Unbreakable Java

The Java Server that Never Goes Down

Thomas Smits

SAP announced "*Always On Java"* based on VM Container technology at SAP TechEd 2004 in San Diego and Munich. This article presents some of the secrets behind the promise of an "unbreakable" Java server.

Unbreakable Java

Developers using Java on clients or in small projects may not believe that there is a fundamental problem with Java's robustness. People working with huge applications and application servers written in Java know about the problem, but may doubt that it is possible to build something like an unbreakable Java architecture. Some may even remember the White Star Line, which promised that their ocean liner Titanic was unsinkable – an iceberg in the North Atlantic proved them wrong and demonstrated that there is no such thing as an unsinkable ship. Is it really possible to build a Java application server that never goes down?

It's all about isolation

The key to the understanding of robust Java is isolation, isolation, and isolation. Robust applications, and especially robust application servers, require a high level of isolation between users. It is not acceptable for an error occurring while processing one user's request to affect all users connected to the system. The complexity of software systems makes it impossible to develop software that is completely free of errors, so errors will always happen. Only isolation can provide real robustness by limiting the impact of errors.

The design of the Java virtual machine ignores the painful lessons operating system vendors have learned in the past 40 years. The concepts of processes, virtual memory management, and different protection modes for kernel and user code can be found in all modern operating systems. They focus on the question of isolation and therefore robustness: an application with errors cannot affect the other applications running in the system [1].

In contrast to this, Java follows the all-in-one-VM paradigm: Everything is processed inside one virtual machine running in one operating system process. Inside the VM, parallelism is implemented using threads with no separation regarding memory or other resources. In this respect Java has not changed since its invention in the early nineties. The fact that Java was originally invented as a programming language for embedded devices may explain this approach [2].

There is no isolation in Java

Java does not have a problem with isolation; there is virtually no isolation at all. Java tries to avoid dangerous concepts like manual memory management (this is like taking some of the icebergs out of the ocean) and it cannot be denied that it provides at least some isolation concepts, but a Java virtual machine is still easy to break. For example, class loaders make it possible to partition an application into parts that cannot see and access each other directly, which provides some isolation. Going back to our nautical example from the very beginning, this is exactly what was supposed to make the Titanic unsinkable: the ship consists of separate compartments and water pouring into the ship is supposed to be stopped by the bulkheads separating the compartments – unfortunately the iceberg was too big and way too many compartments filled up with water. In terms less familiar to the sailor, but more to the developer: all the fancy isolation built with class loaders does not help if you have memory leaks, threads running amok, or even bugs in the VM itself.

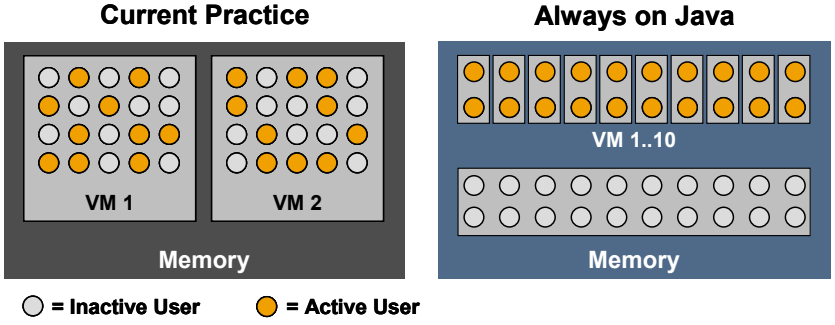
SAP's approach to isolation

SAP's ABAP application server – the powerhouse underlying enterprise-scale R/3 business solutions – was based on the concept of process isolation from the very beginning. It consists of a dispatcher and a bunch of work processes handling the requests. The work processes are normal operating system processes and the OS provides a high level of isolation for free. The dispatcher guarantees that in one moment of time, exactly one user request is processed by each work process. In case of a crash, only the user currently processed in the crashing process is affected. All other users continue their work and the operating system takes care of the resource cleanup [3].

Why is it called VM Container?

The technology behind the *Always On Java* initiative is called VM Container and the name suggests that there is something like a virtual machine and a container housing it. Right!

The name is based on the fact that the ABAP application server already contains a lot of interesting and battle tested services that can be reused to build a robust Java server. The components were reshaped and now provide the container that hosts the Java virtual machine. The VM itself was licensed by SAP and modified to seamlessly integrate into the container and to provide additional features like sharing technologies and enhanced supportability.



In current designs, a high number of users share one virtual machine; problems will affect all users. In the always on architecture, the number of users per VM is reduced and inactive user sessions are kept outside the VMs.

To overstress the ocean liner example a little: the ship is not split up into compartments, but every passenger gets its own ship (a separated process) with some guide (the dispatcher) taking care that all sail the same course and do not hit each other. Using this architecture, an iceberg (a severe error) may still hit one of the ships but it will affect only one passenger.

One passenger per ship sounds weird. Giving each passenger his own private dining room and engines seems to be a huge waste of resources. Two things can be done to handle the resource issue. First of all, it is possible to let the passengers share the ship with some others without meeting them at any time. Some invisible mechanism moves the sleeping passenger out of the ship storing them somewhere outside and puts another active passenger into it, taking care that only one active passenger is in each ship at any moment of time. The second way to address the resource problem is to share as many resources as possible between the little ocean liners.

In the ABAP application server, the state of the user – often called *user context* – is not stored inside the process but in a shared memory area accessible to all work processes. This allows attaching the user context to a free work process when the next request arrives. Attaching user contexts is a very fast operation because no data is copied.

The ABAP virtual machine (yes, ABAP is executed on a virtual machine) was designed from the very beginning to store user contexts in shared memory. All infrastructure (the engines, the dining room) is written in C and able to deal with user contexts being moved between the work processes, too.

User isolation in Java

The *VM Container* technology transfers the ABAP isolation concepts to the Java world. The first step is to increase the number of virtual machines and therefore reduce the number of users handled by each VM. Having a hundred instead of a thousand users assigned to a VM makes a difference in case of a crash, but still affects too many users. Decreasing the number of affected users further without increasing the number of virtual machines requires some extra magic.

Shared Closures

One of the key features of the VM Container technology is the Shared Closures API. It provides a semantic similar to serialization but with a new and very fast implementation. This technology enables middleware developers to share Java objects between virtual machines running on the same computer. For the application developer high level APIs based on Shared Closures are available, for example providing caching or configuration management.

The name Shared Closures already implies that not only single objects but the whole transitive closure of objects reachable from one root object is shared. This behavior is like Java serialization with the difference that the operations are faster and a special mode of operation, called mapping, is supported.

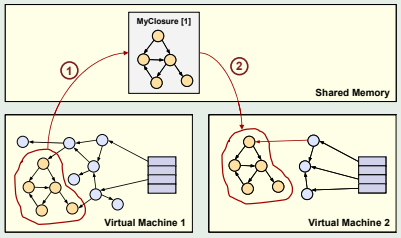
A Shared Closure is created or updated by providing a reference to the root of an object tree to the API. The content of the tree is copied to the shared memory while the objects inside the virtual machine remain unchanged.

An exiting Shared Closures can be used in two different ways:

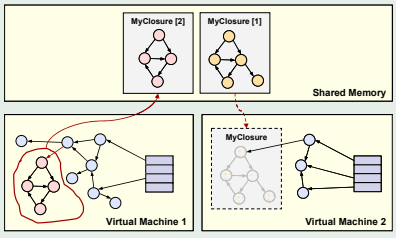
- Copy – the objects in the Shared Closure are copied to the heap of another VM. The objects become normal local objects and can be modified (Picture 1).
- Map – the objects in the Shared Closure are not copied but only mapped into the address space of the virtual machine (Picture 2). This operation is very fast in comparison to copy, because no data is transferred at all. Especially no extra memory is consumed for the mapped objects. The objects mapped into the address space are read only.

An implicit versioning mechanism takes care of the fact that some VMs may have mapped a version of a Shared Closure when another VM wants to publish an update. All previously mapped closures remain unchanged whereas new map requests provide the new version. A distributed garbage collector removes all old versions that are no longer used.

Mapping objects from Shared Closures is the best mode of operation for caches and configuration data that changes rarely. Copying the data of a Shared Closure is used to implement session failover or messaging mechanisms.



Picture 1: Create and copy of a Shared Closure



Picture 2: Versioning and mapping of a Shared Closure

Normally, less than ten percent of the users connected to a system are actively sending requests; the others are thinking about their next action or typing in some data at the front-end (*thinking users*). Keeping the user state (*user session* in Java terms) in a memory area outside the virtual machine allows re-establishing the sessions of all thinking users in case of a crash. This reduces the number of affected users in our example to only ten, or one percent of the thousand users.

The technology used to keep the sessions outside the virtual machine is called *Shared Closures* (see box for details). The session state of a user is saved to shared memory after his request was processed. This guarantees that the shared memory contains a backup of the session state of at least all thinking users and that the data is accessible to all virtual machines. In case of a crash, another virtual machine can copy the user state from shared memory to its local memory and continue processing the user's requests without the user even noticing.

Memory diet for the VM

The drawback of the described approach is that you have more virtual machines, each of them eating up some memory. This requires extra measures to keep the memory footprint of the VMs low; they must be put on a diet. The problem is addressed by *Shared Classes*.

The memory consumed by Java classes can become quite large in real-world applications. Shared Classes is a technology built into the Java virtual machine that shares the runtime representation of the classes, including the native code generated by the JIT compiler, across all virtual machines on one physical box. The classes exist only once in memory reducing the overall memory consumption of the VMs.

In addition to the session backup explained above, Shared Closures can be utilized to reduce the memory footprint of a virtual machine. Configuration data and other application or server-wide information can be shared between VMs. Mapping the data from shared memory will provide access to it without consuming memory in each VM.

Don't forget supportability

Providing a high level of robustness through isolation is half the battle, but robustness without supportability is not sufficient. If something goes wrong in the application server, support personnel must be able to track down and resolve the problem easily.

The virtual machine used in the VM Container has been improved regarding supportability. One of the most interesting features is the ability to switch dynamically into debugging mode and vice versa. The switch can be initiated from the inside (using Java code) or from the outside (using administrative tools). Normally, Java application servers need dedicated debugging nodes, because the Java virtual machine must be switched into debugging mode at start-up. Using the VM Container, debugging is possible at any time, even in productive systems. A sophisticated rights management restricts which parts of an application or server a developer can debug. This prevents misuse of debugging capabilities in production environments.

Besides debugging, the monitoring capabilities of the VM can be used to obtain granular statistics about the running server. The monitoring is built in a way that does not affect the performance of the running application until explicitly switched on.

Virtual Machine

The virtual machine used for the VM Container is based on a SUN CDC/Hotspot VM. It was originally designed for embedded devices, making it very lightweight and easy to port to new platforms [5]. Having a VM with a low memory footprint is important because the isolation approach of the VM Container will increase the number of parallel running VMs. You may imagine the VM Container as a *cluster of Palm Pilots* if you like.

A peek into the labs: full user isolation

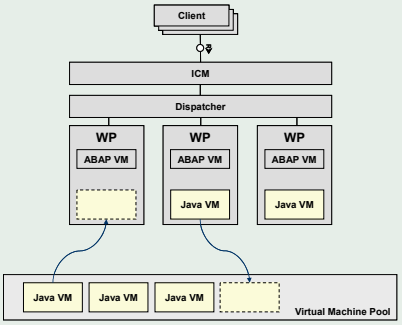
In the development labs at SAP, work is in progress on a solution that goes beyond the approach described in this article: it merges the Java and the ABAP world. Both virtual machines run together in one work process and full user isolation is provided for ABAP and Java programs: in one Java virtual machine, only one user request is processed in one moment of time.

A new paradigm was implemented called *Process Attachable Virtual Machines* [4]. It decouples the VM from the process and makes it a lightweight memory image that can be moved between processes. Using VM templates, new virtual machines for the pool can be created with nearly no runtime effort. VM templates are available which contain a fully bootstrapped virtual machine including the application server and the deployed applications. Using VM templates offers a way to create new virtual machines for the pool instantaneously.

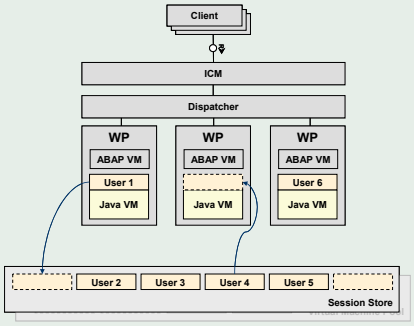
The number of work processes can be configured in a way to guarantee that the working set of all processes fits into the machine's main memory (although the memory is usually too small to hold all VMs at the same time). The number of virtual machines in the pool is normally higher to take into account situations where a virtual machine does blocking I/O or other operations that don't use the CPU. In those cases, the VM is temporarily detached to free the process for new requests.

The operating system schedules preemptively between the processes but the virtual machines are moved in and out of the processes on a semantic base (*semantic scheduling*). This dramatically reduces the problem of thrashing because the working set is only changed after a user request was finished. Controlling the semantic scheduling is easy because the VMs are not operating system processes but attached to processes and detached on demand.

The session state of the users is kept in a special shared memory area accessed via the Shared Closures technology. The VM and the user session are separated after each request. Therefore the VMs can be used independently of the user sessions; there are no sessions sticky to a special VM except in the moment of time when a request is processed.



The virtual machine pool contains process attachable VMs that can be assigned to work processes on demand.



User sessions are kept in shared memory and copied to the virtual machine before the request is processed and copied back after the request was finished..

Summary

The VM Container technology offers improved robustness through isolation. The isolation is provided by reducing the number of users handled in parallel in one virtual machine. Saving the user's session state in a shared memory area improves the fail-over characteristics of the application server. Advanced sharing technology helps to reduce the memory footprint of the virtual machines. Improved monitoring and debugging support makes it easy to detect and fix problems at runtime.

THOMAS SMITS

has been an Development Architect in the SAP NetWeaver™ team since 2002.

References

[1] Tanenbaum, Andrew *Modern Operating Systems (2nd Edition)* Prentice Hall, February 2001

[2] Byous, Jon *Java Technology: the Early Years*, http://java.sun.com /features/1998/05/birthday.html

[3] *SAP Web Application Server Components*, http://help.sap.com

[4] Kuck, Norbert et.al *SAP VM Container: Using Process Attachable Virtual Machines*, Java Virtual Machine Research and Technology Symposium, San Francisco, August 2002

[5] *J2ME CDC HotSpot Implementation Overview*, http://java.sun.com/products/cdc-hi/overview.html