Availability Services Integration with Third-Party MPI Stacks

Mike Heffner
mike.heffner@evergrid.com
v1.2

Overview

Availability Services (AVS) provides transparent, user-level checkpoint/restart support. AVS enables highly optimized checkpointing for large distributed compute jobs running in grid/cluster environments. AVS supports state capture and replay of common InterProcess Communication (IPC) constructs such as: pipes, shared memory, and BSD sockets.

To support grid jobs that utilize high-speed messaging fabrics, such as RDMA, AVS provides hooks to integrate with third-party messaging stacks that provide their own network state capture. The most common approaches to network state capture include: full message logging that ensures correct message replay and quiescence interfaces that stop network traffic before checkpoint state capture. The latter approach is often preferred for zero-copy fabrics where message logging has too much overhead. This document serves to describe the high-level integration hooks AVS provides for supporting checkpoint/restart when using third-party MPI stacks.

The following sections describe the high-level integration hooks required by AVS or provided by AVS to the messaging fabric. The specific parameters and methods of invocation have been left out in-order to concentrate on the high-level operation. Further, in-depth engineering discussions can flush out the detailed integration requirements.

Quiescent Integration Hooks

AVS provides single process checkpointing capabilities for third-party MPI stacks using messaging fabrics other than BSD sockets. AVS handles the recording of all process state required to restart an application except for the state of the underlying messaging fabric. To correctly store a globally consistent checkpoint image, AVS relies on several integration APIs to inform the messaging fabric to quiesce the fabric to prepare for a checkpoint. Once the third-party messaging stack is able to quiesce the fabric, AVS will correctly record the state of each process participating in the distributed application.

The most common method of using AVS is to initiate the checkpoint operation with AVS and for AVS to propagate that checkpoint operation down to the messaging stack. While this is most common, AVS will also support the reverse – a checkpoint operation initiated by the messaging stack. This is described in a later section.

int network_comm_freeze(...)

This operation is invoked from within a signal handler inside AVS during a checkpoint or preempt operation. It is called from the main thread of the process and it is intended that this call should quiesce the messaging fabric. After the return of this function the messaging library should be in a state that a checkpoint can be initiated and, if successful, a restart could be later invoked from. Every process participating in the distributed checkpoint will invoke this call asynchronously. It is understood that one or more barrier operations would likely be required at the messaging layer during this procedure.

int network_comm_unfreeze(...)

This operation is called after AVS has appropriately recorded all process state required to reconstruct a globally consistent checkpoint image. It is invoked only during a checkpoint (NOT a preempt) from within the same asynchronous signal handler that invoked `network_comm_freeze`(). The intent of this function is to inform the messaging fabric that the application is about to be given back control of execution so that the messaging fabric can undo any operations that were required to quiesce the fabric during the call to `network_comm_freeze`().

This function can be called before the entire checkpoint has been completely stored to a secondary storage device. However, when it is called, AVS provides the guarantee that any process state required for the globally consistent checkpoint will be safe. Therefore, the messaging fabric should not be concerned with the *background* checkpoint that will be invoked from AVS.

int network_comm_restore(...)

This operation is called from AVS during a restart operation after the process has been reloaded. On a restart operation, AVS will reload all application memory, saved checkpoint memory zones, application file descriptors and all other application state. The state of the application will be reloaded to where it was at the last successful checkpoint image. The purpose of this call is to reconstruct the state of the messaging fabric after a restart operation.

After this call returns, the messaging fabric should be in a ready state to resume application messaging. It is recognized that the messaging fabric must reconnect and reinitialize all communication channels previously opened at the time of the checkpoint. This function will be invoked asynchronously for each process in the distributed application, so it is common to perform a barrier at the fabric layer during this call.

The distributed application will be allowed to continue execution soon after this function is executed.

Critical Section Support

AVS provides asynchronous checkpointing support for applications. However, there can be times that a critical section of code should not be checkpointed, i.e., a critical section. For these sections AVS provides lightweight mechanisms to block the handling of checkpoints or preempts. However, checkpoints should never be disabled during a blocking operation, as deadlock can occur if the resource blocked on is interrupted by a checkpoint.

void avs_block_checkpoints(...)

void avs_unblock_checkpoints(...)

Block and unblock checkpoints, respectively. These operations support recursive calls, so you can call `avs_block_checkpoints`() as many times as required as long as it is followed by the same number of calls to `avs_unblock_checkpoints`().

Bypassing AVS Virtualization

AVS will virtualize all system resources used by the application (e.g., file descriptors, shared memory handles, POSIX thread handles, etc.). When an application is restarted, all these resources will be correctly restarted and reinitialized to their corresponding states at the time of the last checkpoint.

For resources that must not be virtualized and reconstructed during restart, AVS provides *unvirtualized zones*. These are commonly useful for hardware-level messaging stacks that require access to device driver files under `/dev` that can not be correctly virtualized and restarted by AVS. It is also useful when a resource will be manually restarted outside of AVS and therefore the virtualization penalty can and should be avoided.

The following procedures instruct AVS that the invoking thread is entering and exiting an *unvirtualized zone* and therefore any system calls should be passed directly to the system and bypass the virtualization layer. This will only apply to the executing thread or any thread spawned while within the zone. It is common to use these calls to wrap invocations to underlying hardware access libraries.

void avs_unvirt_enter(...)

void avs_unvirt_exit(...)

Enter and exit an unvirtualized zone in the invoking thread. These calls are not recursive.

Allocation Support

By default, all memory allocated with `malloc(3)` or `mmap(2)` will be recorded by AVS and tracked using memory protection. AVS manages memory by using *memory zones* that track individual application heaps. By default, all application memory is allocated in the default zone which maps to the application `sbrk()` heap. However, the following APIs allow a subsystem to allocate a new memory zone and use it for memory tracking outside of the application heap. Memory that must be checkpointed separate from the application or possibly not checkpointed at all, should be managed within its own memory zone. All `malloc`() and `mmap`() operations that occur while in the memory zone for a thread will be serviced from the memory zone and not the application heap. It is typical for a messaging fabric to create its own zone for any memory allocated and used by the hardware access libraries and to mark it as not checkpointed. This is an easy to way quickly throw-out all memory previously allocated to hardware devices from a previous checkpoint without worrying about memory leaks.

zid_t avs_malloc_new_zone(flags, size)

Allocate a new memory zone with specific flags. Flags change whether the zone is checkpointed or not and whether or not page protection is used to track zone usage. A zone ID is returned identifying the zone.

zid_t avs_malloc_set_zone(zid_t zone)

Set the new zone to be active for the current thread. The previously set zone ID is returned. Zones are inherited across thread creation.

zid_t avs_malloc_get_zone()

Get the active memory zone.

zid_t avs_malloc_destroy_zone(zid_t zone)

Destroy a memory zone.

Memory Protection Management

AVS uses memory page protection (see `mprotect(2)`) to track the active memory used by a process.

Page protection allows AVS to checkpoint only the set of changed memory pages between two incremental checkpoints, providing a very efficient checkpointing method. Additionally, page protection allows AVS to commit checkpoints to secondary storage while allowing an application to continue computation at the same time.

Unfortunately, page protection can interfere with high-speed messaging stacks that require direct access to application memory from hardware. A hardware operation that manipulates memory could go undetected to AVS – causing memory corruption later on – or may cause the hardware operation to fail. Therefore, AVS provides a mechanism to inform it of memory ranges that should not use page protection and therefore should be checkpointed using different methods.

... avs_mem_protection_off(address, length)

Disable memory protection for the specified range of virtual memory. Memory page protection allows AVS to perform more efficient checkpoints, so memory page protection should be disabled sparingly.

... avs_mem_protection_on(...)

Re-enable page protection for the range.

Application Initiated Checkpoints

As described earlier, AVS will typically initiate the checkpoint process by receiving an asynchronous signal during execution. Using the quiescence interface it will notify the messaging fabric of the checkpoint event. However, the following API is provided and can be called by either the application or the messaging fabric to self-initiate the checkpoint process. It will start a checkpoint for the current process. An optional parameter will perform a preempt rather than a checkpoint.

This function does not return on a preempt until the corresponding restart is performed. On a checkpoint, this function will return as soon as the global consistency of the checkpoint has been ensured across all processes. This function may perform a barrier operation across the distributed job, therefore all processes should invoke this function simultaneously.

... avs_checkpoint(bool preempt)