Crash-only Components in Self-healing Systems

Toni Ruottu
Department of Computer Science
Helsinki University
Email: toni.ruottu@iki.fi

***Abstract*—Crash-only components don't provide a method for shutting them down. Therefore crash recovery functionality gets tested every time a crash-only component is restarted. Well working crash recovery makes the components well suited for building self-healing systems based on transparent restarting of failing subsystems. Healing by restart is a widely adaptable primitive which is easy to understand.**

I. Introduction

Creating a system that is able to heal from unknown sicknesses, such as software bugs, is a tricky task. When a developer notices a failure in the system, she usually attempts to locate the problem origin and fix the problem where it is. A developer who creates self-healing abilities, leaves the software broken and writes some medicine code to heal the broken system. While developers try to remove the chronic diseases from software before release, it is useful to prepare for occasional flu by shipping some medicine with the system as a safety measure.

Because the failures healed by self-healing code are often unsuspected, it may be hard to design an algorithm to analyze the problem and find out a solution. One solution for the problem is to restart the failing component and hope that it will start up in a working state. Failures that happen always at component startup are noticed immediately when the component is deployed. Therefore, restarting a component will bring it to a working state unless the failure has been stored to persist over the restart.

Simplicity of the approach makes it suitable for many different purposes, including not only purely technical faults, but also failures enforced on purpose by a malicious party. The approach has been used in micro kernel operating systems to heal form driver failures by restarting the driver. [1] Healing from security attacks by enabling an IDS (intrusion detection system) to restart components has also been suggested. [2]

Restarting a components is a simple and robust way for healing an unsuspected failure for components that forget their state on restart, but not all components forget their state on restart. The components may have an algorithm that is ran at shutdown time to store some parts of the components state to persist over restart. Crashing the failing components instead of shutting them down cleanly may prevent the component from storing the failure. Interrupting operation of an unsuspecting component does not always result in a good state. The component may have stored some state during operation or some state may have been stored in other components. Another problem in sudden interruption is that the interrupted component doesn't get a change to free resources it has reserved. Lease based resource allocation is useful in cases where freeing resources may not be possible.

Crashing happens for programming mistakes, if not by design for constructive purposes, so preparing for them is usually a good idea. Components can be made crash-safe by adding a recovery algorithm for fixing breakages at startup. Carefully designed runtime behaviour can simplify the recovery algorithm making recovery faster. Transaction like techniques can be used for transferring the component from a good restart persistent state, to another good state. The recovery algorithm can then simply cancel or complete the operations that were unfinished when the crash happened. Thus the component will not be stuck in the middle of two states and doesn't fail for that purpose.

In most cases there is no point in using a sophisticated shutdown algorithm with crash-safe components. There are some cases where combination of the two techniques makes sense. The recovery algorithm may not work well and it may be designed to be used only as the last straw when everything else fails. However, working recovery is usually desired. In some special situation a shutdown algorithm may make faster recovery possible. But then again, recovery can often be made reasonably fast without a shutdown algorithm.

Along self-healing, restarting components can be used for rejuvenating the software while it is running. Components can be restarted periodically which heals minor problems before the component breaks up completely. Restarting a component can also take place when the system notices some warning signs. For example continuing memory consumption growth could be a warning sign for memory leak. Restarting the leaking component would then prevent it from filling the run time memory entirely.

II. States That Matter

Everything that is being run on a computer has a state. There are many levels of state. Many pieces of software are not design to keep their state over crashing. With some simple applications storing the state would not even make any sense. For example traditional office software may store the state of a document on request or even periodically. Office software seldom stores the state of operation.

Figure 1 shows a "hello world" component. All it ever does is greeting the user and it greets the user until it is crashed. With a component this simple, crashing it and restarting it will make no difference for its operation. The operation will

```
while( true )
    greet( )
```

Fig. 1. Source code for a ”hello world” component.

```
while( true )
    foreach( customer )
        interview( )
```

Fig. 2. Source code for an imaginary interviewer component.

```
while( true )
    load( )
    shoot( )
```

Fig. 3. Source code for an imaginary cannon controller component.

remain equal whether or not we store any of the programs state. So there is no need for sophisticated state handling.

Figure 2 shows an imaginary component for interviewing customers. It is used in a huge corporation. When the last customer interview is complete, a relatively long time has passed from the first interview and it is time to interview all the customers again. In this example restarting the component changes operation. When the component gets restarted it always starts interviewing from the first customer. If restarting happens often, the first customer gets irritated and it may take ages before the corporation gets interviewing results of the last customer. In a case like this, storing some of the operational state can be used to optimize behaviour of a component even, if storing the state is not a requirement for operation.

Many tasks done with office software are fast, so storing state of a task in order to save time or computing power is not feasible. With some more heavy computational tasks the progress is stored at the level of sub operations. For example a compiler may store state of compiling operation at the level of compiled object files. Once an object is compiled there is no need to recompile it unless the source code changes. Thus, if a compiler crashes while compiling, it is possible to restart work from the last unfinished object.

Computational operations may have an end or they may not have one. Distributed systems are often lacking a final goal that would end all processing. Components are designed to serve other components or users in an on demand fashion. Once the need for a service is gone, operation of the component is ended explicitly by a system administrator.

Large distributed systems may also be hard, if not impossible to restart. For example the Internet has way too many components to be restarted. The systems may also have some availability requirements that could not be met, if the system had to be restarted. Shutting down the Internet completely for a moment doesn't seem a sensible option.

Figure 3 shows an imaginary component that controls an automatic cannon. Application programmers interface of the cannon offers a method that operates the loading mechanism which inserts an explosive shell into the cannon and another method which fires the cannon. In such case remembering the state of operation would be mandatory. A crash could occur after inserting a shell into the cannon. When restarted, the component would insert another shell without shooting the previous one first. This might break the canon or even result in an explosion.

	restart time *(s)*	
	clean	**crash**
RedHat 8	104	75
JBossAS 3.0	47	39
Windows XP	61	48

Fig. 4. Clean and crash reboot durations. (experiment results from Candea and Fox[3])

III. Crashes and Clean Shutdowns

Clean shutdown refers to a shutdown operation that manages to end operation of a software component in a known good state. A sophisticated algorithm that takes in account the inner state of a component may be needed in order to perform clean shutdown. A public method implementing such algorithm is usually called in order to shutdown the component. Each component will then recursively call shutdown methods of the components they use.

Crashes are situations where a component working on a task is interrupted unexpectedly. Crashes are usually enforced by a master component for components that refuse to work by some set of given rules. For example an operating system might crash a process for writing to memory locations reserved for other processes or a user might crash a piece of computer hardware by cutting out its power source, if the machine started to create loud operating noises in extension to computational results. In UNIX systems executing kill -9 is an example of crashing a process. Turning off a machine or a virtual machine, inside which the process is being ran is another example.

Crashes can often be hard to avoid. Sudden interruption of execution at an inconvenient time may leave the system in a broken state. Thus it may be feasible to implement recovery procedures for healing from an occasional crash. The most important task for a recovery method is to heal the component, thus allowing normal operation to continue. Some more ambitious recovery procedures may attempt to bring back data processed at interruption time. For example a word processor might attempt to bring back a document being worked on at interruption time.

Sometimes the shutdown algorithm may take longer time to execute than the recovery time it saves. Figure 4 shows the result table for an experiment where three systems were restarted. The systems included Red Hat Linux 8, JBoss Application Server 3.0 and Windows XP. Red Hat Linux was

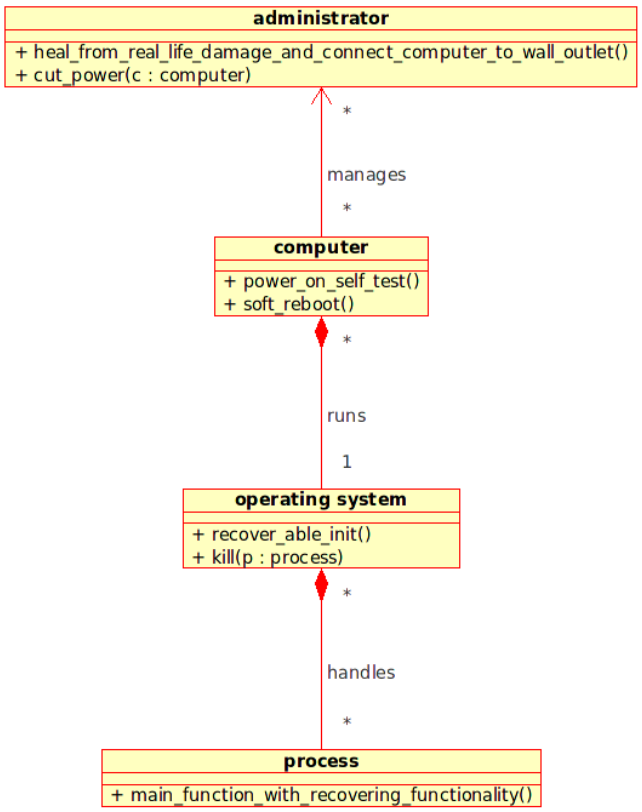

Fig. 5. Examples of Crash-only components with external crash and internal recovery methods.

configured to use ext3 (third extended file system) as its file system. Each system was restarted two times. One of the restarts used a shutdown algorithm for shutting the system down in a *clean* fashion, while the other brought the system down with a *crash*. No important data was lost in any of the experiments.

IV. Crash-only Components

Components that intentionally do not provide any public method for shutting them down are referred to as crash-only[3] components. Lack of public shutdown methods forces user to crash a component in order to shut it down. Crashing is also the only way a crash-only component can shut itself down. At high level the crash-only behaviour can be described by the equations stop=crash and start=recovery. While crash-only components don't provide a shutdown method for themselves, methods for shutting down subcomponents may be provided.

Figure 5 shows a class diagram example for a few crash-only components. Each component provides an internal recovery method. Each component also provides a method to crash their subcomponents externally without dealing with internal state of the component. In the example "process" makes an exception. It doesn't provide any mechanism for crashing subcomponents, as it doesn't have any subcomponents.

Crashing components on each shutdown puts recovery methods on stress. Trivial crashes are usually fixed before the software is released. Therefore the remaining crashes happen seldom and are hard to simulate. The lack of a shutdown algorithm implies crash-safety and shutting a component down always by crashing it results in frequent testing of crash recovery code.

In their description of crash-only design, Candea and Fox[3] limit the discussion mainly to software components. They also describe many requirements for crash-only systems. These requirements include many software specific requirements on used protocols, additional software components, that a crash-only system has to implement, and the way these pieces are glued together to form a system. While the stated requirements provide a useful example for using crash-only components, limiting their use strictly to that design only seems counter-productive.

Reducing all different faults into crashes might make a system hard to debug. When debugging a system it is useful to search for the weakest link. Restart density can be used as a quality metric for a crash-only component. The component that is restarted most often is then the weakest link in the system. Specifics of recover operations can be logged for performing further error analysis at later time while the system is running. This way there would be no analysis overhead at restarting time.

Counting failures is not a new idea. For example Tannenbaum has been advocating LF (Lifetime Failure) as a software quality metric [5]. A quality metric for crash-only components would take in account also failures that are invisible to the user. LF could still be used for measuring quality of the overall user experience.

V. System Composition

Candea, Cutler and Fox[3][4] describe a component level retry architecture model for composing systems out of crash-only components. The architecture includes a service request tree and a state store for storing important state.

The components communicate by sending service requests to each other. The service requests are a way of asking the serving component to perform some computational operation. The service requests are answered with the computational results. When the request has no computational results, successful completion of the given task is reported.

Components are wired together to form a tree shaped structure. Action begins when root component is assigned a task. Root component starts working on the subject and uses a set of worker components, to solve subproblems, by submitting service requests to them. The serving components then behave similarly. Each of them uses a distinct set of components for solving partial problems. Finally processing of the task reaches leaf components that do not need other components for performing their task. In the minimum setup there is only one component which is both a leaf component and the root component of the system.

When component fails to do its task, each component in the subtree behind that component gets restarted along with the

failing component. The restarts may be launched recursively for parallel components or at once for components that are one inside the other. Processes in an operating system would launch restarts of other processes recursively, while turning the power switch of a computer would effectively restart every component running in the computer. Because the system is tree shaped, the components behind the failing component are hidden from the other components. Thus restarting an entire subtree won't cause any side-effects on other components.

Each service request from a component to another component has a related TTL (time to live) value. TTL is the time frame within which the requested operation is expected to complete. At the time of request TTL is set to a positive integer value. The TTL value may be decided on forehand or calculated from duration of past operations or current system load.

The TTL value is then decreased periodically until it reaches zero. TTL values can be handled by timers and they do not need to be forwarded with service requests. If no response is received, at that point, the requesting component resubmits the request. Root component hides all the other components behind it and provides public interface for using the system. When processing moves towards the leaf components, the components hide less functionality behind them. Therefore the operations closer to leaf components are lighter and may have smaller TTL values than the components closer to the root component. Having smaller TTL values closer to leafs is useful, as it allows multiple transparent retries at lower level before the TTL values at upper level run out.

Storing state is often useful, sometimes even mandatory. The architecture model contains a state store where important parts of system state are stored. The state store is itself a crash-only component. Storing state in a dedicated store, is expected to make recovery simpler by introducing one component for managing all important state and pushing all related requirements to that component. Rest of the system becomes stateless and thus free of state related requirements.

Pushing all state handling into just one component simplifies recovery as the component can keep state transitions consistent with techniques similar to transactions. Full ACID (Atomicity, Consistency, Isolation and Durability) requirements may be too much for most systems. Taking in account specifics of the target system, it should be possible to design lighter alternatives.

Transactional crash safety comes with a price. It requires using some additional methods that are used for crash safety purposes only. Letting the system break when it crashes would make the system operate faster.

When a system stores some state with restart persistence it takes the risk of storing a failure in the state information. Thus there is a trade-off between amount of state stored and the systems ability to heal itself by restarting components. A stored failure which would make a component fail again immediately after restart might cause the system to enter a tight restart loop. In this worst case scenario the failing component would not be able to do its task.

VI. Conclusion

Healing by component level restarts is simple and robust which makes the approach suitable for healing from many different unsuspected faults, including purely technical faults and failures enforced on purpose by a malicious party. Preparing for crash is usually a good idea. Crash-safety can be added to components by adding a recovery algorithm that is executed at startup. Carefully designed runtime behaviour can simplify the recovery. In most cases there is no point in using a sophisticated shutdown algorithm with crash-safe components. Sometimes the shutdown algorithm may even take longer time to execute than the recovery time it saves. Restarting components can also be used for component level rejuvenation.

Crash-only components do not provide any public method for shutting them down which puts recovery methods on stress. Weak spots of the system can be found by counting failures that lead to restart of component. In a component level retry architecture model components are wired together to form a tree shaped structure. When a component fails to do its task the system is able to heal by restarting each component in the subtree behind that component along with the failing component. All state handling can be pushed into one component. Having all state stored in a central place simplifies recovery by making most parts of the system stateless. When a system stores some state with restart persistence it takes the risk of storing a failure in the state information.

The discussion around crash-only self-healing seems to concentrate on network services that run within a single administration domain. The approach might work well also in some public systems in which the nodes don't trust each other. Isolation and fail-safety are already important for systems consisting of untrusted nodes. Using a crash-only self-healing strategy in a public p2p-environment might be an area worth further studies. Building user interfaces of crash-only components and allowing user to crash them at any time, might also be an interesting experiment.

References

[1] *Minix website*, http://www.minix3.org/reliability.html (visited 10 March, 2007)

[2] M. Locasto, K. Wang, A. Keromytis and S. Stolfo, *FLIPS: Hybrid adaptive intrusion prevention*, In RAID, 2005.

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

[4] G. Candea, J. Cutler and A. Fox, *Improving availability with recursive microreboots: a soft-state system case study*, Perform. Eval. vol 56, 1-4, 2004, pages 213-248, Elsevier Science Publishers B. V.

[5] *Lifetime Failure* metric was introduced by Andrew Tanenbaum at linux.conf.au in 2007, according to *Tanenbaum outlines his vision for a grandma-proof OS* article by Howard Dahdah, http://www.computerworld.com.au/index.php/id;1942598204;fp;4;fpid;1968336438 (published 24 January, 2007 visited 12 March, 2007)