

# A : An Assertion Language for Distributed Systems

Andrew Tjang, Fábio Oliveira, Richard. P. Martin, Thu D. Nguyen

{*atjang, fabiool, rmartin, tdnguyen*}@cs.rutgers.edu

Department of Computer Science, Rutgers University, Piscataway, NJ 08854

**Abstract.** *Operator mistakes have been identified as a significant source of unavailability in Internet services. In this paper, we propose a new language, A, for service engineers to write assertions about expected behaviors, proper configurations, and proper structural characteristics. This formalized specification of correct behavior can be used to bolster system understanding, as well as help to flag operator mistakes in a distributed system. Operator mistakes can be caused by anything from static misconfiguration to physical placement of wires and machines. This language, along with its associated runtime system, seeks to be flexible and robust enough to deal with the wide array of operator mistakes while maintaining a simple interface for designers or programmers.*

## 1. Introduction

Large and complex distributed systems are expanding in importance as users' dependency on Internet services grows. Reasoning about correct behavior of such systems is difficult because these systems are comprised of complex conglomerates of distributed hardware and software, such as load balancers, loggers, databases, and application servers.

Unlike traditional monolithic applications, these systems require considerable human interaction to keep them operational, and these interactions are a significant source of unavailability [2, 3, 7, 9, 10, 12]. More recently, we found that mistakes are responsible for a large fraction of the problems in database administration [9].

In this paper, we propose a new method of reducing the above class of errors. Our approach uses a new assertion language, called *A*, combined with a runtime monitoring system to directly support abstractions that allow system engineers to both (1) describe correct behavior and (2) reason about human-computer interactions in complex distributed systems. The goals of the *A* language and runtime include both reducing the number and impact of operator mistakes, as well as reducing the human cost of constructing models of correctness for distributed services.

For years, administrators have developed individual, ad-hoc methods of managing operator mistakes and the resulting system errors. Indeed, often significant IT budgets are devoted to developing, maintaining and managing in-house software to operate large-scale systems. We believe that a language specifically tailored to the task of operator-system interaction could improve this process.

Instead of a collection of ad-hoc scripts tailored to a specific site and environment, a language-based approach will allow a concise,

understandable, and verifiable method of specifying correctness. Codifying the process in a specific language would also ease the generalization and parameterizing of operator tasks, thus allowing for the re-use of software to manage operator interactions across different organizations. For example, an organization could re-use existing parameterized libraries to manage the interactions with load-balancers, databases and application servers instead of having to develop custom scripts as they do today.

In this paper we thus argue:

- The abstractions in *A* are suited for verifying correct behavior in distributed systems;
- The *A* runtime supports these abstractions; and
- It is easier to use the *A* language and runtime for preventing and catching mistakes than ad-hoc solutions.

*A* provides a way to name components of a service, access the state of the components (both dynamic attributes and static properties), and conveniently write assertions about these states. The runtime system provides a gateway between *A* programs and a service monitoring infrastructure that provides dynamically measured information about the service to the *A* programs. To abstract human actions, the language has constructs to allow a programmer to describe discrete steps of an operator's task as well how their models may evolve during the execution of the task. One example might be switching a load balancing policy - from a completely balanced one to a weighted one.

We also consider that while modeling performance captures dynamic behaviors, many mistakes result in improper static configurations. In fact, a good number of mistakes resulting in security and latent performance faults fit this pattern. *A* and its runtime system have specific constructs that represent configuration state (e.g., a start-up file) as well as its audit state (e.g., a log). We have found that validating both static and dynamic properties was critical for completeness; that is, both are needed to cover a wide range of mistakes.

In the remainder of the paper we introduce the reader to the *A* language and describe the constructs that make *A* uniquely suited to help system designers formalize their ideas of correct behavior. Section 2 details *A*, with Section 2.2 introducing the running example mistake used throughout this paper. Section 3 provides a brief overview of the supporting runtime system. Section 4 overviews our evaluation methodologies and results. Section 5 describes related research, with Section 5.1 describing current solutions and their inadequacies. Finally, Section 6 looks at ongoing work and future directions.

## 2. The *A* Assertion Language

This section describes the *A* language. We first give an overview. We then describe a common system, a mistake and an example *A* program that catches mistakes for that system. Finally, we give a brief overview of the major *A* language constructs.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

## 2.1 Overview

*A*'s primary constructs are *assertions*, *elements*, and *tasks*. These three abstractions allow the programmer to (1) describe correctness, (2) access system state, and (3) reason about temporal state changes with respect to operator actions.

The *assert* construct was directly inspired by its use in procedural languages such as C and Java. Anecdotally, maintaining a large set of assertions about correct conditions has been critical toward writing correct programs. Indeed, [6], devotes an entire chapter to the subject of how craft assertions for C programs. Some of the key ideas from that work that cross-over into the domain of correctness for the systems we describe are that assertions should (1) be side-effect free, (2) exhibit fail-fast behavior, and (3) should be explicitly labeled to the high-level conditions they are testing.

Elements in *A* were modeled after aggregate types, such as the C `struct`. In our case, an element describes the current system state of devices such as load balancers and databases. It is the runtime's job to map the language-level declared state to the current values of the devices. For example, the number of jobs in the run-queue of the OS on a particular machine might be a field of an element. A special `stat` variable type captures the notion of statistically sampled values, such as load average, for our class of system devices. In addition, because we are concerned with operator mistakes, elements can also describe the configuration state of devices; e.g. values in a configuration file.

Tasks represent sequential execution of an operator's actions. The language-level construct allows for the programmer to specify which assertions should be checked, and provides interval points to wait for operators to complete actions. Tasks also allow the *A* programmer to save state in variables to compare past state with present state. The task abstraction thus directly addresses how to describe how system state should change with time, which is often left unresolved in many modeling languages.

Execution in *A* follows a discrete event model, similar in spirit to triggered databases or user-interface libraries (e.g. the Tk widget library [11]). The run-time takes monitored and measured values from real devices as well as input from human operators and forwards them to a central location. Assertions checks and wait statements in tasks map to a scheduled event stream executed at the central location. Operators are then notified when assertions fail.

## 2.2 Example System, Mistake, and Program

In this section, we briefly introduce an example system, the Linux Virtual Server (LVS), a common mistake made configuring LVS, and an *A* program designed to catch mistakes operating on the LVS system.

One of the main applications of LVS is an advanced load balancing solution. A commonly seen LVS operator misconfiguration [5] occurs when LVS is set up in the direct routing mode with Web servers on the back-end. A popular method to achieve direct routing involves assigning the same IP address that the LVS uses to receive requests from the clients to a virtual interface of a loopback device on each Web server. This causes requests to be handled by the LVS machine, with responses handled directly by the individual Web servers. The caveat is that Web servers must ignore ARP requests for the loopback devices. Clearly, the effect of a misconfiguration here will lead to all Web servers and the load balancer answering ARP requests for the shared IP address, leading to a race condition. The manifestation of this misconfiguration is that some client requests might be sent directly to the Web servers, while others go through the load balancer, resulting in an unbalanced load on the Web servers.

Although the above is only one specific example in a long list of possible mistakes that can be introduced into a distributed system, it is a fairly representative one with several interesting qualities.

In file: assertions.a:

```
1: config LVSCfg{ ...
2:   :lvsadm: ``ipvsadm.pl``
3:   set weight =
         /root/workers/worker/weight, "" ;
4: } ...
5: config ApacheCfg{ ...
6:   :iptables: ``iptables.pl`` ;
7:   single destination =
         /root/chain[...]/destination
8:   :netint: "netinterfaces.pl"
9:   single hidden =
         /root/real-interfaces/all/hidden, "" ;
10: }
11: online ws_all::WebServerGroup(IP=".*")
12:   with config ApacheCfg("path/to/httpd.conf")
13:   with config LVSCfg
14:   assert balanced
15:     (EQUAL(COLLECT(ws_all..cpu.util))) {
16:       on: global;
17:     } else { //print "Web servers load unbalanced" }
18:   assert dest
19:     (ws_all..config['netint'].hidden == 1) {
20:       on: global;
21:     }
22: }
```

**Figure 1.** An example set of assertions and configurations in *A* that help to catch our LVS mistake. Keywords are shown in bold.

Firstly, the mistake has a real, tangible impact on the distributed system. Secondly, the mistake is fairly common and simple to achieve. Finally, the impact of the mistake may or may not result in a latent error.

Figure 1 shows a small *A* program designed to catch mistakes when manipulating an LVS server. At a high-level, lines 1-10 type the configuration state we care about, in this case the LVS and Web server (Apache) configuration files. Lines 11-12 instantiate the elements types against real servers, and lines 13-19 are assertions to describe correct behavior. We give a brief, bottom-up description of these language constructs in the remainder of this section.

## 2.3 Elements

Elements represent states of running service components as reported by runtime monitors. Each element must be declared to be of some element type and consists of a number of fields. Each field can hold a value of a primitive data type, a statistical object, or another element. Rather than being programmer specified, these values within elements are defined by the monitored state of the component. Elements are *bound* to specific components. In the listing, the element *lb* represents a single load balancer bound to a load balancer that has the address *domain.tld*. Similarly, *ws\_all* represents all Web servers, regardless of IP address. Mapping the state of the real system to values in the *A* program is the job of the runtime system. Being able to map and refer to groups of machines is one of many features that makes *A* ideal for distributed systems.

**Primitive types.** *A* supports a standard set of primitive types including `int`, `double`, and `string` with the typical operators.

**Stat type.** A `stat` object represents the tail of a stream of temporally sampled values. The CPU utilization of the Web servers are just such a value.

A `stat` object can be aggregated statistically into a single value via operators such as average, median, standard deviation, min, and max. `Stat` objects can also be compared using the standard relational operators. A statement like "All web server CPU utilization should be less than X" is thus simple to write in *A*. Statistical equality is calculated by using the Box and Jenkins [1] outlier model.

**Configuration and log types.** Since static configurations dictate how a majority of distributed components behave, it is natural to include configuration constructs. Each element may have an attachment of one or more objects of the configuration or log types. Each attached configuration object refers to a set of static information about the service component bound to the element. The definition of a configuration type involves the specification of a set of files or program outputs (Line 2) that can be parsed to obtain the desired static characteristics, a set of drivers that can be used to translate the configuration files into the XML format, and a set of XPath queries to extract the desired characteristics.

In the same vein, definitions of log types, a log object is a stream of information being written to any file.

**Configuration Aggregates.** It is often useful to name an aggregate set of configuration parameters. For our example, the weight of each of the workers seen by LVS is referred to as a *set* (Line 3). A programmer can later refer to all weights using this one parameter.

## 2.4 Assertions

Much like the `assert` directives in C and Java, an assertion is a boolean expression. In *A*, assertions make statements about the values in elements. An assertion evaluating to true models correctness, while one evaluating to false models incorrect behavior.

Each assertion is comprised of four parts: a name, a conditional expression, a control block, and a set of action statements to be executed if the assertion fails. An assertion is thus like a stylized object in that it contains assertion specific parameters (e.g. frequency to be evaluated), as well as implements a single method that returns a boolean value.

**Name.** A common problem with assertions directives in C and Java is that they can become opaque; sometimes even the author cannot remember the higher-level reason a particular assertion was included. It thus becomes difficult to know if the assertion or program is faulty. We hope to mitigate this effect by at least forcing programmers to name the assertions. For example, the name `balanced` in line 13 has a simple intuitive meaning.

**Expression.** An expression returns a boolean value. If the statement evaluates to `true`, the system is correct. If the expression evaluates to `false`, the code in the action block (the `else` clause) is evaluated. In the listing, Line 13 shows an assertion that models the balanced nature of all Web servers.

**Control Block.** The programmer can optionally specify values to control the scheduling of assertions: frequency, delay (when to start an assertion), status (on or off), and type (global or task related).

**Action Block.** The action block (Line 15) refers to what the runtime system should do in the event that the particular assertion has failed. We leave actuation, i.e. adjusting the system in response to failed assertions, as future work.

**Hierarchical Assertions.** The *A* language provides support for abstracting assertions, which can be used to build assertion hierarchies. This allows programmers to think about correct behavior in terms of high-level concepts that eventually resolve to low-level, specific parameter/value checking. For example, the idea that two components are “connected” is a high-level idea that requires more specific checks.

**Aggregates.** Relations in the expression part of an assertion can accept aggregate element types as arguments. For example, the `EQUAL` operator can take an aggregate element (e.g., a set of replicas). The `EQUAL` operation applied to a `stat` field of an aggregate element returns true if all `stat` objects in the field are equal, using the definition of statistical types. The `balanced` assertion shows this usage.

## 2.5 Libraries

Libraries support abstraction over different element instances as well as hide detailed sets of assertions. One can think of libraries as “templates”. Libraries can be organized by components, subcomponents, or by goal. Libraries have the added benefit of allowing code reuse via parameterized assertions.

Our LVS assertions could be logically classified in many ways. The `balanced` assertion could be part of a library of assertions on dynamic properties or could be part of a library filled with other properties that need to be “balanced” (eg. memory usage, network usage, etc.). Similarly the `dest` assertion could be part of a configuration library or could be part of a library indicating connectivity of components. The assertions may also be found in libraries describing their component parts (e.g. an LVS library and a Web server library). The *A* programmer is free to decide how to organize the models in the most suitable and easily manageable way.

## 2.6 Tasks

Tasks represent human-system interactions. One can think of tasks as sets of assertions grouped together with intervening `wait` statements. Tasks, unlike assertions, are stateful and can compare system properties throughout the course of task execution. This is particularly useful when checking “before” and “after” values of assertions with dynamic properties.

Our LVS assertions would be included in a task - perhaps one involving setting up a new Web server. For the sake of brevity, the task itself, has been omitted from the listing. Tasks have a few constructs specific to them:

**Wait statements.** These tell the runtime system to wait on particular operator events, explicitly indicated by the operator.

**Conditional waits.** The runtime waits on a specific condition within the system, e.g. an element field to reach a certain value.

**External waits.** The runtime system waits for operator input. This is especially useful when some assertions depend on information only known to the operator. An example of this is the hostname of component currently being modified.

**Call statements.** These explicitly call for other non-task specific assertions to be evaluated immediately.

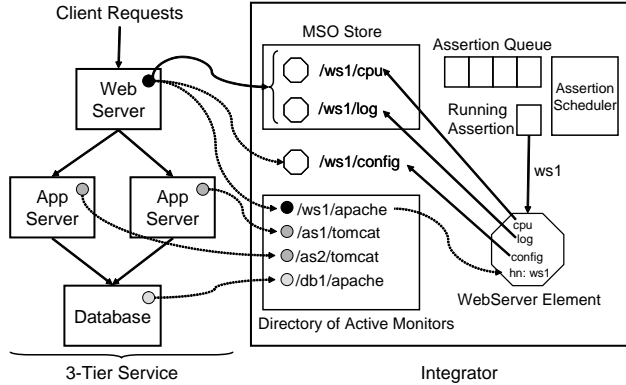
**Variable assignment.** A variable stores the current element field for comparison to future values.

Wait statements allow system engineers a way to segregate operator actions into subspaces - each with its own set of assertions. Wait statements have a timeout (specified in milliseconds), and will cause the task to run the `else` block if the action is not completed by the time allocated. A conditional wait keeps the task from reaching the next block of assertions until an expression evaluating element values becomes true (or it times out).

## 3. The A Runtime System

Recall the most critical job of the run-time is to reflect the real system state in the *A* run-time. For example, the current values of system configuration files and sampled state of running devices must be brought into a single location where the assertions are evaluated. This task is basically system monitoring, which is vast topic in and of itself. In this section we just briefly describe our current implementation.

Configuration is obtained through a series of drivers. These drivers take the vastly different configuration formats and create XML equivalents. *A* programs can then parse these XML fragments in any way. Dynamic information is gathered by a set of *monitors*, which capture the dynamic state of running service components.



**Figure 2.** Architecture of the current *A* runtime system. Shaded circles inside the service components (Web server, application servers, database server) represent monitors. Shaded circles in the directory of active monitors represent monitor callback objects. Hexagons inside the MSO Store represent monitoring stream objects (MSOs). The dynamic streaming of monitoring data to the receiving MSOs is only shown for one monitor for clarity.

All information is then collected by the *Integrator*, and brought to a central location where the tasks and assertions are executed and evaluated. Figure 2 shows the overall architecture of our prototype.

The *Integrator* has three main functions: (1) receive and store observations from all monitors; (2) map accesses to each *A* element to real observations; and (3) schedule and execute assertions from an *A* program.

A key abstraction of the runtime is the Monitoring Stream Object (MSO). There are two categories of MSOs, stat and log, which map directly to *A* stat and log objects.

When an *A* element is instantiated via an *A* binding statement, the runtime system creates an object of the appropriate type, finds an active monitor whose name matches the binding expression, and binds the object to the matching monitor, its callback object, and its MSOs.

For a group binding, the runtime system creates as many objects of the appropriate type as there are active monitors that match the binding expression. Each object is bound to a monitor as described above. A group object is also created to hold the set of created element objects.

### 3.1 Assertion Scheduling

Every assertion is assigned an evaluation time. These are then sorted on a ready queue. After evaluation, the assertion’s next evaluation time is evaluated and it is placed on the queue.

The assertion scheduler distinguishes between two types of assertions, global assertions that have a lifetime that exceeds that of any one task, and task-specific assertions, that have lifetimes associated with specific tasks.

### 3.2 A Run through the *A* System

With a cursory understanding of the language and the runtime, it is now possible to understand how *A* programs are processed. Even before our *A* program is loaded into the runtime system, there are many things occurring. There are monitors on each machine reporting information back to the *Integrator*, which is keeping some historical information and doing statistical analysis.

Once our program is loaded, the runtime immediately attempts to bind the variables it encounters, namely: `lbalancer` and `ws.all`. In the case of the former, it finds the one reporting host

whose identity matches the argument given “domain.tld”. In the latter case, the runtime, matches `ws.all` with all Web servers. These variables are now bound, and their mappings are kept as references. The assertions are then loaded into the assertion scheduler. At regular intervals, as specified in the assertion or by default values, the runtime executes an assertion by comparing the current system state as reported by the monitors on the bound machines. If the expression evaluates to false, the system will then perform the operations listed in the `action` block of the assertion.

If the misconfiguration listed in Section 2.2 were to be performed, the assertion listed on Line 16 would fire immediately after the configuration file was saved. If corrected, the assertion listed on Line 13 (`balanced`) may never have the opportunity to fire. It should be noted that the assertion is still evaluated, so even other causes that make the Web servers unbalanced would also be caught, making one assertion useful in many situations.

## 4. Evaluation

To evaluate our approach, we implemented a prototype runtime system and wrote a set of *A* programs to model what we believed to be correct operation in our evaluation environment - which was a three-tier auction service with a LVS load balancer front-end and a load generator as a client.

We then performed mistake injection experiments to observe how well mistakes were caught. To ensure our mistakes were of comparable complexity and subtlety to real mistakes, we used a combination of mistakes observed from human factors studies in our previous work, those reported by observations in the literature, and those reported by database administrators in the course of a survey.

The assertions found in Listing 1 represent only a small component of the libraries and programs we used to evaluate our methodologies for reducing operator mistakes. While these methods are outside the scope of this paper, we will discuss our findings in the context of our LVS case study.

We injected the mistake of allowing a Web server to answer ARP requests for its loopback device, which is the default behavior. We noticed that only the assertion about the configuration of the loopback device failed. The load was actually correctly distributed across the Web servers behind LVS. The reason was that the load generator had cached the ARP response given by the load balancer. Previous techniques [8, 9] would have overlooked this mistake, due to its latent nature. However, in the interest of completeness, we decided to perform another run after making sure that the ARP cache of the load generator was cold. In this run, all requests were sent directly to one Web server, bypassing the load balancer. This time not only did the configuration assertion fail, but our assertions that CPU utilization should be uniform across the Web servers also failed.

Besides catching the above mistake, we were able to catch 10 of the 11 mistakes we injected - none of which could be found in previous works on validation [8]. This is a very encouraging result because it shows that a high-level understanding of the system, as encoded in our language, will catch a large fraction of unanticipated mistakes.

## 5. Related Work

The related works fall into two broad categories: current monitoring and correctness systems, and language work for formal modeling, performance verification, and event programming. In this short section it is impossible to cover these areas sufficiently; rather, we seek only to point readers to representative works in the areas most related to this work.

## 5.1 Current Monitoring and Correctness Tools.

Assuring correct behavior of distributed systems is often assigned to an organization's system administrators. The Internet is littered with monitoring and alerting scripts that can assist them with the monumental task of keeping a smoothly operating system. Anecdotal evidence suggests the coverage and behavior of these tools is often random and chaotic in nature. Their usage and output leave many things open to interpretation and even the smallest of organizations may have many of them cobbled together to create a suitable solution. In any event, these system administration scripts suffer from a number of problems that show them to be sub-optimal for ensuring correct behavior of distributed systems. 1) They lack continuity: the same script used to monitor and identify problems in network traffic may vastly differ from a script used to monitor CPU utilization, even though one may be tightly related to the other. 2) Manageability/Readability: Adding new alerts or monitors can involve editing multiple scripts/tools. Determining what components, alerts, monitors, exist also involves multiple scripts/tools in multiple formats. 4) Interpretation: Many tools provide the means by which information is gathered, but interpretation of this information is largely left to a person. Different people have differing opinions of correctness.

A seeks to solve these problem areas by providing a clear and concise method for system designers and administrators to formalize policies of correct behavior. The assertions in *A* are expressive enough to encode multi-component dependencies and simple enough to be easily and readily readable and editable. *A* features elements, element groups, dynamic bindings, and configuration structures - all of which make great strides in simplicity and uniformity.

When we look at our LVS case, traditional monitoring and alerting tools might be used to detect excessive CPU usage in an overloaded state, or a saturated link. Usually by this time, the effects of the misconfiguration have been exposed to end users as slow response times, timeouts, or other undesirable behavior. The root cause, however, would remain hidden to administrators.

## 5.2 Modeling, Verification and Event Languages

Other systems have combined assertion languages with dynamic assertion checking. Both PSpec [13] and Pip [14] used assertion checking for performance debugging. While Pip focuses on distributed systems, it also seeks to verify their communication structure. However, because they are concerned with dynamic, run time problems as opposed to detecting operator mistakes, these systems did not consider some important static and structural issues, such as improper system configurations and latent security problems. Operator tasks are first-class concepts in our assertion language, allowing us to detect these types of problems and relate them to specific tasks.

There have also been recent languages and systems designed for temporal, event-based programming. For example, FRP [15] is designed with abstractions for temporal event-based programming. *A* is not as focused on abstracting temporal events handling as it is on expressing correctness about time-varying state. On the systems side, Drools [4] is an event-based rule engine designed to assist enterprises in codifying their business logic. A prime focus of Drools is execution efficiency, and so it can also serve as a framework for other domain specific languages, such as *A*. Its use and implementation are somewhat orthogonal to our work.

## 6. Current Status and Future Work

In this paper we introduced a language for determining correct behavior in distributed systems. We overviewed the many unique constructs in the *A* language that can help system designers model

correct behavior. We also showed the event-based and procedural components that lend themselves nicely to modeling operator tasks.

The question of utility of a new domain specific language versus that of traditional languages is a subjective one. More work must be done to investigate how easily *A* programs are written, and how effective the resulting programs are at modeling correct behavior. Our initial experiences have shown promise in a simple three tiered Internet service. Examining *A* in other distributed scenarios is another avenue for further work. Scalability of *A* programs is an open issue. Our intuition is that *A* programs can be modularized and then combined. This combination raises consistency and efficiency problems that might need to be resolved.

Ultimately, we believe that having a domain specific language like *A* can help to structure, organize and formalize ideas of correctly operating systems, especially in the context of operator actions. We have shown that the carefully designed constructs in *A* fit the domain of distributed systems well.

## References

- [1] G. Box, G. Jenkins, and G. Reinsel. *Time Series Analysis: Forecasting and Control*. Prentice Hall, Inc, Englewood Cliffs, NJ, third edition, 1994.
- [2] J. Gray. Why do Computers Stop and What Can Be Done About It? In *Proceedings of 5th Symposium on Reliability in Distributed Software and Database Systems*, Jan. 1986.
- [3] J. Gray. Dependability in the Internet Era. Keynote presentation at the 2nd HDCC Workshop, May 2001.
- [4] Jboss rules (drools). <http://labs.jboss.com/portal/jbossrules/>.
- [5] Linux virtual server project. <http://www.linuxvirtualserver.org/>.
- [6] S. Maguire. *Writing Solid Code*. Microsoft Press, Redmond, WA, 1993.
- [7] B. Murphy and B. Levidow. Windows 2000 Dependability. Technical Report MSR-TR-2000-56, Microsoft Research, June 2000.
- [8] K. Nagaraja, F. Oliveira, R. Bianchini, R. P. Martin, and T. D. Nguyen. Understanding and Dealing with Operator Mistakes in Internet Services. In *Proceedings of OSDI '04*, Dec. 2004.
- [9] F. Oliveira, K. Nagaraja, R. Bachwani, R. Bianchini, R. P. Martin, and T. D. Nguyen. Understanding and Validating Database System Administration. In *Proceedings of the 2006 USENIX Annual Technical Conference*, June 2006.
- [10] D. Oppenheimer, A. Ganapathi, and D. Patterson. Why do Internet Services Fail, and What Can Be Done About It. In *Proceedings of USITS'03*, Mar. 2003.
- [11] J. Ousterhout. *Tcl and the Tk Toolkit*. Addison Wesley, Reading, MA, 1994.
- [12] D. A. Patterson, A. Brown, P. Broadwell, G. Candea, M. Chen, J. Cutler, P. Enriquez, A. Fox, E. Kiciman, M. Merzbacher, D. Oppenheimer, N. Sastry, W. Tetzlaff, J. Traupman, and N. Treuhaff. ROC: Motivation, Definition, Techniques, and Case Studies. Technical Report UCB/CSD-02-1175, UC, Berkeley, Mar. 2002.
- [13] S. E. Perl and W. E. Wehl. Performance Assertion Checking. In *Proceedings of SOSP'93*, Dec. 1993.
- [14] P. Reynolds, C. Killian, J. L. Wiener, J. C. Mogul, M. A. Shah, and A. Vahdat. Pip: Detecting the unexpected in distributed systems. In *Proc. NSDI 2006*, May 2006.
- [15] Z. Wan and P. Hudak. Functional Reactive Programming from First Principles. In *ACM SIGPLAN Conference on Programming Language Design and Implementation*, June 2000.