Autonomous Recovery in Componentized Internet Applications

George Candea, Emre Kiciman, Shinichi Kawamoto, Armando Fox
Computer Systems Lab, Stanford University
{candea,emrek,skawamo,fox}@cs.stanford.edu

Abstract

In this paper we show how to reduce downtime of J2EE applications by rapidly and automatically recovering from transient and intermittent software failures, without requiring application modifications. Our prototype combines three application-agnostic techniques: macroanalysis for fault detection and localization, microrebooting for rapid recovery, and external management of recovery actions. The individual techniques are autonomous and work across a wide range of componentized Internet applications, making them well-suited to the rapidly changing software of Internet services.

The proposed framework has been integrated with JBoss, an open-source J2EE application server. Our prototype provides an execution platform that can automatically recover J2EE applications within seconds of the manifestation of a fault. Our system can provide a subset of a system's active end users with the illusion of continuous uptime, in spite of failures occurring behind the scenes, even when there is no functional redundancy in the system.

1 Introduction

When Web-connected systems go down, most often the ones who notice first are the system's end users, who then contact technical support, who then contact the system administrators, who can then re-establish the service's operation. This happens frequently: a 2003 study [8] found that, of the 40 top-performing Web sites, 72% had suffered user-visible failures in the application, such as items not being added to a shopping cart or an error message being displayed. These application-level failures include failures where only part of a site goes down, failures where only some users are affected, and failures where functionality is only intermittently unavailable to end users. Our conversations with Internet service operators confirm the difficulty of detecting and localizing the faults causing these failures; partial service failures can sometimes take days to detect and recover. Faster recovery is needed—a study of three major Internet sites found that earlier detection could have mitigated or avoided 65% of the reported user-visible failures [43].

In spite of ever-improving development processes and tools, all production-quality software has bugs; most of these are difficult to track down and resolve, taking the form of Heisenbugs, race conditions, resource leaks, and environment-dependent bugs [19, 45]. When these bugs strike in live systems, they can result in prolonged outages [24, 40]. In the face of system failure, operators do not have time to run sophisticated diagnoses, but rather need to bring the system back up immediately; compounding this, up to 80% of software problems are due to bugs for which no fix is available at the time of failure [57]. Rebooting is often the last and only resort.

Our view is that, to improve dependability of Internet services, we need *autonomous recovery*, which enables systems to recover on the scale of "machine time" rather than "human time." The end result will be a better service experience for its human end users.

Achieving Autonomous Recovery

As suggested by the cited studies, there are two important components in an autonomous recovery strategy: automatic detection and localization of faults, as well as a reliable form of recovery. Neither of these components can afford to be custom designed for the application, because they need to co-evolve with the system that is being cared for (through upgrades, workload changes, etc.). The increasingly large scale of today's Internet systems therefore calls for both detection/localization and recovery to be application-generic.

Two observations about common failures suggest an approach to reduce this impact. First, many application-level failures are non-fail-stop faults; and the time to detect these failures significantly dominates the total time to recover from the fault. One service operator, TellMe Networks, estimates that over 90% of the time they spend recovering from application-level failures is spent just detecting (75%) and diagnosing (18%) them [14]. We have developed an application-generic technique for detection and localization; we describe it in Section 4.

Second, the results of several studies [50, 24, 41] as well as experience in the field [6, 44] suggest that many failures can be recovered by rebooting even if their root causes are unknown. For example, an informal analysis of the failure reports recorded at an Internet service [33] has found many such instances, including: malformed HTTP requests causing

Apache front ends to hang, overload leading to the failure of the middle-tier-hosted product search feature, out-of-memory conditions caused by Google rapidly crawling dynamically created Web pages, complex ad-hoc queries against the production database that caused DB performance to plummet, etc. Despite automatic garbage collection, resource leaks are a major problem for many large-scale Java applications; a recent study of IBM customers' J2EE e-business software revealed that production systems frequently crash because of memory leaks [39]. Some of the largest U.S. financial companies find it necessary to reboot their J2EE applications as often as several times a day [38] to recover memory, network sockets, file descriptors, etc. Servers in Web farms are subject to an even more aggressive rejuvenation regimen [6].

While rebooting does not necessarily address the root cause, it is a quick band-aid fix commonly employed in large systems. While rebooting certainly meets the requirements of a "universal" recovery technique, it tends to be expensive. Thus, a cheaper form of reboot is required; we have developed such a recovery technique and describe it in Section 5.

1.1 Contributions

In this paper we achieve the following three goals:

- We show that application-generic recovery from transient/intermittent failures can be performed autonomously by Internet systems, with no human assistance.
- We propose a design pattern for autonomously recovering systems and explore the relationship between its three components: monitoring, detection/localization, and recovery. We show which parts should be placed inside the system, and which parts belong outside.
- We present and evaluate a prototype application server implemented in the popular J2EE framework. This server leverages application-generic recovery for all J2EE applications that conform to a small set of design guidelines.

1.2 Roadmap

The remainder of the paper is structured as follows: In Section 2 we survey related work. Section 3 gives an overview of our design, that rests on three building blocks: macroanalysis, microrebooting, and a recovery manager; these are described in Sections 4, 5, and 6, respectively. Section 7 is devoted to an evaluation of our prototype. Section 8 summarizes our technical assumptions and discusses limitations of the prototype; Section 9 concludes the paper.

2 Related Work

2.1 Detecting and Localizing Faults

Following discussions with Internet service operators, we concluded that detection methods used in practice fall into three categories. First, *low-level monitors* are machine and protocol tests, such as heart beats, pings, and HTTP error code monitors. They are easily deployed and require few modifications as the service develops, but miss high-level failures, such as broken application logic or interface problems.

Second, *application-specific monitors*, such as automatic test suites, can catch high-level failures in tested functionality. However, these monitors cannot exercise all interesting combinations of functionality (consider, for example, all the kinds of coupons, sales and other discounts at a typical e-commerce site). More importantly, these monitors must be custom-built and kept up-to-date as the application changes, otherwise the monitor will both miss real failures and cause false alarms. For these reasons, neither the sites that we have spoken with, nor those studied in [43] make extensive use of these monitors.

Third, *user-activity monitors* watch simple statistics about the gross behavior of users and compare them to historical trends. Such a monitor might track the searches per second or orders per minute at a site. These monitors are generally easy to deploy and maintain, and at a site with many users, can detect a broad range of failures that affect the given statistic, though they do not give much more than an indication that something might have gone wrong.

Once a failure has been detected, localizing it can be challenging. Event-correlation systems for network management [46, 5] and commercial problem-determination systems typically rely on either expert systems with human-generated rules or on the use of dependency models to assist in fault localization [58, 17]. Aguilera et al. [1] and Brown et al. [7] have used dynamic observation to automatically build such dependency models. Anomaly detection has gained currency as a tool for detecting "bad" behaviors in systems where many assumed-good behaviors can be observed, including Windows registry debugging [54], finding bugs in system code [23], and detecting possible violation of runtime program invariants [26]. Ward et al. [55] proposed anomaly detection as a way to identify possible failures for Internet sites, but started with a statistical model based on 24 hours of observing the system, whereas in real systems one needs to build and adjusts the model dynamically.

2.2 Recovery

Virtually all recovery techniques rely on some form of redundancy, in the form of either functional, data, or time redundancy. In the case of functional redundancy, good processors can take over the functionality of failed processors, as in the case of Tandem process pairs [4] or clusters. Failover to a

standby node is a powerful high availability technique, but cannot be solely relied on (e.g., because whenever a node fails in a cluster, the system as a whole enters a period of vulnerability in which further failures could cripple it).

The benefits of restarting quickly after failures have been recognized by many system designers, as they employed techniques ranging from the use of non-volatile memory (e.g., Sprite's recovery box [2]) to non-overwriting storage combined with clever metadata update techniques (e.g., the Postgres DBMS [49]). A common theme is that of segregating and protecting state that needs to be persistent, while treating the rest as soft state. We see this approach reflected in recent work in Internet services [25] and we adopt it as a basic tenet.

Checkpointing [53] employs dynamic data redundancy to create a believed-good snapshot of a program's state and, in case of failure, return the program to that state. An important challenge in checkpoint-based recovery is ensuring that the checkpoint is taken before the state has been corrupted [56]. Another challenge is deciding whether to checkpoint transparently, in which case recovery rarely suceeds for generic applications [35], or non-transparently, in which case source code modifications are required. In spite of these problems, checkpointing is a useful technique for making applications restartable, and was successfuly utilized in several systems. We believe that maintaining state in a suitable store, however, obviates the need for checkpoints.

Some of the most reliable computers in the world are guided by the same principles we are following, and use dynamic recovery to mask failures from upper layers. For example, in IBM S/390 mainframes [48], computation is duplicated within each CPU and the results are compared before being committed to memory. A difference in results freezes execution, reverts the CPU to its state prior to the instruction, and the failed instruction is retried. If the results now compare OK, the error is assumed to have been transient and execution continues; if they are different, the error is considered to be permanent, the CPU is stopped and dynamically replaced with a spare.

2.3 Autonomic Computing

Autonomic computing [29] seeks to automate complex systems administration as much as possible, often by having a system automatically learn or infer its operating points and then applying automated management techniques based on closed-loop control or statistical modeling. Recent work in automatic inference of the behavior of complex applications relies on collecting fine-grained (component-level) observations and extracting interesting patterns from them [15, 21], whereas recent progress in applying automated management techniques [20, 31] assume a predictable performance cost for triggering management mechanisms such as recovering or activating a node. We combine similar techniques in the context of our three-tier J2EE prototype system.

3 Overview

Our design for autonomous recovery is illustrated abstractly in Figure 1. More than simply combining fault detection/localization and recovery, we needed to add a separate entity that could have end-to-end knowledge of the recovery process and provide the vehicle for implementing a variety of policies—the recovery manager. The autonomous recovery process is made up of three stages: pervasive monitoring, generic fault detection and localization, and generic recovery. The individual pieces have been developed in prior work [30, 12]; the contribution of this paper is to make a first step toward connecting them.

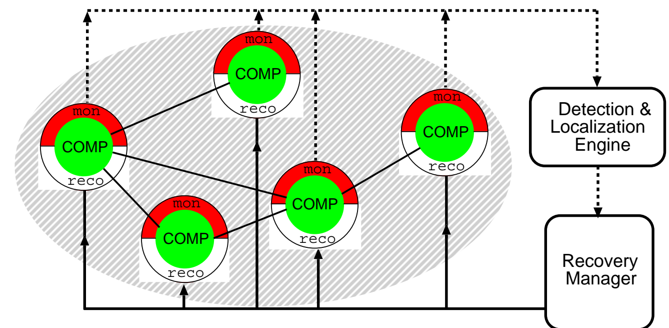

Figure 1: A design pattern common to autonomously recovering componentized applications. In this drawing, the system consists of five components (or subsystems), that are wrapped in a monitoring & recovery harness. Dotted lines represent paths for reporting potential failures, while the solid lines represent paths used by a recovery manager to send recovery instructions to the components' harness.

Monitoring: The system that needs to be recovered must have pervasive monitoring points (probes). Monitors generate low-level observations that are centralized into an analysis engine. The monitoring probes are introduced into the system using various forms of instrumentation; in our case we used interposition leveraging Java's language features.

Fault Detection and Localization: The analysis engine correlates monitoring reports to each other and to other information in order to infer when the system is behaving anomalously. The anomaly reports contain a preliminary location of the fault, listing all the components that are behaving anomalously. All of the reported anomalous components might be faulty and misbehaving, or one truly faulty component might be causing the others to misbehave.

Recovery: The anomaly reports are forwarded to a recovery manager, which is responsible for interpreting anomaly reports and deciding what action, if any, to take. Once the action has been taken, the recovery manager evaluates its effectiveness and decides whether more action is required. Since the cause of the failure is not known, but rather only the loca-

tion, a universal recovery technique is required. In our system we use exclusively recovery based on microreboots, because we focus on transient and intermittent failures; other systems may employ different techniques. Microreboots restart a single component within a larger system, resetting its state.

The time for a system to recover from a manifest fault becomes $T = T_m + T_d + T_r$, where T_m is the time it takes for monitors to pick up the symptoms and report them to the detector, T_d is the time for the analysis engine to correlate the various observations and infer that a failure has occurred in a specific component or set of components, and finally T_r is the time it takes for the recovery manager to decide what recovery action to take and then effect it.

3.1 The J2EE Framework

A common design pattern for Internet applications is the three-tiered architecture: a presentation tier consists of stateless Web servers, the application tier runs the application per se, and the persistence tier stores long-term data in one or more databases. J2EE is a framework designed to simplify developing applications for this three-tiered model; we applied the design described above to a J2EE application server.

J2EE applications consist of portable components, called Enterprise Java Beans (EJBs), together with server-specific XML deployment descriptors. A J2EE application server uses the deployment information to instantiate an application's EJBs inside management containers; there is one container per EJB object, and it manages all instances of that EJB. The server-managed container provides a rich set of services: thread pooling and lifecycle management, client session management, database connection pooling, transaction management, security and access control, etc.

End users interact with a J2EE application through a Web interface, the application's presentation tier. This consists of servlets and Java Server Pages (JSPs) hosted in a Web server; they invoke methods on the EJBs and then format the returned results for presentation to the end user. Invoked EJBs can call on other EJBs, interact with the backend databases, invoke other Web services, etc.

An EJB is similar to an event handler, in that it does not constitute a separate locus of control—a single Java thread shepherds a user request through multiple EJBs, from the point it enters the application tier until it returns to the Web tier.

We implemented our approach to autonomous recovery in JBoss [28], an open-source application server that complies to the J2EE standard. JBoss is widely used: it has been downloaded from SourceForge several million times, was awarded the JavaWorld 2002 Editors' Choice Award over several commercial competitors, and according to one study [3] offers better performance than several commercial competitors. More than 100 corporations, including WorldCom and Dow Jones, are using JBoss for demanding computing tasks.

Figure 2 illustrates the architecture of our prototype. The subsequent sections describe in more detail the modifications and extensions we made to JBoss.

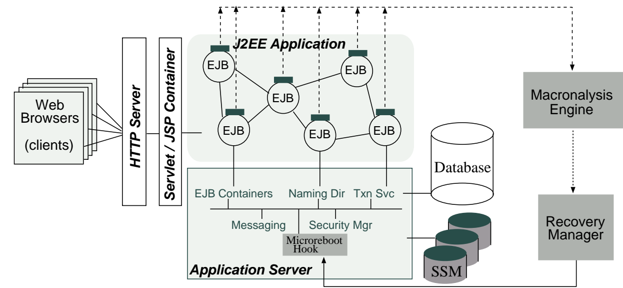

Figure 2: *JAGR*: JBoss with Autonomous Generic Recovery. The dark gray elements represent our additions to vanilla JBoss.

4 Detection and Localization

For detecting and localizing faults, we developed a macroanalysis system, called Pinpoint [30]; it is an application-generic framework for monitoring component-based Internet services, and detecting and localizing application-level failures without requiring a priori knowledge about the application. The basis for pinpoint is the observation that failures affecting user-visible behavior are also likely to affect a system's internally-visible behaviors. By monitoring low-level behaviors that are closely tied to application functionality, Pinpoint develops a model of the normal patterns of behavior inside a system. When these patterns change, Pinpoint infers that the application's functionality has also changed, indicating a potential failure. After noticing an anomaly, Pinpoint correlates it to its probable cause in the system and issues a list of suspected-faulty components.

4.1 Monitoring Instrumentation

We instrumented the JBoss EJB container to capture calls to the naming directory (used by application components to find each other) and the invocation and return data for each call to an EJB. We also instrumented the Java Database Connection (JDBC) wrappers to capture information on interactions with the database tier. Each observation includes the monitored component information, IP address of the current machine, a timestamp, an ordered observation number, and a globally-unique request ID, used to correlate observations.

The information we collect is as follows: in the HTTP server, every time a request enters the system, we record that the request has begun. The request is assigned a unique identifier and a counter initialized to zero; this counter is incremented each time an observation is recorded. Our modified

EJB container intercepts calls to EJBs and records when methods are called and returned, as well as the ID of the request associated with the invocation. We modified the RMI protocol to send the user request ID, and the current counter value to the callee; when the RMI call returns, the protocol returns the modified counter. In our instrumented Java messaging service (JMS) we forward the current request ID along with the message, and record the association between the sender and receiver. Finally, we observe database accesses made through JDBC and collect the SQL queries, updates, and the record IDs of created or read data items, along with the ID of the responsible user request. We do not directly trace background functionality that is not assignable to an individual request, such as garbage collection; the unit of failure that concerns us is the client request, since that is all the end user is aware of.

4.2 Analysis Engine

All observations are sent to a centralized engine for logging and analysis. The expensive work of sending observations over the network is done in one dedicated thread per Java virtual machine, thus keeping in the critical path only the initial packaging of the observation based on immediately available state.

Our engine for analyzing component behavior is implemented using a plugin-based architecture: each algorithm is implemented as a pipeline of data aggregators, sorters, splitters, statistical analysis and data clustering plugins. We use other plugins to interface between our analysis engine and automated or human managers of the system. E.g., a plugin sends failure notifications to our recovery manager; another plugin provides an HTTP interface for viewing the current anomalies and status of the system being monitored. The plugin-based architecture combined with network plugins allows us to forward data between stages of a pipeline running on separate machines. This enables the engine to scale, if necessary, by spreading an algorithm's CPU-intensive analysis stages across multiple machines or, depending on the details of the analysis, parallelize a single stage of the analysis across multiple machines.

4.3 Modeling Component Interactions

We model the behavior of a component C as a set of weighted links between C and other components. Each link is weighted by the proportion of runtime call paths that enter or leave C along that link. If the weights of these links change, then the functional behavior of C is likely to also be changing. We generate a historical model of a component's normal behavior by averaging the weights of links over time. We generate a peer component model by averaging the current behaviors of replicated peers (i.e., components with the same functionality).

Anomalies are detected by measuring the deviation between a single component's current behavior and our model of normal behavior using the χ^2 test of goodness-of-fit:

$$Q = \sum_{i=1}^{k} \frac{(N_i - w_i)^2}{w_i} \tag{1}$$

where N_i is the number of times link i is traversed in our component's behavior, and w_i is the expected number of traversals of the link according to the weights in our model of normal behavior. Q measures the confidence that the normal behavior and observed behavior are based on the same underlying probability distribution, regardless of what that distribution may be. The higher the value of Q, the more likely it is that the normal behavior and the component's behavior were generated by different processes, i.e., the higher the anomaly.

We use the χ^2 distribution with $k-1$ degrees of freedom, where k is the number of links in and out of a component, and we compare Q to an anomaly threshold based on our desired level of significance α, where higher values of α are more sensitive to failures but more prone to false-positives; in our experiments, we use a level of significance $\alpha = 0.005$. While Pinpoint does not attempt to detect problems before they happen, similar statistical techniques could be applied to that problem.

Figure 3 shows sample output from a fault detection run in one of our J2EE applications.

{ [1, CatalogEJB, **9.41**] ,
[2, ShoppingCartEJB, **1.09**] ,
[3, ShoppingControllerEJB, **0.34**] ,
[4, JspServlet, **0.12**] ,
[5, MainServlet, **0.02**] }

Figure 3: The top five χ^2 goodness-of-fit scores of components after injecting a fault into a component called CatalogEJB. The scores are normalized, so 1 is the threshold for statistical significance. CatalogEJB, the most anomalous, has a significantly higher score than other components.

5 Recovery

We argued above that rebooting is a universal form of recovery for many categories of failures; we developed the notion of a component-level "microreboot" as a way to reduce the cost of rebooting.

Reboots have pros and cons. They provide a high-confidence way to reclaim stale or leaked resources, they do not rely on the correct functioning of the rebooted system, they are easy to implement and automate, and they return the software to its start state, which is often its best understood and best tested state. On the other hand, in some systems unexpected reboots can result in data loss and unpredictable

recovery times. This occurs most frequently when the system lacks clean separation between data recovery and process recovery. For example, performance optimizations such as writeback caches and buffers open a window of vulnerability during which "persistent" data is stored only in a cache that will not survive a crash; crash-rebooting this system would recover the process, but would not recover the data in the buffers.

5.1 State Segregation

To ensure the correctness of microreboot-based recovery, we must prevent reboots from inducing corruption or inconsistency in the application's persistent state. The inventors of transactional databases recognized that segregating recovery of persistent data from application logic can improve the recoverability of both the application and the data that must persist across failures.

We take this idea further and require that microrebootable applications keep *all* important state in specialized state stores located outside the application, behind well-defined APIs. We factored out all non-volatile state into dedicated crash-safe state stores: a database for all durable data, and a session state store [34] for semi-persistent ("session") application state. The complete separation of component recovery from data recovery makes unannounced microreboots safe. The burden of data management is shifted from the often-inexperienced application writers to the specialists who develop state stores.

Internet applications, like the ones we would expect to run on JBoss, typically handle three types of important state: long-term data that must persist for years (such as customer information), session data that needs to persist for the duration of a user session (e.g., shopping carts or wokflow state in enterprise applications), and virtually read-only data (static images, HTML, JSPs, etc.). We keep these kinds of state in a database, session state store, and a Linux ext3fs read-only filesystem, respectively. The latter is trivially crash-safe.

Persistent state: There are three types of EJB: (a) entity beans, which map each bean instance's state to a row in a database table, (b) session EJBs, which are used to perform temporary operations (stateless session beans) or represent session objects (stateful session beans), and (c) message-driven EJBs (not of interest to this work). EJBs may interact with a database directly and issue SQL commands, or indirectly via an entity EJB. In microrebootable applications we require that only stateless session beans and entity beans be used; this is consistent with best practices for building scalable EJB applications. The entity beans must make use of Container-Managed Persistence (CMP), a J2EE mechanism that delegates management of entity data to the EJB's container. CMP provides relatively transparent data persistence, relieving the programmer from the burden of managing this data directly or writing SQL code to interact with a database. Our prototype applications conform to these requirements.

Session state must persist at the server for long enough to synthesize a user session from independent stateless HTTP requests, but can be discarded when the user logs out or the session times out. Typically, this state is maintained inside the application server and is named by a cookie accompanying incoming HTTP requests. To ensure the session state survives both microreboots and full reboots, we externalize session state into a modified version of SSM, a session state store [34]. SSM's storage model is based on leases, so orphaned session state is eventually garbage-collected automatically. Many commercial application servers forgo this separation and store session state in local memory only, in which case a server crash or EJB microreboot would cause the corresponding user sessions to be lost.

The segregation of state offers some level of recovery containment, since data shared across components by means of a state store does not require that the components be recovered together. Externalized state also helps to quickly reintegrate recovered components, because they do not need to perform data recovery following a microreboot.

5.2 Containment and Reintegration

For an application to gracefully tolerate the microreboot of a component, coupling between components must be loose: components in a microrebootable application have well-defined, enforced boundaries; direct references, such as pointers, may not span these boundaries. Indirect, microreboot-safe references can be maintained outside the components, either by a state store or by the application platform.

Further containment of recovery is obtained through compiler-enforced interfaces and type safety. EJBs cannot name each others' internal variables, nor can they use mutable static variables. While this is not enforced by the compiler, J2EE documents warn against the use of static variables and recommend instead the use of singleton EJB classes, whose state is accessed through standard accessor/mutator methods. EJBs can obtain references to each other in order to make inter-EJB method calls; references are obtained from a naming service (JNDI) provided by the application server, and may be cached once obtained. The inter-EJB calls themselves are also mediated by the application server via the containers, to abstract away the details of remote invocation (if the application server is running on a cluster) or replication (if the application server has replicated a particular EJB for performance or load balancing reasons).

Since EJBs may maintain references to other EJBs, microrebooting a particular EJB causes those references to become stale. To remedy this, whenever an EJB is microrebooted, we also microreboot the transitive closure of its inter-EJB references. This ensures that when a reference goes out of scope, the referent disappears as well. While static or dynamic analysis [10] could be used to determine this closure, we use the

simpler method of determining groups statically by examining deployment descriptors, which are typically generated for the J2EE application by the development environment. The reference information is used by the application server to determine in what order to deploy the EJBs.

The time to reintegrate a microrebooted component is determined by the amount of initialization it performs at startup and the time it takes for other components to recognize the newly-instantiated EJB. Initialization dominates reintegration time; in our prototype it takes on the order of hundreds of milliseconds, but varies considerably by component, as will be seen in Table 3. The time required to destroy and re-establish EJB metadata in the application server is negligible. Making the EJB known to other components happens through the JNDI naming service described earlier; this level of indirection ensures immediate reintegration once the component is initialized.

5.3 Enabling Microreboots

We added a *microreboot* method to the JBoss EJB container that can be invoked from within the application server, programmatically from outside the server through an HTTP adaptor, or by an administrator through a Web-based management interface. Since we modified the JBoss container, microreboots can now be performed on any J2EE application (however, this is safe only if the application conforms to the guidelines of Sections 5.1 and 5.2). The microreboot method destroys all extant instances of the EJB and associated threads, releases all associated resources, discards server metadata maintained on behalf of the EJB, and then reinstantiates the EJB. This fixes many problems such as EJB-private variables being corrupted, EJB-caused memory leaks, or the inability of one EJB to call another because its reference to the callee has become stale.

The only server metadata we do not discard on microreboot is the component's classloader. JBoss uses a separate class loader for each EJB to provide appropriate sandboxing between components; when a caller invokes an EJB method, the caller's thread switches to the EJB's classloader. A Java class' identity is determined both by its name and the classloader responsible for loading it; discarding an EJB's classloader upon microreboot would have (unnecessarily) complicated the update of internal references to the microrebooted component. Keeping the classloader active does not violate any of the sandboxing properties. Preserving classloaders does not reinitialize EJB static variables upon microreboot, but J2EE strongly discourages the use of mutable static variables anyway (to simplify replication of EJBs in clusters).

6 Managing the Recovery Process

As will be seen in Section 7.4, microreboots are an order of magnitude less expensive than an application restart. This makes them suitable for use as a "first line of defense" recovery mechanism even when failure detection is prone to false positives or when the failure is not known to be microreboot-curable. If a microreboot does not recover from the failure but a subsequent recovery mechanism (such as a full reboot) does, the additional recovery time added by attempting the microreboot first is negligible. Microrebooting is therefore a well-suited match for Pinpoint, and we use it as the sole recovery action taken by our recovery manager.

As shown in Figure 2, the recovery manager is an entity external to the application server. It attempts automated recovery and only involves system administrators when automated recovery is unsuccessful.

The recovery manager listens for UDP-based failure notifications from the Pinpoint analysis engine. These notifications contain a list of suspected-faulty components, along with an anomaly score, as described in Section 4.3. These reports are passed to a configurable *policy* module (in our case, it is the "microreboot-based recovery" module); the policy decides what *action* is appropriate (i.e., to recover or not, and at what level). The policy module invokes the microreboot method in the application server.

6.1 Recovery Policy

The recovery manager keeps track of the previously-rebooted subset of components and, if the new subset is the same, it chooses to restart the entire application instead of again restarting that subset of components. If the problem persists even after rebooting the entire system, the policy module can notify a system administrator by pager or email, as necessary. Right now we only employ two levels of rebooting (microreboot followed by a full reboot), but more levels could be used for other kinds of applications.

The policy employed in our recovery manager is as follows:

1. Given a received failure report, ignore all components with a score of less than 1.0 (since the scores are normalized, 1.0 is the threshold for statistical significance).

2. The existence of several components with a score above 1.0 indicates something is wrong: either one component is faulty and the other ones appear anomalous because of their interaction with it, or indeed we are witnessing multiple simultaneous faults. We choose to act conservatively and microreboot all components that are above the 1.0 threshold. In other types of systems it may be better to just microreboot the top n most anomalous components (for some parameter n).

3. For the next interval of time Δt, failure reports involving the just-rebooted components are ignored, since it takes a while for Pinpoint to realize the system has returned to normal. In our experiments, Δt was set to 30 seconds.

4. If subsequent failure reports (after Δt) indicate that the just-recovered components are still faulty, we can either repeat the reboot-based recovery a number of times, or directly proceed to restarting the entire application. Our current policy implements the latter.

5. If, following a full application restart, the problem persists (after another interval of time $\Delta t'$), then an administrator is notified.

A detailed analysis of the various policies and the tradeoffs involved is beyond the scope of this paper. We have purposely built the recovery manager such that new policy modules can be plugged in, to encourage further research on the topic. An interesting problem we have not addressed is dealing with faults that keep reappearing, either because they are triggered by a recurring input, or are simply deterministic bugs.

One final issue is that of recovery manager availability—should the recovery manager go down, nobody will be watching over the system. We have built the recovery manager such that it can restart quickly and lose only recent recovery history upon doing so. It can be run by a

```
while(true) { run_recovery_manager(); }
```

loop. An alternate design, as proposed in [9], is to have another part of the system (such as the application server) watch the manager and restart it, if it appears to have crashed.

7 Evaluation

In this section, we first evaluate the individual effectiveness of each component in our recovery framework, followed by an evaluation of the end-to-end system, and concluding with an analysis of the performance overheads present in our prototype.

7.1 Test Framework

Applications

We deployed three different applications in our testbed cluster:

PetStore 1.1 [51] is Sun's sample J2EE application, that simulates an e-commerce Web site (storefront, shopping cart, purchasing, tracking, etc.). It consists of 12 application components (EJBs and servlets), 233 Java files, and about 11K lines of code, and stores its data in a Cloudscape database.

Petstore 1.3.1 [51] is a significantly re-architected version, that adds a suite of modules for order processing, administration, and supply-chain management to the previous Petstore. In all, it has 47 components, 310 files, and 10K lines of code.

RUBiS [13] is an auction Web site, developed at Rice University for experimenting with different design patterns for J2EE. RUBiS contains over 500 Java files and over 25K lines of code; there are 21 application components and several servlets.

Workload

In order to simulate realistic clients, we extended and used the load generator that ships with RUBiS. It takes a description of the workload for simulated clients in the form of a state transition table T, with the client's states as rows and columns. These states correspond naturally to the various operations possible in RUBiS, such as *Register*, *SearchItemsInCategory*, *AboutMe*, etc. (29 in total). A table cell $T(s, s')$ represents the probability of a client transitioning from state s to state s'; e.g., $T(\mathrm{ViewItem}, \mathrm{BuyNow})$ describes the probability we associate with an end user clicking on the "Buy Now" button after viewing an item's description.

The simulator uses T to automatically navigate the RUBiS Web site: when in s, it chooses a next state s' with probability $T(s, s')$, constructs the URL representing s' and does an HTTP `GET` for the given URL. Inbetween successive clicks, simulated clients have a think time based on a random distribution with average 7 seconds and maximum 70 seconds, as done in the TPC-W benchmark [47]. The simulator uses a separate thread for each simulated client. In choosing the workload for our tests, we mimic the real workload seen by a major Internet auction site; our workload is described in Table 1.

User operation results mostly in...	Fraction of workload
Read-only DB access (e.g., ViewItem, ViewBidHistory)	32%
Creation/deletion of session state (e.g., Login, Logout)	23%
Exclusively static HTML content (e.g., home page)	12%
Search (e.g., SearchItemsInCategory)	12%
Forms that update session state (e.g., MakeBid, BuyNow)	11%
DB updates (e.g., CommitBid, RegisterItem)	10%

Table 1: Test workload. Overall, 8% of requests both read and write to the database, 61% only read, and 31% do not access the database at all. In terms of session state, 27% of requests both read and write, 60% only read, and 13% do not access session state at all.

In choosing the number of clients to simulate, we aimed to maximize system utilization while still getting good performance; for our system, this balance was reached at 350 concurrent clients for one application server node. This is slightly more aggressive than current Internet services, which typically run their application server nodes at 50-60% utilization [36, 22]. We deployed our application server with an embedded Web/servlet tier on Athlon 2600XP machines with 1.5 GB of RAM; the database, Pinpoint, and SSM were hosted on Pentium 2.8 GHz nodes with 1 GB of RAM and 7200rpm 120 GB hard drives. The client simulator ran on a 4-way P-III 550

MHz multiprocessor with 1 GB of RAM. All machines were interconnected by a 100 Mbps Ethernet switch and ran Linux kernel 2.4.22 with Java 1.4.1 and J2EE 1.3.1.

The RUBiS client simulator is specific to the RUBiS application. Thus, for the two Petstore versions we wrote an HTTP load generator, that plays back traces of several parallel, distinct user sessions with each session running in its own client thread. A session consists of a user entering a site, performing various operations such as browsing or purchasing items, and then leaving the site. We choose session traces so that the overall load on the service fully exercises the components and functionality of the site. If a client thread detects an HTTP error, it retries the request. If the request continues to return errors, the client quits the trace and begins the session again. The traces are designed to take different routes through the Web service, such that a failure in a single part of the Web service will not artificially block all the traces early in their life cycle.

Faultload

We measured end-user-perceived system availability in the face of failures caused by faults we injected, with the recognition that no fault injection experiment can claim to be complete or to accurately reflect real-life faults. As mentioned in Section 1, our work focuses exclusively on failures that can be cured with some form of a reboot. Despite J2EE's popularity as a commercial infrastructure, we were unable to find any published systematic studies of faults occuring in production J2EE systems, so we relied on advice from colleagues in industry who routinely work with enterprise applications or application servers [19, 45, 38, 18, 44, 33]. We also surveyed studies of reported faults in other kinds of systems and the faults injected by other researchers in their experiments [43, 27, 16]. We converged onto the following categories of software-related failures:

- accidental use of null references (e.g., during exception handling) that result in `NullPointerException`
- hung threads due to deadlocks, interminable waits, etc.
- bug-induced corruption of volatile metadata
- leak-induced resource exhaustion
- omission faults (intercept a method call and do not return, or return 0/null if a return value is required)
- simple source code bugs, injected with Polyglot [42]
- various other Java exceptions and errors that are not handled correctly

We aimed to reproduce these problems in our system by adding facilities for runtime fault injection: we can (a) set component class variables to `null`, (b) directly induce deadlock conditions and infinite loops in EJBs, (c) alter global volatile metadata, such as garble entries in the JNDI naming service's database, (d) leak a certain amount of memory per call, and (e) intercept calls to EJBs and, instead of passing the call through to the component, throw an exception/error of choice or drop the call.

We do not inject low-level hardware or OS faults, such as CPU register bit-flips, memory corruptions and IO errors because, empirically, these faults do not manifest as application-level failures that would otherwise go unnoticed [12]. We do not inject all faults in all experiments.

7.2 Detection

To test Pinpoint's effectiveness in detecting failures, we injected faults in three different J2EE applications (both variants of Petstore and RUBiS). The faults took the form of forced runtime Java exceptions, forced declared Java exceptions, call omissions, and source code bugs; the experimental details can be found in [30]. We then placed load on the application and attempted to detect the presence of the faults with three different detectors: one that monitors for HTTP errors, one that parses HTML pages and searches for keywords indicating errors, and Pinpoint, respectively. Table 2 summarizes the results. Although Pinpoint is better, the main improvement in comparison to the other two simple monitors is not that much its ability to detect failures, but the fact that it is a low-maintenance, application-generic solution to high-level fault-detection.

Detector	Declared Exc.	Runtime Exc.	Omissions	Src Bugs
Pinpoint	56% / 89%	68% / 96%	70% / 96%	12% / 37%
HTTP codes	48% / 64%	53% / 70%	44% / 65%	10% / 37%
HTML parsing	28% / 40%	24% / 35%	17% / 20%	2% / 13%

Table 2: For each detector, we show the fraction of injected faults it detected across all our applications. In each cell, the first number indicates how well we detected all faults in the category, while the second indicates how well we detected major outage-causing faults in the category.

We do not include application-specific monitors in our comparison, since deciding what application functionality to test would have been the determining factor in detecting many of these failures; an application-specific monitor can be engineered to test for almost any known/expected fault. While a highly application-specific detector might be able to detect close to 100% of an application's failures (in the extreme, duplicating the application's functionality inside the detector), writing and maintaining it would be very difficult. For this reason, such detectors are seldomly used in practice.

7.3 Localization

The overall results of our localization tests are shown in Figure 4. We measured the efficacy of localization in Pinpoint's component interaction analysis and also compared it to three other algorithms, that use decision trees. These other algorithms are part of Pinpoint, but were not employed in the subsequent experiments of this paper.

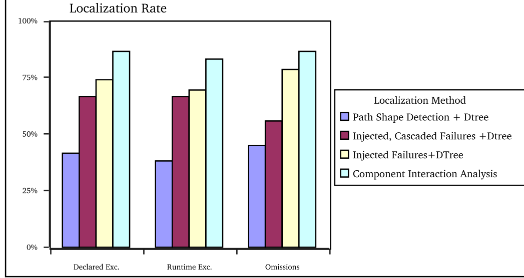

Figure 4: Localization rate of several algorithms per fault type. For this paper we used exclusively "component interaction analysis," which exceeds 85% for all three types of faults.

7.4 Recovery

We evaluated microreboots with a version of RUBiS (the auction application) that maintains session state in SSM [34]. To measure the impact of failures on end users, we defined a new metric, *action-weighted goodput* (G_{aw}). We view a user *session* as beginning with a login operation and ending with an explicit logout or abandonment of the site. Each session consists of a sequence of *user actions* (such as "search for a pair of Salomon ski boots and place a $200 bid on them"); this action is successful if and only if all operations (i.e., HTTP requests) within that action succeed; otherwise, the action fails as a unit. Whenever an operation fails, all operations in the containing action are counted as failed. G_{aw}accounts for the fact that both long-running and short-running operations must succeed for a user to be happy with the site. G_{aw}also captures the fact that, when an action with many operations succeeds, it often means that the user got more work done than in a short action.

In order to isolate the effects of recovery, we wrote for this section a custom, application-specific "perfect" fault detector that is able to detect practically immediately each failure. We injected a sequence of faults into SSM-RUBiS and allowed the system to recover from failure in two ways: by restarting SSM-RUBiS or by microrebooting a component, respectively. We consider recovery to be successful when end users do not experience any more failures after recovery (unless new faults are deliberately injected). JBoss allows for a J2EE application to be hot-redeployed, and this constitutes the fastest way to do reboot-based recovery in unmodified JBoss; we use SSM-RUBiS application restart as a baseline, and call this a *full reboot*.

Table 3 shows comprehensive measurements for all the recovery units in our system; the first three rows are not EJBs and are shown mainly for comparison purposes. Most EJBs generally recover an order of magnitude faster than an application reboot.

Component	Avg	Min	Max
JBoss app server restart	51,800	49,000	54,000
RUBiS application restart	11,679	7,890	19,225
Jetty (embedded Web server)	1,390	1,009	2,005
SB_CommitBid	286	237	520
SB_BrowseCategories	340	277	413
SB_ViewUserInfo	398	288	566
SB_ViewItem	465	284	977
SB_RegisterUser	502	292	681
SB_CommitUserFeedback	509	316	854
SB_SearchItemsByRegion	529	296	906
SB_CommitBuyNow	550	297	1,102
SB_RegisterItem	552	363	837
SB_Auth	554	317	1,143
SB_BrowseRegions	597	241	906
SB_BuyNow	603	303	1,417
SB_ViewBidHistory	623	317	2,058
SB_AboutMe	639	330	1,287
SB_LeaveUserFeedback	680	314	1,275
SB_MakeBid	856	232	2,920
SB_SearchItemsByCategory	911	488	3,019
IDManager	1,059	663	1,547
UserFeedback	1,248	761	1,591
BuyNow	1,421	668	4,453
User-Item	1,828	876	4,636

Table 3: Recovery times (in msec) for whole JBoss, SSM-RUBiS, the Web tier collocated with JBoss, and the various application components. Components prefixed by *SB* are stateless session EJBs, while the rest are entity EJBs. We ran 10 trials per component, under load from 350 concurrent clients.

Figure 5 shows G_{aw} as measured by our client simulator with a load of 350 clients; each sample point represents the number of successful/failed requests observed during the corresponding 1-second interval. In choosing the locations for fault injection, we wanted to study the extremes of recovery time, as well as the spectrum of workload-induced effects. Hence, we injected the first fault in the component that takes the shortest time to recover, the seconds fault in the one that takes the longest, the third fault in the most frequently called component, and the fourth fault in the least frequently called one. Requests are seen to "fail" prior to the injection point because of how $G_{\text{aw}}$is computed: if an operation fails, then all operations within that user action that succeeded in the past are failed retroactively, to reflect that the user's work has been lost.

For a thorough evaluation of microreboots, we refer the

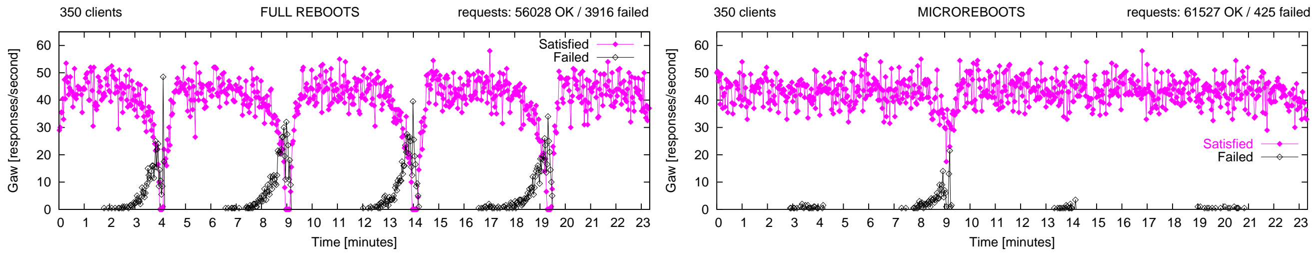

Figure 5: Full reboots vs. microreboots: We injected a null reference fault in SB_CommitBid, then a corrupt JNDI database fault for User-Item, followed by a *RuntimeException* in SB_BrowseCategories, and a Java *Error* in SB_CommitUserFeedback. Left graph shows automatic recovery via full application reboot; right graph shows recovery from the same faultload using microreboots. SB_CommitBid took 387 msec to recover, User-Item took 1742 msec, SB_BrowseCategories 450 msec, and SB_CommitUserFeedback 404 msec. In reaction to the second failure, our recovery service's diagnosis included 6 false positives, resulting in 6 unnecessary microreboots, totaling 4765 msec. In spite of this, there are 89% fewer failed requests (425 vs. 3916) and 9% more successful requests (61527 vs. 56028) in the case of microrebooting.

reader to [12].

7.5 Combination recovers autonomously

In Figure 6 we illustrate the functioning of the integrated system in reaction to a single-point fault injection; this is representative of the reaction to the other categories of faults we injected.

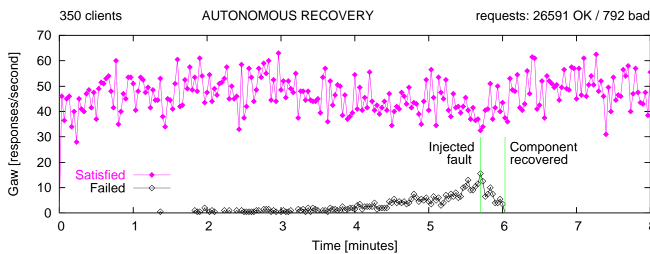

Figure 6: Autonomous recovery: We corrupted an internal data structure in SB_ViewItem, setting it to null, which results in a NullPointerException for SB_ViewItem callers. Labeled light-colored vertical lines indicate the point where the fault is injected and where the faulty component completes recovery, respectively.

In Figure 7 we zoom in on the interval between 5:32 and 6 seconds, to analyze the events that occur; the horizontal axis now represents seconds. We mark on the graph the points (along with the time, to millisecond granularity) at which the following events occur: we inject the fault (t_1), then the first end user request to fail as a consequence is at t_2, the Pinpoint analysis engine sends its first failure report to the recovery manager (t_3), the recovery manager decides to send a recovery command to the microreboot hook (t_4), the microreboot is initiated (t_5), and finally the microreboot completes (t_6) and no more requests fail.

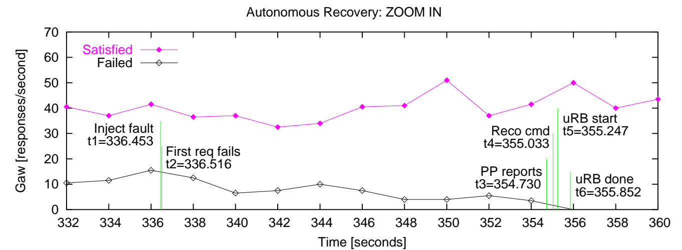

Figure 7: Zooming in on the time interval [5:32, 6:00]. Recovery time is 19.4 seconds from the time the first end user notices a failure; of this interval, 18.5 seconds is spent by Pinpoint noticing that *SB_ViewItem* is faulty.

The system recovers on its own within 19.4 seconds of the first end user failure; of this interval, 18.5 seconds is spent by Pinpoint detecting and localizing the fault. This time compares favorably to the recovery times witnessed in Internet services involving human assistance; recovery there can range from minutes to hours. Note that the kind of faults we inject cannot be noticed by TCP or HTTP-level monitors, because the web pages returned by the server constitute valid HTML. It takes on the order of a second or less to recover from a faulty EJB with a microreboot, compared to a full application reboot.

Finally, in Figure 8 we show the result of a multi-point injection: three different faults in three different components, respectively. Our system notices and recovers two of the components within 19.9 seconds, and the third component 46.6 seconds after the injection. The reason for the delay is that, in our workload, the first two components are called more frequently than the third. Thus, the Pinpoint analysis engine re-

ceives more observations sooner, which gives it a quicker opportunity to detect and localize the injected fault.

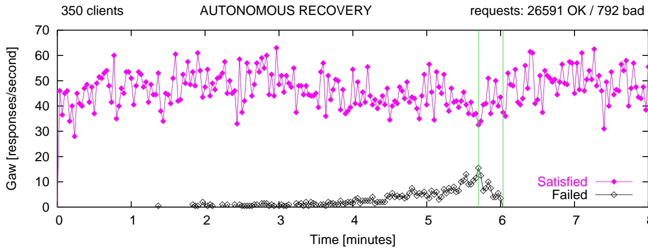

Figure 8: Correlated faults: We simultaneously injected a data corruption fault in SB_PutBid (C_1), a Java Exception fault in SB_ViewUserInfo (C_2), and a Java Error fault in SB_SearchItemsByRegion (C_3). The more frequently-called C_1 and C_3 are detected and recovered sooner (19.9 sec after injection) than C_2 (46.6 sec after injection).

7.6 Application Evolution

To test Pinpoint's resilience against "normal changes", we ran two experiments with Petstore; the historical model we used was based on the behavior of Petstore 1.3.1 under our normal workload for both experiments.

First, we significantly changed the load offered by our workload generator: we stopped sending any ordering or checkout related requests. Second, we upgraded Petstore v1.3.1 to a bug-fix release, Petstore v1.3.2.

In both experiments, Pinpoint did not trigger any false positives. Alhough the gross behavior of application components did change with the workload, the fact that we analyze component interactions in the context of different types of requests compensated for this, and we detected no significant changes in behavior. In the upgrade to Petstore 1.3.2 our component behaviors did change noticeably, but still did not pass our threshold according to the χ^2 test. Though not comprehensive, these two experiments suggest that our fault detection techniques are robust against reporting spurious anomalies when actual application functionality has not changed. This is a benefit of the fact that Pinpoint is application-generic.

7.7 Performance Overhead

The Pinpoint instrumentation of JBoss to trace requests adds a latency penalty of 2-40 msec to each client request, depending on the number of components called, and degrades overall throughput by 17% on the mixed workload used in our experiments. Pinpoint's analysis engine operates on a separate machine and does not directly impact application performance; however, if the machine is slower than the ones used in our testbed, it could lead to engine overload. In this case, we drop observations on the JBoss side, rather than hurt performance. The deployment of commercial instrumentation packages such as IntegriTea (www.tealeaf.com) on large sites such as Priceline.com suggests that fine-grained instrumentation is practical if some engineering effort is focused on the implementation.

We separately evaluated our microreboot-centric design; Table 4 compares three different configurations: vanilla JBoss 3.2.1 release running HttpSession-enabled RUBiS and SSM-enabled RUBiS running on top of both microreboot-enabled JBoss and vanilla JBoss 3.2.1, respectively. HttpSession-enabled RUBiS is a version of the application that uses the in-memory HttpSession mechanism for storing session state, instead of the SSM solution; HttpSessions do not survive reboots.

		JBoss 3.2.1 w/ HttpSession-RUBiS	uRB-JBoss w/ SSM-RUBiS	JBoss 3.2.1 w/ SSM-RUBiS
Throughput [req/sec]		44.8	44.5	44.4
Request Latency [msec]	Ave	39	82	83
	Min	3	3	3
	Max	826	1245	1097
	StDev	.066	.131	.142

Table 4: Performance comparison of microrebootable vs. non-microrebotable design. Results from a 30-minute fault-free run, with 350 concurrent clients against a single node.

Comparing the first two configurations, we see that in spite of the modifications we made to enable microrebooting, throughput remains virtually the same (within 1%). On the other hand, average response time doubles from 39 to 82 msec. However, human-perceptible delay is commonly believed to be 100-200 msec, which means that the 82 msec latency is of no consequence for an interactive Internet service like ours. The worst case response time increases from 0.8 sec to 1.2 sec; while certainly human-perceptible, this level is below the 2-second threshold for what human users perceive as fast [37]. Both latency and throughput are within the range of measurements done at a major Internet auction service [52], where average throughput per cluster node is 41 req/sec, while latency varies between 33 and 300 msec.

A comparison of HttpSession-RUBiS and SSM-RUBiS running on vanilla JBoss reveals that the observed increase in response time is entirely due to our use of SSM. Given that session state is externalized to a separate node, accessing it requires the session object to be marshalled, sent over the network, then unmarshalled; this consumes considerably more CPU than if it were kept in the server's memory. Maintaining a write-through cache of session data in the application server could absorb some of this latency penalty. However, performance optimization was not one of our goals, so we did not explore this possibility.

8 Discussion

8.1 Assumptions

The most fundamental assumption we have made in our design and implementation is that the monitored application is built from interconnected modules (components) with well-defined narrow interfaces. Typical large Internet services will often be written using one of several standard component frameworks, such as .NET or J2EE, and will have a three-tiered architecture.

Another important assumption is that the application has a structure determined by call paths, and components interact with each in accordance to this structure. We assume that, by observing the interactions between components, we can dynamically build a model of "normal" behavior and detect when such behavior becomes anomalous. The more pervasive the monitoring, the more observability we gain into the system and the better we can detect faults.

We are also assuming Internet-style HTTP workloads, consisting of large numbers of relatively short tasks that frame peristent state updates. Such request-reply workloads imply that a single interaction with the system is relatively short-lived, and its processing can be broken down as a sequence of calls to various components, resulting in a tree of named components that participate in the servicing of that request. Failing any one user request (e.g., due to a microreboot) has a negligible impact across the entire user population.

Our analysis engine requires a large workload to exercise all aspects of the application in order to develop a good statistical model of correct behavior. Combining a high volume of largely independent requests from different users allows us to appeal to "law of large numbers" arguments justifying the application of statistical techniques.

Finally, for microrebooting to be effective as a general technique against transient/intermittent failures, the application needs to conform to the guidelines outlined in Section 5; these design principles are collectively known as "crash-only software" [11]. Requests need to either be idempotent, or be easily made idempotent through the use of transactions. This is true of many Internet systems today, but may not carry over to other types of systems.

8.2 Limitations

Pinpoint faces difficulties when an application switches to a different, but correct, operating mode because of external conditions. For example, under heavy load, CNN.com simplifies its "headline" pages rather than deny service [32]; such a large-scale change would likely appear anomalous when it is triggered. If mode switching occurs often, it should be recognized as normal behavior, but activation of rarely-exercised modes will confuse our historical analysis (though not our peer analysis).

When a component exhibits multiple modes of behavior, our component interaction model attempts to capture a non-existent average of the multiple modes, and subsequently detects all the components as deviating from this average, resulting in a deluge of false positives. One possible solution is to use a model that captures multi-modality, though this has the danger of mis-classifying truly anomalous behavior as simply "another mode."

If state updates are atomic, as they are with a database and with SSM, there is no distinction between microreboots and full reboots from the point of view of the state store. The case of non-transactional, shared EJB state is more challenging: the microreboot of one component may leave the state inconsistent, unbeknownst to the other components that share it. A full reboot, on the other hand, would not give the other components the opportunity to see the inconsistent state, since they would be rebooted as well. J2EE best practices do discourage sharing state by passing references between EJBs or using static variables, but we believe they could be enforced by a suitably modified JIT compiler; alternatively, should the runtime detect such practices, it could disable the use of microreboots for the application in question.

Our implementation of microreboots does not scrub application-global data maintained by the application server, such as the JDBC connection pool and various other caches. Microreboots also generally cannot recover from problems occurring at layers below the application, such as the application server or the JVM. In all these cases, a full server restart may be required.

Poor failure diagnosis may result in one or more ineffectual microreboots of the wrong EJBs, leading to microrebooting progressively larger groups of components until the whole application is rebooted. Even in this case, however, microrebooting adds only a small additional cost to the total recovery cost.

9 Conclusion

In this paper we showed how to combine statistical anomaly detection with microrecovery to build an application server that autonomously recovers J2EE applications. Our system is effective in detecting and recovering realistic transient faults, with no a priori application-specific knowledge. This approach combines the generality and deployability of low-level monitoring with the sophisticated failure-detection abilities usually exhibited only by application-specific high-level monitoring. Due to its level of generality, false positives do occur, but cheap recovery makes the cost of these false positive negligible. Microreboots are cheap enough for frequent use as a first line of defense [12], because even if microrebooting ends up being ineffective, it doesn't hurt to try. The synergy between Pinpoint and microreboots offers the potential for sig-

nificant simplification of failure management.

Recovering by microreboot does not mean that the root causes should not eventually be identified and the bugs fixed. However, reboot-based recovery provides a separation of concerns between recovery and diagnosis/repair, consistent with the observation that the latter is not always a prerequisite for the former. Attempting to recover a reboot-curable failure by anything other than a reboot always entails the risk of taking longer than a reboot would have taken in the first place: trying to diagnose the problem can often turn what could have been a fast reboot into a long-delayed reboot, thus hurting availability. Microrebooting has the advantage of confining recovery to a small part of the application, thus protecting the majority of active users from experiencing an outage.

Autonomous recovery enables automated response to failures, that operates in "machine time" rather than "human time," thus improving end user experience. Such autonomy is particularly useful for systems located in zero-administration environments, where access to the system is not immediate. Although we exploited properties of the J2EE programming model to simplify our implementation, we believe the techniques presented here can be applied more generally to non-J2EE systems.

References

[1] M. K. Aguilera, J. C. Mogul, J. L. Wiener, P. Reynolds, and A. Muthitacharoen. Performance debugging for distributed systems of black boxes. In *Proc. 19th ACM Symposium on Operating Systems Principles*, Bolton Landing, NY, 2003.

[2] M. Baker and M. Sullivan. The Recovery Box: Using fast recovery to provide high availability in the UNIX environment. In *Proc. Summer USENIX Technical Conference*, San Antonio, TX, 1992.

[3] M. Barnes. J2EE application servers: Market overview. The Meta Group, Mar. 2004.

[4] J. F. Bartlett. A NonStop kernel. In *Proc. 8th ACM Symposium on Operating Systems Principles*, Pacifi c Grove, CA, 1981.

[5] A. Bouloutas, S. Calo, and A. Finkel. Alarm correlation and fault identifi cation in communication networks. *IEEE Transactions on Communications*, 1994.

[6] E. Brewer. Lessons from giant-scale services. *IEEE Internet Computing*, 5(4):46–55, July 2001.

[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] Business Internet Group. The black Friday report on Web application integrity. San Francisco, CA, 2003.

[9] G. Candea, J. Cutler, A. Fox, R. Doshi, P. Garg, and R. Gowda. Reducing recovery time in a small recursively restartable system. In *Proc. International Conference on Dependable Systems and Networks*, Washington, DC, June 2002.

[10] G. Candea, M. Delgado, M. Chen, and A. Fox. Automatic failure-path inference: A generic introspection technique for software systems. In *Proc. 3rd IEEE Workshop on Internet Applications*, San Jose, CA, 2003.

[11] G. Candea and A. Fox. Crash-only software. In *Proc. 9th Workshop on Hot Topics in Operating Systems*, Lihue, Hawaii, 2003.

[12] G. Candea, S. Kawamoto, Y. Fujiki, and A. Fox. A microrebootable system – design, implementation, and evaluation. In *Proc. 6th USENIX Symposium on Operating Systems Design and Implementation*, San Francisco, CA, Dec. 2004.

[13] E. Cecchet, J. Marguerite, and W. Zwaenepoel. Performance and scalability of EJB applications. In *Proc. 17th Conference on Object-Oriented Programming, Systems, Languages, and Applications*, Seattle, WA, 2002.

[14] M. Chen, A. Accardi, E. Kiciman, D. Patterson, A. Fox, and E. Brewer. Path-based macroanalysis for large, distributed systems. In *First Symposium on Networked Systems Design and Implementation*, 2004.

[15] M. Chen, A. Zheng, J. Lloyd, M. Jordan, and E. Brewer. Failure diagnosis using decision trees. In *International Conference on Autonomic Computing*, New York, NY, May 2004.

[16] R. Chillarege and N. S. Bowen. Understanding large system failures - a fault injection experiment". In *Proc. International Symposium on Fault Tolerant Computing*, June 1989.

[17] J. Choi, M. Choi, and S. Lee. An alarm correlation and fault identifi cation scheme based on OSI managed object classes. In *Proc. of IEEE Conference on Communications*, 1999.

[18] T. C. Chou. Personal communication. Oracle Corp., 2003.

[19] H. Cohen and K. Jacobs. Personal communication. Oracle Corporation, 2002.

[20] K. Coleman, J. Norris, A. Fox, and G. Candea. OnCall: Defeating spikes with a free-market server cluster. In *Proc. International Conference on Autonomic Computing*, New York, NY, May 2004.

[21] A. Diaconescu, A. Mos, and J. Murphy. Automatic performance management in component based software systems. In *First International Conference on Autonomic Computing*, New York, NY, May 2004.

[22] S. Duvur. Personal communication. Sun Microsystems, 2004.

[23] D. Engler, D. Y. Chen, S. Hallem, A. Chou, and B. Chelf. Bugs as deviant behavior: A general approach to inferring errors in systems code. In *Proc. 18th ACM Symposium on Operating Systems Principles*, Lake Louise, Canada, Oct 2001.

[24] J. Gray. Why do computers stop and what can be done about it? In *Proc. 5th Symposium on Reliability in Distributed Software and Database Systems*, Los Angeles, CA, 1986.

[25] S. D. Gribble, E. A. Brewer, J. M. Hellerstein, and D. Culler. Scalable, distributed data structures for Internet service construction. In *Proc. 4th USENIX Symposium on Operating Systems Design and Implementation*, San Diego, CA, Oct. 2000.

[26] S. Hangal and M. Lam. Tracking down software bugs using automatic anomaly detection. In *Proceedings of the International Conference on Software Engineering*, May 2002.

[27] M.-C. Hsueh, T. K. Tsai, and R. K. Iyer. Fault injection techniques and tools. *IEEE Computer*, 30(4):75–82, 1997.

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

[29] J. O. Kephart and D. M. Chess. The Vision of Autonomic Computing. *Computer Magazine*, Jan 2003.

[30] E. Kiciman and A. Fox. Detecting application-level failures in component-based internet services. 2004. In preparation.

[31] E. Lassettre, D. Coleman, Y. Diao, S. Froelich, J. Hellerstein, L. Hsiung, T. Mummert, M. Raghavachari, G. Parker, L. Russell, M. Surendra, V. Tseng, N. Wadia, and P. Ye. Dynamic Surge Protection: An Approach to Handling Unexpected Workload Surges with Resource Actions that have Lead Times. In *Proc. of 1st Workshop on Algorithms and Architectures for Self-Managing Systems*, San Diego, CA, June 2003.

[32] W. LeFebvre. CNN.com—Facing a world crisis. In *15th USENIX Systems Administration Conference*, 2001. Invited Talk.

[33] H. Levine. Personal communication. EBates.com, 2003.

[34] B. Ling, E. Kiciman, and A. Fox. Session state: Beyond soft state. In *First Symposium on Networked Systems Design and Implementation*, 2004.

[35] 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.

[36] A. Messinger. Personal communication. BEA Systems, 2004.

[37] R. Miller. Response time in man-computer conversational transactions. In *Proc. AFIPS Fall Joint Computer Conference*, volume 33, 1968.

[38] N. Mitchell. IBM Research. Personal Communication, 2004.

[39] N. Mitchell and G. Sevitsky. LeakBot: An automated and lightweight tool for diagnosing memory leaks in large Java applications. In *Proc. 17th European Conference on Object-Oriented Programming*, Darmstadt, Germany, July 2003.

[40] B. Murphy and N. Davies. System reliability and availability drivers of Tru64 UNIX. In *Proc. 29th International Symposium on Fault-Tolerant Computing*, Madison, WI, 1999. Tutorial.

[41] 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.

[42] N. Nystrom, M. R. Clarkson, and A. C. Myers. Polyglot: An Extensible Compiler Framework for Java. In *Proc. of the 12th International Conference on Compiler Construction*, Warsaw, Poland, Apr. 2003.

[43] 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.

[44] A. Pal. Personal communication. Yahoo!, Inc., 2002.

[45] D. Reimer. IBM Research. Personal Communication, 2004.

[46] I. Rouvellou and G. W. Hart. Automatic alarm correlation for fault identification. In *Proc. IEEE INFOCOM '95*, 1995.

[47] W. D. Smith. TPC-W: Benchmarking an E-Commerce solution. Transaction Processing Council, 2002.

[48] L. Spainhower and T. A. Gregg. IBM S/390 parallel enterprise server G5 fault tolerance: A historical perspective. *IBM Journal of Research and Development*, 43(5/6), 1999.

[49] M. Stonebraker. The design of the Postgres storage system. In *Proc. 13th Conference on Very Large Databases*, Brighton, England, 1987.

[50] M. Sullivan and R. Chillarege. Software defects and their impact on system availability – a study of field failures in operating systems. In *Proc. 21st International Symposium on Fault-Tolerant Computing*, Montréal, Canada, 1991.

[51] Sun_Microsystems. Java Pet Store Demo. http://developer.java.sun.com/developer/releases/petstore/, 2002.

[52] TBD. A major Internet auction site. Terms of disclosure are being negotiated, May 2004.

[53] Y.-M. Wang, Y. Huang, K.-P. Vo, P.-Y. Chung, and C. M. R. Kintala. Checkpointing and its applications. In *Proc. 25th International Symposium on Fault-Tolerant Computing*, 1995.

[54] Y.-M. Wang, C. Verbowski, and D. R. Simon. Persistent-state checkpoint comparison for troubleshooting configuration failures. In *Proc. of the IEEE Conference on Dependable Systems and Networks*, 2003.

[55] A. Ward, P. Glynn, and K. Richardson. Internet service performance failure detection. In *Proc. Web Server Performance Workshop*, 1998.

[56] K. Whisnant, R. Iyer, P. Hones, R. Some, and D. Rennels. Experimental evaluation of the REE SIFT environment for spaceborne applications. In *Proc. International Conference on Dependable Systems and Networks*, Washington, DC, 2002.

[57] A. P. Wood. Software reliability from the customer view. *IEEE Computer*, 36(8):37–42, Aug. 2003.

[58] A. Yemeni and S. Kliger. High speed and robust event correlation. *IEEE Communications Magazine*, 34(5), May 1996.