Distributed-System Failures: Observations and Implications for Testing

Matthew J. Rutherford, Antonio Carzaniga, and Alexander L. Wolf

Department of Computer Science
University of Colorado at Boulder
Boulder, Colorado, 80309-0430 USA
{rutherfo,carzanig,alw}@cs.colorado.edu

University of Colorado
Department of Computer Science
Technical Report CU-CS-994-05 April 2005

ABSTRACT

Distributed software systems are notoriously difficult to test. As in all software testing, the space of potential test cases for distributed systems is intractably large and so the efforts of testers must be directed to the scenarios that are most important. Unfortunately, there does not currently exist a general-purpose, disciplined, and effective testing method for distributed systems. In this paper we present an empirical study of failures experienced by the users of seven open-source distributed systems. The goal of the study is to understand the extent to which there are patterns, commonalities, and correlations in the failure scenarios, such that one could define an improved testing method for distributed systems. Our results indicate that: a new generation of test-adequacy criteria are needed to address the failures that are due to distribution; the configurations that cause user-reported failures are reasonably straightforward to identify; and generic failure observations (i.e., those for which reusable techniques can be developed) are strongly correlated to the distributed nature of system failures. The second two results in particular imply that there is a reasonable bound on the effort required to organize the testing activity. Overall, the study gives us some early confidence that it is feasible to consider a testing method that targets distributed systems. The results of this study are offered as a basis for future work on the definition of such a method.

1 Introduction

Software testing is a subdiscipline of software engineering that, like all engineering disciplines, is governed by the tradeoff between time and resources on the one hand, and completeness and quality on the other. Given this, it is vital that testing methods be developed with an eye toward guiding testers to the most important failure scenarios while expending the least amount of effort. This paper presents a study aimed at determining whether it is feasible to consider developing such a method for the testing of distributed systems, since a general-purpose, disciplined, and effective method does not currently exist.

Distributed software systems are notoriously difficult to test. Some of these difficulties are related to the problems in selecting effective test data for non-standard inputs, such as the topology of the distributed components and network characteristics of bandwidth or latency. Other difficulties arise from the challenges associated with defining and observing system failures in the face of nondeterminism and a possibly heterogeneous distributed testbed. In considering a testing method, we must find some structure to the problem that will reduce the effort required of a tester to navigate through an otherwise intractable space of potential test cases. If no such structure can be found, then there is simply no basis for defining a method. Thus, this paper represents a first and necessary step in the development of a method. We seek to understand the extent to which there are patterns, commonalities, and correlations in failure scenarios, such that one could justifiably define a testing method for distributed systems.

Testing methods are usually classified by the level at which they occur: unit, cluster, integration, subsystem, and system. Another way of classifying testing methods is according to the role of the group that performs the testing. In most software development organizations there are two distinct groups: developers test the code that they write at the unit or cluster level, while quality assurance (QA) personnel test the application at the subsystem or system level. In general, developers use the properties and structure of their code to guide testing, while QA personnel concentrate on requirement specifications and use cases as their guide. Furthermore, QA testing acts as a backstop, or a last line of defense, that an organization uses to protect itself from failures in the field. No matter how well or how badly developers fulfill their testing responsibilities, the QA personnel have an obligation to ensure that the fewest possible failures escape. Thus, they must act defensively, and assume little about the testedness of the system achieved by developers.

Conducting a study of distributed-system failures presents a number of difficult challenges, but perhaps none more so than simply finding concrete data. While gaining access to details about any aspect of an organization's software development process is not easy (particularly for a commercial organization), reviewing product failure information is virtually impossible due to its sensitive and potentially damaging nature. Fortunately, many popular distributed applications are developed as open-source projects having associated with them publicly accessible defect-tracking systems. It is the failure reports contained in the defect databases of open-source distributed systems that are the raw data of this study. The study population consists of seven systems representing a broad spectrum of distributed-system styles: peer-to-peer, federation, and client/server, as well as a component library.

The use of open-source software in our study is not only convenient, it is actually an asset, since most of these projects function with little or no dedicated QA personnel, in contrast to traditional closed-source commercial systems. Therefore, open-source software is generally released having only been tested by developers, providing us with a unique opportunity to isolate failures that escape this class of testing. Furthermore, although many open-source developers consistently write and apply unit tests, they rarely if ever document and publish them, even within the defect-tracking systems. Under these circumstances, the failure reports originate mainly with users, who naturally concentrate on the features and configurations of most importance to them. In a sense, the users are acting as surrogate QA system-level testers for the purposes of this study, admittedly providing a broadly user-centric, as opposed to a strictly requirements-centric, prioritization of test cases. The question here is not how well those pseudo-QA testers may have performed their job, but rather whether there is something we can learn from the failures they did manage to uncover.

The testing activity involves two key tasks: (1) selecting inputs that are effective at causing failures and (2) constructing and applying oracles that are capable of recognizing that a failure has occurred. The method we developed for this study requires us to identify and classify each scenario described in a failure report in terms of the overall configuration of components, the specific inputs needed to drive the system

to failure, and the technique used to observe the failure. Further, we must determine whether each failure scenario was truly distributed or simply multi-component. A multi-component scenario would involve only the need for one component to provide input to another, while a truly distributed scenario is a multi-component scenario that would additionally involve properties such as concurrency, platform heterogeneity, unreliable communication, delays, timeouts, or the like. This is a critical distinction that helps shed light on the underlying nature of a failure and the testing techniques that might be required to expose it. We also distinguish between failure observations that are generic, in the sense that they are largely applicable to most kinds of distributed systems (e.g., detecting system crashes), and those that are specific, relating to the functionality of the system (e.g., incorrect content of a message). Clearly, generic observations can be supported by reusable techniques, while specific observations often need purpose-crafted techniques.

Our observations resulting from this study hold several important implications for the development of advanced testing methods. First, a new generation of test-adequacy criteria are needed to address the significant number of failures that are due to distribution. Current criteria are largely blind to the relevant issues revealed by our study. Second, the configurations that cause user-reported failures are reasonably straightforward to identify. This means that testers are not required to find exceedingly complex or exotic configurations in order to expose the failures that users would typically encounter. This, in turn, means that there is a reasonable bound on effort. Third, generic observations are strongly correlated to the distributed nature of system failures, which means that there is again a reasonable bound on effort, in this case the effort required to define and construct test oracles.

The results of our study are certainly preliminary, and the population, although representative of high-level distribution styles, is somewhat limited in size. Nevertheless, the study gives us some early confidence that it is feasible to consider a testing method that targets distributed systems. The results of this study are offered as a basis for future work on the definition of such a method.

The next section outlines related research efforts that provide empirical studies based on defect reports. Section 3 describes each of the subject systems we considered. Section 4 describes the method we used to conduct our study, while Section 5 presents the results of the study. Section 6 concludes the paper with thoughts on how the results of the study might lead to a feasible testing method, as well as future work.

2 Related Work

There is a large body of work devoted to the use of defect reports to inform the development of testing techniques and methodologies. For the most part, these are aimed at studying the faults that are identified by the reports, not the types of tests that could have identified the faults earlier. For example Fenton and Ohlsson [4] and Ostrand and Weyuker [8] both describe the use of fault data from successive releases of commercial systems. They present a number of interesting results related to correlations between module size and fault-proneness and the relationships between pre- and post-release faults in modules, but they do not consider the types of tests (particularly for the post-release faults) that could have produced the failures sooner.

Another example of this is the analysis of bug reports and subsequent change requests to determine the root cause of the defects across all phases and activities in the development process [5, 6]. As part of their root-cause analysis, Leszak, Perry and Stoll describe the development of countermeasures to help improve the development effort. One of the countermeasures listed was improving unit testing. However, this is motivated by the root-cause information not the test-input information that could have potentially been gleaned from the initial defect reports.

In orthogonal defect classification (ODC) [1, 2], both a *defect type* and a *defect trigger* are identified for each defect. The defect type is similar to a root cause, and is recorded by the software engineer making the changes needed to fix the defect. Conversely, the defect trigger is a coarsely grained categorization of the circumstances that lead to the defect being reported. At a high level, the defect trigger is used to improve the verification process by determining why the circumstances that caused the defect where not seen during testing. Our input classification is similar in intent to the defect trigger classification. However, our classification is more detailed, as well as being more directly targeted at distributed systems. ODC is

a general-purpose defect analysis technique that is targeted at improving the software development process for a project in general, while we are primarily concerned with exploring the feasibility of creating a new testing method.

More recently, Li et al. [7] studied user defect reports from several open source projects, and fault and failure reports from commercial software, to study defect occurrence models for large, multi-release software systems, including operating systems and a distributed middleware system. While the raw data for their study are similar to ours, Li et al. are primarily interested in frequency data. Interestingly, they also note that user defect reports in open source systems often contain duplicates and invalid reports, though in their study, these reports are considered in scope, since a developer has to review each one, regardless of its ultimate utility.

3 Subject Distributed Systems

Our study reviewed failure reports from seven diverse distributed systems. We selected these systems to represent a broad spectrum of distribution styles: peer-to-peer (P2P), federated, and client/server. We also included in our study population a component from a distributed component library.

3.1 Peer-To-Peer

For our purposes, the defining characteristic of P2P applications is their ability to operate without the resources of a centrally controlled server application. Peers generally discover each other, and then cooperate to deliver services.

3.1.1 Gtk-Gnutella

Gnutella is a popular protocol for decentralized file sharing. Gtk-Gnutella is a Unix-based Gnutella client. A client joins the network by locating a single existing member who can then help bootstrap the new node by sending to the new node its list of known addresses. A decentralized system of web applications collectively known as the Gnutella Web Caching System (GWebCache) stores lists of active hosts and other caches; this too is used for bootstrapping.

The Gnutella protocol runs over both TCP and UDP and includes control messages for joining the network, issuing queries, and responding to queries. HTTP is used to ultimately download files from peers. Gtk-Gnutella also supports the GWebCache protocol for retrieving and updating lists of active peers.

In Gnutella, nodes can operate in "ultrapeer" mode or "leaf" mode. Ultrapeers are typically run on machines having access to large amounts of bandwidth and other resources. Ultrapeers are able to respond to query messages on behalf of leaf nodes to which they are connected, thereby reducing the network usage for leaf nodes.

Gtk-Gnutella is written in C, and supports a rich graphical user interface via the GTK library. It has a large number of configuration options related to interacting with peers, joining the network, and controlling the amount of bandwidth used for uploads and downloads. It is the only application we considered with a dedicated (i.e., non-web-based) graphical user interface.

3.1.2 JBoss Clustering

JBoss is a fully featured enterprise Java (J2EE) application server. A single instance of JBoss can be used as a servlet and Enterprise JavaBean (EJB) container, and provides the typical services of a J2EE container, including management of transactions, security policies, and threading. An advanced feature of JBoss is its support for clustering of multiple JBoss instances to provide load balancing and redundancy. While JBoss itself is typically considered to be a canonical server application, we consider its clustering feature in the P2P category, since the clustering module is completely decentralized and supports automatic discovery of cluster members who cooperatively provide a service.

JBoss clustering supports replication of HTTP session and EJB state, Java Naming and Directory Interface (JNDI) replication, failover and load balancing of JNDI and EJBs, and cluster-wide deployment of J2EE components. JBoss is written in Java.

3.2 Federated

Federated applications are similar to P2P applications in that they are able to cooperate with other programs to collectively deliver a particular service. They differ from P2P applications in that they can provide services when running alone, and they must be manually configured to cooperate with other programs.

3.2.1 Elvin

Elvin[1] is a content-based publish/subscribe communication middleware application [9, 10]. In its most basic configuration, an Elvin application consists of a publishing client and a subscribing client both connected to a single router. The subscriber provides the router with information about its content preferences, and whenever a publisher sends a notification that matches these preferences, the router forwards it to the subscriber. Elvin supports federation of routes located in different administrative domains. It also supports failover, in which routers can cooperate to provide replication for advanced quality-of-service features. Each Elvin router can optionally be managed through a web interface. The core Elvin router is written in C and runs on both Windows and Unix platforms. Client bindings are available in a number of languages.

3.2.2 Squid

Squid is an advanced caching proxy written in C that is able to proxy communications for the HTTP/HTTPS and FTP protocols, among others. Squid provides advanced support for integration with other web caches to improve performance and reliability. Squid can also operate as an HTTP accelerator for a standard web server. It includes built-in support for management via SNMP, and is able to operate as a caching DNS resolver.

3.3 Client/Server

In the traditional client/server model, a single server program services multiple client programs without cooperation from other programs. However, there are very few server programs that are truly independent and do not use other services, such as DNS, sometime during their operation. Therefore, the two applications we selected as servers primarily operate alone to provide a service, but in some configurations may also act as clients of other servers.

3.3.1 AOLserver

AOLserver is a web server designed to be highly scalable and configurable. It is developed and used by America On-Line. The server core is written in C and is configured and programmed using the Tcl API that it exposes. AOLserver also supports server-side programming through Tcl, and integrates with various popular relational databases. Management of AOLserver is accomplished through a telnet-like interface.

Through the server-side programming API, AOLserver can act as an email client, and as an HTTP/HTTPS client to other (not necessarily AOLserver) web servers.

3.3.2 James

James is an email server written in Java. It supports SMTP, POP3, and NNTP. James can also be configured to act as a POP3 client, and supports the use of popular relational databases for storing email and metadata. Management of a running James instance is accomplished through a telnet-like interface. As an email server,

[1] Depending on the interpretation of the term, Elvin may not be considered truly "open-source". However, it has a publicly accessible defect tracking system, which is why it was included in this study.

James makes extensive use of the DNS system, and has a built-in DNS resolver/cache for this purpose. James can be customized through its Mailet interface.

3.4 Distributed Component

With the advent of server frameworks, distributed application programming is often accomplished through the development of components designed to operate within those frameworks. The components are similar to basic code libraries, but they often support special interfaces required by the framework and they must often be aware of the threading model used by the framework.

3.4.1 OpenSymphony Workflow

The OpenSymphony Workflow component provides advanced support for workflows. The component is customized to a particular workflow through a configuration file, and supports various popular databases for persistence of workflow histories. The component is written in Java.

OpenSymphony Workflow can be executed in a standalone fashion, but it is also intended to operate within a J2EE container as a web application (servlet) or EJB.

4 Empirical Method

The raw data used in the study are the reports contained in the databases of the defect-tracking systems used with the systems we considered. A central challenge faced in conducting the study was to transition from the informal application-specific descriptions submitted by users to generalizable descriptions of a test scenario. The method we used consists of four distinct activities. First, among all user reports, we isolate the ones describing defects that are suitable for our study; then, for each suitable report, we describe the architecture of the system in the reported scenario; we classify the inputs that cause the failure; and finally we classify the observations that highlight the failure.

4.1 Report Classification

Defect-tracking systems are used for many things, and particularly in open-source projects some of the uses are not necessarily related to reporting defects. Therefore, our first challenge was to identify a subset of the reports that are relevant to the scope of our study. For a defect report to be included in our study it must satisfy the following criteria:

- it describes a system-level failure;
- it describes a run-time failure; and
- it must have been verified by a developer.

Our initial set of reports for each application consisted of all "closed" reports. Using the querying features of the defect-tracking systems we were able to eliminate some extraneous reports by only considering those marked as "fixed". However, such queries were not available for all defect-tracking systems, so even fixed reports must still be classified manually.

Table 1 shows the report types we used for this initial preprocessing step. In general, reports are eliminated from consideration because they (1) do not describe a failure (marked as `IMPROVEMENT`, `RFE`, `WORKSFORME`, `3RDPARTY`); (2) describe a compilation or documentation defect (marked as `STATIC`); or (3) do not provide enough details about the system-level behavior that causes the failure (marked as `NEI`, `UNIT`).

Most report-type classifications are a straightforward matter of reading through the initial report and the subsequent dialog between developers and users. However, there is some subjectivity in making a `NEI` or `UNIT` classification. The following two examples should help clarify how these categorizations are used.

Report Type	Description	%
INSCOPE	in-scope report	34.17
IMPROVEMENT	code improvement	21.96
STATIC	non-run-time failure	14.47
NEI	not enough information	11.18
RFE	request for enhancement	7.83
UEI	user error / ignorance	3.92
UNIT	unit test	2.84
WORKSFORME	developer cannot replicate	2.72
3RDPARTY	failure is in external software	0.91

Table 1: Classified Report Types

```
The following code generates a java.lang.StringIndexOutOfBoundsException
exception: -

MailAddress mailAddr = new MailAddress("A@B.");

The exception is as follows: -

Jul 05 16:59:59 2001: java.lang.StringIndexOutOfBoundsException: String index
out of range: 4
Jul 05 16:59:59 2001: at java.lang.String.charAt(String.java:503)
Jul 05 16:59:59 2001: at org.apache.mailet.MailAddress.<init>
(MailAddress.java:102)

I can only generate this error if the last character is a fullstop.
```

Figure 1: James Report #5

Figure 1 contains the initial description of a defect report for James. This report was classified as UNIT because it makes no mention of the user-level inputs needed to reproduce the failure or the user-level observations that would be needed to recognize the failure.

```
#0  0x40197d21 in __kill () from /lib/libc.so.6
#1  0x40197996 in raise (sig=6) at ../sysdeps/posix/raise.c:27
#2  0x401990b8 in abort () at ../sysdeps/generic/abort.c:88
#3  0x8061bce in xassert (msg=0x809c4e2 "auth_user_request != NULL",
    file=0x809c4c0 "authenticate.c", line=478) at debug.c:250
#4  0x8050778 in authenticateAuthUserRequestLock (auth_user_request=0x0)
    at authenticate.c:478
#5  0x8058fd3 in clientRedirectDone (data=0x845de90,
    result=0x8211f68 "http://127.0.0.1:2968/") at client_side.c:314
#6  0x8080f34 in redirectHandleReply (data=0x845e600,
    reply=0x8211f68 "http://127.0.0.1:2968/") at redirect.c:70
#7  0x806e7e4 in helperHandleRead (fd=7, data=0x8211f20) at helper.c:691
#8  0x8061050 in comm_poll (msec=878) at comm_select.c:434
#9  0x807a8a8 in main (argc=2, argv=0xbffff7ac) at main.c:720
```

Figure 2: Squid Report #198

Similarly, Figure 2 shows a report that simply consists of a debugger stack trace that was classified as NEI. In this report, there is clearly enough detail for the developers to fix the problem, but there is not enough information to reconstruct the scenario that leads to the failure.

Table 2 shows that after preprocessing we are left with 602 in-scope failure reports. James report #66, entitled "MX Chaining in the RemoteDelivery mailet is broken", is an example of an in-scope report. It begins:

> MX chaining in RemoteDelivery is broken. Basically, the code in place assumes that a given MessagingException generated while sending mail doesn't merit a retry unless it encapsulates an IOException. This doesn't appear to be the exception generated when a server refuses a connection. So the exception is rethrown, and the other SMTP servers are never tried.

This report highlights a fault in the logic that deals with retrying email delivery when a domain has multiple mail exchangers (MX DNS records). The steps needed to reproduce this failure are:

1. start an SMTP server (not necessarily James);

System	Total Reports	In-scope Reports	%
AOLServer	196	69	35
Elvin Router	149	67	45
Gtk-Gnutella	337	115	34
James	203	85	42
JBoss Clustering	36	22	61
OpenSymphony Workflow	289	37	13
Squid Cache	552	207	38
	1762	602	34

Table 2: In-Scope Reports by System

2. configure a DNS server with a test domain that has two MX records: the first points to an invalid address; the second points to the server started in step (1);

3. run a James server configured to use the DNS server started in step (2);

4. send an email, through the James server, to an account at the test domain using an SMTP client;

5. observe that the SMTP server started in (1) does not receive the email.

James report #66 does not explicitly include these steps, but the description provides enough details about the failure that these can be inferred.

There are three major aspects of each scenario that are summarized for each report: (1) architecture, (2) inputs, and (3) observations. The remainder of this section describes them in detail.

4.2 Architecture

The architecture of a scenario refers to the number and organization of any components needed to reproduce the failure. The architecture is separated into the configuration of the application itself, the configuration of any supporting components, collectively referred to as the *harness*, and properties of the underlying network. The harness is divided into *drivers* and *services*, where a driver component is one that initiates communication with the system, while a service component instead awaits communication from the system.

In James #66 the scenario architecture consists of one instance of James, one SMTP client driver, one SMTP service, and one DNS service. There are no special network properties required.

In a few scenarios, the underlying network was required to have certain characteristics. For example, a JBoss Clustering scenario requires that the underlying network not support IP multicast.

# Components	%
1	19.60
2	41.20
3	33.39
4	4.82
5	1.00

Table 3: Architectural Summary

The scenario architectures we identified involved between one and four application instances, between zero and three driver components, and between zero and three service components. Only seven scenarios required specific characteristics of the underlying network. Table 3 shows a summary of the architectures described in the reports. The first column in this table reflects the total number of components involved in a scenario. For example, a scenario involving an AOLserver instance and an HTTP client, and a different scenario requiring two Gtk-Gnutella instances are both considered to involve two components. One notable feature of the data in Table 3 is that over 80% of the scenarios involved more than one component.

4.3 Inputs

In addition to the scenario's architectural elements, the scenario's inputs are identified and organized into general categories. We take a broad view of "inputs" in which we consider all the installation/configuration settings and run-time actions needed to reproduce the reported failure. As with the architectural classification, the inputs are separated by whether the site of the input is the system, one of the harness elements, or the underlying network. The input categories presented in Table 4 were derived from an examination of the reports themselves. In particular, new categories were created as needed, to accommodate an aspect of a scenario that did not fit neatly within the existing ones.

Locus	Category	Description
System	Configuration	configuration activities
	Customization	API programming activities
	Execution	execution style or command line options
	FileData	data files contents
	FileSystem	changes to files or file system structure
	Installation	compilation and installation activities
	OperatingSystem	changes to the operating system
	Platform	specific software platform
	UserInterface	interaction with the system via interface
Harness	Behavior	specific network-related behavior
	Configuration	driver/service configuration
	Platform	specific software platform
	SendMessage	send a particular message
Network	Manipulation	alter underlying network

Table 4: Input Categories

Most categories listed in Table 4 are self explanatory. "Customization", which appears as a system input, requires further explanation. Customization refers to the creation of any plug-in-like software that runs within a component in the system or the harness. For example, many of the failures associated with AOLserver required that a server-side script be created with particular contents, and hence this was classified as a customization.

The inputs categorized for James #66 are: (1) *Harness:SendMessage* for the DNS response and email and (2) *System:Configuration* for the DNS configuration of James.

After performing the initial input categorization, we considered each input independently and determined whether it was truly "distributed". The purpose of this categorization was to enable us to distinguish between scenarios whose architecture is multi-component, simply because the system accepts input at run time through the network interface, and scenarios that actually involve the distributed nature of the system. Inputs that were considered to be distributed fell into these three rough categories.

- *Inherent:* e.g., concurrency; relative differences in processing speed, latency, and bandwidth; unreliable communication; platform heterogeneity.
- *Accidental:* e.g., addresses, ports and well-known port assignments; networking resource limits.
- *Error simulation:* e.g., connection timeouts and termination; invalid or truncated messages.

A scenario was classified as distributed if it required any of its constituent inputs to be distributed. For example, a number of defects in AOLserver are demonstrated by creating a server-side script with particular characteristics and then executing the script by sending the appropriate HTTP request to the server; these are all classified as non-distributed. By contrast, an AOLserver scenario that required a client to connect to the HTTPS port and simply wait for the connection to timeout is classified as distributed.

4.4 Observations

For a system behavior to be classified as a failure, the anomalous behavior must be observed during execution. For each scenario, we described and classified the observation that was made. The categories that we used are summarized in Table 5. As we did for inputs, we separated our observation categories by whether they are applied to the system components, to the harness components, or to the network. The fourth column in Table 5 shows the percentage of all 602 observations that fell into each category.

Locus	Category	Description	%
System	Process	application process	28.57
	LogOutput	log or console messages	17.77
	State	internal state	12.96
	File	files or directories	6.48
	Performance	resource usage	2.16
	OperatingSystem	operating system elements	<1
	Message	network messages	<1
Harness	Message	network messages	25.41
	Networking	networking API	1.82
	State	internal state	1.16
	Performance	resource usage	<1
	LogOutput	log or console message	<1
Network	Transport	transport-layer aspects	<1

Table 5: Observation Categories

In our initial observation classification, we also produced a succinct summary of each observation. These summaries were then used in a second refinement phase to further categorize each observation category. In all, we identified 29 different subcategories of observations. Due to space limitations we cannot show the entire list. As an example, the subcategories of the *Message* category are as follows.

- *Absence:* no message was sent or received when it should have been.
- *Content:* the data transmitted in a message were somehow invalid.
- *Format:* the message format was incorrect.
- *Presence:* a message was sent or received when it should not have been.
- *Type:* a message of a particular type was was observed.

The observation in James #66 was classified as a *Message:Absence*, since the SMTP service should have received an email message.

Finally, each subcategory was labeled as either *generic* or *specific*, according to the level of specificity of observation that was needed. Generic observations are those that can be applied to virtually any software system (e.g., application crashes, errors appearing in standard system logs, or memory leaks), and observations that are tied to the application being tested, but still at a high level (e.g., the presence or absence of network messages). By contrast, specific observations involve detailed judgments about computed data values (e.g., contents of message fields). These observation labels are the basis for our analysis of observations discussed in the next section.

5 Analysis and Discussion

The analysis presented in this section is centered around (1) determining what general trends exist with respect to the complexity of the failure scenarios and (2) examining the specificity of observations that are required to detect failure. We examine both of these aspects from the perspective of two measures of a scenario's distributedness: (1) how many different components must cooperate to cause the failure and

System	# Dist.	# Scenario	%
AOLServer	14	85	16
Elvin Router	13	69	18
Gtk-Gnutella	23	115	20
James	12	67	18
JBoss Clustering	2	37	5
OpenSymphony Workflow	1	22	4
Squid Cache	38	207	18
	103	602	17

Table 6: Distributed Scenarios by System

(2) whether or not the scenario requires any "distributed" inputs as described in Section 4.3. This leads us to distinguish between multi-component and distributed scenarios in the discussion below.

After classifying each input as distributed or not we were able to determine that 17% of the 602 scenarios required truly distributed inputs. While 17% is not a very high percentage, it is important to remember that the scenarios in this study were reported by users, and that failures involving concurrency and timing issues are notoriously hard to replicate. Additionally, as the data in Table 6 show, the percentage of scenarios that are distributed is relatively consistent across all the systems we studied, lending credence to our belief that this class of failures is fundamental across all distributed systems. In any case, our conclusion is that a significant fraction of all the failures is directly related to the distributed nature of the systems. This observation suggests that specialized testing methods and adequacy criteria are needed to address distribution.

The core assumptions underlying our analysis are that a failure scenario with a larger number of components is inherently more difficult for a tester to perform, and that the space of possible test cases grows rapidly as scenarios of higher complexity are considered. Table 3 shows the overall architectural complexity of the reported scenarios in terms of the number of components involved. One interesting observation from the data in this table is that approximately 75% of the scenarios we encountered involved either two or three components, and that 93% of the scenarios can be accounted for with simple scenario architectures involving three or fewer components.

# Components	%
1	12.62
2	36.89
3	41.75
4	6.80
5	1.94

Table 7: Architectural Complexity for Distributed Scenarios

Table 7 shows the same data for the distributed scenarios. It should be pointed out that just over 12% of the distributed scenarios actually only required a single system instance. Most of these defects were related to misconfiguration of ports, addresses and DNS related settings and are thus part of the "accidental" distributed inputs. At a high level, comparing the values in Table 7 with those in Table 3 confirms the observation made above that even for the scenarios requiring distributed inputs, more than 90% of the scenarios we encountered involved three or fewer components.

Table 8 shows the top 8 scenario architectures as represented by the mix of application instances, driver instances, and service instances. Altogether, these eight scenarios account for 96% of those encountered, while the top four alone account for 85%. It is encouraging to note that the four most common scenario architectures involve only simple harnesses, if at all.

While the inputs to a test determine if a fault will occur, for the scenario to generate a failure, the fault

# App. Instance	# Drivers	# Services	%
1	1	1	27.57
1	1	0	24.92
1	0	0	19.93
2	0	0	12.62
1	2	0	3.49
1	0	1	3.16
1	1	2	2.33
2	1	0	2.16

Table 8: Scenario Diversity

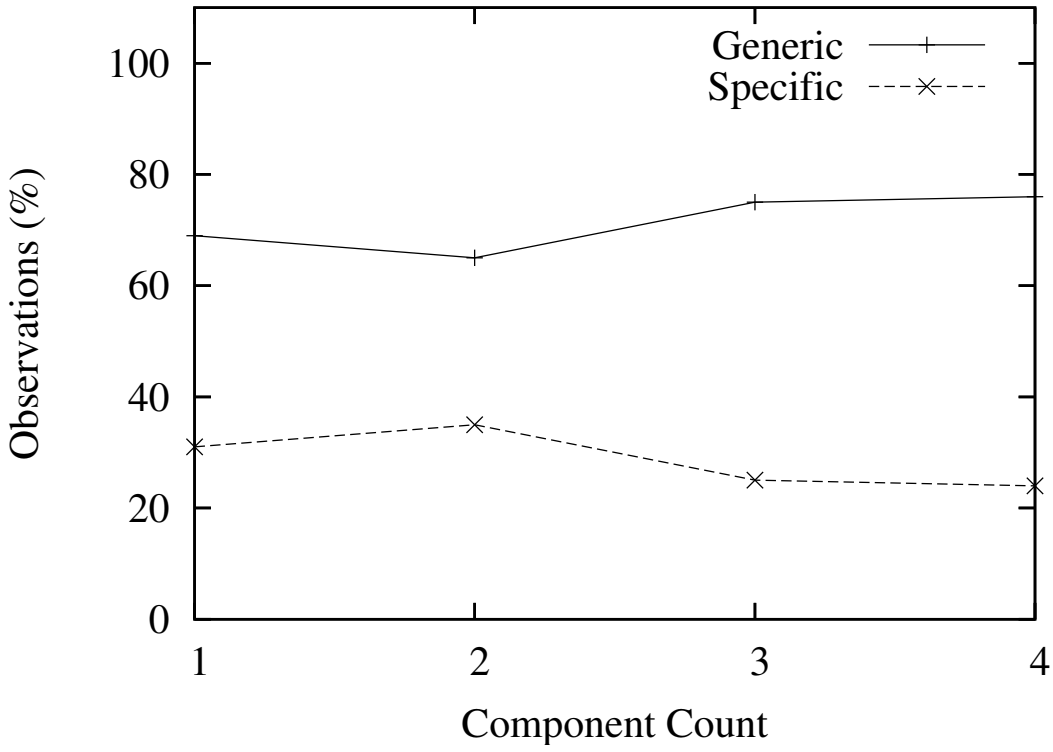

Figure 3: Observation Complexity versus Distribution

must be observed. Figure 3 shows the percentage of observations at each level of architectural complexity for the two main observation labels described in Section 4.4. Data are only shown for scenarios with four or fewer components since there are so few scenarios with five components. This figure shows the encouraging result that the proportion of *generic* observations and *specific* observations remain steady as the scenarios become more distributed.

Within the generic observations, we further identified three groupings: (1) catastrophic, (2) universal, and (3) general. Observations of catastrophic failures are the easiest to make since, by definition, the application stops running. Universal observation are those that can be applied to virtually any software system (e.g. error messages in system logs), while general observations are system-specific, but still broadly applicable within the space of possible behaviors of the system (e.g. message format violations).

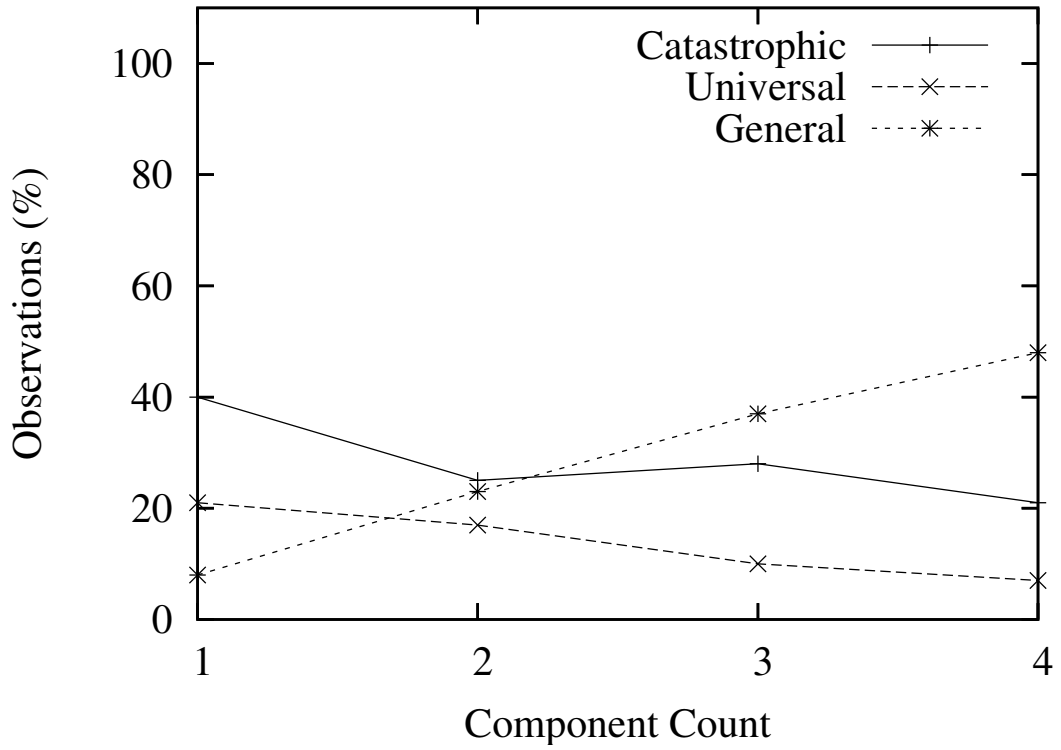

Figure 4: Generic Observations versus Distribution

Figure 4 shows that observations of catastrophic failures are most prevalent in single-component scenarios, and that their frequency levels off at roughly 25% for scenarios with more components. The most striking feature of Figure 4 is the rise of the general observations as the scenario complexity increases.

5.1 Implications for Testing

The prevalence of failures that are directly attributable to the distributed nature of systems suggests that specialized testing methods are needed to address distribution. As for the complexity of these methods, the implications of our two main findings are quite encouraging.

First, our analyses of the structural complexity of scenarios uncover the encouraging finding that the vast majority of scenarios are accounted for with relatively simple architectures. This is even true of scenarios that explicitly include inputs related to the distributed nature of the system. This implies that testers can reasonably prioritize their attention on the simple configurations.

Next, while the proportion of very specific observations appears to remain relatively constant at about 30%, the need for general purpose oracles for a particular system appears to increase as scenarios involve more components. This finding also bears out common sense which dictates that the failures in systems with more communication and interaction will be more subtle to identify, and that techniques and methods for developing system-level test oracles become more important in these cases.

Finally, at a higher level, the experience of conducting this study and attempting to impose some level of structure and categorization on the space of distributed system failures is encouraging. We were able to analyze a large number of failure reports from seven disparate systems using quite a small set of categories and a relatively simple model of the scenario architectures. All of this implies that attempting to develop systematic and rational approaches to high-level testing of distributed systems is not futile, and that general-purpose methods, tools and techniques can be developed with at least the possibility of making a positive impact on the quality of such systems.

5.2 Threats to Validity

While we feel confident that the results and their implications found through this study are reasonable, it is important to recognize threats to their validity.

We state explicitly that we are not claiming that our results can be used to draw conclusions about the testing of all distributed systems; we are only claiming that our study shows that there are some systems to which our analysis applies, although we have tried to look at systems that represent a diversity of distribution styles. There are a number of reasons why our results might not generalize. For example, the systems considered in this study all result from open-source development projects that would be expected to devote fewer resources to quality assurance than would commercial projects. Thus, failures that may be caught through standard means might have escaped into the field.

The failures are only those that have been reported using the defect-tracking systems; it is highly likely that failures are found and fixed by developers while using the system themselves, or that failures are seen by users, but they simply upgrade to the newest release or avoid the problem in some way without reporting it. Therefore, the study is limited to drawing conclusions about the failures that were actually reported by users. We believe that this probably limits the complexity of the scenarios that are included in the study, and it is the primary reason that we consider the results to be applicable to a test goal of exposing major user-visible failures. In other words, even if all the failures mentioned in this study were fixed, there will likely still be failures in the systems that can only be found through scenarios that are more complex than those examined here.

A major threat to validity is the objectivity of the analyst. As pointed out by El Emam and Wieczorek [3], this is a problem with all empirical defect classifications, since it is by nature a subjective activity. In our study, all classifications were performed by a single analyst (the first author), and the results might have been unwittingly skewed in some direction. The process that we have established guards against this by breaking the classification into small steps. In the future, we hope to use a collaborative approach to defect-report classification, where different analysts classify reports independently, and conflicts are resolved anonymously.

Another major threat to validity is the content of the defect reports themselves. It is possible that a report contains too much or too little information. If too many details about a scenario are included in an initial report, and the developer does not make a note of this, the analysis of the report will include more specifics than are really warranted. Conversely, if the initial report does not contain enough detail about the scenario to reliably duplicate the failure, but the developer does not use the defect-tracking system to gather the necessary information from the reporting user, then the report will be given a less-specific classification than is warranted.

6 Conclusion

In this paper we present the results of an empirical study undertaken to understand and structure the space of failure scenarios reported by users of distributed systems. Our intent is to motivate new testing techniques

tailored to distributed systems by presenting patterns, commonalities, and correlations in the user inputs and observations described in defect reports. A secondary goal of our study is to determine how frequently the distributed nature of an application comes into play when system failure occurs. Our results indicate that: a new generation of test-adequacy criteria are needed to address the failures that are due to distribution; the configurations that cause user-reported failures are reasonably straightforward to identify; and generic failure observations are strongly correlated to the distributed nature of system failures.

A secondary contribution of this work is the empirical method we developed. As far as we know, this is the first empirical study of its kind and the method is generally applicable to many varieties of software. In particular, we believe that the classification activities and input and observation categories identified could provide a basis for other defect report analyses aimed at understanding failure scenarios. Additionally, critical examination of defect reports is not a purely academic endeavor; practical techniques derived from our empirical method might provide QA managers with a means of understanding shortcomings of their team's testing processes, analogous to the way root-cause analysis can help development teams improve and evolve their processes.

As already stated, the results of this study suggest that the space of failure scenarios for distributed systems does exhibit some internal structure and is thus amenable to the the development of rational techniques and methods to attack it. In our future work, we plan to concentrate on the development of test adequacy criteria designed to induce QA personnel to execute scenarios similar to the ones described in the defect reports of this study. The input and observation categories we identified provide a rich collection of elements to consider for our adequacy criteria, and our evaluation of architectural complexity suggests that starting with simple scenarios is likely to be quite effective.

The techniques and methods we develop based on this work will be targeted at system-level testing as performed by QA personnel. We believe that the development of advanced techniques in this area is entirely complementary to existing research targeted at improving developer testing since QA teams must be conservative in their assumptions about how effectively developers are testing their code. Experience suggests that much of the complexity of testing a distributed system (e.g., observing the interaction of multiple components subjected to network effects) is not easily addressed by the developer who is testing code at the unit level. Additionally, for some highly decentralized distributed applications, such as routing and peer-to-peer data sharing, key features are emergent and therefore can only be tested at the top-most level, again clearly outside the normal purview of unit-level developer testing. This leads us to hypothesize that system-level testing will play a distinct and critical role in the development of distributed systems, no matter how much we may be able to improve the testing processes at lower levels.

References

[1] Jarir K. Chaar, Michael J. Halliday, Inderpal S. Bhandari, and Ram Chillarege. In-process evaluation for software inspection and test. *IEEE Trans. Softw. Eng.*, 19(11):1055–1070, 1993.

[2] Ram Chillarege, Inderpal S. Bhandari, Jarir K. Chaar, Michael J. Halliday, Diane S. Moebus, Bonnie K. Ray, and Man-Yuen Wong. Orthogonal defect classification-a concept for in-process measurements. *IEEE Trans. Softw. Eng.*, 18(11):943–956, 1992.

[3] K. El Emam and I. Wieczorek. The repeatability of code defect classifications. In *Proceedings of the The Ninth International Symposium on Software Reliability Engineering*, pages 322–333. IEEE Computer Society, 1998.

[4] Norman E. Fenton and Niclas Ohlsson. Quantitative analysis of faults and failures in a complex software system. *IEEE Trans. Softw. Eng.*, 26(8):797–814, 2000.

[5] Marek Leszak, Dewayne E. Perry, and Dieter Stoll. A case study in root cause defect analysis. In *Proceedings of the 22nd International Conference on Software Engineering*, pages 428–437. ACM Press, 2000.

[6] Marek Leszak, Dewayne E. Perry, and Dieter Stoll. Classification and evaluation of defects in a project retrospective. *J. Syst. Softw.*, 61(3):173–187, 2002.

[7] Paul Luo Li, Mary Shaw, Jim Herbsleb, Bonnie Ray, and P. Santhanam. Empirical evaluation of defect projection models for widely-deployed production software systems. In *Proceedings of the 12th ACM SIGSOFT International Symposium on the Foundations of Software Engineering*, pages 263–272, New York, NY, USA, 2004. ACM Press.

[8] Thomas J. Ostrand and Elaine J. Weyuker. The distribution of faults in a large industrial software system. In *Proceedings of the 2002 ACM SIGSOFT International Symposium on Software Testing and Analysis*, pages 55–64. ACM Press, 2002.

[9] B. Segall and D. Arnold. Elvin has left the building: A publish/subscribe notification service with quenching. In *Proceedings of the Australian UNIX and Open Systems User Group Conference (AUUG'97)*, 1997.

[10] Bill Segall, David Arnold, Julian Boot, Michael Henderson, and Ted Phelps. Content based routing with elvin4. In *Proceedings of Australian UNIX and Open Systems User Group Annual Conference, 2000 (AUUG2k)*, Canberra, Australia, June 2000.