Dangers and Joys of Stock Trading on the Web: Failure Characterization of a Three-Tier Web Service

Fahad A. Arshad
School of Electrical & Computer Engineering
Purdue University
West Lafayette, USA
faarshad@purdue.edu

Saurabh Bagchi
School of Electrical & Computer Engineering
Purdue University
West Lafayette, USA
sbagchi@purdue.edu

***Abstract*—Characterizing latent software faults is crucial to address dependability issues of current three-tier systems. A client should not have a misconception that a transaction succeeded, when in reality, it failed due to a silent error. We present a fault injection-based evaluation to characterize silent and non-silent software failures in a representative three-tier web service, one that mimics a day trading application widely used for benchmarking application servers. For failure characterization, we quantify distribution of silent and non-silent failures, and recommend low cost application-generic and application-specific consistency checks, which improve the reliability of the application. We inject three variants of null-call, where a callee returns null to the caller without executing business logic. Additionally, we inject three types of unchecked exceptions and analyze the reaction of our application. Our results show that 49% of error injections from null-calls result in silent failures, while 34% of unchecked exceptions result in silent failures. Our generic-consistency check can detect silent failures in null-calls with an accuracy as high as 100%. Non-silent failures with unchecked exceptions can be detected with an accuracy of 42% with our application-specific checks.**

***Keywords*-failure characterization; detection; multi-tier application;**

I. Introduction

Business critical enterprise applications require high reliability to keep their customer's trust. A customer trusts the service provider if all her web requests are satisfied, and possibly for a small fraction of requests, a noticeable failure is flagged to the customer and the customer's application state is not corrupted. An instance, where a failure of a web request is not made obvious to the consumer, but affects the end functionality, possibly at the time of a later transaction, is highly undesirable. To avoid this, the service provider needs an understanding of how software errors affect his web service, such as which failures occur in a silent manner and which in a non-silent manner; for the latter class, how are the failures logged (where—at the web server, the application server, or the database server, and within each, at which component; and the level of detail in the log). With an understanding of the failures, the service provider can start to design techniques to better log failures and to reduce the incidence of silent failures through appropriate, preferably non-intrusive, detection checks.

In this paper, we show an approach to failure characterization in an online stock trading benchmark application, called DayTrader. The application has the typical structure of a web service in that it is deployed on a web container and an EJB container (specifically, the Glassfish application server), with the persistent state being stored in a back-end database. DayTrader is a benchmark derived from IBM's performance benchmark *Trade6* and is representative of an enterprise-class three-tier web service.

We classify the failures from an end-user perspective, into three categories: *Non-silent, Non-silent-interactive*, and *Silent*. Non-silent failures are characterized by explicit error messages from the infrastructure that are visible to the end user, e.g., a servlet exception that results in an HTTP 500 status in the browser. Non-silent-interactive failures are the ones that a user notices interactively, e.g., a user searches for five valid stock quotes but is returned only three. Thus, these failures need user circumspection for detection. Silent failures disrupt web service without any manifestation to the user. An administrator is unable to detect until informed by an external agent. Non-silent and non-silent-interactive failures are generally logged, while silent failures are normally unlogged. We study each of these classes of failures by developing a fault-injection framework and by observing how DayTrader reacts to each injected fault.

We inject errors that mimic common programming faults. Based on these injected errors, in DayTrader, we find that 49% of errors from null-call injections and 34% of errors from unchecked exceptions cause silent failures [1]. A relevant study [1] shows that 29% of failures in application servers are silent. The silent failures can have serious consequences such as emptying out the portfolio of a user, or giving an incorrect view of the portfolio to the user.

Our fault injection scheme mimics typical software bugs encountered in middle-tier of an enterprise software system.

[1]We provide the raw data files for this paper, together with annotations, at the following URL: https://engineering.purdue.edu/dcsl/projects/characterize.html

Our fault trigger is method invocation and we inject three variants of *null-call* [2], *Null-Return*, where a `null` is returned on a call to a business method , *Null-Object-Return* where a `null` is cast into the return type and returned, *No-Op*, where an empty object of required return type is instantiated and returned. Further, we also inject *unchecked exceptions*, a major fault type, especially in multi-threaded architecture [3]. We find that such exceptions are quite common due to the desire (sometimes a mistaken desire) to keep the code compact enough by omitting exception handling for lots of corner cases.

By observing the behavior of DayTrader to each injected error, we recommend an intuitive application-generic check to handle the *null-call* injections where a null object is returned, and find that on average this low-cost generic check is able to detect 100% of such errors for *Null-Return*, and 29% for *Null-Object-Return*. This kind of check generalizes effortlessly across applications, and this result points web service developers or distributed middleware developers toward building primitives that will make such checks easy to insert in the application. However, one kind of null-call injection, *No-Op*, where an object of the correct return type is returned, but without executing any functionality of the called function, is impossible to detect to any meaningful degree using application-generic checks.

Application-specific checks require a study of application characteristics, but here we focus on checks that can be easily deduced by standard machine learning algorithms. We use learning to induce two application-specific checks—checking the length of any web request (in terms of the number of software components) and checking the first and the last method invoked in any web request. We find, that of the different kinds of web requests, these checks perform well for those that are not data dependent, i.e., the control path is the same regardless of the parameters in the request. However, two of the ten web requests in DayTrader are data dependent and these are challenging to detect and point out the need for checks that have a more detailed understanding of the application.

To summarize, we believe this work makes the following contributions:

- It provides the distribution and classification of silent, non-silent, and non-silent-interactive failures in a representative three-tier web service.
- It presents an application-generic consistency check that can detect a subclass of errors that cause silent failures.
- It presents low-cost application-specific consistency checks that have a high coverage for unchecked exceptions.

The rest of the paper is organized as follows. Section II discusses background and related work. Section III describes the system architecture. Section IV illustrates the fault injection methodology. Section V presents failure classification. Section VI describes consistency checks. Section VII explains experiments and results. We describe the lessons learned in section VIII and conclude with section IX.

II. Background and Related Work

Studying failure characterization involves two general techniques, i.e., analyzing bug databases and employing fault injection. In the former case, multiple bug reports are analyzed and correlated to classify failures according to some criteria, e.g., [4] studies failures in JVM, classifies them based on failure manifestation, failure source and failure frequency. [1] studies failures in application servers and shows that 75% of all failures are related to service semantics and require application-specific efforts. Bug repository based failure analysis has some limitations, i.e., quality of bug reports and subjective administration, etc., therefore many researchers have resorted to fault-injection, e.g., [5] emulates classes of software faults. For evaluation based on fault injection, different injection campaigns depending on *where* and *how* to inject errors exist. Faults can be injected in source code [6], or they can be emulated at runtime using an emulation technique [5]. We emulate faults in the application, and classify them based on manifestation at client's browser. Failures can also be analyzed at different layers in the software system, i.e., operating system [7], [8], network controller [9], virtual machine [4], application server [1] and application [2]. The type of injected fault for a given system also varies depending on the system under test, e.g., [8] shows that, one of the major causes *null pointer* is responsible for system crashes in operating system kernel. We study null-calls in a similar fashion at the application layer.

Detecting injected faults as studied by [10] can be done at a system level or at an application level. The study shows that low-level system monitoring tools are not enough to detect application level errors. Pinpoint [11], [2] treats failures as anomalies and applies machine learning techniques to detect application level failures in an application-generic way. For application-specific evaluation, assessing application problems is important, e.g., [12] applies a set of tests, by mutating call parameters, and analyzes robustness of web services code and application server infrastructure. We follow a similar approach, but instead of robustness failures, we focus on more general emulated failure classes.

Characterizing by failure type, especially variants of silent failures is studied by [9]. The study detects silent failures that impact database clients in a telephone network. It uses software fault injection at database tier and does data audit to reduce silent failures. Our study focuses on middle-tier and is for a three-tier application. Another study on silent errors in OS [7] estimates the percentage of silent failures by comparing the execution of a given API across different operating systems. Application-specific detection, diagnosis and recovery are all important. A study in [13] shows

that application generic recovery techniques are not enough to recover from application failures. Our paper tests this hypothesis from a detection perspective and is helpful to give useful hints for diagnosis. In addition, if detection results by our checks are taken into account by an application logger, then this can improve logging quality as shown in [6].

III. Three-tier Architecture

We build a three-tier system with clients, a middleware and a back-end database. The client layer is composed of a browser emulator, Grinder [14]. It sends web requests to the middleware that hosts an application server, Glassfish. Middleware executes application logic and builds database requests, that are sent to back-end database. Our system architecture, shown in Figure 1, implements DayTrader, a java based web service application.

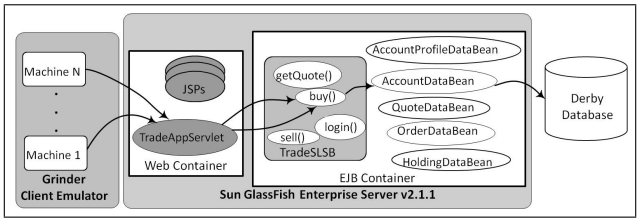


Figure 1. DayTrader System Architecture

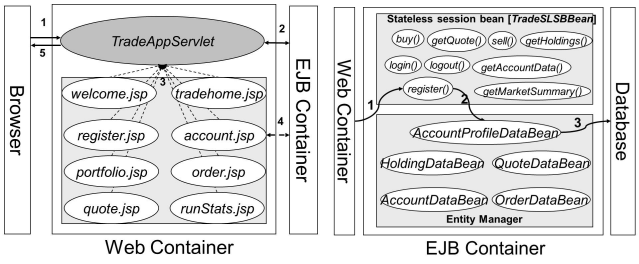
Web Request	EJB Requests
Login	getClosedOrders->login->getAccountData->getHoldings->getMarketSummary
Home	getClosedOrders->getAccountData->getHoldings->getMarketSummary
Account	getClosedOrders->getAccountData->getAccountProfileData
Portfolio	getClosedOrders->getHoldings->getQuote$*$[2]
Quotes	getClosedOrders->getQuote$*$
Buy	getClosedOrders->buy->updateQuotePriceVolume
Sell	getClosedOrders->sell->updateQuotePriceVolume
Logoff	logout
Update	getClosedOrders->updateAccountProfile->getAccountData->getAccountProfileData
Register	register->login->getAccountData->getHoldings->getMarketSummary

Table I
Web Request to EJB Request mapping

A. Middleware

Middleware is composed of an application server, *Glassfish*, that executes application logic functions and builds corresponding database queries at runtime. The application, *DayTrader*, is deployed on top of the application server. On deployment, two major parts of application server: web container and EJB container work to receive and process HTTP web requests from clients.

Web container extends the web server functionalities and gives an environment to run servlets and JavaServer Pages (JSPs). DayTrader has a main controller servlet, *TradeAppServlet* (Figure2(a)) that is responsible for handling incoming HTTP requests and is the entry point for all client requests. It also consists of supporting JSPs: *welcome.jsp, register.jsp, portfolio.jsp, quote.jsp, tradehome.jsp, account.jsp, order.jsp, runStats.jsp* that help to manage dynamic content. Web container executes a sequence of steps for every web request initiated by client browser: (1) *receive* an HTTP request; (2) *send* a request to the EJB in the EJB container (optionally, it may need to instantiate the EJB first); (3) *set-up* the response page template based on the request by instantiating the appropriate JSP; (4) *process* the response from the EJB and then use it to populate the response page template created by the JSP in step (3); (5) send response back to client browser.



(a) Web Functionality (b) EJB Functionality

Figure 2. DayTrader: Web and EJB container in the middleware layer

The EJB container is responsible for executing application's business logic. For DayTrader, it contains a *stateless session bean* and several *entity* beans as shown in Figure 2(b). The stateless session bean, *TradeSLSBBean*, maintains state only for a given business method call, e.g. `TradeSLSBBean.buy()`, and does not maintain state across business methods. Entity beans represent relational data corresponding to database tables with one-to-one mapping. Each entity bean corresponds to a table in the database where as each instance of the entity bean corresponds to a row in that table. On receiving an EJB request from web container, EJB container executes the following sequence of steps, also shown in Figure 2(b): (1) *invoke* the requested EJB business method in *TradeSLSBBean* (2) *invoke* one or more entity beans (3) *send* SQL query to the database table corresponding to the entity bean to insert, update or load from back-end database.

B. Client Requests

DayTrader customers send web requests to the middleware layer. Each web request results in several EJB method invocations in a stateless session bean, which we call *EJB requests*. An example invocation is *"Login"* web request, where a user enters her username and password, and sends the web request to the web container. The web container calls the *TradeSLSBBean.login()* in EJB container. This in turn calls several different entity bean methods according to application logic. Entity bean methods invoke *load, update* and *insert* operations on corresponding tables in the

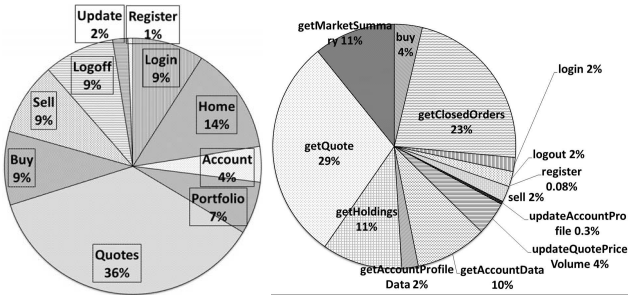
[2]zero or more invocations

database.

The client layer is represented by Grinder [14], a load testing distributed framework that emulates simultaneous users. We use a client workload provided by the DaCapo benchmark suite [15] for stressing DayTrader. The workload comprises of a set of stocks, users, and user sessions. Each user session starts with *"Login"*, followed by other possible web requests as defined by the application design. Each request ends with *"Logout"*. For a new user, the session starts from a *"Register"* request. A typical user session may be: *Login, Home, Portfolio, Sell, Account, Update, Quote, Buy, Logout*. We show the client web request distribution from our workload in Figure 3(a). Each web request results in multiple EJB requests (Table I). The distribution of EJB requests is shown in Figure 3(b). We observe that *"Quotes"*, a read web request, is exercised with the highest percentage. This is expected of a real online user, who looks up stock quotes most of the time. Write web request like *"Buy"* and *"Sell"* have equal percentage.

C. Database

Database layer is composed of an independent database management system (DBMS) that stores tables specified by application designer. The DBMS, *derby*, is responsible for processing database requests, i.e., load, update and insert SQL queries.



(a) Web (b) EJB

Figure 3. Web and EJB Request Distribution for 750 users

IV. Fault Injection Methodology

In order to characterize faults, we built a fault injector that ultimately leads to silent and non-silent failures in web service based applications. Our faultload is representative of typical faults in three-tier applications. Our injected faults are similar to faults injected in [2], and we claim that they are general enough to cover a wide range of low-level programming errors. The key questions while designing our fault injector were: *What kind of faults to inject*? *Where to inject these faults*? *When to inject the faults*?

We inject *undeclared* runtime exceptions, and study the effect of injection. We find that the practice of not handling

```
foo(){
...
RObject x = bar();
...
}
RObject bar(){
...
return RObject;
}
```
Original Code

```
foo(){
...
//RObject x = bar();
Class RObjectClass = RObject.getClass();
RObject x = RObjectClass.cast(null);
...
}
```
Null-Object-Return Code

```
foo(){
...
//RObject x = bar();
RObject x = null;
...
}
```
Null-Return Code

```
foo(){
...
//RObject x = bar();
RObject y = new RObject();
RObject x = y;
...
}
```
No-Op Code

```
foo(){
...
RObject x = bar();
...
}
```

```
RObject bar(){
throw new java.lang.RuntimeException();
...
return RObject;
      }
```
Unchecked Exception Code

Figure 4. Example code for emulated error types

such exceptions is quite common due to the desire (sometimes a mistaken desire) to keep the code compact enough by omitting exception handling for lots of corner cases. We decided not to inject *declared exceptions* as these exceptions are normally handled and masked by the application code itself. We also inject *Null-call*, where the called component is never executed. Specifically, we experiment with three variants of null-call, *Null-Return, Null-Object-Return, No-Op* as shown in Figure 4 where `foo()` calls `bar()`.

Null-Return: By injecting a Null-Return error in EJB container, we mean that when a call to a method is made, a `null` is returned without executing the logic of the method.

Null-Object-Return: By injecting a Null-Object-Return error in EJB container, we mean that when a call to a method is made, then on return a `null` is cast into the return type of the method.

No-Op: By injecting a No-Op error in EJB container, we mean that when a call to a method is made, an empty object of the correct return type is created by using the `new` operator. This empty object is returned to the caller.

Unchecked Exceptions: On a call to a method, this throws an unchecked exception in the callee. The list of unchecked exceptions, that we inject, along with the typical reasons for their occurrence is: *ArithmeticException* due to a divide by zero, *ArrayIndexOutOfBoundsException* due to out of bound array access and *ClassCastException* due to bad casting, e.g., converting non-numeric to integer.

In deciding fault locations, we had two major goals: Injection should be non-intrusive, which means parsing the source code to find good locations is avoided. Secondly, injections should mimic real programming mistakes. To achieve this, we emulate the manifestation of fault conditions in EJB container. We emulate injection in *TradeSLSBBean's*

methods in EJB container using interceptors. In addition, our chosen error types, for injection, are representative for capturing the low-level programming mistakes as shown by previous studies. Specifically, [1] shows that 60% of all failures are manifested as thrown exceptions. The number of previous works about *Null Pointer Exceptions* [8] is a proof of how common this type of failure is, and the motivation to inject null-call variants in our work. Next we choose fault *triggers*, i.e., when to inject a fault. We inject whenever a call is made to invoke one of the methods of *TradesSLSBBean* in EJB container. Our fault injection implementation is generic and can inject these faults in any typical three-tier web application.

We use *interceptors* for EJB container and *filters* for web container to implement the generic injection and detection framework. An interceptor intercepts a call whenever a call is being made or returned from a java session bean. Similarly, a filter intercepts calls to web container. Both interceptors and filters can run custom java code and can be configured to throw exceptions at runtime according to some failure injection rate.

V. Failure Classification

We classify failures with respect to their manifestation at user end. We relate this manifestation to whether the failure is originated from the application, or from the underlying container. We categorize failure manifestation in three groups: *Non-silent, Non-silent-interactive* and *Silent* as shown in Figure 5.

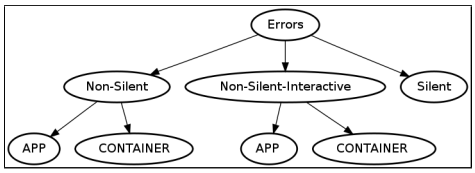


Figure 5. Failure classification

Non-silent: On receiving an HTTP web request from user's browser, if the response shows a blank page or an exception, we classify it as a non-silent failure. Non-silent failure can be either logged by the application or by the container as shown in Figure 5. A non-silent failure logged by an application comes from a `println` statement in application's logger. It shows that the developer of the application anticipated the failure. A non-silent failure that is logged by other modules in the container e.g. `javax.enterprise.system.container.web` is an example of a failure for which the application developer did not anticipate.

Non-silent-interactive: A non-silent-interactive failure occurs when a user interactively realizes in the same transaction that a failure has occurred. An example of a non-silent-interactive failure is a Null-Return injected in the `login()` method, which reports "Could not find account for + xxxxxx" in the browser for an existing user. Another example is when a user searches for five quotes but only three quotes show up in the result. Non-silent-interactive failures can be either logged by the application, or by the container.

Silent: Silent failures are those whose manifestation is never captured in logs until after multiple web requests, when a user recognizes that her transactions did not go through. An example would be when a user buys a stock, and when after a few days wants to sell it, she cannot find that stock in her portfolio. When the user contacts the administrator, the administrator is unable to determine from logs that the user purchased such a stock. This is the kind of failure mode that keeps owners of web services awake at nights and provides employment to many legions of lawyers.

VI. Consistency Checks

A. Application Generic

Null-call check: A rule is placed as an EJB interceptor which checks for `null` on return from every EJB request. Whenever a session bean's method is called, this rule flags if the return value is `null`. This rule has false positives as some of EJB requests like `logout()` do return `null` in normal case. We quantify the accuracy and precision of this rule.

B. Application Specific

Call-length check: For a given web request type, this check counts the number of corresponding EJB requests invoked, e.g., "Account" web request invokes getClosedOrders->getAccountData->getAccountProfileData, which makes the call-length of this web request three. Call-length for each web request is learned from training as explained in the next sub-section. We observe that call-length of web requests fall into two categories: *data-dependent* and *data-independent*. Data-independent web requests have fixed call-length in normal case, e.g., "Account" invokes three EJB requests. Data-dependent web requests have variable call-lengths based on the entries in the database or entries entered by the user in a search box. Call-lengths learned through training are used for this check; for data-dependent requests, the minimum observed call-length is used. This check is triggered in a filter on return from a servlet, that handles the web request in web container. If the observed length differs from the length inferred through training, an error is flagged.

For implementing this check, we use the `java.lang.ThreadLocal` API to monitor the web request as it passes through the three-tier system. ThreadLocal is used to create and update a local vector variable, called *call-vector*. As the call passes through to the EJB container, an interceptor adds every EJB request for the given web request to the call-vector. On return to the filter in the web container, call-vector has the given

web request along with all the EJB requests invoked for that web request.

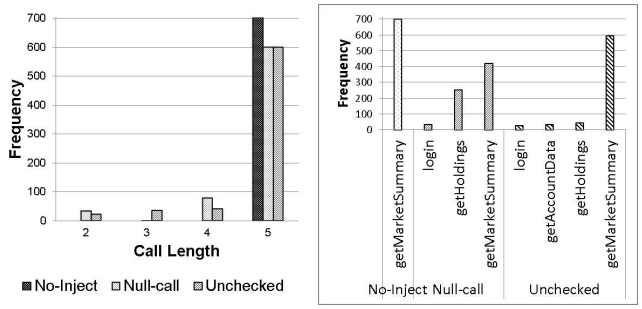
Head-tail check: This check requires the normal tail EJB request names, that are learned in a training run as described in next sub-section. For a given web request, this check monitors head and tail EJB request, e.g., in `foo()->bar()->baz()` chain, if `foo()` and `baz()` are invoked in a web request, then the web request is assumed to successfully complete and no alarm is raised. The trigger for this check is the same as for call-length check, i.e., return from the servlet in a filter.

For implementing this check, we use the same call-vector, as in previous check. Errors like unchecked exceptions that can result in shortening of web requests will be caught by this check. The head-tail check is able to detect errors in some data dependent web requests which call-length cannot detect, e.g., in "Portfolio" as long as the length is ≥ 2, call-length check is satisfied, but the head-tail check can catch the error if either the head or the tail does not match.

C. Learning Parameters for Application-Specific Checks

To learn the rule parameters, we run the workload in a training mode and find frequency distribution of call-length for each web request, as shown in Figure 6(a) for "Login". We find normal call-length for a given web request by choosing the call-length with the highest frequency, e.g., for "Login" this is 5. For data-dependent web requests, "Portfolio" and "Quotes", the call-length check is of the form "$\geq$*value*". We set the *value* equal to the minimum call-length observed in training. This keeps the processing for the check simple and not too tied to the application, because we do not have to extract at runtime the part of data which determines the call-length for data-dependent web requests.

For the second kind of rule, we follow the same training approach as for data-independent web request in the call-length rule, and find the head and the tail EJB methods that give the highest frequency. Figure 6(b) shows the distribution of tail EJB methods, and shows that `getMarketSummary()` is the normal tail EJB request for "Login" web request.



(a) Frequency Distribution of Call-length for Login web request

(b) Frequency Distribution of Tail EJB requests for Login web request

Figure 6. Distributions for learning parameters for application-specific checks

VII. Experiments and Results

Our experiments quantify the distribution of occurrence of failure classes, for all types of injected errors. Further, we quantify the accuracy, precision and cost for our consistency checks. In all our experiments, we uniformly inject in 5% of all EJB requests.

A. Testbed and Metrics

For our experiments, we use 750 concurrent users running the workload. The emulation of 750 users is achieved by Grinder framework that uses 15 client machines each emulating 50 users. Users send their web requests concurrently to a machine running the Glassfish application server, which sends the corresponding database calls to a derby database running on a third machine. The client machines are Quad-Core AMD Opteron 2.2 GHz with 8182 MB of memory. The application server and the database machines are an Intel Xeon 3.4 GHz with 1024 MB of memory.

To evaluate each of our consistency checks, we define *Accuracy* as the ratio of the number of correctly detected errors to injected errors. *Precision* is defined as the ratio of the number of correctly detected errors to the number of total detected errors. For application generic check, we give accuracy and precision at the granularity of an EJB request. For our application specific checks, we give accuracy and precision at the granularity of a web request, as they are evaluated at the end of each web request.

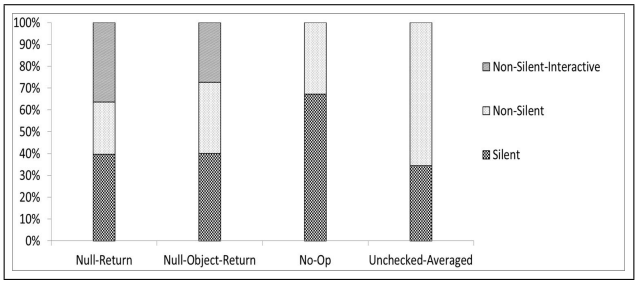


Figure 7. Distribution by Failure Class

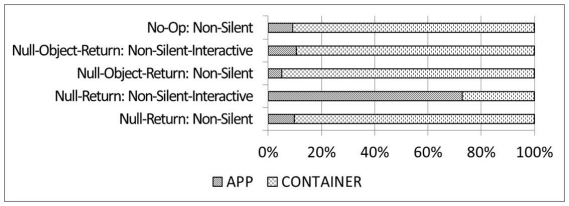


Figure 8. Failure Distribution Breakdown for the two kinds of Non-Silent Failures

B. Failure Distribution in EJB Container

In this section, we determine the distribution of failure classes across each injection type in EJB container. For the three kinds of unchecked exceptions, we aggregate, and give average accuracy, precision, and overhead in our results. Figure 7 shows that 40% of injected Null-Return errors are

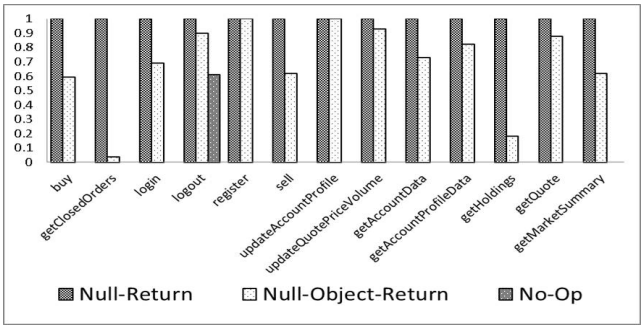
silent, which neither the end user nor the administrator can detect and they are not logged at the server either by the application or by the container. Non-silent failures constitute 24%, while non-silent-interactive failures are 36%. For Null-Object-Return type, silent failures remain almost the same as in Null-Return, while non-silent failures increase to 33% and non-silent-interactive decrease to 27%. This increase is because, when the callee returns, in Null-Object-Return, it returns an expected return type object, but with `null` in it. On return, caller continues with its processing, until it tries to use the returned object, and this is when a problem is detected. For No-Op errors, we observe 67% silent failures and 33% non-silent failures with no occurrence of non-silent-interactive failures. This happens, because for this type of injected error, we instantiate with `new` operator an object of the return type, and then return it to caller. For example, in `getQuote()` EJB request, the caller expects a "QuoteDataBean" object on return. When we inject Null-Return, the caller handles it in a general catch block (also logs it), and the quote result is not shown in the browser. Thus, the user interactively detects the failure. When we inject No-Op, the catch block at caller is never activated as the caller gets a "QuoteDataBean" object on return. Caller continues its buggy execution without logging anything and therefore a silent failure happens.

The objective of the next analysis is to see that for the failures that are logged (i.e., are non-silent), where are they logged—by the application or by the EJB container. We show error class breakdown, as logged by the application and the container in Figure 8 for both non-silent and non-silent-interactive failures. For Null-Return, out of all non-silent failures, 10% are logged by application, while 90% by container's logger. This suggests the need of null checking mechanisms at more places in application code. On the other hand, the majority of non-silent-interactive Null-Returns are detected (73%) by the application. This is understandable as an application developer would focus more on handling scenarios where a user should not see a failure interactively. The distribution for both non-silent and non-silent-interactive failures for Null-Object-Return errors is similar with container logging majority of errors. This is conceivable for non-silent failures, but a low percentage of non-silent-interactive (11%) failures logged by application entails for extensive sanity checks on returned objects. For No-Op errors, 91% of the non-silent errors are detected by the container while the rest are detected by the application. This is because the developer did not anticipate the occurrence of No-Op errors and did not handle them in application code.

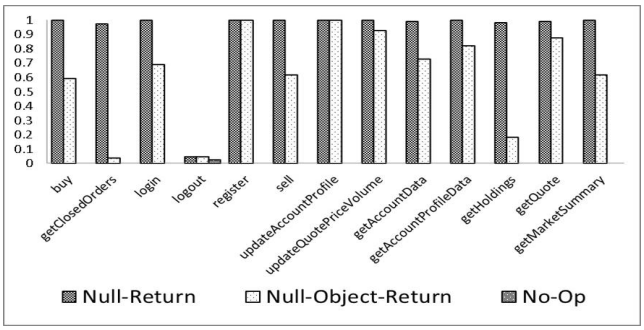
For unchecked exceptions, we see from Figure 7 that they are either silent or non-silent. Also, those that are silent (33%), i.e., normal operation in user's browser, are logged at the server. Majority of unchecked exceptions (67%) are non-silent, and all of them are logged by the container.

Accuracy of Null-Call Check		
Type of Injection	Failure Class	Accuracy
Null-Return	Silent	100%
	Non-Silent	100%
	Non-Silent-Interactive	100%
Null-Object-Return	Silent	30%
	Non-Silent	22%
	Non-Silent-Interactive	35%
No-Op	Silent	1%
	Non-Silent	0%
	Non-Silent-Interactive	0%

Table II



(a) Accuracy



(b) Precision

Figure 9. Null-call Check: Performance of application-generic check that matches `null` on return

C. Application Generic Check

We implement our null-call application generic check, as defined in section VI. The accuracy and precision of this check is shown in Figure 9(a) and Figure 9(b) respectively, for each of the thirteen types of EJB requests. As we check for `null` on return from a method call, this check shows an accuracy of 100% for Null-Return.

For Null-Object-Return, accuracy is less than Null-Return. This type of error escapes the null-call check at caller because an object of the same type as expected is returned. For No-Op, no method except `logout()` is detected by null-call check. The reason for this is that, `logout()` expects a null on its return and null-call check catches it. This also raises false alarms as seen by low precision (4%) for `logout()`. In general, as we move from Null-Return to Null-Object-Return to No-Op, accuracy reduces. Table II shows accuracy of null-call check, for each failure class in each of the injection types. Unchecked exceptions are not

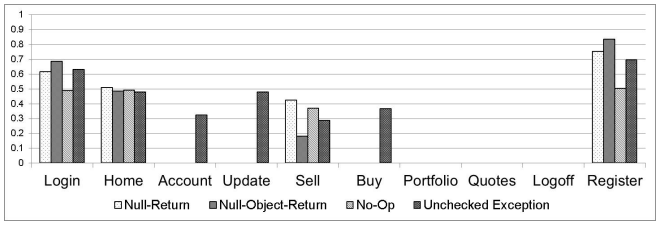
detected at all by the null-call check.

D. Application Specific Checks

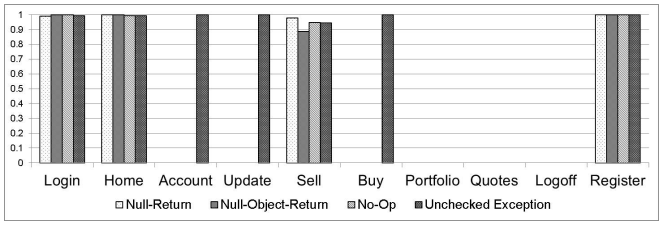
We show accuracy and precision of each check in Figure 10 and Figure 11.

Accuracy of Call-Length Check		
Type of Injection	Failure Class	Accuracy
Null-Return	Silent	0%
	Non-Silent	48%
	Non-Silent-Interactive	30%
Null-Object-Return	Silent	1%
	Non-Silent	57%
	Non-Silent-Interactive	11%
No-Op	Silent	0.2%
	Non-Silent	60%
Unchecked	Silent	0%
	Non-Silent	41%

Table III



(a) Accuracy



(b) Precision

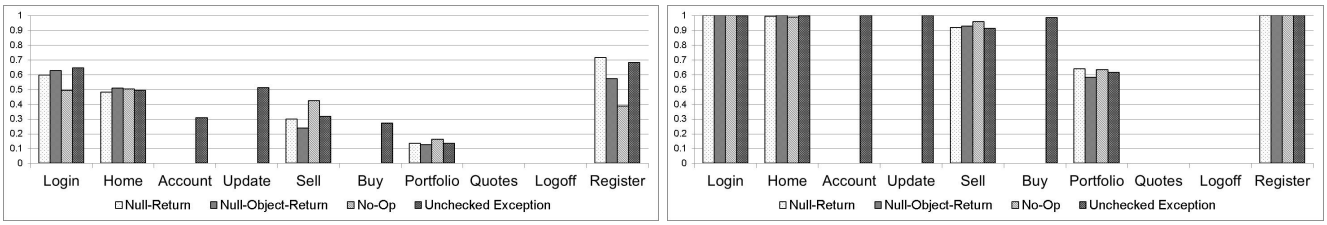
Figure 10. Call-Length Check: Performance of application-specific check that matches the normal length of a web request to detect

Call-length check: We observe in Figure 10(a), that all three variants of null-call in 6 of the 10 web requests ("Account", "Update", "Buy", "Portfolio", "Quotes", and "Logoff") are not detected by this check. From DayTrader's application code for the above 6 web requests, we find that there is no code that checks for returned `null` and handles it. Handling `null` means that there is specialized code that executes when a `null` is returned, typically to terminate the web request. Therefore, for these web requests, subsequent EJB requests are made and the total call-length is not affected. For web requests that handle a returned `null`, EJB request sequence is cut short, and an error is detected by this rule—in "Login", "Home", "Sell" and "Register". For unchecked exceptions, this check does not detect in "Portfolio", "Quotes" and "Logoff", while it detects with an average accuracy of 46% across the rest of the web request types. The rule for "Portfolio" and "Quotes" is of "$\geq$ *value*" form with learned *value* equal to 2 and 1 respectively, and also confirmed by Table I. The reason for failing to detect is that the caller for the first EJB request, `getClosedOrders()` (always called since it is the first EJB request), contains a generic catch block, that handles the exception, and proceeds with the next EJB request. This means that "Quotes" would never be detected by this check as it checks for ≥ 1 EJB request only. Also, since `getClosedOrders()` always proceeds with the next EJB request, `getHoldings()` in "Porfolio" is always invoked. This means that this check will not detect "Portfolio", as the rule is that the EJB invocation chain length is ≥ 2. For web requests that are detected by this check, precision in general is good as shown in Figure 10(b).

Accuracy of call-length check within each failure class for each injected error is shown in Table III. It detects 57% of non-silent failures for Null-Object-Return type, which is an improvement when compared with null-call check. It also detects 60% of non-silent failures for No-Op, which the null-call check did not detect. In addition, it is able to detect 41% of unchecked non-silent failures because the length of web requests is changed.

Head-tail match: This check requires knowledge of first and last EJB request names for a given web request. The results of this check in Figure 11 show that, it is able to detect errors in "Portfolio", which the previous rule did not detect. To see why this happens, note that the "Portfolio" web request comprises of the EJB requests `getClosedOrders()`, `getHoldings()`, followed by zero or more invocations of `getQuote()`. The number of invocations will depend on how many stocks the user has in her portfolio. When the target of the error injection happens to be either `getClosedOrders()` or `getHoldings()`, this check detects the error because `getQuote()` never gets invoked. However, due to the nature of the workload (most users have multiple stocks in their portfolio), in a large majority of the cases (more than 80%) `getQuote()` is invoked at least once. In these cases, the head-tail match check fails. The averaged precision for "Portfolio" is 62%. For "Quotes" web request, this check does not detect for the same reason as in call-length check. The accuracy and precision results for the rest of the web requests are similar to call-length check results. Accuracy of head-tail check within each failure class for each injected error is shown in Table IV. The results are similar to call-length check.

Overhead: To understand the time overhead of each rule, we measure the time spent per web request, in the web container, the EJB container and the back-end database. The time spent in database is an overestimate as it includes network latency between the EJB container and database. We observe this latency to be small relative to time spent in the database. We compare these times to a baseline case, that has no detection mechanism. We show the overhead of call-length check in Figure 12. We aggregate the results for Null-Return, Null-Object-Return and unchecked types into



(a) Accuracy (b) Precision

Figure 11. Head-tail Match: Performance of application-specific check that matches the first and the last EJB request names to detect

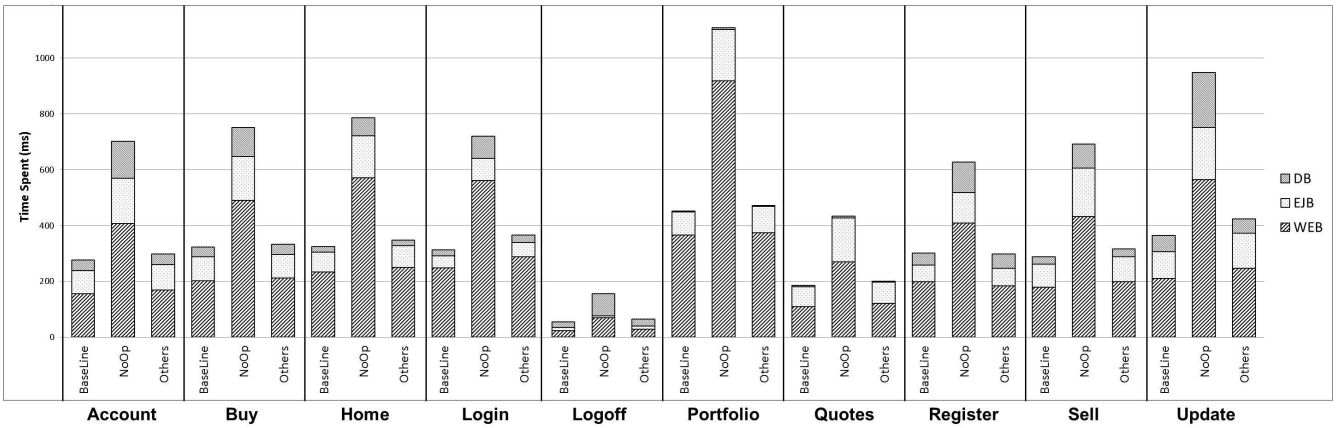


Figure 12. Overhead of application-specific checks in WEB, EJB and DB in terms of time spent per web request

Accuracy of Head-Tail Check		
Type of Injection	Failure Class	Accuracy
Null-Return	Silent	0%
	Non-Silent	45%
	Non-Silent-Interactive	36%
Null-Object-Return	Silent	1%
	Non-Silent	60%
	Non-Silent-Interactive	11%
No-Op	Silent	0%
	Non-Silent	68%
Unchecked	Silent	0%
	Non-Silent	42%

Table IV

an "Others" category. We do this as results for injections in "Others" category are similar. We observe, that overhead is minimal for time spent in each of WEB, EJB and DB for the "Others" category. No-Op, however has considerable overhead. The time spent for No-Op in WEB is 2.5x, in EJB 1.9x and in DB 3x. The reason for this extra overhead is from the fact, that in No-Op, we instantiate a new object of the return type, which the caller accepts and processes for a longer time. The overhead results for head-tail check are similar, since it uses the same underlying monitoring.

VIII. Lessons Learned

In this section, we point out the lessons that we deduce from our injections and analysis of the resulting data. These lessons may be instructive to an application developer of web services.

(1) A caller method should flag a returned *null* as an incorrect operation and cause a failure notification to the end user. An application developer should log these cases, thus causing a non-silent error. While this does happen with some methods, further validation of checking that the returned object type is correct and its values are reasonable is not done at all in this application. Such validation should also be a part of careful application development. The first kind of validation can be created through static analysis of source code, while the latter will need application-specific checks inserted by the developer.

(2) In case of a correct return type, i.e., *Null-Object-Return*, a caller should have some sanity check, e.g. check size of returned object is greater than a threshold etc., before proceeding with the rest of its operations. This would detect the error immediately on return rather than delaying it till the null-return-object is used, thus reducing error propagation.

(3) The No-Op errors is very hard to detect in an application-generic manner because the right kind of data structure is returned, only no logic is executed in the invoked function.

(4) We find that detecting failures caused by unchecked exceptions is difficult. Therefore, for application developers aiming for high reliability of their developed web services, we would suggest putting explicit `catch` blocks specifically for each of the common unchecked exceptions. Our list of three unchecked exceptions is a good candidate set for the ones for which exception handling should be provided because we find that they occur most frequently.

(5) Silent failures constitute a significant percentage of the injected errors. Mechanisms both internal and external to the

application to log possible manifestations to logs and convert some silent errors to non-silent will be very valuable.

(6) Data dependent web requests are hard to detect. We speculate they will require more sophisticated data-specific checks or stateful monitoring in the web container. They will require extracting and interpreting data in the request, something our current application-specific checks cannot do.

IX. Conclusion

This paper presents an experimental study on characterizing silent and non-silent failures in a three-tier web service application. It uses a fault-injection based scheme and classifies failures from a user's perspective. Three variants of null-call and three types of unchecked exceptions are used in the fault injection scheme. The reaction of injecting each of these faults in DayTrader is classified into three failure classes, *Non-Silent*, where the server notifies the client of the failure, *Non-Silent-Interactive*, where a client detects by observation of an obvious failure manifestation in the same transaction, and *Silent*, where neither the client nor the administrator is able to detect. This study shows that 49% of errors averaged across three null-call variants result in silent failures, while 34% of unchecked exceptions cause silent failures. The majority of non-silent (90%) failures in Null-Return are logged by container, while the majority of non-silent-interactive (73%) failures are logged by application.

A second issue investigated is the applicability of consistency checks to detect the injected faults. A simple null-call check that looks for `null` on return gives an accuracy of 100% for Null-Return in each of the three failure classes. Null-call check is an application-generic rule that can be implemented with almost zero cost. Our study also includes application-specific checks, i.e., *Call-length check*, where length of a web request is monitored , and *Head-tail check*, where the first and last EJB request name in a given web request are matched. Quantitatively, these two checks are able to detect between 40%-70% of the various error classes, except for the No-Op errors which go mostly undetected. In ongoing work, we wish to develop more targeted application-specific checks learned through machine learning techniques. We are also expanding the work to include errors due to concurrency and timing perturbation.

References

[1] J. Li, G. Huang, J. Zou, and H. Mei, "Failure analysis of open source j2ee application servers," in *Quality Software, 2007. QSIC '07. Seventh International Conference on*, 2007, pp. 198 –208.

[2] M. Chen, E. Kiciman, E. Fratkin, A. Fox, and E. Brewer, "Pinpoint: problem determination in large, dynamic internet services," in *Dependable Systems and Networks, 2002. DSN 2002. Proceedings. International Conference on*, 2002, pp. 595 – 604.

[3] B. Jacobs, P. Muller, and F. Piessens, "Sound reasoning about unchecked exceptions," in *Software Engineering and Formal Methods, 2007. SEFM 2007. Fifth IEEE International Conference on*, 2007, pp. 113 –122.

[4] D. Cotroneo, S. Orlando, and S. Russo, "Failure classification and analysis of the java virtual machine," in *Distributed Computing Systems, 2006. ICDCS 2006. 26th IEEE International Conference on*, 2006, p. 17.

[5] H. Madeira, D. Costa, and M. Vieira, "On the emulation of software faults by software fault injection," in *Dependable Systems and Networks, 2000. DSN 2000. Proceedings International Conference on*, 2000, pp. 417 –426.

[6] M. Cinque, D. Cotroneo, R. Natella, and A. Pecchia, "Assessing and improving the effectiveness of logs for the analysis of software faults," in *Dependable Systems and Networks (DSN), 2010 IEEE/IFIP International Conference on*, 28 2010-july 1 2010, pp. 457 –466.

[7] C. Shelton, P. Koopman, and K. Devale, "Robustness testing of the microsoft win32 api," in *Dependable Systems and Networks, 2000. DSN 2000. Proceedings International Conference on*, 2000, pp. 261 –270.

[8] W. Gu, Z. Kalbarczyk, Ravishankar, K. Iyer, and Z. Yang, "Characterization of linux kernel behavior under errors," in *Dependable Systems and Networks, 2003. Proceedings. 2003 International Conference on*, june 2003, pp. 459 – 468.

[9] S. Bagchi, Y. Liu, K. Whisnant, Z. Kalbarczyk, R. Iyer, Y. Levendel, and L. Votta, "A framework for database audit and control flow checking for a wireless telephone network controller," in *Dependable Systems and Networks, 2001. DSN 2001. International Conference on*, july 2001, pp. 225 –234.

[10] L. M. Silva, "Comparing error detection techniques for web applications: An experimental study," in *Proceedings of the 2008 Seventh IEEE International Symposium on Network Computing and Applications*. Washington, DC, USA: IEEE Computer Society, 2008, pp. 144–151.

[11] E. Kiciman and A. Fox, "Detecting application-level failures in component-based internet services," *Neural Networks, IEEE Transactions on*, vol. 16, no. 5, pp. 1027 –1041, 2005.

[12] M. Vieira, N. Laranjeiro, and H. Madeira, "Assessing robustness of web-services infrastructures," in *Dependable Systems and Networks, 2007. DSN '07. 37th Annual IEEE/IFIP International Conference on*, june 2007, pp. 131 –136.

[13] S. Chandra and P. Chen, "Whither generic recovery from application faults? a fault study using open-source software," in *Dependable Systems and Networks, 2000. DSN 2000. Proceedings International Conference on*, 2000, pp. 97 –106.

[14] "The grinder, a java load testing framework," `http://grinder.sourceforge.net/`.

[15] "The dacapo benchmark suite," `http://www.dacapobench.org/`.