control Center Operations Module, 10, 11, 13, 14, 20
CCPM Control Center Partitioning Module, 10–12
CCSM Control Center StatsMon Module, 12, 15–17
CCTM Control Center Topology Module, 10, 11, 13

DSUT Distributed System Under Test, 8–20, 23–27

JDBC Java Database Connectivity, 28
JMX Java Management Extensions, 5, 16, 17, 26
JUNG Java Universal Network/Graph, 28
JVM Java Virtual Machine, 16, 23, 26
JVMTI Java Virtual Machine Tool Interface, 5, 16, 17, 26, 27

NPA Network Partitioning Agent, 11

RMI Remote Method Invocation, 27

SHA-1 Secure Hash Algorithm, 18

XML Extensible Markup Language, 16, 17, 28

1 Introduction

GODS is an ecosystem for controlling the deployment, monitoring and evaluation of large scale distributed applications in an emulated wide-area network environment to which it can apply churn models and network partitioning models, in the context of reproducible experiments.

At the heart of GODS sits ModelNet [25, 21, 34, 4, 36], a large-scale network emulator that allows users to evaluate distributed networked systems in realistic Internet-like environments.

While ModelNet provides the network emulation, GODS enables effortless handling of the complexity of managing and monitoring thousands of distributed application nodes. It provides global knowledge of select system state in the form of aggregated statistics and a global view of the system topology. GODS allows users to trace the execution of distributed algorithms, to trigger various operations, to collect statistics for user-defined experiments and to fully automate and replay experiments. Automated experiments can be used for collecting evaluation data, performance tuning, regression testing, and benchmarking similar systems.

Next, we give the motivation for building GODS and in the following section we describe the functional features of GODS. Then, in Section 3, we present the GODS architecture. In Section 4 we describe the experiments and collected statistics. In Section 5 we present some use cases. In Section 6 we give some implementation details and in Section 7 we discuss our contributions in conjunction with the Section 8 detailing related work. Finally, Section 9 is the GODS user's guide.

1.1 Motivation

There are essentially three ways to validate research on distributed systems and algorithms. One is to to analytically verify the correctness and efficiency of a system. Another approach is to verify the results by means of simulation, whereby a model of the system is built and used for simulation. Finally, one can deploy the system itself and do black-box observations on its behaviour. The first two methods have the disadvantage that they draw conclusions about a model, rather than the real system. Hence, they may fail to spot real bottlenecks or consider practical issues. The last method, as it is applied currently, is too coarse and does not give specific insight into every component.

We would like to verify and analyse the actual distributed system, rather than a model of it. Consequently, the real system with all its intricacies and shortcomings would be studied, enabling us to make changes and fix bugs in only one version of the system. In addition, we would like to use real-world scenarios and input data, when running the deployed system, to be able to exactly pin-point hot spots, resource consumption causes, bandwidth usage of each component, and to catch defects. Moreover, we would like to run automated batch experiments that allow fine tuning of system parameters by running experiments multiple times for a wide range of parameter values.

Currently, the largest public real-world network test-bed, PlanetLab [29], provides around 600 nodes distributed around the world. We need larger-scale test-beds that retain the property of real-world latencies and bandwidths between the nodes. Furthermore, we need to study distributed systems behaviour under various churn models and network partitioning models, in completely controlled and reproducible experiments. To our knowledge, to date, this is only achieved through simulation.

ModelNet [25, 21] is a large-scale network emulator that allows the deployment of thousands of application nodes on a set of cluster machines. ModelNet provides a realistic network environment to the deployed application. This has the advantage of offering a real-world large-scale test-bed in a controlled environment. But the large scale comes with great complexity in management and evaluation of the deployed system, so we need a tool that enables us to control that complexity. Apart from that, ModelNet provides only a static network model. We need to add dynamism to that model in the form of dynamically partitioning the network, controlling nodes' presence in the network, and also by varying the properties of the network links.

We have identified the need for real-world, large-scale test-beds, for easily manageable, controlled and reproducible experiments, that provide detailed insight into the operation of the observed systems. To address this need, we have decided to design and develop GODS, to benefit the community of distributed systems researchers and developers.

1.2 ModelNet Overview

ModelNet is a wide-area network emulator that can be deployed on a local-area cluster. Using ModelNet, one can deploy a large-scale distributed system on a local-area network, providing a realistic Internet-like environment to the

deployed system.

Nodes of the distributed application, which we shall call *virtual nodes*, run on some physical machines in the cluster, called *edge nodes*, and all traffic generated by the virtual nodes is routed through a ModelNet *core*, consisting of one or more physical machines. This core emulates a wide-area network by subjecting each packet to the delay, bandwidth, and loss specified for each link in a virtual topology.

The target topology can be anything from a hand-crafted full-mesh, populated with real-world latencies taken from DIMES [8, 30] or King datasets [18, 15], to a complete transit-stub topology, created by available topology generators such as GT-ITM [14], Inet [16] or BRITE [2, 3]. Each link in the virtual topology is modelled by a packet queue and packets traversing the network are emulated hop-by-hop, thus capturing the effects of cross traffic and congestion within the network.

ModelNet maps virtual nodes to edge nodes, binding each virtual node in the topology to an appropriate IP address in the 10.0.0.0/8 range. For relatively low CPU, low bandwidth applications, it is possible to run 10's or even 100's of instances of an application on one edge node. In the case when the virtual nodes running on one edge node begin to contend for resources like CPU or bandwidth, additional edge nodes can be added to allow the network size to scale.

All packets generated by virtual nodes are routed to the core even if both the source and the destination virtual nodes live on the same physical machine. ModelNet builds a shortest path global "routing table" that contains the set of pipes to traverse for each source-destination path. The core applies the characteristics of all pipes along a path to packets that should travel on that path and then routes the packets to the destination virtual node.

When the amount of traffic generated by the virtual nodes becomes unbearable for the ModelNet core, additional machines can be added as core nodes. The emulation load is balanced across all core nodes, each core node being responsible for a part of the virtual topology, thus emulating a subset of the pipes. Packets need to be handed over to a different core node when they have to hop through a pipe that is handled by that respective core node.

2 Functional Features

GODS is a companion tool that assists the development and maintenance of large-scale distributed applications throughout their life-cycle. GODS facilitates the *deployment and management*, *monitoring and control*, *tracing and debugging*, *performance tuning*, *bandwidth accounting*, *regression testing*, and *benchmarking* of distributed applications, in *reproducible automated experiments* emulating real-world networking environments. Moreover, GODS allows its user to observe the influence of nodes with *Byzantine behaviour* on the distributed application. Let us discuss now each of these features in turn.

2.1 Deployment and Management

GODS enables its users to automatically deploy thousands of instances of a distributed application onto an Internet-like wide-area network, emulated on a local cluster. GODS manages the lifetime of these application instances. They can be launched, shut down, and killed, thus emulating nodes joining, leaving, and failing in the distributed system.

The join, leave, and fail events can be modelled as Poisson processes. Given these specifications, GODS generates a churn event script, that can be executed and reproduced in multiple experiments. GODS keeps track of each application instance's status and distinguishes between controlled and uncontrolled failures, thus accurately emulating node failure.

GODS can dynamically change the deployment network environment by varying the latency and bandwidth of links, or emulating network partitions. GODS can emulate non transitive network connectivity that sometimes occurs in the Internet due to firewalls or routing policies. Network variation events can be reproduced using a script similar to the churn event script.

2.2 Monitoring and Control

GODS enables the user to monitor the application instances by watching select state variables and notifying updates to these variables. Select application methods can also be watched, all calls to these methods being notified. Before starting an experiment, the user specifies the variables and methods to be watched. Then, during experiment execution, GODS collects and logs all state update and method call notifications.

Debugging mechanisms are used to watch state updates and method calls. For distributed applications written in the Java programming language, these mechanisms are provided by the Java Virtual Machine Tool Interface (JVMTI) and the Java Management Extensions (JMX), making it unnecessary to change the application's source code.

Given that GODS collects global knowledge about the system state, global statistics can be compiled across all the application instances.

GODS allows the user to control the distributed application's behaviour by issuing application specific operations on certain application instances. Operation invocations can be specified as Poisson processes, from which GODS generates an operation invocation script that can be executed and reproduced in multiple experiments. Operation invocations effectively emulate users of the distributed system.

2.3 Tracing and Debugging

GODS collects all notifications about state updates and method calls into a centralised log. The centralised log is an interleaving of notifications occurring at different machines in the cluster. GODS synchronises the clocks of the cluster machines before starting an experiment. All notifications are timestamped with the local clock of the machine where they occur, and the centralised log is sorted by notification timestamp and machine id.

The user can tag some application methods as events representing the sending or receipt of a message whereby the message id is one of the method's parameters. Thus, some method call notifications are tagged as send or receive events, providing causal order among notifications. GODS verifies whether the timestamp total order of the notifications log satisfies causal order and warns the user if that is not the case. Causal order should always be satisfied if the minimum message propagation delay is larger than the maximum difference in machines' local clocks.

The notifications log can be used to replay the logged view of the experiment execution. The user can step forward or backward or jump to any time in the notifications log. She can trace the execution of distributed algorithms by watching select state updates and message passing between application instances.

2.4 Bandwidth Accounting

GODS timestamps and logs all the traffic generated by the distributed application. Through the timestamps of traffic packets and the timestamps of events in the application, GODS correlates traffic to certain operations and modules of the application. Hence, GODS provides bandwidth accounting for the observed distributed application, and allows the user to observe how various changes to the application influence bandwidth consumption.

2.5 Automated Experiments

The user can define experiments that are executed automatically multiple times. The definition of an automated experiment contains the ModelNet topology of the network that is emulated while running the experiment. Next, the experiment definition contains a churn event script, that drives virtual nodes to join and leave the system or fail, a network variation script, that drives network partitionings and link failures, and an operation invocation script, that drives the operations executed by the virtual nodes, emulating their users.

The output of the experiment is also specified. Besides the execution replay log, that can be used for tracing, the data collected in an experiment run contains the results and timings of the invoked operations, the bandwidth consumption log, and various other system measurements.

Automated experiments can be driven by existing automation systems. GODS provides bindings to popular scripting and programming languages, so that external GODS clients can be implemented. Notification email is sent upon experiment completion or failure.

2.6 Performance Tuning

Automated experiments are leveraged to fine tune parameters of the observed distributed system. The user specifies ranges of values for a set of parameters taken as input by the observed system. Also an experiment description is given, specifying the performance metrics to be collected.

GODS executes the experiment multiple times, varying the values of the parameters. Each parameter is varied while keeping the other ones constant. This allows the user to observe the influence of each parameter on the various performance metrics. Hence, the user is enabled to spot different trade-off points.

2.7 Regression Testing

Some bugs in distributed systems manifest only in some certain "unfortunate" timing conditions. Thus, reproducing the network conditions and operations timing is crucial in reproducing these bugs. To cover certain code paths in the observed distributed application, tests supplying specific timing conditions need to be crafted. A suite of such tests may need to be run to asses the functionality of the application. Tests for uncovering regression bugs are usually added to the test suite.

Automated experiments are again leveraged to run regression test suites. Effectively, each test in the suite is run as an automated experiment, with specific network conditions, churn and operations timing. Tests using the same virtual network topology are grouped together to avoid unnecessary ModelNet network deployments.

2.8 Benchmarking

GODS serves as the foundation for benchmarking large-scale distributed systems with qualitatively comparable functionality. Identical automated experiments can be executed for different distributed systems. Hence, two or more systems can be run under the same network conditions, subjected to the same churn scenarios and the same service requests or operations. Measurements for various performance or resiliency metrics are collected in the experiments and used to compare the evaluated systems.

2.9 Byzantine Behaviour Observation

Groups of nodes with different characteristics can be defined in a GODS experiment. Such a group may be comprised of nodes running a modified version of the application that behaves maliciously. Having a group of malicious nodes around, and being able to control them, enables the user to observe how the rest of the nodes cope with the malicious behaviour. Furthermore, using automated experiments, the user can observe the limit on the number of malicious nodes where the functionality of the application becomes disrupted.

3 Architecture

The GODS architecture is depicted in Figure 1. On each machine in the cluster there are *n slots* created by ModelNet. In each slot, one of the Distributed System Under Test (DSUT) nodes can be run. We say that a slot is *unused* if no DSUT node is currently running on that slot, that is, no process has bound the IP alias provided by the slot. Otherwise, we call the DSUT node running on the slot, a *virtual node*.

On each machine runs an *agent* which is in charge of managing all the slots and virtual nodes on that machine. The agent is able to start, gracefully shutdown or kill the local virtual nodes, thus offering mechanism to simulate join, leave and failure of DSUT nodes. The agent functionality is provided by a handful of modules (*Topology*, *Churn*, *Operations*, *StatsMon*) driven by their counterparts in the *control center*. Their role is described later on.

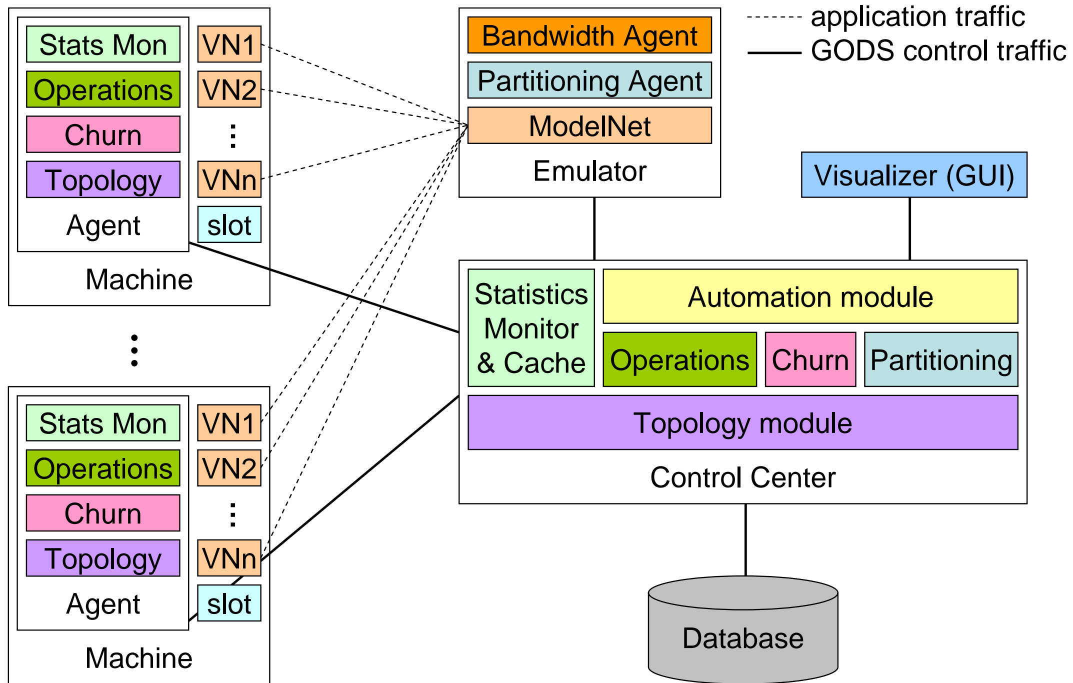

Figure 1: GODS Architecture

ModelNet *core* nodes, or *emulators*, are special machines running the FreeBSD 4.x operating system. The ModelNet emulator is implemented as a 4.x FreeBSD[1] kernel module and relies on the IP forwarding kernel module for routing traffic

[1] A Linux implementation of the ModelNet emulator is under development

between virtual nodes and subjecting the traffic to the delay, bandwidth, and loss specified by the target topology. When a single emulator machine becomes a bottleneck for the traffic generated by the virtual nodes, more emulators can be added to share the load. On each emulator machine runs a *partitioning agent* able to manage IP forwarding rules on the emulator, thus offering mechanism to simulate network partitions and link failures, and a *bandwidth agent* responsible of tracing DSUT bandwidth usage.

On a separate dedicated machine runs the *control center*, a daemon in charge of orchestrating the activity of all virtual nodes in the DSUT, aggregating statistics and providing global knowledge about the DSUT. It is comprised of the *Topology*, *Churn*, *Operations* and *StatsMon* modules that control their counterparts in the agents, a *Partitioning* module that controls its counterparts in the emulators, and an *Automation* module that allows carrying out repeated experiments. The control center exports its services to an external *Visualiser* and to external automation tools, and relies on a database to store the data collected in the automated experiments.

In the following subsections we describe the requirements for each module.

3.1 Topology Module

The Agent Topology Module (ATM) is responsible of slots accounting, i.e., keeping track of used and unused slots. Slots are static in the sense that no new slots appear or disappear after the module is started, but their status may change, that is, new virtual nodes may be launched or virtual nodes may fail as a result of software defects or misconfiguration. The ATM has to actively make sure that started virtual nodes are still alive. In order to be able to correctly enforce churn models, we need to be in control of VN failure, that is, VNs that we think are alive, should be alive.

If the ATM finds uncontrolled VN failure it reports it to the Control Center Topology Module (CCTM) and the running experiment is deemed failed. The ATM checks with the Agent Churn Module (ACM) the presupposed legitimate status of VNs before triggering an experiment failure.

The ATMs aggregate information about the status of all local slots and push it to the CCTM, so topology information flows from the ATMs to the CCTM. At ATM startup all local slots are accounted and the local view is pushed to the CCTM. As DSUT nodes are launched, slot status updates are incrementally pushed to the CCTM.

The CCTM receives slots statuses from all ATMs and compiles a global view of available resources (slots) and virtual nodes. This view constitutes input to the Control Center Churn Module (CCCM), the Control Center Partitioning Module (CCPM), and the Control Center Operations Module (CCOM). Each slot has a numeric identifier and the global slots view is a mapping from slot IDs to slot information structures containing: slot IP, machine IP, slot status, DSUT node ID, VN PID, slot status history, etc.

The CCTM is also in possession of the all-pairs latency and bandwidth maps that are part of the deployed ModelNet target topology. The CCTM reads these from the ModelNet model files upon initialisation.

3.2 Churn Module

At the churn layer information flows from the control center to agents. The ACM is controlled by the CCCM and is responsible of enforcing churn models.

The ACM implements functionality for launching DSUT nodes, shutting them down gracefully or killing them. While launch and kill operations are external to the DSUT nodes, graceful shutdown may require interfacing with the DSUT application (in cases when the DSUT application does not handle a specific signal for graceful shutdown). We discuss the DSUT application interface in Section 6.3.4.

The CCCM provides functionality for globally executing a churn event script in the context of an automated experiment, and issuing a single join/leave/fail command for the purpose of monitoring/visualising the execution of the respective join/leave/fail algorithms in the DSUT application.

How to define and implement a churn model is an open issue. One possibility is to take as input a node lifetime distribution. Another is to take as input absolute join/leave/fail rates (#/s) or join/leave/fail rates relative to other DSUT application operations. This would relate to an operations model applied by the CCOM. Given these models and the cluster and emulators traffic handling capacity, absolute churn rates can be devised. Starting from the churn model, a churn event script is generated.

The churn event script execution relies on the global slots view, provided by the CCTM, to decide which slots to launch/shutdown/kill.

3.3 Network Partitioning Module

The CCPM is responsible of executing a network variation script in the context of an automated experiment and issuing a single network partitioning or link failure for the purpose of visualising the DSUT application behaviour in effect of the partitioning or failure.

How to define a network partitioning and link failure model is still an open issue. One possibility is to split the network and re-merge it repeatedly. Another is to have multi-level splits and recombined merges. For instance:

$$N_{1234} \rightarrow N_{12} \wedge N_{34} \rightarrow N_1 \wedge N_2 \wedge N_3 \wedge N_4 \rightarrow N_{13} \wedge N_{24} \rightarrow N_{1234}$$

Non-transitivity scenarios, whereby host A can connect to host B and host B can connect to host C but host A cannot connect to host C, and firewall scenarios, whereby host A can connect to host B but host B cannot connect to host A, should also be described in the model.

The CCPM relies on the global slots view, provided by the CCTM, to decide how to cut the network. Starting from the network variation model, a network variation script is generated. While executing the network variation script, the CCTM sends commands to the Network Partitioning Agents (NPA) running on the emulator nodes.

The NPAs are responsible of enforcing the split, merge, and link failure commands received from the CCPM, by installing traffic filtering rules in the operating system kernel. Therefore, in order to minimise the number of filtering rules, the easiest way to split the network is by IP address space.

Currently we deployed ModelNet with a full-mesh target topology. However, in the case of deploying a complete transit-stub topology, for the sake of realism, it would make sense to cut the network on a transit-transit link. This would be accomplished by setting the packet drop rate on one or more transit-transit links to 100%.

3.4 Statistics Monitoring, Aggregation and Caching Module

The Agent StatsMon Module (ASM) is in charge of collecting, aggregating and caching statistics from all virtual nodes running on the same machine. How statistics are defined and the aggregation policies are discussed in Section 4.1. Suffice to say here is that we have three types of information: pushed state, pulled state and notifications. Hence, at the StatsMon layer, information flows

both ways between the agents and the control center.

The ASM exports an interface to the DSUT nodes, allowing them to publish pushed state and install state update and method call notification handlers, upon being launched. The ASM builds a state aggregation structure and also pushes it to the Control Center StatsMon Module (CCSM). The CCSM collects this structure from all ASMs and builds a global state aggregation structure which it makes available to the Visualiser.

Pushed state is immediately updated in the ASM, as soon as their value changes in the DSUT nodes. The ASM aggregates the recently received stat updates with the state it keeps in its aggregation cache. Then, the ASM pushes the cache updates forward to the CCSM. The CCSM aggregates the received updates with the state it keeps in its aggregation cache, keeping a consistent global view of the observed DSUT state. As both the ASM and the CCSM cache pushed data, only the differences need to be communicated.

Pulled state is not updated in the ASM when it changes. Instead, the DSUT nodes register getters that can be called by the ASM to retrieve the state. Pulled state retrieval is triggered by the Visualiser or by an external GODS client. It asks the CCSM for the state which in turn asks the ASMs which in turn call the DSUT getters. Pulled state is not cached but it can be aggregated.

Method call notifications are sent from the DSUT nodes to the ASM, and the ASM forwards them to the CCSM which logs them in the database and reports them to the Visualiser or to an external GODS client. Notifications are used for the DSUT nodes to report internal events like the receipt of a message or failure detection, and allow the visualisation of distributed algorithms execution. Notifications are not cached and are not aggregated.

3.5 Operations Module

The operations layer provides functionality for invoking DSUT application operations. The Agent Operations Module (AOM) exports an interface to the DSUT nodes, allowing them to publish callable operations. The AOM collects all the DSUT callable operations and reports them to the CCOM. The CCOM provides a view of all DSUT callable operations to the Visualiser or to an external GODS client, which can then trigger operation calls through the CCOM and the AOMs.

The CCOM provides functionality for implementing operation invocation models. An operation invocation model can be specified either declaratively, in terms of operation types and operation rates, or through an operations scripting

language. The operation invocation model implementation relies on the global DSUT nodes view, provided by the CCTM, to decide which DSUT nodes to issue operations on. Starting from an operation invocation model, an operation invocation script is generated. This script is executed by sending operations commands to the AOMs.

The AOM is responsible of invoking operations as commanded by the CCOM, timing each invoked operation and reporting the registered times to the CCOM. The CCOM collects the timings and reports them to the Visualiser or to an external GODS client, for single operation calls, or stores them in a database, when the operations are invoked from an operation invocation script, part of an automated experiment.

3.6 Automation Module

The Automation Module (AM) provides functionality for devising complex experiments with specific churn models, network partitioning models and operation invocation models. It allows for running the same experiment multiple times for a statistic evaluation, composing experiments and parameterising experiments.

It is still an open issue to design a language for expressing churn models, network partitioning models and operation invocation models, for composing basic experiments into more complex ones, for parameterising experiments, for specifying data collected in the experiment and so on.

The AM is responsible for batch scheduling of automated experiments carried out as part of performance tuning experiments, executing regression test suites, or benchmarking experiments.

3.7 Bandwidth Accounting Module

The Bandwidth Accounting Agent (BAA) runs on the emulator node and is in charge with measuring bandwidth consumption by the DSUT. Because DSUT traffic is routed to the emulator machines and GODS control traffic is routed to the Control Center machine, by placing the BAAs on the emulator machines, DSUT bandwidth consumption can be measured accurately.

The BAAs measure DSUT bandwidth usage by installing traffic filtering rules in the operating system kernel, for logging the size of every forwarded packet having source and destination IPs in the 10.0.0.0/8 address space.

In the case of multiple emulator machines, there is a BAA running on each emulator. To avoid duplicate measurement, packet size is only recorded when it exits the core, that is, when routed to an edge machine.

The BAA can continuously report bandwidth usage, or it can be turned on and off by the CCCM or the CCOM, to measure bandwidth used by join/leave/fail protocols and operations respectively. To some extent, bandwidth consumed by DSUT operations and bandwidth consumed by DSUT correction algorithms triggered by churn can be distingueshed using the timestamps of the operations and the timestamps of the churn events, respectively.

4 Statistics and Notifications

The purpose of GODS is to put the DSUT evaluator in the front row. She should be able to easily observe DSUT internal events or internal state, be it a DSUT node's routing table or state variables of a specific algorithm.

We have briefly sketched the nature of statistics and notifications in Section 3.4. Collected statistics offer global knowledge about the DSUT internal state. Notifications offer global knowledge about DSUT internal events. Let us enter the details of statistics and notifications collection, now.

4.1 Statistics

In order to capture DSUT internal state we need statistics of a few basic data types: `integer`, `long`, `double`, `string` and probably two composite data types: `set` and `structure`, for unstructured and structured collections, respectively.

In order to cope with the large scale of DSUT applications, which can run on thousands of virtual nodes, and to offer a compact view of the DSUTat the same time, we need to aggregate statistics. Stats aggregation is done on two levels: at the agent level and again at the control center level.

Each stat has a name, a type and a value. Statistics of the same name and type are aggregated across local virtual nodes at the agent level and across all agents at the control center level.

We envision two aggregation techniques. For the `integer`, `long` and `string` stats, the aggregated view could be the number of occurrences of a distinct value across all the virtual nodes. For instance, if 10 local VNs present values: 2, 3, 2, 1, 4, 4, 5, 2, 3, 2 to the ASM, the ASM aggregates them as {(1, 1), (2, 4), (3, 2), (4, 2), (5, 1)}. On the next level, if two ASMs present {(1, 1), (2, 4), (3, 2), (4, 2), (5, 1)} and {(1, 3), (2, 3), (3, 1), (5, 2), (6, 4)} to the CCSM, the CCSM aggregates them as {(1, 4), (2, 7), (3, 3), (4, 2), (5, 3), (6, 4)}.

For the `integer`, `long` and `double` stats, the aggregated view could be the number of values occurring in a range. For instance, if the 10 local VNs present values: 1.3, 1.4, 0.5, 3.7, 4.2, 2.5, 2.3, 2.4, 3.2, 3.5 to the ASM, the ASM aggregates them as {(0.0-1.0, 1), (1.0-2.0, 2), (2.0-3.0, 3), (3.0-4.0, 3), (4.0-5.0, 1)}. CCSM aggregation follows similarly.

It is still an open issue whether `set` and `structure` stats would need some form of aggregation. Not all DSUT state is suitable to aggregation, for instance, DSUT routing tables. Instead, it may be suitable to caching. DSUT nodes' rout-

ing tables can be cached at both the ASM and CCSM levels. Thus, the CCSM offers a global view of the DSUT topology, and because only changes need to be communicated, DSUT nodes' routing table updates can quickly be reflected in the global view.

As we mentioned before, stats can be either pushed or pulled. Either way, we need to get up to date stats, that is, the DSUT internal state should not be far ahead of the reported view. Therefore, both DSUT internal code and the ASM should access the same data or memory locations.

One possibility is to have all DSUT observed state as instances of an *ObservedVariable* class. This class contains the name, type, value and push/pull flags for one stat. The value is accessed through getters and setters both by the DSUT code and by the ASM. Pulled stats are retrieved by the ASM through getters. For pushed stats, the setter checks whether the new value is different from the old value. If that is the case, an update is sent to the ASM. The ASM updates its cached state and sends a cache update forward to the CCSM. The CCSM updates its cached state and triggers the Visualiser to update its view.

Using *ObservedVariables* requires changes to be made to the DSUT source code which is prone to introducing subtle bugs. Another possibility is to implement the *ObservedVariable* functionality using the debugging mechnisms offered by JVMTI and accessing them through JMX. No changes need to be made to the source code. The observed DSUT state, as fully qualified member names (package.class.member), together with push/pull flags, are specified in an Extensible Markup Language (XML) stats descriptor file. This XML file is read by the ATM upon initialisation and the corresponding watches are installed into the Java virtual machines at DSUT node launch time. This process is described in Section 6.3.4.

One may think about *heisenbugs*, the kind of bugs that do not reproduce under the debugger, often synchronisation errors, and wonder whether instrumenting the Java Virtual Machine (JVM) may lead to similar situations. Instrumenting the JVM is the least instrusive and lightest way of instrumenting a Java application. Even if *heisenbugs* are instroduced this way, that is out of the scope of GODS.

Both using *ObservedVariables* and bytecode injection are suitable for DSUT applications written in the Java programming language. For DSUT applications written in other languages, source code modifications are needed and a special application interface to the ASM.

Because the DSUT traffic is subject to the ModelNet target topology latencies, and hence a bit delayed, and the GODS traffic is not, it is likely that DSUT state

updates, made while executing distributed algorithms, are observed in real time in the Control Center.

4.2 Notifications

Notifications are a means of live reporting of DSUT internal events to the Control Center, allowing the user to monitor the execution of DSUT distributed algorithms. In particular they can be used to report the receipt of select messages, but in principle they could be used to report any kind of event, not necessarily related to receiving a message. For instance, actions that the DSUT node decided to take as a result of running a local periodic algorithm, failure detection events, select method calls, or events that may or may not be triggered by a message receipt, that is, events that are not definitively implied by a message receipt, could all be reported.

Notifications should be described in an XML notifications descriptor file. This XML file is read both by the ASM and the CCSM upon initialisation. Each notification type has a unique identifier allowing it to be distinguished in the CCSM and properly shown in the Visualiser, maybe using colour coding.

If we merely want to report message receipts or method calls, we can easily use the JVMTI to watch and a method call and trigger the notification. The respective fully qualified method names and message handlers should be specified in the XML notifications descriptor file, for the ASM to know how to install the watch. If we want to report intra-method events other that state updates, changing the DSUT source code is required. Probably the safest choice is to refactor the code so that the respective events are extracted in their own methods.

The communication between the DSUT nodes and the ASM for both pushed stats and notifications, is done through the JMX protocol. Once again, for DSUT applications written in programming languages other than Java, source code refactoring and a special application interface to the ASM is needed.

5 Use Cases

We envision two major usage scenarios for GODS, namely *interactive control and monitoring* a DSUT and running batch *automated experiments*. The user would first experience using GODS interactively, driving it from the Visualiser, and after getting used to its features she would start setting up automated experiments, thereafter driving GODS from external automation tools.

First, let us review the basic mechanisms offered by GODS and then we look at how they can be leveraged into complex evaluation experiments. GODS provides:

- a global view of slots topology including all pairs latencies and bandwidths;
- controlled launch, shutdown and kill of DSUT nodes;
- partitioning the emulated network and changing network links properties;
- global knowledge about published DSUT parameters, DSUT internal state, DSUT topology and DSUT exchanged messages;
- timed DSUT operations invocation;
- DSUT traffic bandwidth measurement;

We strive to keep GODS independent of DSUT as much as possible. However, for the simplicity of presentation, the following use cases refer to evaluating a specific DSUT, DKS[9], a large-scale distributed hash table.

5.1 Interactive Control and Monitoring

In interactive control and monitoring, the user is presented with the slots topology. She can select slots on which to launch DKS nodes. She can manually assign DKS IDs to slots or use the Secure Hash Algorithm (SHA-1) hashes of the slots' IPs. She can also check which slot's IP hashes closer to a given identifier.

The Visualiser should draw the DKS ring topology. DKS nodes should be selectable. When a DKS node is selected, its neighbours are highlighted and its fingers are drawn. On the selected DKS node, the user can issue lookup operations. Using notifications for lookup messages, the user can visually inspect a lookup path. The time it took for the lookup to return is reported to the user, as well as the latency between the lookup initiator node and the lookup destination node.

The user can activate bandwidth accounting and issue another lookup. Besides the lookup time and the latency between the end nodes, the user is also presented with the number of messages and the bandwidth consumed by the lookup.

The user can launch a new DKS node and using notifications for the messages exchanged in the local atomic join protocol, the user can visualise the execution of the local atomic join. Pushed stats for the state variables in the atomic join protocol, can also be used. The Visualiser should allow colour coding for message notifications and different state changes. The drawn DKS nodes should change colour according to the received notification so the user can easily visually inspect the execution of the algorithm. Finally, the user is presented with the number of messages and bandwidth consumed by the local atomic join protocol.

The user can kill a DKS node and watch how its neighbours detect the failure and initiate a topology maintenance protocol. Inspecting how the DKS nodes change colour, the user is able to see all the nodes affected by the maintenance protocol. The user can also see how their routing tables are updated. Again, the user is presented with the total number of messages and bandwidth consumed by the topology maintenance protocol.

A DKS plug-in to the Control Center, should report at all times, the number of stale fingers in the system. This is possible because the plug-in has global knowledge about the topology, and can check whether all the fingers point where they should. This is possible even when using Proximity Neighbour Selection as global knowledge about all pairs latencies is available. If the PNS scheme relies upon Vivaldi [6, 7], this would also reveal the accuracy of the latency prediction given by the synthetic coordinates.

The user can issue a network partitioning and observe how DKS reacts to that. The Visualiser should be able to draw two or more rings if new DKS rings are formed as an effect of the partitioning. A storm of topology maintenance messages is expected after a network partitioning. The user should be presented with the total number of these messages and the total bandwidth consumed. The user can merge the network partitions and again observe the DKS behaviour.

The user should be able to record her actions, replay them and combine them into automated experiments.

5.2 Automated Experiments

Besides combining recorded actions, the user should be able to specify an operation invocation model, a churn event model and a network variation model for an automated experiment. The user can also define groups of nodes with different models. Groups of nodes running a modified version of the application can be defined, to observe how the application copes with Byzantine behaviour. Starting from these models, experiment scripts are generated, so that an experiment can be later reproduced.

The user also specifies the measurements collected in an automated experiment. For some experiments meant to be executed as part of a regression test suite, for performance tuning or benchmarking, some input parameters with ranges of possible values need to be specified. The Automation Module will then, schedule this experiment one time for each different input parameter value. When an automated experiment is started, collected measurements are automatically stored in a database. Notification email should be sent upon completion of an automated experiment or in the case of an experiment failure.

The Visualiser should be able to hook into a running automated experiment and allow its user to monitor the DSUT activity. This is particularly useful for inspecting the status of an experiment running for a couple of hours or even days.

One drawback of using a real-world setup with real-world latencies and actual DSUT code running, is that it leads to lengthy experiments. For instance, let's suppose we run 2000 DSUT nodes and we want to measure the DSUT stretch by issuing all pairs lookups and pings. We need to execute about 4 million lookups and 4 million pings. If we issue one lookup and one ping every second we need around 46 days. So we need to issue around 50 lookups and 50 pings per second to finish in one day. Should the ModelNet emulator traffic capacity become a bottleneck, we could scale it up by adding more emulators. In this scenario, the CCOM needs to balance the issued operations evenly across the emulator machines.

6 Implementation Details

In this section we explain the implementation details of GODS. First, we explain the core concepts of GODS, then we explain its software architecture, followed by extension mechanisms. Finally, we explain the technologies used for each part and the motivation for our choices.

The *control center* and *agents* are collectively referred to as *components* of GODS in this section, while, the individual *modules* of each of these as described in the previous Section are still referred as *modules*.

6.1 Core Concepts

GODS has been developed entirely as an event-driven system. It functions as a set of *modules* interacting through *events*. The *modules* subscribe for and publish the *events*. In the following sections we explain the core concepts of GODS, before we delve into the implementation details. Then, we would explain the extension mechanisms of GODS.

6.1.1 Event

An *event* in GODS besides representing the events in traditional approaches also represents asynchronous requests and replies for interaction among *modules* as well as the *control center*, *agents* and *visualiser*. Events are the only way for cooperation among internal *modules* of the *components*. The *components* of GODS however, can interact by alternative means which are explained later in this chapter.

The events are subscribed and triggered by *modules*. An *event* can be subscribed by multiple *modules* specifying their respective *event handlers* and can also be triggered by multiple *modules*.

The events are prioritised, and can be grouped with similar *events* into *event topics*. The grouping of *events* into *event topics* is to facilitate subscription of event groups as a whole. This feature is currently however not being used.

6.1.2 Module

A *module* in GODS represents an object encapsulating state, to be modified only by a set of *events* to which the *module* subscribes. Each *module* has its own thread and a blocking priority based queue of *event handlers*. A *module* is thus a unit of execution in GODS and can ideally be loaded or unloaded independently of

other modules. A *module* subscribes to *events* of interest just before starting its thread.

As described earlier a *module* subscribes to the *events* along with a specific *event handler*. When an *event* is triggered the corresponding *event handler* is enqueued by the events broker in the *module's* queue of *event handlers*. An *event handler* is only associated with a single *module*, that is it can only modify the data in a single *module*.

This mechanism serialises the access to a *module's* data structures and since *modules* cannot be accessed by any other means there is no need for explicit synchronisation of a *module's* data structures. This approach however has the disadvantage of some threads being unutilised or underutilised, in case of a *module* having less or no workload.

6.1.3 Event Handler

An *event handler* as opposed to its name is *not* a method, but an object encapsulating the *event* to be handled, an instance of the *module* responsible for executing the *event handler* and a method *handle* which actually handles the *event*. *Event handlers* are prioritised based on the priorities of the *event* they handle.

The motivation for modelling the *event handler* as an object rather than methods of modules would be explained later in this chapter.

6.1.4 Subscriptions Registry

The *subscription registry* simply keeps a list of all *subscriptions* for an *event type* in a hash map. It provides a simple interface to its clients through the methods *addSubscription* and *getSubscriptions*. The *addSubscription* method adds a *subscription* to the list of previous subscriptions against an *event* type, and the *getSubscriptions* method returns the list of all subscriptions for an *event type*.

A *subscription* is a 3-tuple, containing the *event type*, a reference to the *module instance* that is interested in the particular *event type* and the *event handler type* to handle the *event*.

The *subscription registry* is not an event broker itself, but only a utility for GODS *components* which act as event brokers for their internal modules.

6.1.5 Task

A *task* represents an execution request from *control center* to an *agent* that is to be executed synchronously. This is in contrast to an *event* which is enqueued and then scheduled for a *module* subscribing to it. This is however not being used currently.

6.2 Event Handling

The *control center* and *agents* also act as event brokers for their respective *modules*. All *modules* register their interest in *events* as soon as they are started, with the *component* to which they belong. The *component* stores these *subscriptions* in a *subscription registry*.

When an *event* is triggered by a *module* or rather by an *event handler*, it is enqueued in the *events* queue of the *component* to which the *module* belongs. The *component* consumes these *events* from the head of a priority queue. For every *event*, it searches in its *subscription registry* and for the list of *subscriptions* for the *event type*. For each *subscription* in the list it instantiates the corresponding *event handler*, initialises it and enqueues it in the subscribing *module*.

Event handling between *components* that is the *control center* and *agents* is explained in detail later in this chapter.

6.3 Software Architecture

In this section we explain the internals of each of the GODS component and the interfaces they expose to each other.

6.3.1 Overview

The *control center* and the *agent* are modelled as singletons [11] that is there exist only one instance of them in a JVM.

The applications either update their state to the local *agent* or the *agent* pulls the data from application nodes running on the same physical node as the *agent*. Similarly, an *agent* can either update the *control center* or the *control center* can request for an update from any of the *agents*. The components thus have to provide interfaces for both type of scenarios.

6.3.2 Control Center

The Control Center orchestrates the execution of experiments. It can be controlled manually from the *visualiser* for interactive experiments but it also needs to be controlled by external GODS clients for automated experiments. Hence, we need to provide interoperability with various automation tools.

The Control Center is implemented as a set of cooperating *modules,* as described in Chapter 3 which interact amongst themselves through *events*. Since, the *control center* also acts as an event broker between its *modules*, it also has to provide an interface for the *modules* to subscribe and trigger *events*. This functionality is exposed by the *ControlCenterInterface* with the *subscribe* and *enqueue* methods.

The functionality that is required by the *agents* or for control and monitoring of the DSUT is provided through the *ControlCenterRemoteInterface* and is implemented by the *ControlCenterRemote* object. Although, *ControlCenterRemote* is modelled as a separate entity for reducing responsibility on the *control center* however, it is encapsulated by the *control center*. When, the *ControlCenterRemote* receives an *event* from an *agent,* it is simply enqueued in the *control center*, as that is also to be dealt by one of the *modules*.

When, GODS is started first of all the configuration properties are loaded and all the modules are started. A *BootEvent* which contains the command line arguments is then enqueued in the *control center* and it is then started. The *BootEvent* is processed by the *control center*'s *DeploymentModule* which is explained in section 6.3.2.

Next, we discuss the *control center*'s internal modules.

DeploymentModule The *control center*'s *DeploymentModule* currently handles the process from booting up of GODS to the state when GODS is ready for an experiment. As described earlier in the section that the *event handler* for the *BootEvent* is enqueued in the *DeploymentModule* at startup. The handler deploys *agents* code on the physical machines, starts the *agents* and changes GODS state to *JOINING.*

The *DeploymentModule* then waits for a *JoinedEvent* from each of its *agents.* This is a synchronisation point or barrier. After receiving *JoinedEvent* from all *agents*, the *control center* determines the requirement for the number of slots (virtual nodes) required for the experiment as specified in its configuration. It then equally distributes the number of nodes on each machine and sends the range of

slot ids to each *agent* in the *PrepareEvent*. It then waits for a response from each of the *agents*.

As the *agents* are ready for the experiment, they start sending in *ReadyEvent* to the *control center*. After receiving a *ReadyEvent* from each of the *agents*, the *control center* sorts all the slots received from each *agent* and sends an update to the *visualiser*. The physical location of slots is thus transparent to the user.

ChurnModule The *ChurnModule* of the *control center* currently handles the *launch*, *stop* and *kill* application events for the *control centerİ*n order to *launch* an application the *ChurnModule* handles a *LaunchApplicationEvent* from the *control center*. It first determines the physical location of the slots on which the DSUT is to be launched. In case slots lie on different machines, the *ChurnModule's event handler* for *LaunchApplicationEvent* breaks the *LaunchApplicationEvent* into separate *LaunchApplicationEvents* one for each machine. The *control center* receives *ApplicationLaunchedEvent* from each of the *agents* to which the slots belonged. The *control center* then updates the *visualiser*.

A similar procedure is followed for the *KillApplicationEvent* and *StopApplicationEvent*.

TopologyModule The *TopologyModule* is to handle the events related to changes in topology. The topology is however static currently, and so it is not handling any *events*. It however, keeps the global information on all slots, and references to all *agents*.

6.3.3 Agent

An *agent* is responsible for executing the *control center* requests and updating status of all application nodes on a single machine, as discussed in Section 3. It needs to provide an interface to the *control center*as well as the DSUT. Additionally, it needs to provide an interface for its internal modules for *events* handling.

The interface for *agents* internal *modules* is exposed by the *AgentInterface*. The interface exposes the methods *subscribe* and *enqueue* for subscription and triggering of *events* respectively.

The interface for the *agent's* interaction with *control center* and DSUT is exposed through *AgentRemoteInterface*. This interface exposes two methods for interaction with DSUT that is *notifyDSUTEvent* and *updateDSUTState* for push and pull updates respectively. Interactions with the *control center* is carried out with

notifyEvent and *executeTask* methods. While, the former serves the purpose of asynchronous communication, the latter is used for synchronous execution of a task from the *control center*. The *executeTask* method is not in use currently.

Agents are started by the *control center* on each of the physical machines as part of processing the *BootEvent*. An *agent* loads its configuration properties, starts its own modules and then enqueues the *AgentBootEvent* in its queue. This starts the *deployment* process as explained in section.

AgentDeploymentModule The *AgentBootEvent* is the first *event* handled by the *AgentDeploymentModule*. In response to this *event* it only sends *JoinedEvent* to the *control center*.

As an *agent* receives *PrepareEvent* from the *control center*, it executes the *PrepareEvent* handler, which gathers the information of all slots on the machine as that of the *agent*, assigns them *slot ids* within the range specified by the *control center* in the *PrepareEvent* and sends this information to the *control center* in a *ReadyEvent*.

AgentChurnModule The *AgentChurnModule* handles the churn events from the *control center*. As it receives a *LaunchApplicationEvent* from the *control center* it launches the application each time taking a different set of arguments if provided, collects their *process ids* against their *slot ids* and sends this information to the *control center* in *ApplicationLaunchedEvent*. The *control center* then updates its data and sends an update event to the *visualiser* as well.

The *KillApplicationEvent* and *StopApplicationEvent* are handled similarly, and if successful an *ApplicationKilledEvent* and *ApplicationStoppedEvent* is sent to the *control center* respectively.

If any launching, killing or stopping operation is unsuccessful in a churn event, the experiment is completely aborted.

AgentTopologyModule The *AgentTopologyModule* does not handle any *events* currently, due to static topology, however it keeps information of all slots on the same physical machine.

6.3.4 Application Interface

GODS needs to interact with the DSUT application bi-directionally for issuing operations and receiving state update and method call notifications respectively. Cooperation from the application is needed for triggering certain events and

also for executing the operations issued by GODS. Therefore an application's behaviour needs to be changed in order to offer this kind of cooperation.

Changing the application's source code, to make it GODS aware, is one solution but we believe that this is prone to introducing bugs in the code, on the one hand, and it would make the adoption of GODS less attractive, on the other hand. Therefore, we prefer to leave the application's source code intact and to change the application's behaviour by instrumenting its executing virtual machine. Of course this is possible only for applications written in the Java programming language. For applications written in other programming languages, the application source code needs to be changed to implement the GODS application interface.

We use a JVMTI [33] agent to watch and trigger write accesses to certain fields in the application and to notify entry and exit points of certain methods. These notifications are sent to an MBean [23] server local to the JVM. This MBean server will forward the notifications to the GODS agent through the JMX [24] protocol. Hence, the GODS agents are JMX compliant clients. The GODS agents can also send operation invocation commands to the JVMTI agent, again through the JMX protocol.

At initialisation time, the GODS agent reads the pushed state and method call notifications descriptor files, and commands the JVMTI agents in all application instances to install the necessary watches. The same watches are installed in every virtual machine of a newly launched application instance.

Using JVMTI permits dynamic installation and removal of watches, allowing the user to watch new fields and methods, specified at experiment run-time. Hence, GODS behaves as a true debugger for distributed applications.

6.4 Extension Mechanisms

There can be greatly varying requirements for testing of different DSUTs. Taking this into consideration, there are various interfaces are provided in GODS. This section discusses in detail the various requirements that might arise and how they can be accomplished.

Events can be added through either the *Event* interface or the *AbstractEvent* class in order to observe behaviour specific to a DSUT. Moreover, for any added event, an *EventHandler* should be written and the pair should be added to a *module's* subscription list.

It might however be a case where an *event*only needs to be handled differently. For this case the abstract *EventHandler* class should be extended. Two

different cases can be considered here, one where an *event*needs to be handled in a completely different way than it is being currently handled, or adding some functionality to the current *EventHandler*. For the former case an *EventHandler* should be extended directly from the *EventHandler* class, while for the latter a new *EventHandler* can be extended from the specific *event handler*.

Moreover, different DSUTs require different arguments at command line. In case a single DSUT is to be launched that can be easily achieved in the *interactive control and monitoring* mode from the *visualiser*. However, for launching multiple nodes, arguments to an application can be generated. For this purpose the *ArgumentGenerator* interface has been provided. The *ArgumentGenerator* can be provided an optional configuration file, to generate different types of arguments. For example, in case a ring based DHT is being tested, we might like the application nodes to be either uniformly distributed over the ring, randomly distributed or in a sequence.

6.5 Technologies Used

The GODS infrastructure is written in the Java programming language. Java was our foremost choice considering the requirement of portability. For the cooperation between agents and the Control Center we make use of Remote Method Invocation (RMI) [32]. RMI was chosen because on top of its inherent extensibility from Java, it provides great ease in extending the protocols for interaction amongst remote components.

For instrumenting the DSUT application we make use of the JVMTI [33]. The other alternative was developing our own instrumentation component using Byte Code Engineering Library (BCEL) [1] or the high level instrumentation API Javassist [17]. The motivation for using JVMTI has already been explained in Section 6.3.4. Java reflection [13] classes are used to aid an evaluator to see which of the available state variables would she like to observe in a distributed system.

As a generic testing and debugging platform for distributed systems, GODS requires configurations and scripts for testing specific distributed software. The configuration files are in XML due to its extensibility and the availability of tools available for reading and validating these. Scripts are currently written for the bash [2] environment.

The Visualiser is built around the Java Universal Network/Graph (JUNG) [10]

[2]Bash is an sh-compatible shell, or command language interpreter. http://www.gnu.org/software/bash/

framework, for extensible topology visualisation. We use a MySQL database through Java Database Connectivity (JDBC), for storing execution statistics and event logs.

We plan to add bindings to popular programming and scripting languages (C, C++, Python, Perl, Tcl, Ruby), to allow the scheduling and automatic execution of various experiments.

7 Contributions

In this section we briefly describe the contributions of this work, that are controllability and reproducibility of experiments, emulating network partitioning, bandwidth accounting of application nodes on Modelnet, emulating churn and the behaviour of users of large-scale distributed systems (load injection).

7.1 Controllability and Reproducibility

Modelnet provides a good foundation for testing of large-scale distributed systems on a realistic wide-area network, this however, comes with great complexity in management and evaluation of the deployed system. GODS enhances Modelnet by providing a mechanism for having full control over the experiments and being able to reproduce them. It provides global knowledge of *select system state* in the form of *aggregated statistics* and a global view of the *system topology* emulated by Modelnet.

7.2 Emulating Churn

GODS sits over Modelnet to orchestrate churn in a large-scale distributed system. This is useful to observe the behaviour of the distributed system under test when individual nodes are joining, leaving and failing. For example, DHTs based on ring based overlay networks are said to be highly resilient to churn [31], various DHTs can be tested under churn to evaluate their performance.

7.3 Emulating Network Partitioning

We enhance the Modelnet emulated network to provide *network partitioning*. Our tool adds dynamism to the emulated environment in the form of dynamically partitioning the network, besides modifying network link characteristics. This is useful for example to observe the behaviour of a large-scale overlay network in case of a network partition.

7.4 Bandwidth Accounting

GODS correlates traffic to certain operations and modules of the application to account for the bandwidth usage by a distributed application. This helps to ob-

serve the bandwidth consumed by different algorithms or their implementations by a distributed application.

7.5 Emulating User Behaviour

Besides, emulating churn and network performance anomalies we also provide emulation of user-behaviour by controlling the application through probing into the application. This can be useful in evaluating a distributed application with models of user-behaviour for example to observe what happens as a user sends a query in a distributed application.

8 Related Work

We do not know of any other tool that provides the complete functionality offered by GODS, however we are aware of some related work that overlaps parts of our goals.

8.1 Application Control and Monitoring Environment (ACME)

Application Control and Monitoring Environment (ACME) is a scalable, flexible infrastructure that can perform the tasks of benchmarking, testing, system management, scalability and robustness of large-scale distributed systems [28]. ACME extends the scope of emulation environments such as Emulab [35] and Modelnet [25] by adding a framework to automatically apply workloads, and under faults and failures to measure complete distributed services, based on a user's specification [27].

ACME is built over the metaphors of sensors and actuators. The sensor metaphor is used to describe the mechanism for monitoring distributed systems and actuator metaphor is used to describe the mechanism for controlling distributed systems being evaluated. It has two principal parts, a distributed query processor (ISING) that queries Internet data streams and then aggregates the results as they travel back through a tree-based overlay network. The second part is an event triggering engine ENTRIE, that invokes the actuators according to user-defined criteria, such as killing processes, during a robustness benchmark [28].

ACME has been used to monitor and control two structured peer-to-peer overlay networks, Tapestry[37] and Chord[26] on Emulab[35].

8.2 Distributed Automatic Regression Testing (DART)

DART [5] is a framework for distributed automated regression testing of large-scale network applications. It provides distributed application developers with a set of primitives for writing distributed tests and a runtime that executes distributed test in a fast and efficient manner over a network of nodes. Besides, the programming environment and test script execution DART also provides execution of multi-node commands, fault injection and performance anomaly injection.

DART supports automated execution of a suite of distributed tests, where each test involves (1) setting up or reusing a network of nodes to test the application on, (2) setting up the test by distributing code and data to all nodes,

(3) executing and controlling the distributed test, and finally (4) collecting the results of the test from all nodes and evaluating them.

To automate, the aforementioned tasks, DART relies on its components which are *network topology*, *remote execution and file transfer*, *scripting and programming environment*, *preprocessing, execution and postprocessing*, *fault injection* and *performance anomaly injection*.

ACME [28] and DART [5] though related to GODS do not have any mechanisms for probing into the application under test itself e.g., invoking particular operations or observing selected variables. This limits their evaluation of the application only to the application environment and not user-behaviour.

Moreover, these platforms are targeted towards PlanetLab [29] or Emulab [35] which also limits the controllability and reproducibility of the experiments.

The following works are also related to GODS in their vision, and have their own contributions for solving the problem. We, thus explain them briefly and discuss our differences with them.

8.3 WiDS

WiDS [20, 19] is an ecosystem of technologies for optimising the development and testing process for distributed systems, currently developed at Microsoft Research Asia. GODS shares part of the WiDS vision, but there are some differences between the two. WiDS allows developers to test their distributed systems by either simulation or emulation. WiDS also allows debugging of distributed systems, by running all the protocol instances in a single process address space.

WiDS puts more weight on assisting the user throughout the development process, by enabling distributed algorithms modelling and code generation. In contrast, GODS focuses on the evaluation of readily available implementations.

The vision that a single code base should be used for development and evaluation, by either simulation or emulation, is shared, but while one could use GODS for evaluation of an existing implementation without changing it, in order to use WiDS, the developer needs to write a WiDS driver and adjust her implementation to be driven by that driver. The distributed application code has to be encapsulated in WiDS objects, which represent protocol instances.

The WiDS objects can be run either in simulation mode, in emulation mode or in debugging mode. Simulation is either local and all WiDS objects are run part of the same address space, or distributed across a set of cluster machines. In emulation mode each WiDS object is run in its own process and these are distributed

across the cluster machines. The traffic generated in emulation mode is routed to a network emulator, just like in GODS. In debugging mode, WiDS objects are run locally within a single process space, under the control of a debugger.

WiDS users work with the same code base across different development stages and link it to appropriate libraries accordingly. WiDS provides multiple runtime environments for simulation, emulation and debugging respectively. The price to pay for this versatility is that source code has to be WiDS targeted.

In describing their experience with WiDS [20], the authors agree that protocol bugs that are more difficult to find, only surface in emulation mode. Because event handling can take arbitrarily long in network execution mode, as opposed to one (simulated) clock tick in simulation, the sequence of events can differ in unexpected ways, making it difficult to discover those bugs in the simulation environment. For this reason, they enhanced WiDS with a module for logging execution in emulation mode. The logs are then used to drive the execution in debugging mode within a single process. We believe that GODS logging and deterministic step-by-step replay achieves the same functionality.

Finally, WiDS is a proprietary tool while GODS is open source.

8.4 Liblog

Liblog [12] is a tool that enables replay debugging for distributed C/C++ applications. When running the distributed application, the Liblog library is preloaded and all activity, including exchanged messages, thread scheduling and signal handling is logged.

Each process logs its own activity locally. Post-mortem, all logs are fetched to a central machine where an interesting subset of processes are replayed in step-by-step debugging mode. Liblog integrates GDB into the replay mechanism for simultaneous source-level debugging of multiple processes. Liblog uses Lamport timestamps in exchanged messages to ensure that replay is consistent with the causal order of events.

There are a number of challenges that Liblog overcomes. Because, no change to the debugged application is made, Liblog transparently adds timestamps to messages and removes them before delivering the messages to the application. As the replay is done on a different machine, in the case where processes access the file system, Liblog has to recreate the same file system view for the replayed processes.

We believe there are some limits to the scalability of Liblog given the fact

that all exchanged messages have to be logged, and all the processes have to be replayed on a single machine.

Liblog strives on deterministic replaying of the execution, by logging and reproducing thread scheduling, signal handling and exchanged messages, therefore reproducing race conditions and non-deterministic failures. GODS offers both replay of the execution (with no deterministic guarantees) and exact replay of the view of the execution, still guaranteeing that the execution view is causally consistent with the real execution. Liblog allows replay debugging of one uncontrolled execution at a time. In contrast, GODS allows for controlling the execution and thus enabling the preparation of corner tests for the application and subjecting the application to a suite of stress test executions for maximal coverage.

Next, we describe a tool that only provides monitoring of distributed systems, it is however worth mentioning here because of the similarity of the challenges they share with the problem we have targeted.

8.5 Ganglia

Ganglia [22] is a scalable distributed monitoring system for high performance computing systems such as clusters and Grids. It has been built to address the challenges in modern distributed systems such as reliability, scalability, heterogeneity, manageability and system evolution over time. Ganglia serves the purpose of monitoring distributed systems to detect software and hardware failures for quick repair. It also provides means for evaluation of interactions between different components of a distributed system by recording such interactions and gives a global view of the system.

Ganglia is based on a hierarchical design focusing on federation of clusters. It relies on a multi-cast based listen-announce protocol to monitor state within clusters and uses a tree of point-to-point connections amongst representative cluster nodes to federate clusters and aggregate their state. It incurs low overhead on the system being monitored, is robust, and has already been ported to very diverse types of distributed systems.

Ganglia has interesting properties and positive results for monitoring of distributed systems. It is however different from GODS that it is monitoring deployed systems rather than providing a test-bed for distributed systems. GODS vision besides monitoring is to be able to control distributed system over a fully manageable emulated network, hence providing means to evaluate system prop-

erties and aid in debugging of distributed systems software.

9 Users Guide

In this section we firstly discuss in detail about how to configure and run GODS. Later we discuss generating and running experiments through GODS.

9.1 Preliminaries

9.1.1 Enable Password Less Login

The machine running the GODS Control Center should have password less access to the remote cluster machines which will be running the application under test.

Generating RSA key pair for the Control Center machine, and adding its public key to authorized keys on each machine that hosts a virtual node.

9.1.2 Setup a Webserver

The GODS Control Center and Agents communicate using Java RMI. This requires class files to be available on web. Hence a webserver is required with class files shared on it.

9.2 Configuring GODS

9.2.1 GODS config file

The main config file for GODS is specified as a command line argument. Currently the *gods.config.file* in *config* folder is being used. The parameters required in the config file are described below:

1. **gods.slots** is the number of slots in the network model
2. **gods.net.model.dir** is the path to the network model directory. GODS will read the *machines* file from this directory to get the hosts and emulator machines.

9.2.2 Deploying Agents

The agents are deployed automatically using the *deploy-agent.sh* script when GODS is started. However, the following parameters are to be changed in the script.

1. **SOURCE_PATH** is the path to GODS class files.

2. **DESTINATION_PATH** is the path where GODS is to be deployed. It will be deployed on each physical machine that is a host in the Modelnet network.

3. **REMOTE_USER** is the user name assumed to be same for each host machine.

9.2.3 Configuring Agents

An Agent is started on each machine hosting the Modelnet virtual nodes by GODS automatically, using the *run-agent.sh* script in *scripts* folder. This script is passed a file *agent-setup-snusmumrik.sh* in the *scripts* folder for example which sets up the environment. The variables in the script are described below:

1. **AGENT_HOME** is the folder where agent has been deployed. All paths accessed from the Agent are relative to this path.

2. **JAVA_PATH** is the path to jre on remote machines.

3. **CLASSPATH** is the path to GODS class files on these machines.

4. **CODEBASE** is a web url to GODS class files. This is required by Java RMI.

5. **JAVA_POLICY** is the java policy file. The file *java.policy4* in the *javapolicies* folder can be used to start with. *java.policy4* does not have any security related restrictions.

6.

7. **CCHOSTNAME** is the hostname of the machine on which Control Center is running.

8. **REMOTE_USER** is the remote user being used to run GODS experiments.

9. **AGENT_CONFIG** is the config file describing parameters required by the Agent at runtime. Currently, *gods.agent.config.xml* in the *agent.config* folder is being used for this purpose. This does not require any changes.

10. **AGENT_LOG** is the config file for the logger being used by GODS. A sample logger configuration file is *agent.log4j.config* in the *agent.config* folder.

9.2.4 Deploying Application

The application under test is also deployed automatically by GODS using the deploy script for an application provided by evaluators. For the dummy application this is *deploydummy.sh* script in *dummy_app* folder provided in *utilities*. However, the following parameters are to be changed in the script.

1. **SOURCE_PATH** is the path to the application.

2. **DESTINATION_PATH** is the path where the application is to be deployed. It will be deployed on each physical machine that is a host in the Modelnet network.

3. **REMOTE_USER** is the user name assumed to be same for each host machine.

9.3 Running GODS

The *run-gods.sh* and *test-gods.sh* scripts in the *scripts* folder can be used to run GODS.

1. **test-gods.sh** can be used to run gods in test mode. It starts GODS through gods.tests.GodsTestSuite which performs any tests to be conducted on GODS itself. It will start GODS and then wait for any key input before it starts the tests.

2. **run-gods.sh** is used to run GODS in the normal mode, where *visualizer* can join in between and send commands to the Control Center.

Both of these scripts can have two parameters as arguments at command line. However at least one parameter is compulsory which is the file specifying environment variables. Details about this file are mentioned in this section.

The second argument is a boolean which takes in the value of *true* or *TRUE* and is false in all other cases. The default value is also *true*. If false the GODS boot up mechanism will not start Agents automatically. This option is for debugging initial startup or deployment problems.

9.3.1 Environment Setup File

gods-setup-snusmumrik or *gods-setup-korsakov* in folder *scripts* can be considered as sample setup files. Following describes each of the parameters in the file.

1. **GODS_HOME** is the full path of the folder which contains all files related to GODS. Besides being used inside the script GODS considers other paths relative to this path.

2. **CLASSPATH** is the path to GODS class files.

3. **HOSTNAME** is the hostname of the machine on which Control Center is to be started.

4. **CODEBASE** is a web url to the class files. This is required by Java RMI.

5. **POLICY** is the java policy file. The file *java.policy4* in the *javapolicies* folder can be used to start with. *java.policy4* does not have any security related restrictions.

6. **LOG_CONFIG** is the config file for the logger being used by GODS. A sample logger configuration file is *log4j.config* in the *config* folder.

7. **CONFIG_FILE** is the config file for GODS. It contains all the runtime parameters required by GODS. *gods.config.file* in *config* folder can be used as a sample config file. The details of this configuration file are described in detail in Configuration section.

9.3.2 Running Visualizer

The *run-visualizer* script in *scripts* folder can be used to run the Visualizer. This script requires the same enviornment setup file as required by the scripts for running GODS that is *gods-setup-snusmumrik.sh* can be used as one.

Currently, the *Visualizer* can join in a running Control Center. Some minor fixes are required so that it starts up GODS if it is launched before GODS itself.

9.4 Experiments

Conducting an experiment involves the following steps:

9.4.1 Generating an Experiment

To generate an experiment the script *generateExperiment.sh* in the *scripts* folder can be used. The following parameters in the script should be modified

1. **GODS_HOME** This is the path to the folder where GODS is deployed. Paths for the following variables are relative to this folder. In most cases changing only this variable will suffice.

2. **CLASSPATH** This includes the path to GODS class files as well as the *log4j* library.

3. **LOG_CONFIG** The logger configuration file. The *log4j.config* file in *config* folder can be used for this purpose.

4. **ARG_GENS_FILE** This is the file listing all classes that implement the *gods.churn.ArgumentGenerator* interface against their *Display Name*. This is by passed if a link to such a file is provided in command line arguments. More details about Argument Generators and their need is described in the Experiment Generation Parameters section.

The script takes in at most two arguments at command line. First is the Experiment Generation parameters file and an optional second argument is the file that lists all classes implementing gods.churn.ArgumentGenerator interface.

Experiment Generation Parameters GODS provides automatic experiment generation through *gods.experiment.ExperimentGenerator* class. It takes the experiment generation parameters as a file in the command line arguments. Two such files *dummy.exp.xml* and *dks.exp.xml* in the *expgens.config* folder can be viewed as examples of such files. The parameters in these files are described in detail in the following lines:

1. **experiment.name** All files related to this experiment will be stored in a folder with this name appended to the *experiments.dir.path*.

2. **experiment.dir.path** Path to the directory where a folder with *experiment.name* will be created.

3. **experiment.gen.class** Fully qualified Java Class name of the ChurnEventGenerator. There are currently two types of churn event generators in *gods.experiment.generators* package. One is a *DummyEventGenerator* which generates join, leave and fail events on virtual nodes chosen randomly with a random time interval between events. The other generator is the *JoinEventGenreator* which only generates application join events after with random time intervals between joins.

4. **totaltime.int** Total time for which the events for experiment are to be genrated in seconds

5. **seed.int Seed** The seed that will be used for all pseudo-random number generation.

6. **slots.int** Total number of virtual nodes in the deployed network model.

7. **network.model.path** Full path to the network model folder.

8. **app.home.path** Path to the home folder of the application that is to be evaluated.

9. **app.remote.home** Path of the folder on which this application should be deployed on the host machines (machines hosting the virtual nodes).

10. **app.deploy.script** Script to be used to deploy the application. This path is relative to the variable *app.home.path*

11. **app.init.script** Script that is to be run just before beginning an experiment to clean up all previous instance(s) of the application and clearing its logs.

12. **app.launch.script** Script to launch an application on encountering a join event. The path to this script must be relative to *app.remote.home*. The *app.deploy.script* should thus also deploy it on the host machnies.

13. **app.remote.log** Path to the file on which these application would log. The path is relative to *app.remote.home*.

    Currently all instances of an application on a single machine are assumed to log their joining, leaving and failing into a single log file. This is so because after each experiment GODS collects the logs from machines and compares it with an experiment valdation file, if all events happened within a certain threshold of actual time.

14. **app.arggen.displayname** Argument Generator display name for this Application. Argument generators are required if each instance of an application requires a different argument or set of arguments. For example in the case of a ring based DHT each instance of an application might require a unique id. In a case where all instances of an application require same set of arguments, then these can be provided through the launch script, and an argument generator is not required.

Argument generators in GODS are recognized by their display names instead of Java class names. Argument generators are registered with GODS against their display names through a config file currently chosen to be *gods.churn.arggens.xml* in the *arggens.config* folder. The value for this property must be an argument generators display name *not* class name. In a case where argument generator is *not* required *void* should be specified as the value for this attribute.

An argument generator must implement the *ArgumentGenerator* interface. Argument generators already implemented in the *gods.churn.arggens* package can taken as examples to start with. These argument generators are specifically for generating arguments required by an implementation of Distributed K-ary System (DKS).

15. **app.arggen.config.file** An argument generator might need some configuration parameters for generation of arguments. Consider the case where each application instance is a node on a DHT ring, the ids to be generated for them must be within a range. GODS provides a mechanism to specify such configuration parmeters in files with which an argument generator is initialized. There is no restriction for the format of such files or how they are interpreted by an argument generator.

    The value of this parameter should be a config file valid for the argument generator specified in *app.arggen.class*. This parameter is not ignored if *app.arggen.class* is *void*.

    The configuration files for already implemented argument generators are in the *arggens.config* folder.

16. **app.kill.signal.int** Integer signal number to be sent to the application before killing it. GODS, currently kills the application with KILL signal that has the value of 9. So, this value is currently ignored.

17. **app.stop.signal.int** Integer signal number to be sent to the application for it to leave the system gracefully. This feature has not been implemented completely.

9.4.2 Running an Experiment

An experiment can be conducted by selecting to run an experiment through the Visualizer. For running the Visualizer refer to Running Visualizer section.

GODS deploys the network required for the experiment automatically, however a network deployed after startup is not updated, so the network will have to be deployed manually for an experiment and the network model and number of slots in it should be mentioned in *CONFIG_FILE* specified in GODS startup parameters.

References

[1] Apache Software Foundation . Bcel, 2002–2006. http://jakarta.apache.org/bcel/.

[2] BRITE. http://www.cs.bu.edu/brite, 2000-2001.

[3] BRITE. http://www.cs.bu.edu/faculty/matta/Research/BRITE, 2001.

[4] Jay Chen, Diwaker Gupta, Kashi Venkatesh Vishwanath, Alex C. Snoeren, and Amin Vahdat. Routing in an internet-scale network emulator. In Doug DeGroot, Peter G. Harrison, Harry A. G. Wijshoff, and Zary Segall, editors, *MASCOTS*, pages 275–283. IEEE Computer Society, 2004.

[5] Chun. DART: Distributed automated regression testing for large-scale network applications. In *International Conference on Principles of Distributed Systems (OPODIS), LNCS*, volume 8, 2004.

[6] Russ Cox, Frank Dabek, Frans Kaashoek, Jinyang Li, and Robert Morris. Practical, distributed network coordinates. In *Proceedings of the Second Workshop on Hot Topics in Networks (HotNets-II)*, Cambridge, Massachusetts, November 2003. ACM SIGCOMM.

[7] Frank Dabek, Russ Cox, Frans Kaashoek, and Robert Morris. Vivaldi: A decentralized network coordinate system. In *Proceedings of the ACM SIGCOMM '04 Conference*, Portland, Oregon, August 2004.

[8] DIMES. http://www.netdimes.org, 2004-2006.

[9] Distributed K-ary System. http://dks.sics.se, 2003-2006.

[10] Java Universal Network/Graph Framework. http://jung.sourceforge.net/, 2003-2006.

[11] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Elements of Reusable Object Oriented Software*. Addison Wesley Professional Computing Series. Addison Wesley, 1995. http://www.aw.com.

[12] Dennis Geels, Gautam Altekar, Scott Shenker, and Ion Stoica. Replay debugging for distributed applications. In *Proceedings of USENIX Annual Technical Conference*, pages 289–300, 2006.

[13] Glen McCluskey. Using Java Reflection, Jan 1998. http://java.sun.com/developer/technicalArticles/ALT/Reflection/.

[14] GT-ITM. http://www-static.cc.gatech.edu/projects/gtitm, 2000.

[15] Krishna P. Gummadi, Stefan Saroiu, and Steven D. Gribble. King: estimating latency between arbitrary internet end hosts. In *IMW '02: Proceedings of the 2nd ACM SIGCOMM Workshop on Internet measurment*, pages 5–18, New York, NY, USA, 2002. ACM Press.

[16] Inet. http://topology.eecs.umich.edu/inet, 2002.

[17] Javassist. http://www.jboss.org/products/javassist, 1999-2004.

[18] King. http://www.mpi-sws.mpg.de/~gummadi/king, 2002.

[19] Shiding Lin, Aimin Pan, Rui Guo, and Zheng Zhang. Simulating large-scale p2p systems with the wids toolkit. In *MASCOTS '05: Proceedings of the 13th IEEE International Symposium on Modeling, Analysis, and Simulation of Computer and Telecommunication Systems*, pages 415–424, Washington, DC, USA, 2005. IEEE Computer Society.

[20] Shiding Lin, Aimin Pan, Zheng Zhang, Rui Guo, and Zhenyu Guo. Wids: An integrated toolkit for distributed system development. In *Proceedings of the 10th USENIX Workshop on Hot Topics in Operation Systems*, Santa Fe, NM, USA, June 2005.

[21] Priya Mahadevan, Ken Yocum, and Amin Vahdat. Mobicom poster: emulating large-scale wireless networks using modelnet. *Mobile Computing and Communications Review*, 7(1):62–64, 2003.

[22] Matthew L Massie, Brent N Chun, and David E Culler. The ganglia distributed monitoring system: design, implementation, and experience. *Parallel Computing*, 30(7):817–840, July 2004.

[23] Sun Microsystems. Introducing MBeans, 1995–2007. http://java.sun.com/docs/books/tutorial/jmx/mbeans/index.html.

[24] Sun Microsystems. Overview of the JMX Technology, 1995–2007. http://java.sun.com/docs/books/tutorial/jmx/overview/index.html.

[25] ModelNet. http://modelnet.ucsd.edu, 2002-2006.

[26] Robert Morris, David Karger, Frans Kaashoek, and Hari Balakrishnan. Chord: A Scalable Peer-to-Peer Lookup Service for Internet Applications. In *ACM SIGCOMM 2001*, San Diego, CA, September 2001.

[27] David Oppenheimer, Vitaliy Vatkovskiy, and David A. Patterson. Towards a framework for automated robustness evaluation of distributed systems. In *FuDiCo II: S.O.S. Survivability: Obstacles and Solutions*, 2nd Bertinoro Workshop on Future Directions in Distributed Computing, Jun 2004. www.cs.utexas.edu/users/lorenzo/sos/SOS/oppenheimer-fudico.pdf.

[28] David L. Oppenheimer, Vitaliy Vatkovskiy, Hakim Weatherspoon, Jason Lee, David A. Patterson, and John Kubiatowicz. Monitoring, analyzing, and controlling internet-scale systems with ACME. *CoRR*, cs.DC/0408035, 2004.

[29] PlanetLab. http://www.planet-lab.org, 2002-2006.

[30] Yuval Shavitt and Eran Shir. Dimes: let the internet measure itself. *SIGCOMM Comput. Commun. Rev.*, 35(5):71–74, 2005.

[31] Daniel Stutzbach and Reza Rejaie. Understanding churn in peer-to-peer networks. In Jussara M. Almeida, Virgílio A. F. Almeida, and Paul Barford, editors, *Proceedings of the 6th ACM SIGCOMM Conference on Internet Measurement 2006, Rio de Janeriro, Brazil*, pages 189–202. ACM, 2006.

[32] Sun Microsystems. Java Remote Method Invocation - Distributed Computing for Java, 1994-2007. http://java.sun.com/javase/technologies/core/basic/rmi/whitepaper/index.jsp.

[33] Sun Microsystems. JVM Tool Interface, 2004. http://java.sun.com/j2se/1.5.0/docs/guide/jvmti/jvmti.html.

[34] Amin Vahdat, Ken Yocum, Kevin Walsh, Priya Mahadevan, Dejan Kostic, Jeff Chase, and David Becker. Scalability and accuracy in a large-scale network emulator. In *OSDI '02: Proceedings of the 5th symposium on Operating systems design and implementation*, pages 271–284, New York, NY, USA, 2002. ACM Press.

[35] Brian White, Jay Lepreau, Leigh Stoller, Robert Ricci, Shashi Guruprasad, Mac Newbold, Mike Hibler, Chad Barb, and Abhijeet Joglekar. An integrated experimental environment for distributed systems and networks. In *OSDI02*, pages 255–270. USENIXASSOC, dec 2002.

[36] Ken Yocum, Ethan Eade, Julius Degesys, David Becker, Jeffrey S. Chase, and Amin Vahdat. Toward scaling network emulation using topology partitioning. In *MASCOTS*, pages 242–245. IEEE Computer Society, 2003.

[37] B. Y. Zhao, J. D. Kubiatowicz, and A. D. Joseph. Tapestry: An infrastructure for fault-tolerant wide-area location and routing. Technical Report UCB/CSD-01-1141, UC Berkeley, April 2001.