Recovery-Oriented Computing: Main Techniques of Building Multitier Dependability

Yi Ding

Abstract—**Frequent freezes and crashes on current systems bring tremendously heavy loads to the system administration, directly resulting in an undesirable increase on the total cost of ownership (TCO). Obviously, it is time to broaden the long lasting performance-dominated research focus, which has neglected other aspects of computing such as dependability, availability and stability. Deeming that software bugs, hardware faults and operator errors are facts to be coped with, not problems to be solved, Recovery Oriented Computing (ROC) concentrates on building systems that recover fast when a fault does occur, instead of aiming for systems that never fail. To reach high dependability in the Internet service environment, ROC implements two building blocks for recovery, microreboot and system-level undo, which have proven effective in handling failures. A suitable benchmarking method is developed to quantify the impact of these effects on the system dependability.**

Index Terms—**Microreboot, undo/redo, benchmarking**

I. Introduction

In spite of the ever-improving design on hardware and software, computer systems still crash or freeze frequently. Web sites seem to become unavailable when we most need their services. Human operators regularly bring disorders to the systems they administer. Apparently, system administration constitutes a major part of total cost of ownership, with a large fraction of this cost going to recovering and managing failures. A detailed survey for cluster-based services with respect to total cost of ownership (TCO) [3] implies that the TCO/Purchase ratios are from 3.6 to 18.5 as shown in Table I.

The survey reminds us a fact that a majority of TCO goes to either recovering from or preparing against system failures. Since large portion of system administration is dealing with failure handling, reducing the costs from failure recovery reduces the total cost of ownership (TCO) as well. As a joint research effort of Stanford University and University of California, Berkeley, Recovery-Oriented Computing (ROC) group studies techniques of facilitating system to recover from inevitable failures. By regarding failures as a fact of life, ROC group strives on building systems that can recover as fast as possible, rather than on trying to prevent failures.

To reach a multitier dependability, two techniques are of the main concern in this paper: microreboot and system level undo. As a front line recovery, microreboot provides an acceptable mechanism to achieve high availability at relatively low costs. On the other hand, system level undo renders a more comprehensive method for recovering from state-corrupting failure, and also provides operators a forgiving operation environment to help address an unfulfilled need that although system operations play a critical role in maintaining system dependability, the operators lack powerful tools to help them do so.

ROC researches are geared mainly toward Internet services due to the unique challenges presented by them: Google has more than 100,000 computing nodes subjected to perpetual evolution; with weekly software updates in common and various workloads by orders of magnitude over the day, they are expected to provide stable service on 24/7 within this dynamic environment. ROC believes that lessons and experiences learned from Internet can also shed lights on other computing environments such as smaller network services and desktops.

In the reminder of this paper, an overview on multitier dependability design space is given in section II. Section III provides a description of microreboot prototype and lessons obtained from the implementation. In section IV, the general 3R model "Rewind, Repair and Replay" for operator undo is introduced, followed by the integration of generic undo design with the specific application of an e-mail store. Section V evaluates the effectiveness of techniques with benchmarking, and we conclude in section VI.

II. Multitier dependability design space

A. General design space

To define the system availability in terms of the relationship between mean-time-to-fail (MTTF) and mean-time-to-recover (MTTR), there is a common formula:

Availability = MTTF / (MTTF + MTTR)

As implied by the formula, there exist two ways of increasing availability: increase the MTTF to infinity thus approaching 1/1 (hardly ever fail), or bring down the MTTR close to zero which means short recovery time. ROC argues

Yi Ding is a master student at Department of Computer Science, University of Helsinki (e-mail: yi.ding@cs.helsinki.fi).

Operating system/Service	Linux/Internet	Linux/Collab.	Unix/Internet	Unix/Collab.
Average number of servers	3.1	4.1	12.2	11.0
Average number of users	1150	4550	7600	4800
HW-SW purchase price	\$127,650	\$159,530	\$2,605,771	\$1,109,262
3 year Total Cost of	\$1,020,050	\$2,949,026	\$9,450,668	\$17,426,458
TCO/HW-SW ration	8.0	18.5	3.6	15.7

Table I: Ratio of three tear total cost of ownership to hardware-software purchase price [3].

that it is easier for us today to approach high availability through reducing MTTR as:

$$\lim_{MTTR \to 0} (availability) = 100\%$$

At the same time, it is of equal importance that recovery should not only be fast but correct as well. To reach of goal of flawless recovery, Recovery-oriented systems must defend themselves in depth by adopting multiple layers of recovery. Two principal axes are proposed to present the design – cost of recovery and percentage of recoverable failures, as showed in Figure I: One axis captures the amount of breadth a technique can achieve in terms of failures it cures; the other axis refers to the general cost of employing that technique. In a sense, these axes capture a cost benefit ratio for recovery.

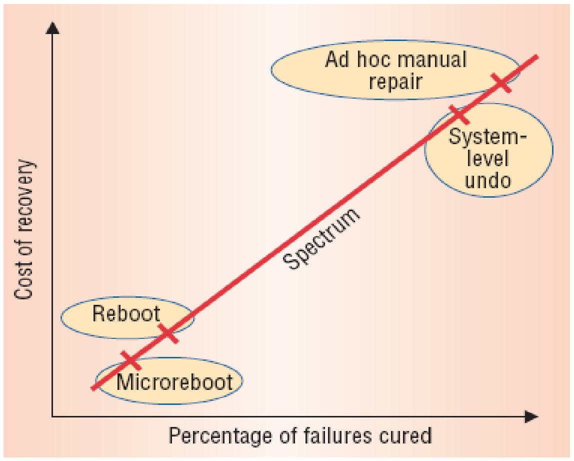

Figure I: Software recovery design space [6].

B. *Principles to guide research*

Through the research projects done by ROC group [6], four principles are learned as listed below, to guide further researches.

1) Safeguarding system state is required by fast recovery:

Recovering a failed system means to bring it back to a point where the system can function as well as it used to before failing. The system state, a union of all the data which programs use to guide their operations, largely determines the capacity of serving requests, and the cost of recovery depends on how long it takes to reconstruct the state of the system.

Take the Web server for example, which is a typical stateless system. When requests come, each individual request does not require the server to maintain a state on its behalf. Therefore the system can be rebooted respectively safely, and subsequent HTTP requests will not notice that server was maintained. On the other hand, an operating system is a stateful program which reads and changes states on behalf of the applications it runs. Discarding the state would bring down the entire system. To recover from such a problem as one corrupted Windows registry, usually require us to reinstall the whole operating system and its corresponding applications again, in order to recreate a large amount of state. This process might take from hours to many days.

Most existing dependable systems rely on certain form of rollback recovery, using state checkpoints or activity logs to help restore after a failure occurs [4]. Because rollback can be expensive, the key to fast recovery is to protect the state from corrupttion by safeguarding it as much as possible from the program logic. Overall dependability will increase with the level of separation of data management from application. For this to be effective, our programs should access the data only via well-defined interfaces.

2) Fine-grained workloads speed up recovery:

Another factor influencing the system recovery is the workload. For instance, when rebooting a Web server, the lost state is the HTTP requests that happened during the interval of recovery. The stateless nature of HTTP, combined with the retrying from client side, keeps the correctness across short term failure. If the system designers can break the workload into small units independent of each other, then to recover from failure needs little state reconstruction, thus accelerating the recovery.

3) Platform recovery is cost-effective:

Two prototypes described in this paper implement recovery in the platform rather than on the application level. Although it is harder to reach, this approach might leverage recovery code across multiple applications including developed and tested ones, both present and forthcoming.

4) We can’t improve what we can’t measure:

One goal of ROC research is to develop suitable ways to benchmark the dependability of systems designed by ROC group. The process of defining such benchmarks teaches us where the system vulnerable points are, and using them to guide the future design. An extra benefit of benchmarking effort is that it can make the recovery techniques more quantitative and persuasive, thus promoting the prospect of adopting those methods in real systems.

III. Microreboot – a Technique for Cheap Recovery

In industry, rebooting is often regarded as a universal cure for software failures in that it is easy to implement and automate, and can bring failed applications back to start state which is a best-tested state [7]. Unfortunately, the recovering process could take a long time if lots of state reconstructions are needed. What is more, unexpected reboots in a system that is not crash-safe could even result in data loss.

One major requirement for the front-line defense in building multitier dependability is to keep both cost and overhead at a low level with a good possibility to fix the problem, and low opportunity cost in case it does not work. The technique designed by ROC – Microreboot [5], renders a refined recovery mechanism that can restore most of the same failures as full reboot, but does so an order of magnitude faster thus achieving an order of magnitude savings in lost work. As microreboot preserves the advantages of reboot while mitigating the corresponding drawbacks, ROC group therefore chooses it as a promising candidate for the front-line recovery in building multitier dependability.

A. *Microreboot design principles*

A microboot is the selective crash-restart of only those parts of a system that trigger the observed failure [5]. It is regarded as an effective mechanism for system recovery due to the fact that a small subset of components is often responsible for system failure. Here we summarize essential parts of microreboot design.

1) Fine-grain components:

Component-level reboot time depends largely on how long it takes for the underlying platform to restart a component and reinitialize it. Thus a microrebootable application aims for components that are as small as possible in terms of startup time and program logic. Although the task of partitioning a system into components is specific and difficult, developers can benefit a lot from existing component-oriented programming framework such as J2EE [5].

2) State segregation:

To ensure the recovery correctness, ROC requires that microrebootable applications should keep all important state in dedicated state stores located outside the application, safeguarded behind strong high-level APIs. Besides enabling the safe microreboot, the complete segregation of data recovery from application recovery also improves the system robustness. Because the segregation can shift the burden from application program writers to the experienced specialists who develop state stores.

3) Decoupling:

If applications are to tolerate microrebooting, components should be loosely coupled. The well-defined and well-enforced boundaries are needed. Direct references, such as pointers are not allowed. If cross component references are indeed needed, they must be stored outside the component, either in the application platform or inside a state store.

4) Retryable requests:

To reintegrate microrebooted components smoothly, inter-component interactions use timeout. When one component invokes a currently microrebooting component, it receives a retry after (t) exception, and the call can be re-issued after the estimated time t, if it is idempotent [5]. For non-idempotent calls, rollback could be used. If components transparently recover requests in this way, we can hide intra-system component failures and microreboot from end users.

5) Leases:

To improve the reliability of cleaning up after microreboots, resources in a frequently-microrebooting system should be leased. In addition to memory and file descriptors, CPU execution time should also be leased meaning that if a computing hangs and does not renew its execution lease, it should be terminated with a microreboot. If requests can carry a time-to-live (TTL) which indicates how long the request could be regarded as valid, then stuck requests can be automatically purged once the TTL expired.

B. *Prototype implementation and lessons*

ROC group implements the microreboot-approach in JBoss, an open-source application server written in Java supporting J2EE's component-based programming framework [5]. The refined JBoss modification allows selective microrebooting of small groups of Enterprise Java Beans (EJBs). All applications store their session state in a dedicated state repository optimized for fast recovery. A session state must remain persist between user's login and logout, but is not needed after the session ends. In this prototype, microreboots recover from a large category of failures, including deadlock, memory leaks and corrupted volatile data for which system administrators usually choose to restart the application.

Microreboots are largely as effective as full reboots but 30 times faster [6]. It not only reduces recovery time but also minimizes the side-effects on system end users, as shown in Figure II.

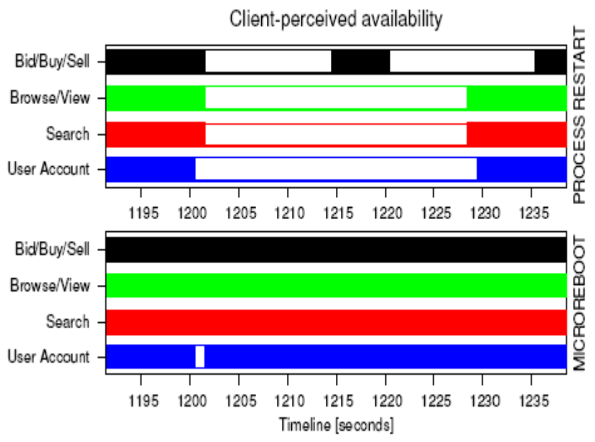

Figure II: Service functionality availability [5].

The graphs illustrate end-user-perceived availability of an online auction service. The white gap of an interval indicates that some requests processing during this period eventually fail, suggesting that site is down. When recovering with microreboot, end-users are almost unaware of the restoring process, with close to no visible global down-time.

If a component failure can not be corrected by microreboot, larger subsets of components are restarted progressively. This is like navigating upward on the spectrum to find the most advantageous cost-benefit ratio as shown in Figure I.

Based on the implementation, two lessons are learned. Firstly, fine-grained recovery requires accurate fault localization. Concerning the workloads faced by Internet services which often consist of many short-term tasks, by localizing recovery to a small subset of components, microreboot minimizes the number of state loss thus yielding a transparency to end-users for recovering. One challenge we face is how to identify the location of faults more accurately. To address it, ROC group built an application-generic fault detection and localization program – Pinpoint (http://pinpoint.standford.edu) – which uses statistical learning techniques to detect and localize application-level faults in component-based Internet services. Although Pinpoint does exhibit false positives, the integration of microreboot and Pinpoint offers higher dependability than ordinary rebooting.

For the second, Microreboot is not a cure-all. Microrebooting works best on software failures triggered by so-called "Heisenbug", which is a computer bug that disappears or alters its characteristics when it is researched. Microreboot is also effective against resources leak and corruption of volatile data structures [6]. Although these fault classes are important and hard to prevent with existing quality assurance process, they do not represent all system failures. Some failure types such as corruption of persistent data and misconfigurations can hardly be fixed by microrebooting.

IV. UNDO AND REDO

Software bug is not the only culprit for bringing down the systems; based on the analysis [4] of three large-scale Internet services, it is found that 1) human error is a major cause of failures for Internet service systems, 2) configuration errors are the largest category of operation errors, and 3) operator error is the largest contributor to mean-time-to-recover (MTTR). The related problems include accidental data deletion, improper component shutdown and incorrectly performed update. To address these types of problems, ROC group adds a second line of defense to the multitier dependability based on the Three-R's undo pattern.

A. *Design of Three-R model:*

The model of Three-R [1] includes three fundamental steps referred as "Rewind, Repair and Replay". In the first Rewind step, all system state from application to OS is physically rolled back to a point before any damage occurred. In the Repair step, the operator in charge alters the rolled-back system to avoid reoccurring of problems. Finally, in the Replay step, the repaired system is rolled forward to the present by selectively replaying portions of the previously-rewound timeline.

The essence of Three-R's Undo is that it preserves the timeline: it restores lost updates and incoming data via replay in a manner that retains their intent but not the bad results of their original processing and this is also the property that distinguishes it from traditional approaches such as backup and restore.

Three design decisions are important in the Three-R's undo model: First one is the choice to perform Rewind physically and Replay logically. In this approach, "undo" is implemented by one single operation of restoring a previous snapshot of system hard state, while "redo" is achieved by re-executing a sequence of recorded user-level operations. As ROC undo system makes no assumption about the possible corruption it might encounter, physical Rewind provides flexibility in the recovering in that the corrupt state can not escape from roll-back. On the other hand, logical replay preserves the intent of user operations without reference to the original corrupted state together with respecting repairing process. While the logical replay may increase the complexity of undo system, ROC group constructs the undo system so that the replay code is implemented as part of normal system operation, thus flushing out bugs before replay during an emergency. Another key decision is that Repair should be as unconstrained as possible to allow the full flexibility for operator in designing solutions to repair the system problems. The last one is the fault model that makes minimal assumptions about the correctness of undoable application. Although it could limit the possibility of formal analysis, the fault model is the key to practical recovery from problems that altered the system operation in unknown ways, due to the fact that the most confounding and error-prone problems are the ones that have never been seen before.

B. *Implementation in an E-mail store*

Comparing with today's productivity applications where the undo design is used so commonly, in the administration and operator environment, it is still virtually unheard of. Trying to change this situation, ROC group first implements the undo method on an E-mail store system. The general architecture of undo system is shown in Figure III.

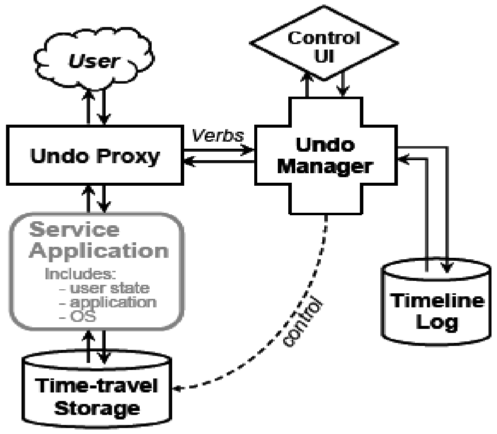

Figure III. Undo system architecture [2].

The heart of the undo system is the undo manager, which coordinates the system timeline. The proxy and time-travel storage layer wrap the service application, capturing and replaying user requests from above and providing physical rewind from below.

The service application and its hosting operating system are left virtually unmodified; the undo system interposes itself both above and below the service. By keeping the undo system isolated from the service, this wrapper-based approach supports the fault model. Below the operating system, a time-

Verb	Protocol	Changes?	Externalizes state?	Async?	Description
Deliver	SMTP	√		√	Delivers a message to the mail store via SMTP
Append	IMAP	√			Appends a message to a specific IMAP folder
Fetch	IMAP	√	√		Retrieves headers, messages, or flags from a folder
Store	IMAP	√	√		Sets flags on a message (e.g., Seen, Deleted)
Copy	IMAP	√			Copies message to another IMAP folder
List	IMAP		√		Lists extant IMAP folders
Status	IMAP				Reports folder status (e.g., message count)
Select	IMAP				Opens an IMAP folder for use by later commands
Expung	IMAP	√	√		Purges all messages with Deleted flag set from a folder
Close	IMAP	√			Performs a silent expunge then deselects the folder
Create	IMAP	√			Creates a new IMAP folder or hierarchy
Rename	IMAP	√			Renames an IMAP folder or hierarchy
Delete	IMAP	√			Deletes an IMAP folder or hierarchy

Table II. Verbs defined for undo e-mail store [2].

travel storage layer provides the ability to physically roll the system's hard state back to a desired point. Above the service application is a proxy which interposes between the application and end users. The undo system can intercept the incoming user request stream to record the system timeline and can inject its own requests to affect replay. The proxy and time-travel storage layer are coordinated by the undo manager, which maintains a history of user interactions comprising the system timeline.

Because the undo manager in the system has no knowledge of the service and corresponding semantics, to address the translation problem when application-specific proxy communicates with undo manager, ROC group proposes the concept "verb", which becomes the fundamental construct to represent events in the system timeline. A verb is an encapsulation of an end-user interaction with the system – a record of event causing service state to be changed or externalized (exposed to an external observer). It contains all the application-specific information needed to execute or re-execute user interaction, and at the same time appears to the undo manager as a generic data type with interfaces that only exposes information to manage the recording and execution. Rather than recording the contents of state or the effects of interactions on state, verb records the intent of user interactions at the protocol level. The flow of verbs during normal operation and during Replay is illustrated in Figure IV.

During normal operation, the verb flow follows the solid black arrows, with verbs created in the proxy and looped through the undo manager for scheduling and logging. During replay, verb flow follows the heavy dashed arrow, with verbs being reconstructed from the timeline log and re-executed via the proxy.

With the undo system architecture described, we now turn to the implementation of e-mail store service, which represents a leaf node in global e-mail network, delivering e-mail via SMTP and making it available for reading via IMAP [2].

1) Verbs for E-mail

ROC defines 13 verbs for their undoable e-mail system that together capture important interactions in the IMAP and SMTP protocols, as listed in Table II. Each e-mail verb is implemented as a Java class realizing a common verb interface; the verb

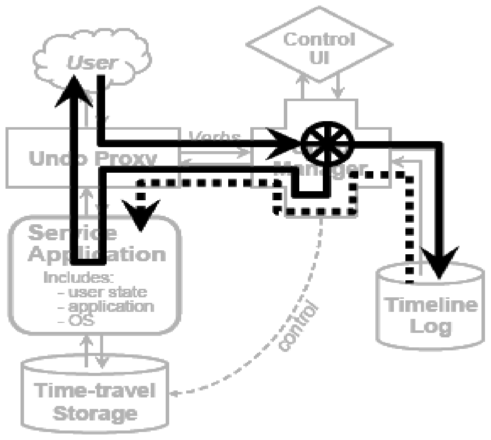

Figure IV. Illustration of verb flow [2].

interface is defined by the undo manager and it declares an API that maps the routines into Java function declarations. All verbs contain a tag which is a container structure wrapping the information needed to execute the verb and to check external consistency. The tag also includes a record of whether the execution succeeded or failed.

To define verbs for existing protocols like IMAP and SMTP, it is important to capture the necessary context needed by the verb replay. For SMTP, the system captures the parameters passed to each SMTP command and stores them in the verb's tag. As to IMAP, in order to be able of replaying IMAP verbs in situations where repairs have changed the system context, ROC group defines the notion of an UndoID, which is a time-invariant name independent of system context and capable of being translated to IMAP name for verb execution. The proxy is responsible for converting UndoID with IMAP names based on current context.

2) E-mail proxy

The e-mail proxy in the system is responsible for intercepting all SMTP and IMAP traffic directed at the server, converting it into verbs and interacting with the undo manager. It accepts connections on SMTP and IMAP ports and dispatches threads to handle each incoming connection. Each connection is handled by a thread running in a loop which decodes each incoming SMTP or IMAP, packages it into a verb and invokes the undo manager to sequence, execute and record the verb.

3) Time-travel storage layer

At the base of ROC undoable e-mail system is the time travel storage layer, which provides stable storage for the e-mail store's hard state and holds the ability to physically restore previous versions of target state. The storage layer design aims at application-neutral, and has neither the knowledge of the e-mail store, nor any customization to e-mail semantics

4) Undo manager

The undo manager stores system timeline as a linear append-only verb log. The log is implemented as a BerkeleyDB database, with each verb assigned a log sequence number (LSN) which is the fundamental internal representation of time to the undo manager [2]. The undo manger mediates execution of verbs during normal operation.

The detailed information related to undoable e-mail store is discussed in Brown *et al* [2].

V. Suitable Dependability Benchmarks

A dependability benchmark is made up of a system specification, a faultload, a workload and a metric [3]. Due to the fact that no standard existed for benchmarking the dependability of Internet service which is of the main concern, ROC develops one through experiment and discussion with industrial experts.

Action-weighted-throughput [5] is adopted to evaluate availability, which accounts for user interaction with a Web-based services as well as different weights of various operations. ROC assumes that a user session begins with login operation and ends up with logout or abandonment of the site. Each session consists of a sequence of actions; each user action is a sequence of operations; each operation in an action must succeed if the corresponding user action can be considered successful. When an operation fails, the entire action fails.

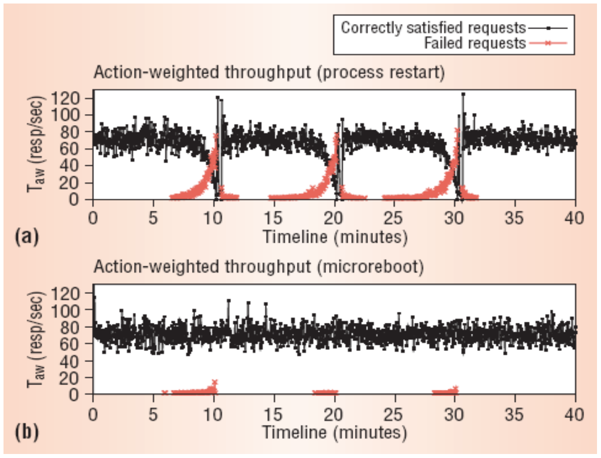

Figure V. Action-weighted-throughput measurement [6].

Figure V shows results of an evaluation for an online auction site using action-weighted-throughput. Individual operations are normal HTTP requests and user actions take the form, "Put $50 bid on X." ROC injected a sequence of three faults such wrong data into the system for every 10 minute. As illustrated by comparing two graphs, microbooting keeps the number of successful served requests up and failed ones down. Overall, 11,752 requests failed when using restart; at the same time, 233 requests failed when recovering with microreboot.

Action-weighted-throughput takes into account end users, but it still does not provide quantitative metric for system administrators. ROC develops a new form of human-aware dependability benchmark [2] to correlate the cause and observed effects and applied it to measure the effectiveness of the system-level undo prototype. ROC benchmarks the correctness and availability of the e-mail server with and without the undo recovery mechanism, under two state corrupted failure scenarios, using 12 student subjects to perform recovery in 7 case scenarios. As demonstrated in Figure VI, the number of incorrectly handled messages greatly decreased in each case with undo/redo. For each of seven scenarios, the graph plots results of correctness and availability with and without undo recovery tool

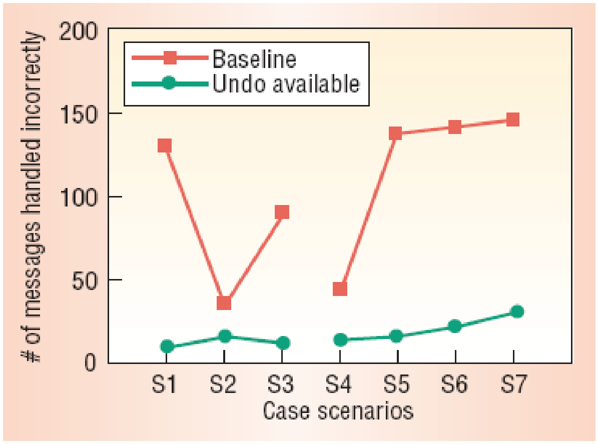

Figure VI. Benchmarking the human administrator component [6].

Besides the implementations of microreboot and system-level undo, ROC group also tries to improve recovery predictability and generates a new direction of research to improve the predictability of system behavior as a whole with reference at (http://predictable.standford.edu/). To broaden the undo/redo application space, ROC uses "spheres of undo" structuring concept to look at providing system-level undo in more complex systems such as Window-based desktop systems and distributed services [6]. With the exception of human errors, ROC finds out a fact that the noticing of an error often takes longer than to diagnose and repair it, especially when repair is a low-cost microreboot. Therefore, the research group regards the fault detection and diagnosis as an open challenge, which could be a promising direction for continuing the research to build high dependable systems.

VI. Conclusion

To achieve the goal of multitier dependability, two techniques were designed and illustrated in this paper, providing a front-line to back-up-line defense in case of failure. Microreboot, serving as the first-line-of-defense, is driven primarily by the desire to decrease the mean time to recover (MTTR) as a way to improve availability. Accepting software bugs as fact, this cheap reboot-based recovery method provides us a potential path toward dependable large-scale software. To extend the recovery coverage to more complex failure types, system-level undo was proposed. This approach creates a forgiving environment for system operators to help address the challenges facing the human operators, who exert a crucial influence on dependability. Through the adaptation of specifically designed benchmarking methods which take into account human operators as well as end users, we evaluated the prototype implementations of the two techniques at the final stage. The results showed that the combination of microreboot and system-level undo yielded promising solutions to complex problems within the large-scale Internet service environment.

APPENDIX

Source code for ROC undo framework and e-mail proxy is available at http://roc.cd.berkeley.edu/undo/.

ACKNOWLEDGMENT

Special thanks go to my tutor Ms. Tiina Niklander for being available when needed with wholehearted supports, to Janne Metso for pointing out the inconsistency in the draft, and to Timi and Sabine for reviewing and precious feedback.

REFERENCES

[1] A.B. Brown and D. A. Patterson, “Rewind, Repair, Replay: Three R's to Dependability,” 10th ACM SIGOPS European Workshop, Saint-Emilion, France, September 2002, pp. 70-77.
[2] A.B. Brown and D.A. Patterson, “Undo for Operators: Building an Undoable E-mail Store,” Proc. Usenix Ann. Tech. Conf., Usenix Assoc., 2003, pp. 1-14.
[3] D. A. Patterson, A. Brown, et al., “Recovery-Oriented Computing (ROC): Motivation, Definition, Techniques, and Case Studies,” UC Berkeley TR UCB//CSD-02-1175. Berkeley, CA, March 2002.
[4] D. Oppenheimer, A. Ganapathi, and D. A. Patterson, “Why do Internet services fail, and what can be done about it?” Proc. 4th USENIX Symp. on Internet Technologies and Systems. March, 2003, pp. 1-16.
[5] G. Candea et al., “Microreboot – A Technique for Cheap Recovery,” Proc. 6th Symp. Operating Systems Design and Implementation (OSDI), Usenix Assoc., 2004, pp. 31-44.
[6] George Candea, Aaron B. Brown, Armando Fox, and David Patterson, “Recovery-Oriented Computing: Building Multi-Tier Dependability,” IEEE Computer, Volume 37, Number 11, November 2004, pp 64-67.
[7] Oppenheimer, D. and D. A. Patterson, “Architecture, operation, and dependability of large-scale Internet services: three case studies,” IEEE Internet Computing special issue on Global Deployment of Data Centers, September/October 2002, pp 41-49.