Appears in Proceedings of the 1st Workshop on Evaluating and Architecting System Dependability (EASY), July 2001

Designing for High Availability and Measurability

George Candea Armando Fox
Stanford University
{candea,fox}@cs.stanford.edu

Abstract

We propose a structuring model, called recursive restartability, aimed at controlling the amount of end-to-end *un*availability and improving the measurability of software infrastructures with high availability requirements. Recursive restartability exploits the benefits of restarts at various levels within complex software systems and relies on an execution infrastructure to monitor, cure, and rejuvenate software components. We show how system architects can measure and reason about the availability of their systems, as long as these are recursively restartable.

1 Introduction

Reboots can and should be turned into a reliable tool for achieving high availability in critical software infrastructures. Although reboots are not considered a graceful way to keep a system running, mainly because most systems are not suitably designed, restarting has a number of valuable properties. We propose a structuring model centered around the notion of fixing bugs through reboot. This model systematizes a number of known techniques, in an attempt to turn "HA folklore" into a well-understood tool.

We also contend that, besides high availability, measurability is an important requirement for dependable software infrastructures, and must be an intrinsic part of the design. Just like debugability and performance, measurability cannot be an afterthought.

The paper provides motivation for using restarts as a way to achieve high availability in section 2, and section 3 argues for why measurability should be included as a first-class citizen in the design of software infrastructures for highly available services. Section 4 describes recursive restartability – our proposed structuring model for highly available software systems – and section 5 describes how recursively restartable systems are highly measurable and make availability evaluation tractable. Section 6 describes briefly our plans for future work, and section 7 concludes.

2 Why Restart-Based High Availability?

We believe that reboots are a valuable tool for running infrastructures that provide services with high availability requirements. In this section we will show how the "restart concept," commonly viewed as an evil sledgehammer, can in fact be turned into a fine grain scalpel that improves the availability of suitably structured systems.

One of the largest sources of unavailability in today's software systems are latent and pseudo-nondeterministic bugs [13, 12, 3, 8, 1]. Commonly known as Heisenbugs, they are difficult to reproduce, or depend on the timing of external events. Tripped by a race condition or other unforeseen circumstances, they cause systems to crash, deadlock, spin, livelock, leak memory, create and use dangling pointers, corrupt the heap, etc. System administrators will readily attest that, often the best high-confidence way of bringing such a "wedged" system back to normal is to restart. Even if the source of such bugs can be tracked down, it may be more cost-effective to simply live with them, as long as they occur sufficiently infrequently and rebooting allows the system to work within acceptable parameters. As complexity increases, we can expect Heisenbugs to become even more frequent, making it increasingly more difficult to achieve high availability.

Although reboots are dreaded by most of us, they do have three important properties that make them de-

sirable. First, a restart will *unequivocally return software to its start state*, which is usually the best understood and best tested state of the system. For example, the Patriot missile defense system, used during the Gulf War, had a bug in its control software that could be circumvented only by rebooting every 8 hours [14]. It was due to an error in a mathematical computation, which would accumulate and eventually render the system ineffective.

Second, reboots provide a *high confidence way to reclaim resources* that are stale or leaked. At a major Internet portal, the front end Apache web servers are routinely quiesced, killed and restarted, in order to control known memory leaks that accumulate quickly under heavy load.

Finally, the third advantage is that rebooting is *easy to understand and employ*. Simple mechanisms, in general, benefit from being easy to implement, easy to debug, and easy to automate.

Based on these properties, we argue that it is possible to improve overall system availability for any infrastructure, provided it is suitably structured and we can independently restart components that have gone astray. It is critical that the system structure synergize with a restart-based high availability solution, and we describe such a structure in section 4. It is called *recursive restartability* and, in proposing it, we are inspired by the benefic effects that the ACID transaction model and the object model had on the quality of software. Not only did these concepts greatly simplify the design of complex software systems, but they also provided a clean framework within which to reason about the behavior of such systems.

3 Measurability – A Critical System Property

Reasoning about software infrastructures is becoming increasingly more difficult, due to their scale and complexity. A pervasive example is that of the Internet, which appears to have taken on a life of its own; most of the recent studies treat it as a little-understood system that requires experimental rather than theoretical investigation. The systems that are particularly subjected to such increases in scale are the infrastructures we depend on in our day-to-day lives. From air traffic control to health services, banking, and e-commerce, we interact and count on the availability of various software systems at every step we make.

To counter the increasing incomprehensibility of software, we must give infrastructure operators better tools for designing, acquiring, and tuning highly dependable systems. Such tools require a reliable way of measuring, as well as reasoning about, the availability of their systems.

Availability is usually defined in terms of the mean time between failures (*MTBF*) and the mean time to repair (*MTTR*) as the expression $\frac{\mathrm{MTBF}-\mathrm{MTTR}}{\mathrm{MTBF}}$. Although the expression is simple, measuring *MTBF* is difficult, because it requires that we compress time intervals of years or decades to short time spans for experimental purposes. The difficulty of such experiments is compounded by the need to develop a faultload that is representative of what a deployed system may encounter. Moreover, experiments aimed at measuring the *MTBF* of large scale infrastructures are simply intractable.

Consequently, the availability characteristics of today's production systems are usually evaluated based on the "gut feelings" of more or less skilled people. We contend that, assessing the availability of a system, whether based on gut feelings or using comprehensive tools, can be made a lot easier when the system was designed with that task in mind. We therefore argue for *measurability as a built-in property* of the system, and believe that recursive restartability provides a sufficiently flexible framework for measurability.

Deployed software systems undergo constant change and reconfiguration, but change is the anathema of dependability. Highly available infrastructures must, therefore, provide the right structure and hooks for ongoing availability measurements. One example of such a hook are statistics of how often and for how long each subsystem was unavailable/failed; based on how these subsystems are composed into the larger system, we can reason about the overall availability characteristics, aided by structure, modularity, and abstraction.

Similar mechanisms have long been provided for the sake of performance measurements, both at the level of platforms and applications. Examples range from performance counters and cache miss counters in CPUs, to lock grab/delay counters in database systems, and

access metrics in web servers. Performance measures are even used in some devices to provide fail stop behavior, such as I/O subsystems that attempt to predict imminent failures.

Finally, a system's availability cannot be considered in isolation. End-to-end availability is the result of a tight interplay between software, environment, and administration process. In fact, the Gartner group estimates that 40% of downtime in corporate information infrastructures are due to management process failures, with the remainder of 40% due to application failures, and only 20% due to hardware breakdown [17]. It is for this reason that we provide each recursively restartable system with an execution infrastructure, described in the next section, to improve an administrator's ability to reason about the entire system.

Having motivated the use of restart-based high availability and the inclusion of measurability as a core system property, we now proceed to describe recursive restartability, a structuring model which satisfies both these requirements.

4 Recursively Restartable Structures

The main goal of recursively restartable structures is to control the amount of end-to-end *un*availability. This is achieved by:

- System structure that leads to good fault containment, hence increasing *MTBF*.
- System structure that allows for bounded portions of the system to be restarted, in order to decrease restart time, i.e., *MTTR*.
- Monitoring software that reacts rapidly to subsystem unavailability, to decrease *MTTR*.
- Proactive rejuvenation of software components, to avert failures related to poor resource management, and thus increase *MTBF*.

In this section we will summarize the recursive restartability concept and direct the reader to [2] for details.

To define a recursively restartable (RR) system, we take both a functional and a constructional approach. From a functional point of view, a RR system is one in which collections of components/subsystems can be graciously restarted with little or no advance warning. One possible way to build such a system is to use the following constructional recursive definition: The simplest RR system (*base case*) is a software component that can be safely restarted with little or no advance warning. A general RR system (*recursive case*) is an assembly of recursively restartable subsystems, composed according to the following five guidelines.

It's OK to say "No". The interfaces along which we "glue" components together should provide sufficiently weak guarantees, so they can occasionally restart with no advance warning, yet not cause their callers to hang or crash. This guideline pushes the responsibility of dealing with "No" from the provider of a service to the user of that service. Inspired by the work on distributed data structures [9] and the end-to-end argument [16], this guideline achieves a high level of decoupling and fault containment between subsystems.

Trade precision/consistency for availability. Complementary to the previous one, this guideline is based on the observation that applications can often trade precision or consistency for higher availability [10, 6, 18, 20]. The ability to make such tradeoffs dynamically and automatically during transient failures makes a system much more amenable to partial restarts. Inter-component protocols should allow components to make this type of decisions dynamically; they should provide application-specific consistency/precision measures and a consistency/precision utility function (e.g., "absolute consistency is twice as good as missing the last two updates").

Use soft state with announce/listen. Soft state and announce/listen have long been the favorites of wide area network protocols [21, 4, 5, 15]. Announce/listen makes the default assumption that a component is unavailable unless it says otherwise; soft state can provide information that will carry a system through a transient failure of the state's authoritative data source. Keeping most shared state soft will therefore increase the system's tolerance for restarts. The use of announce/listen with soft state allows restarts and "cold starts" to be treated as one and the same, using the same code path. Moreover, complex recovery code is no longer required, thus reducing the potential for la-

tent bugs and speeding up recovery.

Use fine grain workloads and communication. Glue protocols should enforce fine grain interactions between subsytems, to allow these subsystems to complete outstanding work rapidly. This property helps in reducing the *MTTR*, because a reboot can be done much sooner. A good example of fine grain workload requirements is HTTP: the Web as a whole exhibits the property that individual server processes can be quiesced rapidly, since HTTP connections are typically short-lived, and servers are extremely loosely bound to their clients, given that the protocol itself is stateless. This makes them highly restartable and leads to the simple replication and failover techniques found in large cluster-based Internet services.

Decompose functionality orthogonally. Independent subsystems that do not require an understanding of each other's state machines in order to function are said to be mutually orthogonal. Compositions of orthogonal subsystems exhibit high tolerance to component restarts, allowing the system as a whole to continue functioning, perhaps with reduced utility, in spite of transient failures. Examples of orthogonal mechanisms include deadlock resolution in databases [7], software-based fault isolation [19], the use of branch history tables in CPUs, etc. We advocate that subsystems be centered around an independent locus of control, and interact with other subsystems via events posted using an asynchronous mechanism.

A recursively restartable structure leads to a number of desirable properties. Due to the very loose coupling enforced by the assembly rules, components are able to tolerate temporary unavailability of peer components and can gracefully survive failures that are resolved by restart, hence improving fault tolerance. At the same time, due to the ability to restart only a bounded part of the system, it becomes possible to survive failures with little or no loss in availability. If we picture a RR system as a "restartability tree" of components/subsystems, we can see that parent nodes can provide progressively degraded service as their children become unavailable, and then gracefully return to full service as the children recover.

In addition to these intrinsic properties, we can use an *execution infrastructure* (EI) to further improve availability. The EI is a layer underneath a RR system, that is in charge of monitoring the progress of components/subsystems and restart them whenever necessary. Monitoring is done by using application-specific probes, along with end-to-end checks, such as verifying the response to a well-known query. The execution infrastructure performs two types of restarts: reactive, in order to repair a failed component, and proactive, in order to rejuvenate components that are about to fail [11]. In addition to monitoring, the execution infrastructure can also perform a variety of runtime measurements.

The use of the EI automates administrative tasks and offers considerable flexibility, such as tailoring the rejuvenation regimen in an application-specific way. It enables gradual rejuvenation of the entire system through rolling subsystem rejuvenations, without ever discontinuing the end-to-end service. Finally, combining recursive restartability with the EI provides a way of increasing availability that is complementary to other high availability mechanisms, such as redundancy with failover.

5 Measuring Recursively Restartable Systems

Having briefly described recursive restartability, we will now show some preliminary ideas on how a RR structure can help in assessing a system's availability based on the characteristics of its subsystems. These ideas have not beend validated in practice yet, but provide a starting point for our research agenda.

To evaluate a RR system, one must start from the leaves of the restartability tree suggested in section 4 and proceed toward the root. We first reason in isolation about the components at each level and, based on the protocols that glue them together, we combine the characteristics into a property of the composition. Ongoing, runtime measurement performed by the execution infrastructure closes a feedback loop that corrects the reasoning at each level.

There are a number of factors that enter in the individual evaluation of a leaf component. We must obtain an availability metric, which in its simplest form may be just a measure of how available a component is over its lifetime (e.g., "has 4 nines"). This can be

obtained using various methods, such as fault injection, heuristic evaluations, etc. We also need to capture the restartability characteristic of each component, i.e. how tolerant it is to being restarted, how much advance notice it requires, how long recovery takes, etc.

Once we have an availability and a restartability value for each leaf component, we proceed up the restartability tree and compose their values. Following are some examples of how to reason about the aggregations of components/subsystems based on the composition methods.

The "OK to say No" rule, combined with dynamically trading consistency/precision for availability, helps us backpropagate availability and restartability characteristics of components through the system call graph. Each component A making a call to components B and C is a multiplicative filter, with B's and C's characteristics as inputs, and their product as an output characteristic.

Using the "soft state with announce/listen" rule, we can place upper bounds on the availability of the soft state's authoritative sources, and then compute for how long the subsytems that depend on that soft state can survive without the soft state having been refreshed. This can easily be done based on the timeout associated with the freshness of the soft state [15]. The difference, if any, between the availability of the authoritative source and that of the soft state will translate in unavailability exposed to the dependent components.

We advocated fine grain workloads and communication between components in order to allow for rapid quiescing of subsystems. For a given subsystem, we can place an upper bound on the quiesce time by taking the maximum of the expected completion times of the tasks it serves. This will be the upper bound on how long it takes for the entire subsystem to be quiesced and, hence, readied for restart. Reducing the time to quiesce will reduce the *MTTR*.

Measuring the orthogonality of functionality decomposition and its effects is difficult. However, we should observe that each subsystem represents a finite state machine; two such state machines will be orthogonal if they do not share any states and are completely decoupled. If no blocking calls cross the boundary of the subsystem and no hard state is shared with other parts of the RR system, we can conclude that the two systems are orthogonal and can survive each other's unavailability.

This type of inferences with regard to availability must be subjected to a feedback process that adjusts them based on continuous, run-time data gathering. One example of such a feedback loop is the one driving the rejuvenation schedule: component rejuvenation is initially done based on the availability characteristics of each subsystem. As the history of reactive restarts that cure failures in that component accumulates, we fine tune the availability metric and adjust the rate of rejuvenation up or down, as needed: frequent reactive restarts will prompt more often rejuvenation. A recursively restartable structure allows this to be done easily, because each component can be subjected to an individualized rejuvenation regimen.

6 Future Work

As proponents of recursive restartability, we need to provide software architects and developers with suitable tools for building and evaluating RR systems. The only way to turn recursive restartability into an easy-to-use structuring tool is by describing a simple model and providing the right software support for applying that model.

Two major categories of tools are required. First, we need a tractable way of evaluating how restartable a given software component is. For this we need to develop a representative restartability metric and an accurate way to measure it.

Second, we need tools for determining to what degree the various assembly guidelines are applied in a given software infrastructure. It is necessary to evaluate each rule using its own specific metrics. For example: the metrics described in [20] can provide a basis for evaluating consistency/availability tradeoffs; "hardness" of shared state can be measured using the soft state model from [15].

7 Conclusion

When measuring availability, it's important to measure the system, the environment, and the process of running the system. We try to address these issues by (a) structuring the system appropriately, (b) adding a level

of control to the environment through the execution infrastructure, and (c) automating the administration process.

Building RR systems in a systematic way requires a change in the way we architect systems, and a framework consisting of well-understood design rules. We have made a first attempt at formulating such a framework, while advocating the paradigm of building applications as distributed systems, even when they are not distributed in nature. We showed how, through RR, we can reduce the *MTTR* and increase the *MTBF*, hence increasing availability. By reducing $MTTR$, we decrease the importance of the $MTBF$ in assessing availability.

We set forth a research agenda aimed at validating these ideas and verifying that RR can be an effective supplement to existing high availability mechanisms. We hope recursive restartability will provide a simple way to build highly available, highly measurable software systems.

References

[1] E. Adams. Optimizing preventative service of software products. *IBM J. Res. Dev.*, 28(1):2–14, 1984.

[2] G. Candea and A. Fox. Recursive restartability: Turning the reboot sledgehammer into a scalpel. In *Workshop on Hot Topics in Operating Systems*, Elmau, Germany, 2001.

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

[4] S. Deering, D. Estrin, D. Farinacci, V. Jacobson, C. Liu, L. Wei, P. Sharma, and A. Helmy. Protocol independent multicast (PIM), sparse mode protocol: Specification, March 1996. Internet Draft.

[5] S. Floyd, V. Jacobson, C. Liu, and S. McCanne. A Reliable Multicast Framework for Light-Weight Sessions and Application Level Framing. In *ACM SIGCOMM '95*, pages 342–356, Boston, MA, Aug 1995.

[6] A. Fox and E. A. Brewer. ACID confronts its discontents: Harvest, yield, and scalable tolerant systems. In *Seventh Workshop on Hot Topics In Operating Systems (HotOS-VII)*, Rio Rico, AZ, March 1999.

[7] J. Gray. Notes on data base operating systems. In R. Bayer, R. M. Graham, J. H. Saltzer, and G. Seegmüller, editors, *Operating Systems, An Advanced Course*, volume 60, pages 393–481. Springer, 1978.

[8] J. Gray. Why do computers stop and what can be done about it? In *Proc. Symposium on Reliability in Distributed Software and Database Systems*, pages 3–12, 1986.

[9] S. D. Gribble, E. A. Brewer, J. M. Hellerstein, and D. Culler. Scalable, distributed data structures for internet service construction. In *Symposium on Operating System Design and Implementation*, page ??, San Diego, CA, October 2000.

[10] J. M. Hellerstein, P. J. Haas, and H. J. Wang. Online aggregation. In *ACM–SIGMOD International Conference on Management of Data*, Tucson, AZ, May 1997.

[11] 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, 1995.

[12] D. Milojicic, A. Messer, J. Shau, G. Fu, and A. Munoz. Increasing relevance of memory hardware errors. a case for recoverable programming models. In *ACM SIGOPS European Workshop "Beyond the PC: New Challenges for the Operating System"*, Kolding, Denmark, Sept. 2000.

[13] B. Murphy and N. Davies. System reliability and availability drivers of Tru64 UNIX. In *Proceedings of the 29th International Symposium on Fault-Tolerant Computing*, Madison, WI, February 1999. IEEE Computer Society. Tutorial.

[14] U. G. A. Office. Patriot missile defense: Software problem led to system failure at Dhahran, Saudi Arabia. Technical Report GAO/IMTEC-92-26, 1992.

[15] S. Raman and S. McCanne. A model, analysis, and protocol framework for soft state-based communication. In *Proceedings of the ACM SIGCOMM Conference*, Cambridge, MA, Sept. 1999.

[16] J. Saltzer, D. Reed, and D. Clark. End-to-end arguments in system design. *ACM Transactions on Computer Systems*, 2(4):277–288, Nov. 1984.

[17] D. Scott. Making smart investments to reduce unplanned downtime. Tactical Guidelines Research Note TG-07-4033, Gartner Group, Stamford, CT, 1999.

[18] D. B. Terry, A. J. Demers, K. Petersen, M. J. Spreitzer, M. M. Theimer, and B. B. Welch. Session guarantees for weakly consistent replicated data. In *Proceedings of the International Conference on Parallel and Distributed Information Systems*, pages 140–149, Austin, TX, Sept. 1994.

[19] R. Wahbe, S. Lucco, T. E. Anderson, and S. L. Graham. Efficient Software-Based Fault Isolation. In *Proceedings of the 14th ACM Symposium on Operating Systems Principles (SOSP-14)*, 1993.

[20] H. Yu and A. Vahdat. Design and evaluation of a continuous consistency model for replicated services. In *Proceedings of the Fourth Symposium on Operating Systems Design and Implementation*, page pp. ???, Oct. 2000.

[21] L. Zhang, S. Deering, D. Estrin, S. Shenker, and D. Zappala. RSVP: A New Resource Reservation Protocol. *IEEE Network*, 7(5), Sept. 1993.