Appears in Proceedings of the International Conference on Dependable Systems and Networks (DSN-2002), June 2002

Reducing Recovery Time in a Small Recursively Restartable System

George Candea, James Cutler, Armando Fox, Rushabh Doshi, Priyank Garg, Rakesh Gowda
Stanford University
{candea,jwc,fox,radoshi,priyank,rgowda}@cs.stanford.edu

Abstract

We present ideas on how to structure software systems for high availability by considering MTTR/MTTF characteristics of components in addition to the traditional criteria, such as functionality or state sharing. Recursive restartability (RR), a recently proposed technique for achieving high availability, exploits partial restarts at various levels within complex software infrastructures to recover from transient failures and rejuvenate software components. Here we refine the original proposal and apply the RR philosophy to Mercury, a COTS-based satellite ground station that has been in operation for over 2 years. We develop three techniques for transforming component group boundaries such that time-to-recover is reduced, hence increasing system availability. We also further RR by defining the notions of an oracle, restart group and restart policy, while showing how to reason about system properties in terms of restart groups. From our experience with applying RR to Mercury, we draw design guidelines and lessons for the systematic application of recursive restartability to other software systems amenable to RR.

1. Introduction

The Software Infrastructures Group (SWIG) and Space Systems Development Lab (SSDL) at Stanford are collaborating on the design and deployment of space communications infrastructure to make collection of satellite-gathered science data less expensive and more reliable. One necessary element of satellite operations is a ground station, a fixed installation that includes tracking antennas, radio communication equipment, orbit prediction calculators, and other control software. When a satellite appears in the patch of sky whose angle is subtended by the antenna, the ground station collects telemetry and data from the satellite. In keeping with the strong movement in the aerospace research community to design ground stations around COTS (commercial off-the-shelf) technology [11], part of the collaboration between SSDL and SWIG includes the design and deployment of Mercury, a prototype ground station that integrates COTS components.

A current goal in the design and deployment of Mercury is to improve ground station availability, as it was not originally designed with high availability in mind. Our first step in improving the availability of Mercury was to apply recursive restartability [4], an approach to system recovery that advocates "curing" transient failures by restarting suitably chosen subsystems, such that overall mean-time-to-recover (MTTR) is minimized. Recursive restartability is a concrete example of the recovery-oriented computing (ROC) philosophy [12], as applied to COTS-based systems. To our knowledge, this paper describes the first deployed system to be systematically retrofitted to exploit recursive restartability (RR).

We had two main goals in applying RR to Mercury. The first was to partially remove the human from the loop in ground station control by automating recovery from common transient failures we had observed and knew to be curable through full or partial restarts. In particular, although all such failures are curable through a brute force restart of the entire system, we sought a strategy with lower MTTR than full system reboots. The second goal was to identify design guidelines and lessons for the systematic future application of RR to other systems. For example, we found that, if one adopts a transient-recovery strategy based on partial restarts, redrawing the boundaries of software components based on their mean-times-to-failure (MTTFs) and mean-times-to-recover (MTTRs) can minimize overall system MTTR by enabling the tuning of which components are restarted together. In contrast, most current system and software engineering approaches establish software component boundaries based solely on considerations such as amount and overhead of communication between components or amount and granularity of state sharing.

The paper is organized as follows: we provide necessary background on the architecture of our ground station and its failure detection mechanisms in section 2, followed by an overview of recursive restartability concepts in sec-

tion 3. In section 4 we describe a set of transformations by which we improve the mean-time-to-recover of the ground station, hence improving its availability. In section 5 we extract lessons from our experience. We address related work in section 6, propose future avenues of research in section 7, and conclude in section 8.

2. Mercury Overview

In this section we describe Mercury, our ground station prototype. Although the focus of this paper is the effectiveness of recursive restartability as a recovery strategy, recovery is only possible once failures are detected. As the original Mercury design had little in the way of failure detection, we describe the simple failure detection mechanisms we added to enable recursive restartability.

2.1. Ground Station Architecture

The Mercury ground station communicates with low earth orbit satellites at data speeds up to 38.4 kbps. For the past two years, the Mercury system has been used in 10-20 satellite passes per week as a primary communication station for Stanford's satellites Opal [6] and Sapphire [14].

The station, composed primarily of COTS hardware and software written mostly in Java, is controlled both remotely and locally via a high-level, XML-based command language. Software components are independently operating processes with autonomous loci of control and interoperate through passing of messages composed in our XML command language. Messages are exchanged over a TCP/IP-based software messaging bus.

The general software architecture is shown in Figure 1. fedrcom is a bidirectional proxy between XML command messages and low-level radio commands; ses (satellite estimator) calculates satellite position, radio frequencies, and antenna pointing angles; str (satellite tracker) points antennas to track a satellite during a pass; rtu (radio tuner) tunes the radios during a satellite pass; mbus passes XML-based high-level command messages between software components. REC and FD will be described in the next section.

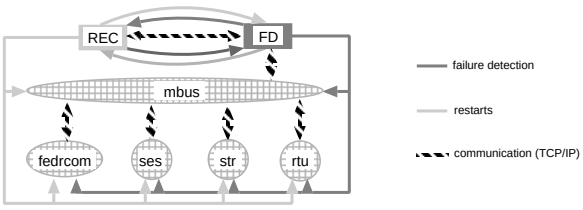


Figure 1. Mercury software architecture

The ground station components are easily restartable, since most are stateless; they use only the state explicitly encapsulated by received messages from mbus. Hard state exists, but is read-only during a satellite pass and is modified off-line by ground station users. In addition, the set of Mercury failures that can be successfully cured by restart is large, and in fact this is how human operators recovered from most Mercury failures before we implemented automated recovery.

2.2. Adding Failure Detection to Mercury

Adding failure detection to this architecture was motivated by the need to automate detection of several common failure modes; we understood these modes from extensive past experience with Mercury. All the failures we focused on were fail-silent: when components failed, they simply stopped responding to messages (e.g., when the JVM containing a component crashed). Moreover, all failures were curable through restart of either a single software component or a group of such components.

Given the fail-silent property, we chose application-level liveness pings (i.e., "are you alive?" messages) sent to a component via the software message bus, mbus. The pings are encoded in and replied to in a high-level XML command language, so a successful response indicates the component's liveness with higher confidence than a network-level ICMP ping. Application-level liveness pings are simple and low-cost, and effectively detect all fail-silent failures that humans were detecting before in the ground station, thus satisfying the immediate goal of automated failure detection.

Figure 1 illustrates Mercury's simple failure detection architecture, based on the addition of two new independent processes: the failure detector (FD) and the recovery module (REC). FD continuously performs liveness pings on Mercury components, with a period of 1 second, determined from operational experience to minimize detection time without overloading mbus. When FD detects a failure, it tells REC which component(s) appear to have failed, and continues its failure detection. For improved isolation, FD and REC communicate over a separate dedicated TCP connection, not over mbus; mbus itself is monitored as well. REC uses a restart tree data structure and a simple policy to choose which module(s) to restart upon being notified of a failure. The policy also keeps track of past restarts to prevent infinite restarts of "hard" failures. Once REC restarts the chosen modules, future application-level pings issued from FD should indicate the failed components are alive and functioning again. However, if the restart does not cure the failure, FD will redetect it and notify REC, which may choose to restart a different module this time, and so on.

Given the above strategy, two situations can arise, which we handle with special case code. First, FD may fail, so we wrote REC to issue liveness pings to FD and detect its fail-

ure, after which it can initiate FD recovery. Second, REC may go down, in which case FD detects the failure and initiates REC's recovery, although the generalized procedural knowledge for how to choose the modules to restart and initiate recovery is only in REC.

Splitting FD and REC requires the above two cases to be handled separately, but it results in a separation of concerns between the modules and eliminates a potential single point of failure. Our enhanced ground station can tolerate any single and most multiple software failures, with the exception of FD and REC failing together.

It is important to note that, in our system, restarts are a recovery mechanism based on detecting failures, not faults. Response to a failure is independent of the fault that caused the failure. Restarting can be used in addition to other recovery strategies, not necessarily in place of them, so we do not believe that anything we have done precludes the use of more sophisticated failure detection or high availability mechanisms (such as redundancy) in the future.

3. Recursive Restartability

It is very common for bugs to make software systems crash, deadlock, spin, livelock, or develop such severe soft state corruption—memory leaks, dangling pointers, damaged heaps—that the only high-confidence way of continuing is to restart the application(s) or reboot the system [3, 7, 13, 1]. Recursive restartability is predicated on our belief that this state of affairs will remain a fact of life, both due to the increasing complexity of software and the increasing cost of chasing and resolving elusive bugs. While definitely not an encouragement to develop poor quality software, recursive restartability (RR) provides a way to deal with some of the drawbacks of using inexpensive COTS software, particularly after deployment.

Restarts provide an effective and immediate workaround for transient failures, as they (a) unequivocally return software to its start state, which is usually the best understood and tested state of the system, (b) provide a high confidence way to reclaim resources that are stale or leaked, and (c) are easy to understand and employ. Unfortunately, most systems do not tolerate restarts well: restarting is often very expensive in terms of time-to-recover, may cause loss of hard state, and bounded restarting of only those subsystems that are faulty is usually not supported.

A *recursively restartable system* gracefully tolerates successive restarts at multiple levels. Due to its fine restart granularity, an RR system enables bounded, partial restarts that recover a failed system faster than a full reboot. Availability is generally thought of as the ratio MTTF/(MTTF +MTTR); recursive restartability improves this ratio by reducing MTTR with reactive restarts of failed subsystems, and by increasing MTTF with a bounded form of software rejuvenation [9]. The focus in this paper is on reducing MTTR, both because that is the emphasis of recovery-oriented computing and because it is easier in industrial and research practice to measure MTTR than MTTF.

3.1. Restart Trees

A recursively restartable system can be described by a *restart tree*—a hierarchy of restartable components, in which nodes are highly fault-isolated and a restart at a node will restart the entire corresponding subtree. A restart tree does not directly capture functional or state dependencies among components, but rather the "restart dependencies," expressing how each component is affected by the restart of other components around it. It must be recognized, however, that the very definition of components is tightly related to functional and state dependencies among the parts of the system, so the restart tree does embody such considerations, albeit indirectly. In Figure 2 we show a simple restart tree with 5 nodes, called *restart cells*. A restart cell is the unit of recovery in a recursively restartable system. Each cell R_A, R_B, R_C, R_{BC}, R_{ABC} conceptually has a "button" that can be "pushed" to cause the restart of the entire subtree rooted at that node.

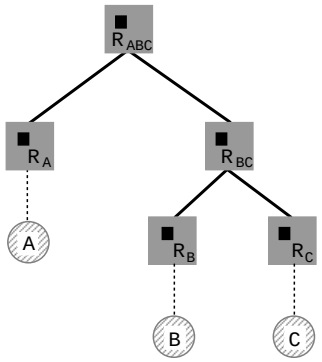


Figure 2. A restart tree

We attach to the leaves of the restart tree annotations as to which actual software components (round nodes) are restarted. In this example, nodes A, B, and C are software components; when we push the button on R_{BC}, both B and C are restarted; when we push the button on R_B, only B is restarted.

3.2. Restart Groups, MTTF, and MTTR

Subtrees in the restart tree are called *restart groups*, in close analogy with process groups in UNIX. The fact that a number of nodes are together in a restart group captures the fact that there is commonality among them with respect to restarting. The tree in Figure 2 contains 5 restart groups:

three trivial ones (R_A, R_B, R_C) and two non-trivial ones (R_B, R_C, R_{BC} form a group rooted at R_{BC}, and R_A, R_{BC}, R_B, R_C, R_{ABC} form another restart group rooted at R_{ABC}). When we say that we restart a group, that means that all components attached to leaves of the respective subtrees will be restarted; in the case of the group rooted at R_{BC}, these components would be B and C. The system as a whole is always a restart group. As will be shown in section 4, we can recursively reason about the availability properties of the system in terms of restart groups.

For consistency with industrial practice, we will refer to the MTTF and MTTR of individual components, restart groups, and the system as a whole. The time-to-recover a component includes the time it takes to detect that it failed—downtime starts when the failure occurs, not when it is detected.

We must note that MTTF and MTTR constitute an impoverished representation of the failure behavior of a system or subsystem. Without knowing more about the distribution of failure or recovery times, one cannot predict the probability that a subsystem will fail during a particular time interval, which may be important since not all downtime is equally expensive. The techniques we will describe in this paper for constructing and evolving restartability trees are based on the assumption that MTTF and MTTR represent the means of distributions with small coefficients of variation. We have confirmed through experiment that this is the case with our system, and for compactness we will henceforth use the notations MTTF_S and MTTR_S to refer to the MTTF and MTTR, respectively, of subsystem S. In particular, we assert that the MTTF for a restart group G containing components $c_0, c_1, \ldots, c_n$ is $\text{MTTF}_G \leq \min(\text{MTTF}_{c_i})$, and that the corresponding MTTR is $\text{MTTR}_G \geq \max(\text{MTTR}_{c_i})$.

3.3. The Recoverer and the Oracle

The restart tree plays a central role in keeping a recursively restartable system alive, in conjunction with a *recoverer*, which performs the actual restarts. A recoverer does not make any decisions as to which component needs to be restarted—that is captured in the *oracle*, which represents the restart policy. Based on information about which component has failed, the oracle tells the recoverer which node in the tree to restart. If a restart at that point fixes the problem, then the system continues operation normally. However, if the failure still manifests (or another failure appears) even after the restart completes, the oracle moves up the tree and requests the restart of the node's parent. This process can be repeated up to the very top, when the entire system is restarted. In our ground station, we have collocated the recoverer and the oracle in the REC component, shown in Figure 1.

We say a failure is n-curable if it is cured by a restart at node n or any of its ancestors in the restart tree. A *minimally n-curable* failure is a failure that is n-curable and n is the *lowest* node in the tree for which a restart will cure the failure. Admitting that mean-time-to-repair is non-decreasing as we move up the tree, a minimal cure implies the failure is resolved with minimal downtime. For a given failure, it is possible for n to not be unique (e.g., if restarting the parent of n is no more expensive than restarting n itself). A perfect oracle is expected to embody the *minimal restart policy*, i.e., for every minimally n-curable failure, it recommends a restart of node n. In section 4.4 we illustrate what happens when the oracle is imperfect.

4. Evolving the Restart Tree

Having introduced the concept of a restart tree, we show on the left side of Figure 3 a simple restart tree for Mercury (tree I), consisting of a single restart group. The only possible policy with this tree is to reboot all of Mercury when something goes wrong. The system MTTF is at least as bad as the lowest MTTF of any component, and its MTTR at least as bad as the highest MTTR of any component. Table 1 shows rough estimates of component failure rates, made by the administrators who have operated the ground station for the past two years. The components that interact with hardware are particularly prone to failure, because they do not fully handle the wide variety of corner cases.

Component	mbus	fedrcom	ses	str	rtu
MTTF	1 month	10 min	5 hr	5 hr	5 hr

Table 1. Observed per-component MTTFs

In Mercury, each software component is failure-isolated in its own Java virtual machine (JVM) process, a failure in any component does not necessarily result in failures in others, and a restart of one component does not necessarily entail the restart of others. This suggests the opportunity to exploit partial restarts, as discussed in section 3, to lower MTTR. Set against this opportunity is the reality that some failures do propagate across JVM boundaries. Moreover, restarting some components can cause other components to need a restart as well. Both result in observed correlated failures. In the former case, a state dependency leads to a restart dependency; in the latter case, a functional dependency leads to a restart dependency. In the rest of this section we describe how to modify the trivial restart tree to reduce the MTTR of the overall system, illustrating which tree modifications are most effective under specific conditions of correlated failures.

We describe three techniques: *depth augmentation*, that results in the addition of new nodes to the tree, and *group consolidation* and *node promotion*, that result in the removal

of nodes from the tree. Since the focus of the present work is investigation of a recovery strategy designed for transient failures, we make the following simplifying assumption, that does hold for our system:

$\mathcal{A}_{\text{cure}}$: *All failures that occur are detectable by* FD *and curable through restart.*

This assumption is consistent with the fail-silent and restart properties of our system's components. It remains to future work to detect non-fail-silent failures, and to diagnose failures as restart-curable or not, either proactively, or as the result of being unable to resolve through restart.

Another assumption, $\mathcal{A}_{\text{entire}}$, arises when there is no functional redundancy in the system; it would not necessarily apply, for example, to a cluster-based Internet server with hot standby nodes or similar functional redundancy:

$\mathcal{A}_{\text{entire}}$: *A failure in any component will result in temporary unavailability of the entire system.*

4.1. Simple Depth Augmentation

A failure in any component of tree I will result in a maximum-duration recovery. For example, rtu takes less than 6 seconds to restart, whereas fedrcom takes over 21 seconds. Whenever rtu fails, we would need to restart the entire system and wait for all components, including fedrcom, to come back up, hence incurring four times longer downtime than necessary. In this argument we implicitly assume that components can restart concurrently, without significantly affecting each other's time-to-restart.

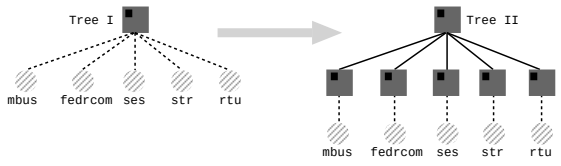


Figure 3. Simple depth augmentation gives tree II.

The "total reboot" shortcoming can be fixed by modifying the tree to allow for partial restarts, which can cure subsystems containing one or more components without bringing the entire system down. Figure 3 illustrates this transformation.

To measure the effect this transformation has on system recovery time, we cause the failure of each component (using a SIGKILL signal) and measure how long the system takes to recover. We log the time when the signal is sent; once the component determines it is functionally ready, it logs a timestamped message. The difference between these two times is what we consider to be the recovery time. Table 2 shows the results of 100 experiments for each failed component.

In the new restart tree II, each restart group (except the root) contains exactly one component. Because of $\mathcal{A}_{\text{entire}}$, the system's MTTF has not changed under the new tree, but its MTTR is lower, because a failure in a component can potentially be cured by restarting a subset of the components, possibly only the failed component.

Specifically, for a restart group G,

$$\text{MTTR}_G^{\text{II}} \leq \sum f_{c_i} \text{MTTR}_{c_i}$$

where c_i is G's i-th child, and f_{c_i} represents the probability that a manifested failure in G is minimally c_i-curable. As mentioned earlier, all observed failures in our ground station prototype were restart-curable, so the sum of f_{c_i} in any G is 1. As long as our system contains some component c_k such that $f_{c_k} > 0$ and $\text{MTTR}_{c_k} \neq \max(\text{MTTR}_{c_i})$, the result will be that $\text{MTTR}^{\text{II}} < \text{MTTR}^{\text{I}}$, since $\text{MTTR}^{\text{I}} = \max(\text{MTTR}_{c_i})$. Note in Table 2 that $\max(\text{MTTR}_{c_i})$ is different in the two trees. A whole system restart causes contention for resources that is not present when restarting just one component; this contention slows all components down.

Failed node	mbus	ses	str	rtu	fedrcom
MTTR^{I}	24.75	24.75	24.75	24.75	24.75
MTTR^{II}	5.73	9.50	9.76	5.59	20.93

Table 2. Tree II recovery: time to detect failed component plus time to recover system (in seconds).

Given that restart tree II now has more than one restart group, we must assume that the oracle is perfect, as described in section 3 (in section 4.4 we will relax $\mathcal{A}_{\text{oracle}}$):

$\mathcal{A}_{\text{oracle}}$: *The system's oracle always recommends the minimal restart policy.*

Another assumption we have made in this transformation is that the restart groups are independently restartable:

$\mathcal{A}_{\text{independent}}$: *Restarting a group will not induce failure(s) in any component of another restart group.*

This assumption is important for recursive restartability, as it captures the requirement of strong fault-isolation boundaries around groups. In section 4.3 we describe how to transform the restart tree so that it preserves this property even when the design of our components impose the relaxation of $\mathcal{A}_{\text{independent}}$.

4.2. Augmenting Depth of Tight Subtrees

An interesting observation is that components may be decomposable into sub-components that have highly disparate MTTR and MTTF. In our system, the fedrcom component connects to the serial port at startup and negotiates communication parameters with the radio device; thereafter, it translates commands received from the other com-

ponents to radio commands. Due to the hardware negotiation, it takes a long time to restart fedrcom; due to instability in the command translator, it crashes often. Hence fedrcom has high MTTR and low MTTF—a bad combination.

We therefore split fedrcom into the pbcom component, which maps a serial port to a TCP socket, and fedr, the front end driver-radio that connects to pbcom over TCP. pbcom is simple and very stable, but takes a long time to recover (over 21 seconds); fedr is buggy and unstable, but recovers very quickly (under 6 seconds). After restructuring the code and augmenting the restart tree (Figure 4), it becomes possible to restart the two components independently. We show the intermediate tree II', which is identical to tree II, except the fedrcom component is split.

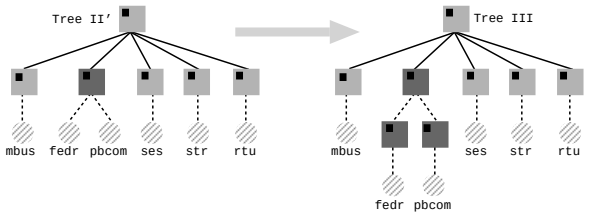


Figure 4. Subtree depth augmentation: tree III.

The new tree III has no effect on the system's MTTF, as the split did not affect the failure characteristics of what used to be fedrcom.

All failures that were previously minimally curable by a restart of fedrcom are now minimally curable by a restart of pbcom, a restart of fedr, or a restart of both. Since $\text{MTTR}_{\text{fedr}} \ll \text{MTTR}_{\text{pbcom}}$ and $\text{MTTF}_{\text{fedr}} \ll \text{MTTF}_{\text{pbcom}}$, most of the failures will be cured by quick fedr restarts and a few of the failures will result in slow pbcom restarts, whereas previously they would have all required slow fedrcom restarts. Therefore, the overall MTTR is improved.

Our measurements confirm this expected improvement: while before it took the system 20.93 seconds to recover from a fedrcom failure, it now takes 5.76 seconds to recover from a fedr failure and 21.24 seconds to recover from the seldom occurring pbcom failure. The increased value of pbcom's recovery time is due to communication overhead.

Some failures that manifest in either of the two new components may only be curable by restarting both, i.e., we have not succeeded in separating fedrcom into completely independent pieces. We observed that multiple fedr failures eventually lead to a pbcom failure. We suspect this is due to the fact that, when fedr fails, its connection to pbcom is severed; due to bugs, pbcom ages every time it loses the connection and, at some point, the aging leads to its total failure. The presence of such correlated failures after splitting a component into pieces is in accord with software engineering reality.

The depth augmentation resulting from insertion of a joint node [fedr, pbcom], as opposed to having fedr and pbcom be top-level nodes under the root, is called for because correlated failures between fedr and pbcom exist, i.e., $f_{\text{fedr,pbcom}} > 0$. Subtree depth augmentation enables us to cure such correlated failures by restarting both components in parallel without restarting the entire tree. If the two components could be made completely independent, then in theory we would have no correlated failures between fedr and pbcom ($f_{\text{fedr,pbcom}} = 0$), and there would be no benefit to the joint node.

We should note that the lower MTTR is achieved only if the oracle makes no mistakes when indicating which node to restart, i.e., $\mathcal{A}_{\text{oracle}}$ holds. Section 4.4 will show why this assumption is necessary to realize the lower MTTR, and will examine the effect of relaxing $\mathcal{A}_{\text{oracle}}$.

From this example we may conclude the following: suppose we have a subsystem containing modules A and B, that any failure in the subsystem is guaranteed to be curable through a partial or complete restart of the subsystem, and that $f_A, f_B, f_{A,B}$ correspond to the probabilities that a failure in the subsystem can be minimally cured by a restart of A only, B only, or [A,B] only, respectively. Then, if $f_{A,B} > 0$, in the engineering sense of being statistically significant, depth augmentation should be used to enable all three kinds of restarts. The same argument holds for the case when $f_A + f_B > 0$.

4.3. Consolidating Dependent Nodes

In the above example, the newly-created fedr and pbcom components, which started out as one, exhibited occasional correlated failures due to bugs in both components. In other cases, components such as ses and str exhibit correlated failures due to functional dependencies. Although ses and str were built independently, they synchronize with each other at startup and, when either is restarted, the other will inevitably have to be restarted as well. When restarted, both ses and str block waiting for the peer component to resynchronize. Such artifacts are not uncommon, especially when using COTS software. In fact, our experience with these components indicated that $f_{\text{ses}} \approx f_{\text{str}} \approx 0$, whereas $f_{\text{ses,str}} \approx 1$. That is, we observed that a failure/restart in one of these components substantially always leads to a subsequent failure/restart in the other.

However, the oracle does not know this ahead of time: under tree III, the oracle will restart ses/str when the component fails, then be told there is another failure, that was induced by the curing action, because of failure to resynchronize with str/ses. It will then restart the peer component. Note that this does not violate $\mathcal{A}_{\text{oracle}}$: if the oracle made a mistake in its restart choice, the original failure would persist. Here, the curing of the failure generates a new, related failure. This violates $\mathcal{A}_{\text{independent}}$.

To fix this, we encode the correlated-failure knowledge into the structure of a new restart tree, as shown in the transformation of Figure 5. It is also possible for the oracle to learn these dependencies over time, but we have not yet implemented this idea. With the new restart tree, whenever a failure occurs in either ses or str, it will force a restart of both, yielding a recovery time proportional to $\max(\text{MTTR}_{\text{ses}}, \text{MTTR}_{\text{str}})$, instead of $\text{MTTR}_{\text{ses}} + \text{MTTR}_{\text{str}}$. This intuition is confirmed by experiment: with tree III it took on average 9.50 and 9.76 seconds to recover from a ses and str failure, respectively; with tree IV the system recovers in 6.25 and 6.11 seconds, respectively.

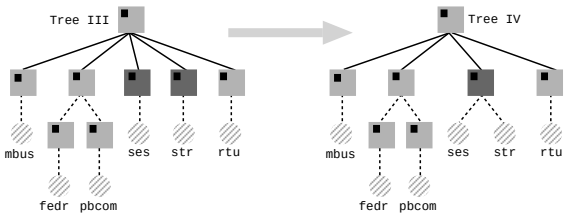


Figure 5. Group consolidation leads to tree IV.

Group consolidation and depth augmentation are duals of each other. We have seen that in a subsystem containing modules A and B, with the probabilities f_A, f_B, $f_{A,B}$, if the ability to restart each component is useful, then the group's depth should be augmented. Similarly, this section has shown that, if the ability to restart each component is not useful (i.e., $f_A + f_B \ll f_{A,B}$), then the restart group should be consolidated.

4.4. Promoting High-MTTR Nodes

There are only two kinds of mistakes an imperfect oracle can make, which we call "guess-too-low" and "guess-too-high". In guess-too-low, the oracle suggests a restart at node n, when in fact a restart at one of n's ancestors is the minimum needed to fix the problem. In this case, the time spent restarting only n will be wasted, because n's ancestor, and hence n as well, will eventually also need to be restarted. In guess-too-high, the oracle suggests a restart at a level higher than minimally necessary to cure the failure. The recovery time is therefore potentially greater than it had to be, since the failure could have been cured by restarting a smaller subsystem, with lower MTTR.

Guessing wrong is particularly bad when the MTTRs of components differ greatly, as is the case for fedr (5.76 sec) and pbcom (21.24 sec). However, we can structure the restart tree to minimize the potential cost incurred from oracle failures: keep low-MTTR components low in the tree, and promote high-MTTR components toward the top, as illustrated with pbcom in Figure 6. As mentioned earlier, there exist failures that manifest in pbcom but can only be cured by a joint restart of fedr and pbcom. We ran an experiment with a perfect oracle, that always correctly guessed when to do a joint restart, as well as with a faulty oracle that guessed wrong 30% of the time (we chose this percentage arbitrarily). The faulty oracle restarts pbcom, then realizes the failure is persisting, and moves up the tree to restart both fedr and pbcom, which eventually cures the failure. Our measurements confirm the impact of node promotion on system recovery time: in tree IV, Mercury took 29.19 seconds to recover from a pbcom failure in the presence of the faulty oracle, in tree V it only takes on average 21.63 seconds to recover with the same faulty oracle.

Intuitively, the reason this structure reduces the cost of oracle mistakes is because mistakenly guessing that a pbcom-only restart was required ultimately leads to pbcom being restarted twice: once on its own, and then together with fedr. Tree V forces the two components to be restarted together on all pbcom failures. Because tree IV is strictly more flexible than tree V, there is nothing that a perfect oracle could do in tree V but not in tree IV. Therefore, tree V can be better only when the oracle is faulty.

Node promotion can be viewed as a special case of one-sided group consolidation, induced by asymmetrically correlated failure behavior. If the correlated behaviors were reasonably symmetric, as was the case for ses and str, then full consolidation would be recommended.

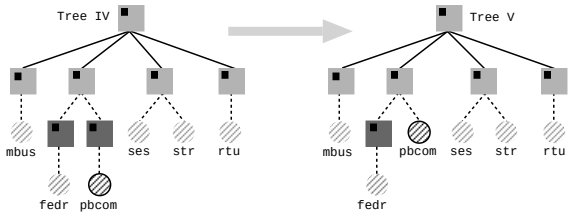


Figure 6. Node promotion yields tree V.

An interesting observation that we have not yet fully explored is the fact that a "free" fedr restart not only accounts for the possibility that the oracle guessed wrong, but also constitutes a prophylactic restart that rejuvenates [9] the fedr component, hence improving its MTTF. Remember that $\text{MTTF}_G^{\text{V}} \leq \min(\text{MTTF}_{c_i})$. Rejuvenation of fedr will likely increase $\text{MTTF}_{\text{fedr}}$, so in the cases in which the next component destined to fail would be fedr, with tree V this would happen later than with tree IV. Therefore, $\text{MTTF}^{\text{V}} \geq \text{MTTF}^{\text{IV}}$.

In this section we have seen how the restart tree was first augmented: we added an entire new level of nodes across the tree, then we added an extra level in one of the subtrees. Then we started reducing the tree, by consolidating nodes within a restart group and by promoting a high-MTTR component up the restart tree. Table 3 on the next page summarizes the tree transformations and reasoning behind them;

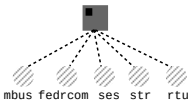
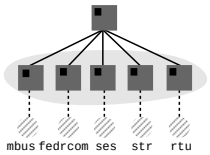
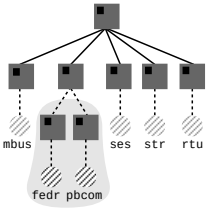
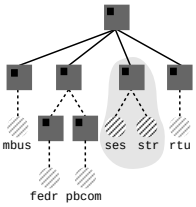
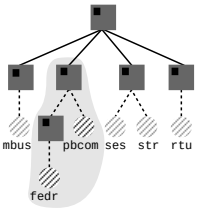
Original Tree	Augmentations		Reductions	
				
Original restart tree. Any component failure triggers a restart of the entire system.	Allows components to be independently restarted, without affecting others.	Saves the high cost of restarting pbcom whenever fedr fails (fedr fails often).	Reduces the delay in restarting component pairs with correlated failures (ses and str).	Encodes information that prevents oracle from making guess-too-low mistakes.
Embodies $\mathcal{A}_{\text{cure}}, \mathcal{A}_{\text{entire}}$	Embodies $\mathcal{A}_{\text{independent}}, \mathcal{A}_{\text{oracle}}, \mathcal{A}_{\text{cure}}, \mathcal{A}_{\text{entire}}$	Embodies $\mathcal{A}_{\text{independent}}, \mathcal{A}_{\text{oracle}}, \mathcal{A}_{\text{cure}}, \mathcal{A}_{\text{entire}}$	Embodies $\mathcal{A}_{\text{oracle}}, \mathcal{A}_{\text{cure}}, \mathcal{A}_{\text{entire}}$	Embodies $\mathcal{A}_{\text{cure}}, \mathcal{A}_{\text{entire}}$
Useful only if all component MTTRs are roughly equal.	Useful when $f_{A,B} > 0$ or $f_A + f_B > 0$	Useful when $f_{A,B} > 0$ or $f_A + f_B > 0$	Useful when $f_A + f_B \ll f_{A,B}$	Useful when oracle is faulty i.e., it can guess wrong.

Table 3. Summary of restart tree transformations

our measurements are centralized in Table 4.

Tree	Oracle	mbus	ses	str	rtu	fedr	pbcom	fedrcom
I	perfect	24.75	24.75	24.75	24.75	—	—	24.75
II	perfect	5.73	9.50	9.76	5.59	—	—	20.93
III	perfect	5.73	9.50	9.76	5.59	5.76	21.24	—
IV	perfect	5.73	6.25	6.11	5.59	5.76	21.24	—
IV	faulty	5.73	6.25	6.11	5.59	5.76	29.19	—
V	faulty	5.73	6.25	6.11	5.59	5.76	21.63	—

Table 4. Overal MTTRs (seconds). Rows show tree versions, columns represent component failures.

5. Discussion

The Mercury ground station is by design loosely coupled, its components are mostly stateless, and failure detection is based on application-level heartbeats. These are important RR-enabling properties and Mercury provides a good example of a simple RR system. The RR techniques described here are applicable to a wider set of applications. For example, we have found that many cluster-based Internet services [3] as well as distributed systems in general are particularly well suited to RR; in fact, many of the RR ideas originated in the Internet world.

In this section, we extract from the Mercury experience some general principles we believe are fundamentally useful in thinking about applying RR to other systems.

5.1. Moving Boundaries

The most interesting principle we found was the benefit of drawing component boundaries based on MTTR and MTTF, rather than based solely on "traditional" modularity considerations such as state sharing. In transforming tree III to tree IV, we placed two independent components, ses and str, into the same restart group. Presumably these two components are independent, yet we are partially "collapsing" the fault-isolation boundary between them by imposing the new constraint that, when either is restarted, the other one is restarted as well.

A dual of the above example is the splitting of fedrcom into the two separate components fedr and pbcom. As described in the text, these two components are intimately coupled from a functionality perspective; it is not an exaggeration to say that either is useless without the other. That is why in the original implementation fedrcom was a single process, i.e., communication between the components that became fedr and pbcom took place by sharing variables in the same address space. Post-splitting, the two components must explicitly communicate via IPC. This clearly adds communication overhead and complexity, but allows the two components to occupy different positions in the restart tree, which in turn lowers MTTR. We conclude that if a component exhibits asymmetric MTTR/MTTF characteristics among its logical sub-components, rearchitecting along the MTTR/MTTF separation lines may often turn out to be the optimal engineering choice. Balancing MTTR/MTTF characteristics in every component is a step toward building a more robust and highly available system.

As explained in [4], RR attempts to exploit strong existing fault isolation boundaries, such as virtual memory, physical node separation, or kernel process control, leading to higher confidence that a sequence of restarts will effectively cure transients. To preserve this property, restart-group boundaries should not subvert the mechanisms that create the existing boundaries in the first place.

5.2. Not All Downtime Is the Same

Unplanned downtime is generally more expensive than planned downtime, and downtime under a heavy or critical workload is more expensive than downtime under a light or non-critical workload. In our system, downtime during satellite passes (typically about 4 per day per satellite, lasting about 15 minutes each) is very expensive because we may lose some science data and telemetry. Additionally, if the failure involves the tracking subsystem and the recovery time is too long, the communication link will break and the entire session will be lost. A large MTTF does not guarantee a failure-free pass, but a short MTTR can provide high assurance that we will not lose the whole pass as a result of a failure.

6. Related Work

The rebooting "technique" embodied in recursive restartability has been around as long as computers themselves, and our work draws heavily upon decades of system administration history. The RR model refines and systematizes a number of known techniques, in an attempt to turn the "high availability folklore" into a well-understood tool. Moreover, recursive restartability is a concrete example of a newly emerging trend in system design, called ROC—recovery-oriented computing [12].

The idea of gracefully terminating an application and restarting it at a clean internal state is called software rejuvenation and was proposed by [9]. Although in this paper we have focused on reactive rather than proactive restarts, rejuvenation is an integral part of the RR strategy. Rejuvenation has also found its way into Internet server installations based on clusters of hundreds of workstation nodes; many such sites use "rolling reboots" to clean out stale state and return nodes to known "clean" states, Inktomi being one example [3]. IBM's xSeries servers also employ rejuvenation for improved availability.

The ability to treat operating system services as separate components, and the ensuing benefits and drawbacks, have long been known to the builders of microkernels [2]. More recent work seeks similar benefits by using lightweight virtual machines [15] for hosting services in third-party Internet infrastructures, thus turning these infrastructures into RR-amenable systems.

The space systems community, and particulary the small-satellite research community, has recently shown tremendous interest in moving to COTS. In fact an entire conference, the Symposium on Reducing the Cost of Spacecraft Ground Systems and Operations, is devoted largely to such issues. This community is recognizing that the opportunities of COTS present challenges of dependability; we believe that our application of recovery-oriented techniques such as RR to space-based systems provides evidence for the feasibility of this approach.

7. Future Work

Restarting cannot recover from a hard failure in a disk drive or other hardware component such as the radio, which is likely to happen eventually. We are in the process of implementing component health summary beacons, which include a digest of internal metrics such as resource usage, data structure consistency, connectivity checks, latency between key code points, warnings of suspect behavior that has not yet caused a failure, and if applicable, information about detectable hard failures. Comprehensive failure detection and logging were not the goals of this effort, though they are long-term goals for Mercury; health summaries constitute one step in that direction.

We have described three types of restart tree transformations: depth augmentation, group consolidation, and node promotion. In Mercury, the choice of transformations and recovery policy was based to a great degree on estimated values of f_{c_i}, i.e., the probability that a manifested failure in the observed group is minimally c_i-curable. These values are generally more difficult to measure than actual failure rates, but are easily determined from experience, after some trial-and-error. In future work we intend to extend the oracle with the ability to learn from its mistakes and this way generate estimates for f_{c_i} values. We also plan to identify specific algorithms for transforming restart trees.

We are applying RR to another test system: iROS, the software infrastructure for the Stanford Interactive Workspace. As described in [10], the functional components of iROS were specifically written to be restartable, and state management in the system as a whole is sensitive to the possibility that components may be restarted at nearly arbitrary times. This design decision was motivated in part by the desire to leverage simple robustness solutions based on mechanisms such as RR. We expect to report on this work shortly.

Interesting work in software rejuvenation focuses on analytic modeling of system uptime to derive optimal rejuvenation policies that maximize availability under a modelled workload [8]. Although we made many simplifying assumptions (all consistent with the behavior of our system) to allow meaningful use of MTTR and MTTF in our argument, we expect to explore a more detailed analytic model in future work.

Applying RR requires that components either be stateless or utilize soft state [4]. For cases where some of the system's components are using hard state, we are developing a general model of *recursively recoverable* systems. With recursive recovery, we can accomodate a wider range of recovery semantics, since each component is recovered

using a custom procedure; restart is just one example of a recovery procedure. An example of where the general model is needed would be complex e-business infrastructures, that combine storage services with databases, application servers, and web servers.

In the process of recovering, the various subsystems actively trade off certain system properties against each other, such as performance or consistency for availability. A new project we have started [5] has identified five basic axes for making these tradeoffs and is developing a utility-function-based model for dynamically optimizing these tradeoffs to maximize system dependability.

8. Conclusions

We applied recursive restartability to a system with which we had extensive "manual" experience. To improve the system's availability, we reduced its time to recover from various types of failures we had observed over nearly 2 years in production use. We achieved automated failure detection and recovery that was better than manual operation, even though the system was not purposely designed to accomodate these goals.

The most significant lesson was that constructing the "optimal" (lowest-MTTR) restart tree required collecting components into restart groups based on their MTTFs/MTTRs and degree of failure correlation, and that this requirement in turn may impose constraints on the way components are architected. This suggests that when module boundaries are drawn at design time, consideration should be given to these properties in addition to the ones traditionally considered functional orthogonality, degree and granularity of state sharing, etc.

By employing recursive restartability we were able to improve recovery time of our ground station by a factor of four. Although we have not thoroughly measured the benefits resulting from automating the failure detection, we have observed them to be significant—in the past, relying on operators to notice failures was adding minutes or hours to the recovery time. There is an increasing trend toward complex, hard-to-manage software systems that integrate large numbers of COTS modules; we believe that recovery-oriented computing approaches hold a lot of promise as a dependability technique in such systems.

9. Acknowledgements

We are indebted to the anonymous reviewers and our colleagues at Stanford University for their insights and helpful comments on our paper. We thank the National Aeronautics and Space Administration (NASA) for supporting our work under award NAG3-2579, and the National Science Foundation (NSF) under Career Award 133966.

References

[1] Patriot missile defense: Software problem led to system failure at Dhahran, Saudi Arabia. Technical Report of the U.S. General Accounting Office, GAO/IMTEC-92-26, GAO, 1992.

[2] M. J. Accetta, R. V. Baron, W. J. Bolosky, D. B. Golub, R. F. Rashid, A. Tevanian, and M. Young. Mach: A new kernel foundation for UNIX development. In *Proceedings of the USENIX Summer Conference*, pages 93–113, 1986.

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

[4] G. Candea and A. Fox. Recursive restartability: Turning the reboot sledgehammer into a scalpel. In *Proceedings of the 8th Workshop on Hot Topics in Operating Systems*, pages 110–115, Elmau, Germany, May 2001.

[5] G. Candea and A. Fox. Making sound tradeoffs in state management. In preparation, 2002.

[6] J. W. Cutler and G. Hutchins. Opal: Smaller, simpler, luckier. In *Proceedings of the AIAA Small Satellite Conference*, Logan, Utah, September 2000.

[7] A. DiGiorgio. The smart ship is not enough. *Naval Institute Proceedings*, 124(6), June 1998.

[8] S. Garg, A. Puliafito, M. Telek, and K. Trivedi. Analysis of software rejuvenation using Markov regenerative stochastic Petri nets. In *Proceedings of the 6th International Symposium on Software Reliability Engineering*, pages 180–187, Toulouse, France, Oct 1995.

[9] Y. Huang, C. M. R. Kintala, N. Kolettis, and N. D. Fulton. Software rejuvenation: Analysis, module and applications. In *International Symposium on Fault-Tolerant Computing*, pages 381–390, Pasadena, CA, 1995.

[10] B. Johanson, A. Fox, P. Hanrahan, and T. Winograd. The event heap: An enabling infrastructure for interactive workspaces. Technical Report CS-2001-02, Stanford Computer Science Department, Stanford, CA, 2001.

[11] J.-J. Miau and R. Holdaway, editors. *Reducing the Cost of Spacecraft Ground Systems and Operations*, volume 3. Kluwer Academic Publishers, 2000.

[12] D. Patterson, A. Brown, P. Broadwell, G. Candea, M. Chen, J. Cutler, P. Enriquez, A. Fox, E. Kiciman, M. Merzbacher, D. Oppenheimer, N. Sastry, W. Tetzlaff, and N. Treuhaft. Recovery oriented computing (ROC): Motivation, definition, techniques, and case studies. Technical Report UCB/CSD-02-1175, UC Berkeley, Berkeley, CA, March 2002.

[13] G. Reeves. What really happened on Mars? RISKS-19.49, Jan. 1998.

[14] M. A. Swartwout and R. J. Twiggs. SAPPHIRE - Stanford's first amateur satellite. In *Proceedings of the 1998 AMSAT-NA Symposium*, Vicksberg, MI, October 1998.

[15] A. Whitaker, M. Shaw, and S. D. Gribble. Denali: Lightweight virtual machines for distributed and networked applications. In *Proceedings of the USENIX Annual Technical Conference*, Monterey, CA, June 2002.