Appears in Proceedings of the 3rd IEEE Workshop on Internet Applications (WIAPP), San Jose, CA, June 2003

Automatic Failure-Path Inference: A Generic Introspection Technique for Internet Applications

George Candea, Mauricio Delgado, Michael Chen, Armando Fox
{candea,mdelgado,fox}@cs.stanford.edu, mikechen@cs.berkeley.edu

Abstract

Automatic Failure-Path Inference (AFPI) is an application-generic, automatic technique for dynamically discovering the failure dependency graphs of componentized Internet applications. AFPI's first phase is invasive, and relies on controlled fault injection to determine failure propagation; this phase requires no a priori knowledge of the application and takes on the order of hours to run. Once the system is deployed in production, the second, non-invasive phase of AFPI passively monitors the system, and updates the dependency graph as new failures are observed. This process is a good match for the perpetually-evolving software found in Internet systems; since no performance overhead is introduced, AFPI is feasible for live systems. We applied AFPI to J2EE and tested it by injecting Java exceptions into an e-commerce application and an online auction service. The resulting graphs of exception propagation are more detailed and accurate than what could be derived by time-consuming manual inspection or analysis of readily-available static application descriptions.

1. Introduction

We would like to build autognosis into complex software systems, and enable them to purposely expose themselves to faults in order to determine how they react to failures. A generic mechanism for this kind of introspection would help complex software systems become more autonomous, as the resulting information could be used, for example, in automated recovery. Knowing how far a fault has propagated may allow the system to limit recovery to only those parts of the system that are affected; such surgical microrecovery can result in significantly lower time-to-recover (TTR) than whole-system recovery [8]. Especially in Internet systems, lowering TTR is often even more important than increasing reliability [15]. Failure propagation information can also be used for root-cause analysis [30, 18] to help target the debugging effort.

To date there have been at least three challenges in realizing the above goals. First, most problem-determination work assumes the dependency graph has already been built. Failure propagation information (e.g., a fault tree) is manually constructed based on detailed knowledge from the designers; the process is tedious and error-prone, and the resulting dependency graph reflects reality only to the extent designers' knowledge is accurate and complete.

Second, if the system evolves, analysis must be redone. For many interesting systems, re-analysis would quickly become the bottleneck to system evolution. Furthermore, it is difficult to keep a system and its manually-generated documentation in sync [6]. Internet systems evolve particularly rapidly due to market pressures [17], making manual analysis impractical.

Third, transient and intermittent failures, in particular, may be triggered not only by the application itself, but by bugs in the underlying runtime system (OS, hardware, etc.) or by idiosyncratic interactions between the application and the runtime system. Capturing these interactions requires examining the whole system, not just the application. This makes the problem size larger and places part of the problem outside the detailed knowledge of the application designer, who may not even be available.

The contribution of this paper is to address these problems with a new technique, *automatic failure-path inference* (AFPI), for automatically capturing dynamic fault propagation information. Using instrumented middleware, AFPI discovers points in the application where it might fail. Using controlled fault injection and observation of the faults' propagation, AFPI builds a failure propagation graph, or *f-map*, that captures dependencies between components.

The rest of the paper is organized as follows. Section 2 describes the specifics of our approach. Section 3 presents experimental results validating the accuracy of our f-maps and confirming that there is no performance overhead in applying AFPI. We highlight specific cases in which AFPI finds interesting fault propagation paths that would have been difficult to find manually, as well as cases in which it eliminates paths that appear in static dependency graphs but do not actually propagate faults in practice. Section 4 describes future work, and section 5 compares AFPI to related work, especially recent research focused on dynamic discovery of failure dependencies.

2 Automatic Failure-Path Inference (AFPI)

We applied AFPI to J2EE, a widely-used middleware platform for running enterprise and Internet-based applications; here we give a brief overview of this technology and the components of its runtime system. We then explain the AFPI algorithm and its implementation.

2.1. J2EE Applications and Application Servers

J2EE is a standard for constructing enterprise applications out of reusable Java modules, called Enterprise Java Beans (EJBs). An EJB is a Java component that conforms to a particular programmatic interface and provides services to remote clients. The bean has an associated *deployment descriptor*—an XML file that conveys runtime characteristics, such as whether the EJB's operations must run within a transaction. The EJB architecture runs distributed components within a special container, that is provided and serviced by the application server (see Figure 1).

A J2EE application must be deployed on a J2EE application server, which provides a standard environment and a set of runtime services specified by J2EE. The application server instantiates the EJBs in *containers*, deploying and undeploying components on demand. It also allows communication via remote method calls among EJBs, JavaServer Pages (JSPs, used to format content for Web output), and Java servlets. JSPs and servlets may invoke EJBs, which may in turn invoke other EJBs and/or communicate with persistent database(s) over JDBC, a standard mediator. The application server also provides standard runtime system services such as a naming and directory service to allow components to refer to each other using symbolic names, distributed transaction facilities, authorization and authentication, etc. Finally, the application server integrates with a Web server and a servlet/JSP container that manages servlets and JSPs, thereby making the application accessible through a Web page interface.

2.2. The Introspection Process

Automatic failure-path inference (AFPI) consists of two phases. In the (invasive) staging phase, the system actively performs fault injection, observes its own reaction to the faults, and builds the f-map. In the (non-invasive) production phase, the system passively observes fault propagation when faults occur during normal operation, and uses this information to evolve the f-map on an ongoing basis. In this paper we focus on the staging phase.

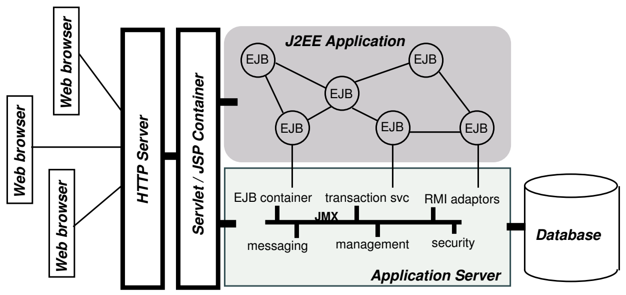

Figure 1. A typical J2EE infrastructure. The actual application consists of the EJBs in the shaded area, plus servlets and/or JSPs.

Our choice for a J2EE application server was motivated by three facts. First, J2EE is an increasingly popular platform for large scale Internet services; many businesses depend for their critical operations on J2EE application servers. Second, Java offers features such as reflection and other mechanisms that allow AFPI to discover relevant faults to inject. Third, J2EE enforces a particular application structure in which components have a small number of well-defined entry points, component state is either explicit or externalized in a database or other persistent store, and communication among components is mediated by the application server's naming/broker system and Java RMI.

2.3. AFPI Algorithm

The general algorithm for the staging phase is as follows:

1. Initialize a global fault list to be empty.
2. Start the application and any external components it uses (e.g., a separate database).
3. Every time a new component C of the application is deployed (whether at startup or during the operation of the application server), use reflection to discover the methods exported by its interface.
4. For each method M_i of C, use reflection to discover the set $\mathbf{F}$ of Java exceptions declared as throwable by M_i, and for each $F_j \in \mathbf{F}$, add the triplet (C, M_i, F_j) to the global fault list.
5. Also add to the global fault list any exceptions corresponding to "environmental" faults that could occur

during execution of method M_i. The set $\mathbf{E}$ of all such environmental exceptions could include network-related exceptions, disk I/O exceptions, memory-related exceptions, etc. In our current approach, we add a triplet (C, M_i, E_j) for *every* exception type $E_j \in \mathbf{E}$. Injecting application-defined exceptions exercises robustness to application bugs, while injecting environment-related exceptions primarily probes the paths through which application-external faults can propagate.

6. Once all EJBs have been deployed, select a triplet (C, M, F) from the list of faults, and arrange to inject exception F into component C the next time method M is called. Section 2.4 describes how this is done.

7. Start the load generator, in order to exercise the application.

8. As failures occur, the monitoring agent is notified by a separate fault detector. As the monitor receives notifications, it builds up an *f-map* for each type of fault. An f-map is a directed graph that captures failure propagation: the presence of a directed edge (u, v) means that a fault in component u propagated and caused component v to fail. If, however, u's fault propagates but is properly handled by v, then no externally visible behavior is reported to our monitor, so edge (u, v) is not added to the f-map. F-maps for different fault types may differ, because some faults propagate from the callee up the call tree and others do not. In all our experiments, the injected faults always resulted in observed failures, i.e., exceptions that propagated to at least one EJB.

9. Save the current f-map and list of faults to stable storage, shut down and restart the application, and continue with the next (C, M, F) triplet. The fault injection experiments end when the list of faults has been exhausted. We restart the application between injections to avoid spuriously representing cascading failures in the f-map.

Two notable differences emerge in comparing our fault injection approach to other recent work. First, in step 4, we directly induce application-visible exceptions, in contrast to prior work that injects low-level hardware faults. Determining which (if any) application-visible failures result from low-level hardware faults requires the construction of a fault dictionary [20], which has proven difficult.

Second, some recent work [16] attempts to narrow the possible faults injected at a particular point based on static or dynamic analysis; for example, if a particular bytecode sequence is known to specify a read from a network stream rather than from a file, one might inject only network-related hardware errors rather than low-level disk errors at that point. As stated in step 5, we avoid such narrowing and simply enumerate all possible environmental conditions that can be expressed as Java exceptions. Besides the fact that deriving more specific information using static analysis can be cumbersome, we would like to minimize our assumptions about whether, in fact, the executing method would really be unaffected if an "unrelated" low-level fault occurred (consider a method that does no disk I/O, but a low-level disk fault occurs while that method is trying to page in data or code). In fact, all we assume is that it is *possible* that a given exception might be thrown by a particular method. We return shortly to the question whether most low-level faults do in fact manifest as application-level exceptions.

A major flaw in many fault injection experiments in the literature is that they do not account for correlated faults. However, in large scale production systems, true independent failures are rare [5, 1]. Therefore, once the AFPI sequence of single-point injections completes, our system does multi-point injections, to simulate correlated failures. The monitoring agent adds to the f-map any additional edges that it detects this way.

Once the multi-point injection phase completes, the system can be deployed into production. The monitoring agent continues to observe the system's reactions to "naturally occuring" (i.e., non-injected) faults, and modifies the f-map based on these observations. Thus, the f-map is a continuously evolving representation of the application. This passive phase is in no way dependent on the active, fault-injection phase—it would work even without an initial draft of the f-map, but it would take much longer to converge onto a correct representation of the dependencies. We can therefore think of the fault injection phase as an optimization.

2.4. Modifying JBoss to Enable AFPI

To test AFPI, we used JBoss [19], an open-source implementation of J2EE; it is free and we can modify its source code to perform fault injection and monitoring. JBoss performs well, is very popular (has been downloaded from SourceForge [29] over 3 million times) and is used by more than 100 corporations, including WorldCom and Dow Jones.

JBoss consists of a microkernel, with the various services being held together through Java Management Extensions (JMX). The services are hot-deployable, which implies that one can replace an existing service with our modified version at runtime, and the server will properly reintegrate it. We built our failure monitoring and fault injection facilities as two separate services of the JBoss microkernel, so that failures are detected independently of any specific knowledge that they are being injected.

In addition, we modified existing JBoss code in three ways. First, whenever a new EJB is deployed (i.e., the bean is instantiated in a new container), the fault injector uses Java reflection to enumerate the bean's methods and the exceptions each method can throw, in addition to those thrown by the JVM itself (i.e., step 4 of the AFPI algorithm). This process is identical whether the EJB is deployed as part of regular application deployment, or as part of a live upgrade or bug fix.

Second, we provide a new container method by which we can instruct the EJB that the next call to method M should throw exception X, i.e. step 6 in the algorithm. During the fault injection stage, the fault injector will systematically inject every kind of exception that can be thrown by each method in the bean, one at a time. It will attempt to throw both declared exceptions (i.e., those explicitly declared by the bean's Java method signature, some of which may be handled using `catch` and others which may be passed up to the caller) and non-declared exceptions (e.g., a runtime condition such as OutOfMemoryError, to simulate more generic system-level problems that most beans would not try to detect directly, as in step 5 of the algorithm).

Third, we modified the EJB container so that, when an exception is thrown by an EJB, the stack trace is parsed and we extract the identity of the calling component (bean/servlet/JSP) and the invoked component. This information is passed to the failure monitor service, which uses the information to build up a failure propagation map.

Since we modified the application server rather than a specific application being deployed, we can invoke the new functionality on *any* J2EE application that runs on JBoss.

3. Experiments

Our experiments address three issues: the suitability of our injected faults, the accuracy and usefulness of our f-maps, and at what cost in terms of time and performance we can obtain these f-maps. To determine the cost of obtaining such graphs, we measure the performance overhead introduced by our modifications to JBoss in section. The section ends with a discussion of AFPI's shortcomings.

All experiments, except performance overhead, were performed on an Intel Pentium–4 1GHz machine, with 512 MB RAM, running Linux 2.4.9. We modified JBoss 3.0.3 running on Sun Java 2SE 1.4 and Sun Java 2EE 1.3.1. The applications we chose for our experiments are Petstore 1.1.2 and RUBiS 1.3. Petstore is a freely available sample J2EE application from Sun that implements an e-commerce site where users can maintain an account, update their profile and payment information, browse a merchandise catalog, add/remove items to/from a shopping cart, complete a purchase, etc. It consists of 233 Java files and about 11K lines of code. RUBiS, developed at Rice University, implements a web-based auction service modeled on eBay. It contains 582 Java files and about 26K lines of code. For each of these applications, the invasive staging phase took between two and three hours to complete.

We wrote a load generator that plays back a trace of clients' interactions with the web site; such traces are saved using a request/response recording proxy. The recording of client activity can be done on live traffic during the production phase, to generate a trace that is absolutely specific to the web site in question.

3.1. Suitability of Injecting Java Exceptions

Since we intend for our technique to be used on real systems, it is important that we inject faults representative of those that would occur in real systems. We inject application and JVM-visible faults (exceptions, application-visible or OS-visible resource exhaustion) rather than low-level faults (bit flips, stuck-ats) for three reasons. First, low-level faults are not that common in Internet systems, unlike, e.g., spaceborn applications exposed to radiation; Internet services go down mainly due to software bugs and operator errors [24, 12, 28, 2, 26]. Second, faults not corrected by the hardware will often manifest as some higher-level fault, but if transient, the mapping may appear nondeterministic. A "random" bit flip *may* corrupt an important/live data structure or trash the heap, but in many cases turns out to be harmless [10]. Getting reliable failure propagation for this type of faults is difficult; we are not aware of prior work that systematically addresses this problem.

An important question concerns coverage of failures: have we exercised all the conditions that could trigger a particular exception, especially given that the way the exception is handled may depend on the particular state of the program at the time the exception occurs? Although the experiments performed so far have not provided evidence for this, we anticipate that our coverage is not complete. However, in cases where partial recovery fails due to incorrectly determining the subset of components that must be recovered, a full reboot could always be used [8].

In addition to injecting the kinds of faults the application designer originally thought of, we choose faults that are in fact commonly observed as software-related transients. Exceptions explicitly declared by the beans correspond to the designer's own knowledge of particular "expected" failure modes. With respect to undeclared exceptions, it is reasonable to ask whether the kinds of

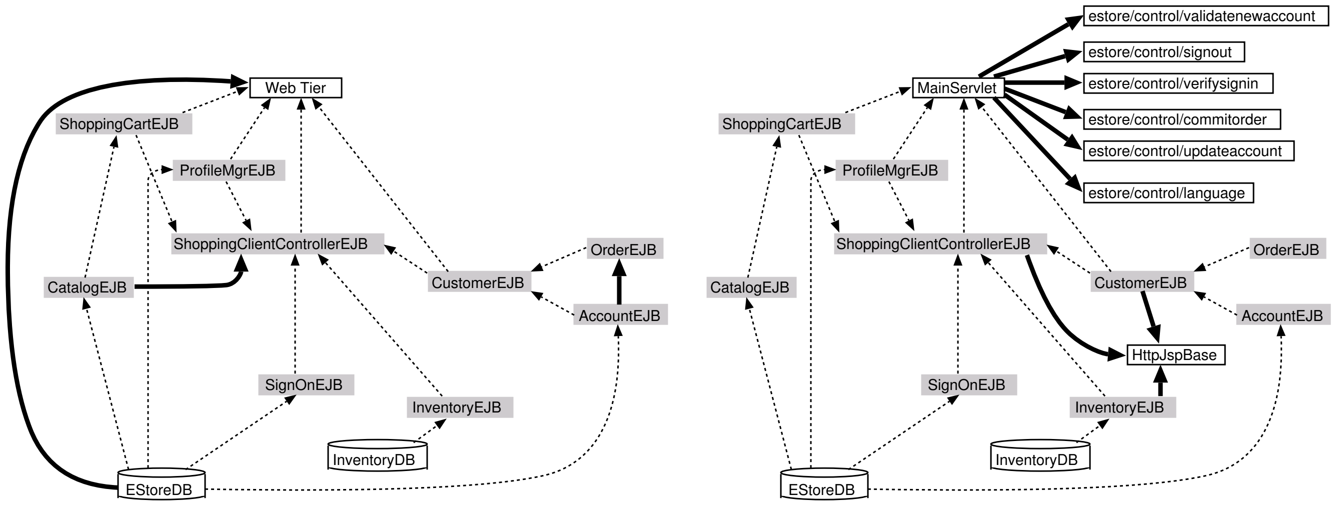

Figure 2. Two f-maps for Petstore: one obtained through manual inspection of deployment descriptors (left) and one obtained automatically by our introspective system (right). The bold edges are those that are not common between the two f-maps.

failures that are independent of application semantics—OS resource exhaustion, network connectivity problems, manifestation of bugs in the OS kernel or libraries—are indeed manifest as Java-visible exceptions.

When an internal error or resource limitation prevents the Java virtual machine from implementing the semantics of the Java programming language, it throws an exception that is a subclass of VirtualMachineError; these exceptions are shown in Table 1.

Exception	Possible real life cause
declared exceptions	application-expected faults
OutOfMemoryError	memory exhaustion
StackOverflowError	code bugs
IOException	failed or interrupted I/O operations
RemoteException	remote method invocation failure
SQLException	database access error
NullPointer	code bugs and data errors

Table 1. Exception types used in our experiments.

We also injected timing faults, in which calls to various EJBs were delayed, but due to the blocking nature of RMI (Java's RPC-like remote method invocation mechanism), such timing faults did not induce any failure in Petstore or RUBiS.

We did not find literature that documents the extent to which different JVMs actually translate such low-level faults into Java-visible exceptions. We performed a number of ad hoc experiments (see Table 2 for a few examples) to determine whether Java exceptions were indeed a reasonable way to simulate such faults. The results were satisfactory: using the Sun HotSpot JVM, all faults we injected at the network level (e.g., severing the TCP connection), disk level (e.g., deleting the file), memory (e.g., limiting the JVM's heap size), and database (e.g., shutting DBMS down) resulted in one of the exceptions shown in Table 1. Our validation, however, is certainly not exhaustive. Fault model enforcement [25] is an interesting technique that has been recently proposed for ensuring that real-life faults do manifest as one of a set of faults specified in a fault model.

Induced failure	Exception
bad server in registry	ConnectException
RMI registry unreachable	ConnectException
server not in registry	NotBoundException
server crashes during call	UnmarshalException

Table 2. Examples of RMI experiments that test whether real faults turn into Java exceptions. All exceptions are subclasses of RemoteException.

3.2. F-Maps Compared to Existing Structures

In this section we examine whether our f-maps are as accurate as those obtained using other techniques, and, if so, whether they are any better. We compare f-maps obtained through our introspective method with a dependency graph built by manual inspection of J2EE deployment descriptors—a programmer-supplied XML document for each J2EE component, that describes a component's deployment settings. A bean's descriptor declares, among other things, what other beans this bean calls; this suggests using the collection of descriptors as an approximation of the static call graph.

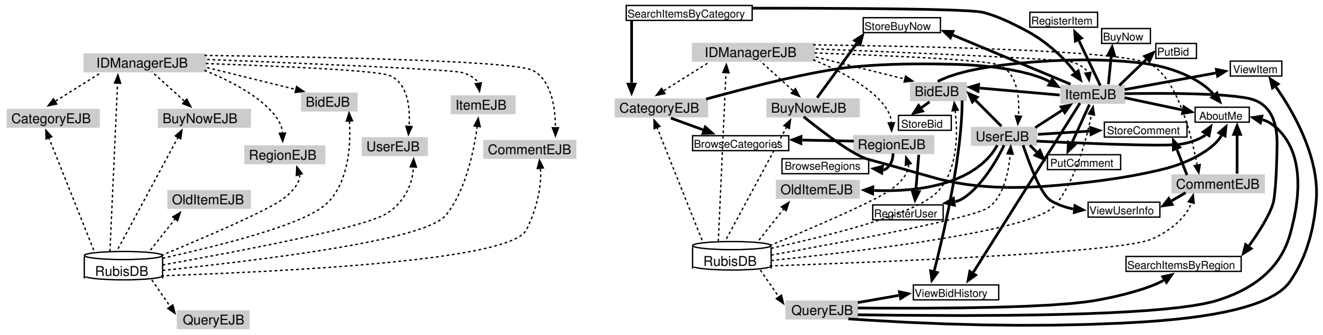

Figure 3. Two f-maps for RUBiS: one obtained through manual inspection of deployment descriptors (left) and one obtained automatically by AFPI (right). The bold edges are those that are not common between the two f-maps. Shaded components are EJBs, clear boxes indicate servlets.

In our implementation, the nodes in an f-map represent EJBs, servlets, JSPs, and special data access objects (DAOs) used for accessing databases. Figure 2 shows two versions of the Petstore f-map; there are two types of differences between the f-maps. First, the injection-based f-map is missing some edges present in the descriptor-based f-map: AccountEJB → OrderEJB, CatalogEJB → ShoppingClientControllerEJB, and EStoreDB → web tier. Second, the injection-based f-map has additional nodes and edges that are not present in the descriptor-based f-map: HttpJspBase, MainServlet, and six JSPs, with the corresponding edges. The deployment descriptors group all web components (servlets and JSPs) into one entity, called the web tier.

We were most concerned about the missing edges, because they seemed to imply that our technique failed to discover existing dependencies. However, upon inspection of the code, we found that the injection-based f-map was correct. In the case of the AccountEJB → OrderEJB edge, OrderEJB did indeed maintain a reference to AccountEJB, but it never interacted with that EJB, hence no opportunity for a fault propagation from AccountEJB to OrderEJB. In the case of CatalogEJB → ShoppingClientControllerEJB, ShoppingClientControllerEJB did not even have a reference to CatalogEJB, so the deployment descriptors were simply wrong. This illustrates that, contrary to our initial expectations, the descriptors cannot be relied on for understanding the structure of the application.

Finally, the EStoreDB → web tier edge is present in the descriptors due to a servlet which runs at setup time and populates the database with default information (users, merchandise, etc.). This servlet is run once at installation and never again during normal operation, which is why our f-map correctly indicates there is no direct fault propagation edge from the database to any of the web components.

The second difference is that components are discovered at a finer grain than can be captured in the deployment descriptors. For example, what descriptors showed as the "web tier," our system dissected into the individual servlets and JSPs, along with fault propagation information. These additional f-map nodes naturally result in new edges along which faults can propagate, edges that can be used in better pinpointing the source of failure and thus provide a more powerful tool in deciding which components to recover. An interesting case is that of the HttpJspBase node; we tried to find it in the Petstore source code to understand its role, but it turned out it is not part of the application. HttpJspBase is the superclass of all JSP-generated servlets and it is part of Jetty, the servlet/JSP container that we used with JBoss. Hence, our technique was able to identify interactions with components which, although not part of the application, are still parts of the system delivering the service.

Obtaining the f-maps for RUBiS, as indicated in Figure 3, confirmed the properties observed with the Petstore f-maps. This time, the deployment descriptors turned out to be quite conservative, in that the AFPI-based f-map contained strictly more information than the descriptors. Many of the inter-EJB dependencies fail to be indicated as references in the deployment descriptors. Dependencies between EJBs and servlets are not captured in deployment descriptors either. While much of this information could be obtained through static analysis, certain conditions cannot be found, such as code that is dead, owing to a dependency on the current date.

We did not modify JBoss between the Petstore and RU-

BiS experiments, thus demonstrating that our approach is application-agnostic. Any J2EE application could be subjected to AFPI by simply dropping the .jar file in JBoss's deployment directory. True application-generic recovery for UNIX applications with no middleware has been shown to be infeasible [23]. AFPI does better because J2EE constrains the structure of the application and allows us to modify the runtime. One could claim for non-J2EE applications that the OS/kernel is the runtime, and try to modify that; UNIX applications, though, are usually non-modular and their state is not well-circumscribed, which explains the results in [23].

AFPI needed no application-specific knowledge to derive the correct set of dependencies between components, and didn't even require access to the application's source code. As was mentioned previously, the collection of deployment descriptors does not need to reflect a correct static call graph in order for the application to run, and in fact one would expect bean evolution to make erroneous descriptors more common. This is especially true given that developers often maintain the descriptors manually [3], although expensive commercial tools are available to programmatically generate deployment descriptors from bean sources. Moreover, even if the deployment descriptors were correct and complete, they would capture all possible paths that *might* propagate faults at runtime, whereas we are interested in those that actually *do* propagate faults.

Since deployment descriptors fail to provide an accurate static call graph, we considered obtaining the call graph using a tool that directly inspected the source code. Of course, such a call graph would have to be significantly pruned in order to make it useful; the static call graph is normally exponential in the program size, and researchers currently attempting to apply static analysis to improve fault coverage admit they will have to use a smaller-sized approximation of the full static call graph [16]. Even if we could work with a compact representation of the static call graph, it would show paths along which faults might propagate, not only the ones along which faults do propagate, and it would not reveal edges that propagate faults unrelated to individual method invocations, such as an errant thread that overallocates memory and causes another unrelated thread in the same JVM to get an out-of-memory exception.

3.3. Fault-Class-Specific F-Maps

Unlike general dependency graphs, our technique allows the system to obtain and maintain separate f-maps for each fault class. Such maps enable targeted recovery in the case of specific faults. Since one motivation of obtaining f-maps is fast recovery, we do want such fine-grained information about failure propagation so that we can recover the minimal subset of failed components. By eliminating edges that do not reflect actual (observed) propagation paths, we can do more efficient microrecovery.

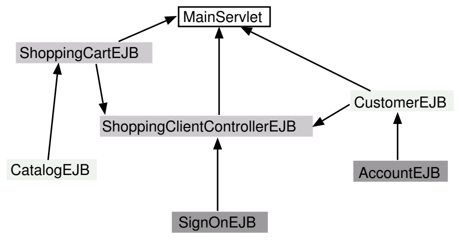

Figure 4. Petstore f-map based exclusively on application-declared exceptions.

Figure 4 shows a Petstore f-map discovered by injecting exclusively application-declared exceptions. Such an f-map may help us in deciding how useful it would be to suppress environmental/external sources of faults (e.g., by purchasing more reliable hardware). Notice that this f-map is considerably simpler than the previous f-maps.

We also examined Petstore's five fault-class-specific f-maps for undeclared exceptions, i.e., those obtained by injecting OutOfMemoryError, StackOverflowError, IOException, RemoteException, and SQLException, respectively. To our suprise, each of these f-maps was virtually identical to the cumulative one in Figure 2, which represents the union of all the fault-class-specific f-maps. Inspecting the Petstore code provided a simple explanation: the application contains almost no code to handle exceptions that may occur from its interaction with the environment. Such lack of robustness causes any external exception to appear as "something bad" that the affected EJB does not handle gracefully.

3.4. Injecting Correlated Faults

As described in section 2.2, the second phase of f-map generation consists of injecting correlated faults. The algorithm for this second phase is similar to the first phase, with one exception: in order to generate the fault list, we take the Cartesian product of the original fault list with itself, and then eliminate any 6-tuples for which the same component is involved in both the first and the second fault of the tuple. This gives all possible combinations of 2 faults that involve distinct components. An analogous algorithm is used for building the list for n-point injections, where $n > 2$. If new propagation paths are discovered through multi-point injection, they are merged into the f-map.

We were particularly interested in verifying the robustness of the information in the f-maps to correlated fault scenarios. The f-maps converged much quicker onto their final form, because of the richer set of faults that the system was exposed to. The unexpected result, however, was that we obtained f-maps identical to those resulting from single-point injection. While this might suggest the f-maps are robust, we actually believe the result is due to another reason: in a request/response system with very little recovery code, when a request enters the system and hits a faulty component, it will almost always fail at that point and not proceed further through the system. Hence, the correlated faults are likely being consumed by separate requests, as if the faults had been injected separately.

3.5. Overhead

We evaluated the performance impact of our modifications by comparing the performance of unmodified JBoss to that of our instrumented version, with the same Petstore workload we used for generating the f-maps. We found no statistically significant difference in performance when no exceptions are thrown, i.e., under steady state operating conditions. This suggests that it is feasible to implement our technique in a production system.

JBoss and the database were hosted on the same Intel-P3 450MHz-based machine with 512MB of RAM. The workload generators ran on a dual P3/1GHz machine with 1.5GB of RAM. The purpose we chose machines with such different characteristics was to allow us to saturate the server. Both machines ran Linux 2.4 with Sun's JVM 1.4.1 and were connected via 100Mbps Ethernet. The server's CPU was saturated at 100% during each run. Our modified JBoss completed each test run on average in 93 seconds, compared to 94.8 seconds for unmodified JBoss; standard deviation was 5.8 in both cases. We consider this improvement in the running time of the application to be just statistical noise, as we do not think any of our changes would make JBoss run faster.

3.6. Weaknesses

A problem specific to our experiments is the fact that we only injected exceptions and timing faults. First, there may be low-level faults that do not manifest as exceptions in the Java VM, although we haven't found any in our ad hoc experiments. Second, we haven't explored yet data-level faults, in which a called component provides wrong data. Most of these faults require application-specific knowledge to detect, unless we transparently employ redundancy combined with a voting scheme. A special kind of data faults are null pointers, which we inject using NullPointerException.

Problems also arise from the fact that we currently do not collect all the information we could about the exceptions, thus preventing us from distinguishing between propagation paths composed of a subpath that propagates fault X, chained to a subpath that propagates fault Y. We do collect per-fault-class f-maps, but in the case of application-declared exceptions, our current version of the system would indicate the entire path as a fault propagation path, without discriminating among the faults.

We cannot claim that any particular set of AFPI experiments will find *all* failure propagation paths, e.g., because we depend on the applied workload to exercise all affected components. However, when using AFPI in recovery via recursive microreboots [8], we can view the use of the f-map as an optimization that impacts recovery performance but not recovery correctness. If the failure had been non-transient, then neither full recovery nor partial recovery guided by the f-map would have cured it.

An important use for f-maps is automated microrecovery: based on the f-map, determine the minimal subset of the system that must be recovered to cure the failure, for example, restarting the database and undeploying/redeploying a bean or set of stateful session beans. We found a significant time-to-recovery advantage to this approach compared to whole-system restart, and showed that availability can be improved by 78% when AFPI and microreboots are used together [9].

4. Future Work

We are extending our work in two main areas:

Application evolution. When we add a new EJB in an application, AFPI currently updates the f-map correctly. We want, however, to modify AFPI such that the f-map gets updated also when a component changes or is removed from the application. Similarly, sometimes the same logic bug in an application will manifest differently on different runtimes. For example, a logic bug in a Java application that caused one system to livelock manifested as a JVM crash on a different system [13]. AFPI should capture such cases that would be missed by analyzing only the application; we would like to verify this in practice using another JDK/OS combination for one or more of the above applications.

Correlation vs. causality. Our technique currently captures only causality information: an edge (A, B) is added to the f-map if component A propagates a fault to component B. We believe there is high value in capturing *correlation* in addition to causality, by adding perhaps a special kind of edge (A, B) for when statistical observation indicates that components A and B fail together most of the time, even though no direct propagation was observed. This would capture indirect coupling between components due to an external resource, e.g., one component corrupts a data structure that is shared by another component, but maintained in a third component such as a database. Such correlation information seems to be a good candidate for the passive, post-deployment phase of AFPI, in which the system is observed in production during normal operation.

5. Related Work

A significant body of work exists on using dependency graphs for root cause analysis and fault diagnosis. For example, alarms detected at various components of the system can be mapped to nodes of the dependency graph, and the graph can be used to determine where the source of the alarms may be [30, 18]. A dependency graph can also be used as a guide for which components to examine directly in order to determine the root cause of failure [22]. There are a number of more sophisticated algorithms [27] that can use dependency graphs to obtain higher precision in fault diagnosis, at the expense of timeliness.

Others have recognized, as we have, the importance of automatically determining dependency information without extensive application-specific knowledge; aggressive software evolution in Internet applications makes this ability essential [17]. Some researchers have used existing deployment information [21] or correlation of behaviors [14] to provide indirect evidence for dependencies. Others have used some form of fault injection to observe the effects of a fault in one system component on overall system behavior: [7] induced application-specific errors in an e-commerce application by selectively locking tables in a database that certain e-commerce transactions needed to access, then recording which types of transactions were affected.

More recently, Bagchi et al. [4] extended this approach to determine which component(s) are most likely to be the root cause of a particular observed type of performance degradation, considering both faults that lead to failures and conditions such as resource starvation that lead to poor performance. Both their work and that of Fu et al. [16] rely on the concept of a fault dictionary [20], which maps "low-level" faults (e.g. a hardware failure in a network link) to application-level observed failures (e.g. exceptions or application crashes): rather than directly inducing application-level exceptions as we do, they inject low-level faults that would produce particular application-level failures. In [4] the construction of the fault dictionary is not addressed; [16] admit that the construction of the dictionary is complicated by the many software layers between the low-level hardware and the application, and they propose to use two kinds of static analysis to infer which application-level exceptions could possibly result from particular low-level faults. We believe that directly injecting exceptions may avoid some of these difficulties. In any case, our goal is to use this information to enable fast partial recovery, whereas most prior work has focused on using it to assist in root-cause diagnosis or system hardening.

Pinpoint [11] used a combination of fault injection and data clustering analysis to determine with high accuracy which components were most likely to be at fault when a particular end-user-visible failure was observed in an Internet application. Unlike Pinpoint, our approach uses an entirely application-generic fault injection mechanism, that utilizes reflection instead of a predetermined list of faults; this enables introspection. We have combined Pinpoint's analysis tools with f-maps, to enable fast accurate failure diagnosis, which is useful for both speeding recovery [9] and targeting system debugging efforts.

6. Conclusions

We focused on applying Automated Failure-Path Inference to applications built on Java 2 Enterprise Edition middleware, because such applications tend to be highly modular and rely on a well-defined set of runtime services whose implementation we can change to add fault injection and monitoring behaviors.

The experiments we described show that AFPI automatically and dynamically generates f-maps that find runtime dependencies that static call graph analysis might miss. AFPI-generated f-maps correctly omit dependencies that appear in the static call graph but do not result in observed fault propagation at runtime. The dynamically generated f-maps can capture dependency information per fault type, providing higher resolution than many static techniques.

Injecting Java exceptions to represent real operational faults is reasonable, and in particular, certain common classes of application-generic faults (such as resource exhaustion) are often manifest as JVM exceptions. While injecting correlated faults, we did not find any new f-map edges, but expect this to be due to the simple, single-threaded application, rather than to the AFPI implementation. For similar reasons, we have not yet investigated in depth the stability of f-maps in the presence of quasi-deterministic faults, such as those introduced by pernicious firmware bugs.

AFPI's staging phase took about three hours for a non-trivial application consisting of over 26K lines of code and required no manual inspection or knowledge of the application itself; the additional overhead of leaving AFPI in place during production operation was negligible. Although AFPI may not capture all propagation paths, we aim for a solution that is the right tradeoff between complexity/difficulty/cost and effectiveness. Failure dependency graphs have many uses, and in [9] we demonstrate its value for automated microrecovery.

References

[1] A. Acharya. Reliability on the cheap: How I learned to stop worrying and love cheap PCs. In *2nd Workshop on Evaluating and Architecting System Dependability*, San Jose, CA, 2002. Invited Talk.

[2] T. Adams, R. Igou, R. Silliman, A. M. Neela, and E. Rocco. Sustainable infrastructures: How IT services can address the realities of unplanned downtime. Research Brief 97843a, Gartner Research, May 2001. Strategy, Trends & Tactics Series.

[3] L. Arellano. Efinance, inc. Personal Communication, 2002.

[4] S. Bagchi, G. Kar, and J. Hellerstein. Dependency analysis in distributed systems using fault injection: Application to problem determination in an e-commerce environment. In *12th International Workshop on Distributed Systems: Operations and Management (DSOM 2001)*, Nancy, France, October 2001.

[5] W. Bartlett. HP NonStop server: Overview of an integrated architecture for fault tolerance. In *2nd Workshop on Evaluating and Architecting System Dependability*, San Jose, CA, Oct 2002. Invited Talk.

[6] F. P. Brooks. *The Mythical Man-Month*. Addison-Wesley, Reading, MA, Anniversary edition, 1995.

[7] A. Brown, G. Kar, and A. Keller. An active approach to characterizing dynamic dependencies for problem determination in a distributed environment. In *Proceedings of the 7th IFIP/IEEE International Symposium on Integrated Network Management (IM 2001)*, Seattle, WA, May 2001.

[8] G. Candea, J. Cutler, and A. Fox. Improving availability with recursive microreboots: A soft-state system case study. *Performance Evaluation Journal*, 56(1-3), March 2004.

[9] G. Candea, P. Keyani, E. Kiciman, S. Zhang, and A. Fox. JAGR: An autonomous self-recovering application server. In *Proc. 5th International Workshop on Active Middleware Services*, Seattle, WA, June 2003.

[10] D. Chen, A. Messer, P. Bernadat, G. Fu, Z. Dimitrijevic, D. J. F. Lie, D. Mannaru, A. Riska, and D. Milojicic. JVM susceptibility to memory errors. In *Submitted to Java Virtual Machine Research and Technology Symposium*, 2001.

[11] M. Chen, E. Kiciman, E. Fratkin, E. Brewer, and A. Fox. Pinpoint: Problem determination in large, dynamic, Internet services. In *Proc. International Conference on Dependable Systems and Networks*, Washington, DC, June 2002.

[12] T. C. Chou. Beyond fault tolerance. *IEEE Computer*, 30(4):31–36, 1997.

[13] J. Cutler. Personal communication, 2002.

[14] C. Ensel. Automated generation of dependency models for service management. In *Workshop of the OpenView University Association*, Bologna, Italy, 1999.

[15] A. Fox and D. Patterson. When does fast recovery trump high reliability? In *Proc. 2nd Workshop on Evaluating and Architecting System Dependability*, San Jose, CA, 2002.

[16] C. Fu and R. M. et al. Compiler-directed program-fault coverage for highly available internet applications. Technical report, Rutgers University Computer Science Dept., 2003.

[17] J. Gray. Internet reliability. Keynote presentation at Second Workshop, High Dependability Computing Consortium, Santa Cruz, CA, May 2001. `http://www.hdcc.cs.cmu.edu/may01`.

[18] B. Gruschke. Integrated event management: Event correlation using dependency graphs. In *Proceedings of the 9th IFIP/IEEE International Workshop on Distributed Systems Operation and Management*, 1998.

[19] JBoss. Homepage. http://www.jboss.org/docs, 2002.

[20] Z. Kalbarczyk, R. K. Iyer, G. Ries, J. Patel, M. Lee, and Y. Xiao. Hierarchical simulation approach to accurate fault modeling for system dependability evaluation. *IEEE Transactions on Software Engineering*, 25(5):619–632, September/October 1999.

[21] G. Kar, A. Keller, and S. Calo. Managing application services over service provider networks: Architecture and dependency analysis. In *Proc. 7th IEEE/IFIP Network Operations and Management Symposium*, Honolulu, HI, 2000.

[22] S. Kätker and M. Paterok. Fault isolation and event correlation for integrated fault management. In *Proc. 5th IFIP/iEEE International Symposium on Integrated Network Management*, pages 583–596, San Diego, CA, 1997.

[23] D. E. Lowell, S. Chandra, and P. M. Chen. Exploring failure transparency and the limits of generic recovery. In *Proc. 4th USENIX Symposium on Operating Systems Design and Implementation*, San Diego, CA, 2000.

[24] B. Murphy and T. Gent. Measuring system and software reliability using an automated data collection process. *Quality and Reliability Engineering International*, 11:341–353, 1995.

[25] K. Nagaraja, R. Bianchini, R. P. Martin, and T. D. Nguyen. Using fault model enforcement to improve availability. In *Proc. 2nd Workshop on Evaluating and Architecting System Dependability*, San Jose, CA, 2002.

[26] D. Oppenheimer, A. Ganapathi, and D. Patterson. Why do internet services fail, and what can be done about it? In *Proc. 4th USENIX Symposium on Internet Technologies and Systems*, Seattle, WA, 2003.

[27] N. S. V. Rao. Expected-value analysis of two single fault diagnosis algorithms. *IEEE Transactions on Computers*, 42(3):272–280, March 1993.

[28] D. Scott. Making smart investments to reduce unplanned downtime. Tactical Guidelines Research Note TG-07-4033, Gartner Group, Stamford, CT, March 19 1999. PRISM for Enterprise Operations.

[29] SourceForge.Net. The world's largest open source software development repository. http://www.sourceforge.net/, 2003.

[30] S. Yemini and S. K. et al. High speed and robust event correlation. *IEEE Communications Magazine*, 34(5):82–90, 1996.