the Availability Digest

Microrebooting for Fast Recovery

March 2007

In our article from last month entitled Recovery-Oriented Computing, we described the research efforts being undertaken by the Recovery-Oriented Computing project at UC Berkeley and Stanford University in California. We first summarize that article below and then describe an important contribution of the ROC project toward achieving very fast recovery times of failed systems – microrebooting.

The Recovery-Oriented Computing Project

In the past, high availability was brought to the marketplace by large mainframes and fault-tolerant systems. However, with the advent of the Internet, enterprises have turned to large complexes of heterogeneous off-the-shelf components. The availability of these hybrid configurations as compared to mainframes and fault-tolerant systems has suffered for a number of reasons:

- The large number of components – servers, routers, databases, and so on - has led to significantly higher failure rates, which must be managed.
- The heterogeneous systems have been acquired from different vendors, with each system requiring its own form of system administration.
- The applications are characterized by rapid innovation. Consequently, there is no time for formal software design and testing which leads to less reliable application software.
- Because of the system complexity, it is often difficult to locate the source of a failure.
- Because of the complexity of system management, there is a high incidence of operator errors.

The user perception of availability is more important than actual system availability. It is the user that counts. Perceived availability is different from actual availability because the user does not see faults if they are repaired quickly enough. If faults can be recovered fast enough, there has been no fault so far as the user is concerned.

The goal of the Recovery-Oriented Computing project is to

- reduce operator and software failure rates,
- automatically locate the source of faults, and
- rapidly recover from those faults.

Fast recovery is the essence of improving the users' perception of system availability.

To provide fast recovery from operator errors, the ROC project is investigating the obvious – an "undo" function for operator commands that simply doesn't exist now. When an operator makes an error, he usually knows it immediately and could correct it if he simply had an undo key.

A ROC facility for quickly locating software faults uses a tracing technique to determine the processing path that each request takes. By recording these paths for both successful and failed requests, the facility can determine the most suspicious software component in the path that has caused a fault.

Fast recovery depends upon microrebooting in which the smallest component that might be causing the fault is simply rebooted. If the microreboot is fast enough so that other components invoking it are successful on a retry, the user is unaware of the fault. Microrebooting is the subject of this paper.

Microrebooting

Rebooting a system lets the software start with a clean slate. Its parameters are properly reinitialized. Resources such as memory and file descriptors which it has leased are returned to the system pools. Whatever problems that the system had been experiencing have been eliminated. Rebooting is the ultimate fix to a software problem short of tracking down the bug and correcting it. In effect, rebooting cures aging software.

The problem with rebooting as a generic recovery procedure is that it can be time consuming. It can take minutes or more. Furthermore, all user sessions are lost; and critical data may also be lost. To the user, the system is certainly seen as being down during the reboot time.

Usually, when a software fault has occurred, it is only in one application. If it is possible to reboot only that application and not the entire system, this is a step in the right direction.

Microrebooting[1] takes this philosophy to the extreme. In an object-oriented environment, it attempts to deduce the most likely object that caused the fault and reboots just that object. Microrebooting can be done typically in a fraction of a second and is orders of magnitude faster than a full reboot. Even more important is that the system unavailability is transparent to the typical user because of the short recovery time and the fact that he does not lose his session.

Pinpointing the Problem

In Internet applications, the area in which ROC focuses, a software fault generally results in either an error message returned to the user in response to a request or in that request timing out. To correct the error, the first step is to locate the problem. The ROC project has built a prototype facility that does this quite efficiently. It is called PinPoint.

PinPoint is a cross between a trace facility and a data mining utility. When operational, it traces the path that each request type takes through the software at a very fine level, object-by-object. It records these sequences for each request whether the request was successful or not.

By later analyzing the unsuccessful request paths against the successful request paths, PinPoint can deduce the object that most likely created the fault. When a software fault occurs, it is this object that is first rebooted to attempt to correct the problem.

[1] G. Candea, S. Kawamoto, Y. Fujiki, G. Friedman, A. Fox, Microrebooting – A Technique for Cheap Recovery, Proceedings of the 6th Symposium on Operating Systems Design and Implementation; December, 2004.

PinPoint overhead has been determined to be about 10% in a JBoss application server environment.

Restart Tree

Supporting PinPoint is a Restart Tree that is compiled based on knowledge of the application that is being instrumented. The Restart Tree shows the interdependencies between the various objects in the application. When an object is rebooted, so are all of the objects that are subservient to it.

Architectural Requirements

In order for microrebooting to work, the system components must be well isolated and stateless. Isolation is important so that other components can tolerate the temporary unavailability of a component which is being rebooted.

Isolation is generally achieved by having loosely-coupled components interacting by messages only. In this way, a component can be defensive and ensure that it approves any change in state or response which it may give. In no way should one software component be allowed to directly invoke internal logic of another component or to change its state. Common memory architectures should be avoided. These are all attributes of objects in today's technology.

Stateless means that the component's important state must be physically stored externally to the component in some sort of state store that is independent of the component. This creates what ROC calls a *crash-only* software module. That is, the module can be rebooted without losing any state. Crash-only software is discussed in more detail later.

These considerations lead to the ROC framework for a microrebootable system:

- The system must be composed of fine-grain components (such as objects).
- The components must be loosely coupled so that they can tolerate the momentary outage of a component whose services they are trying to invoke.
- Requests that one component makes to another component must be retryable.
- The components must not encapsulate any state information.

These are all well-known principles of robust programming, though simply in finer detail.

The combination of the isolation of components and the retryability of requests means that a component will not fail simply because a service that it needs from another component is temporarily unavailable. If a component is being rebooted, any component attempting to invoke its services simply waits a bit and then retries. The invoking component need not panic and abort.

Isolation and retryability, coupled with a fine-grain structure and crash-only components, sets the stage for microrebooting.

The Reboot Procedure

Microrebooting is intended to achieve as fast a recovery as possible. However, there is no guarantee that microrebooting at the lowest level will work.

Therefore, the strategy is to attempt recovery by rebooting at ever more coarse levels. The first attempt is to reboot the suspicious lowest-level component. Typically, this is so fast that if it

doesn't work, there is no significant recovery time penalty. If it doesn't work, the next highest level component is rebooted. This continues until the only recourse is to reboot the entire system.

The hope is that the system will properly recover at some lower level than a full reboot.

Microrebooting Prototype

The Recovery-Oriented Computing project has prototyped microrebooting with some impressive results. As a test vehicle, they chose an auction application which runs under JBoss and modified it to be crash-only. They also added a microreboot method to JBoss to perform microreboots on any crash-only J2EE application. They injected faults into the system, and the response to the faults was monitored. The recovery was deemed unsuccessful if the user received an error message and had to retry his request.

In order to inject faults, they created hooks in JBoss to cause errors such as deadlocks, infinite loops, memory leaks, JVM (Java Virtual Machine) memory exhaustion, Java exceptions, and data structure corruption. They also used a utility called FIG to introduce faults. FIG stands for Failure Injection in glibc. glibc is the GNU C library used by JBoss.

Running in the J2EE framework, JBoss applications are implemented via Enterprise Java Beans (EJBs). Java Beans are fine-grained objects that are isolated and retryable so that they are ideal for microrebooting.

PinPoint, described above, was used to monitor the application to determine the likely Java Beans involved in a fault. When a Java Bean detected an error, it reported that error to a Recovery Agent before retrying. Determining the likely culprit from the PinPoint study, the Agent would reboot that Java Bean and all of its subservient Beans. This happened very quickly.

If errors were still reported, the Agent would reboot the application reporting the errors. If that did not correct the problem, JBoss was rebooted, followed by the JVM, and finally the operating system. If none of this worked, a message was sent to the operator requesting human intervention. The operator was also notified of recurrent faults to avoid continual microrebooting.

A microreboot of a Java Bean took on the order of 500 milliseconds. Any reboot beyond this resulted in a user error. The application took about 8 seconds to reboot, the JVM 20 seconds, and further reboots were measured in the minutes.

The result was that failed user requests were reduced by 98%! Clearly, microrebooting has a future.

The Bottom Line

The advantage that microrebooting brings is that a sick node can continue to process requests while it is being recovered. If microrebooting is successful, it is done within the timing parameters of the system. The result is that the user is oblivious to the node failure.

Application to Clusters

Can microrebooting be advantageous in clusters? Definitely.

When a node in a cluster fails, the system must fail over to another node in the cluster. User sessions are lost, and it takes several minutes to bring up the application in the new node. During this time, the system is not available to its users.

If microrebooting is used, there is a good chance that a failed node can be salvaged. No user sessions will be lost, and the node will be repaired without failover.

In tests on an eight-node cluster, user errors were reduced by 93%. However, as the size of the cluster grows, this effect is reduced because there is a smaller percentage of users on each node. Thus, the advantage of microrebooting becomes less significant (though it will always result in better recovery performance).

Microrebooting added less than 2% overhead to each node.

A side benefit of cluster microrebooting is that there is less of a need to over-provision the cluster to provide sufficient capacity and response time during a failure.

Microregeneration

Another use for microrebooting is *microregeneration*. Some shops make it a practice to preemptively reboot their systems periodically to recover resources such as memory leaks (which occur despite garbage collection) in order to prevent unanticipated problems.

Microrebooting can be a better solution to system regeneration. Using memory usage as an example, the amount of available memory can be monitored. When it falls below an acceptable level, microrebooting can be employed on a rolling basis to reboot objects until available memory has been returned to an acceptable level. This process is concurrent with normal use. The users see no outages.

Initially, the choice of object order is arbitrary. But by keeping track of the amount of memory returned on each microreboot, the system can learn the optimum order for microrebooting.

Crash-Only Software

As we have said, microrebooting requires that the components that are being rebooted be crash-only software.[2] This means that the components must have the capability of being stopped and rebooted, then continue on from where they left off.

This implies that components cannot embed within themselves any state which will be lost if they are stopped and then rebooted. They must be stateless so that they can survive microreboots.

To accomplish this, important state must be stored in a dedicated state store. The state store need not be persistent. It just needs to be stored outside of the components so that it survives microreboots. If it is persistent, it will survive higher-level reboots.

Summary

Microrebooting is the rebooting of fine-grain software components following an application fault. Microrebooting can achieve rapid recovery, so rapid that user sessions are not lost and the users are unaware of the fault. If the microreboot is not successful, ever-coarser reboots are attempted until the application is returned to service.

In order to be microrebootable, an application must comprise fine-grain components that are loosely coupled, retryable, and stateless.

---

[2] G. Candea, A. Fox, Crash-Only Software, Proceedings of the 9th Workshop on Hot Topics in Operating Systems; May, 2003.

In prototype experiments, microrebooting has shown a 98% reduction in user-perceived faults. This allows a failed node to continue in operation without having to reboot it.