# Use Cases

This section provides a non-exhaustive list of use cases presenting how the DAOS storage model and stack could be used on a real HPC cluster.

This document contains the following sections:

* Storage Management and Workflow Integration
* Workflow Execution
  + Bulk Synchronous Checkpoint
  + Producer/Consumer
  + Concurrent Producers
* Storage Node Failure and Resilvering

## Storage Management & Workflow Integration

In this section, we consider two different cluster configurations:

* Cluster A: All or a majority of the compute nodes have local persistent memory. In other words, each compute node is also a storage node.
* Cluster B: Storage nodes are dedicated to storage and disseminated across the fabric. They are not used for computation and thus do not run any application code.

At boot time, each storage node starts the DAOS server that instantiates service threads. In cluster A, the DAOS threads are bound to the noisy cores and interact with the FWK if mOS is used. In cluster B, the DAOS server can use all the cores of the storage node.

The DAOS server then loads the storage management module. This module scans for local storage on the node and reports the result to a designated master DAOS server that aggregates information about the used and available storage across the cluster. The management module also retrieves the fault domain hierarchy (from a database or specific service) and integrates this with the storage information.

The resource manager then uses the DAOS management API to query available storage and allocate a certain amount of storage (i.e. persistent memory) for a new workflow that is to be scheduled. In cluster A, this allocation request may list the compute nodes where the workflow is supposed to run, whereas in case B, it may ask for storage nearby some allocated compute nodes.

Once successfully allocated, the master server will initialize a DAOS pool covering the allocated storage by formatting the VOS layout (i.e. fallocate(1) a PMEM file & create VOS super block) and starting the pool service which will initiate the Raft engine in charge of the pool membership and metadata. At this point, the DAOS pool is ready to be handed off to the actual workflow.

When the workflow starts, one rank connects to the DAOS pool, then uses local2global() to generate a global connection handle and shares it with all the other application ranks that use global2local() to create a local connection handle. At that point, new containers can be created and existing ones opened collectively or individually by the application tasks.

## Workflow Execution

We consider the workflow represented in the figure below.

../graph/Fig\_007.png

Each green box represents a different container. All containers are stored in the same DAOS pool represented by the gray box. The simulation reads data from the input container and writes raw timesteps to another container. It also regularly dumps checkpoints to a dedicated ckpt container. The down-sample job reads the raw timesteps and generates sampled timesteps to be analyzed by the post-process which stores analysis data into yet another container.

### Bulk Synchronous Checkpoint

Defensive I/O is used to manage a large simulation run over a period of time larger than the platform’s mean time between failure (MTBF). The simulation regularly dumps the current computation state to a dedicated container used to guarantee forward progress in the event of failures. This section elaborates on how checkponting could be implemented on top of the DAOS storage stack. We first consider the traditional approach relying on blocking barriers and then a more loosely coupled execution.

Blocking Barrier

When the simulation job starts, one task opens the checkpoint container and fetches the current global HCE. It thens obtains an epoch hold and shares the data (the container handle, the current LHE and global HCE) with peer tasks. Each task checks for the latest computation state saved to the checkpoint container by reading with an epoch equal to the global HCE and resumes computation from where it was last checkpointed.

To checkpoint, each task executes a barrier to synchronize with the other tasks, writes its current computation state to the checkpoint container at epoch LHE, flushes all updates and finally executes another barrier. Once all tasks have completed the last barrier, one designated task (e.g. rank 0) commits the LHE which is then increased by one on successful commit. This process is repeated regularly until the simulation successfully completes.

Non-blocking Barrier

We now consider another approach to checkpointing where the execution is more loosely coupled. As in the previous case, one task is responsible for opening the checkpoint container, fetching the global HCE, obtaining an epoch hold and sharing the data with the other peer tasks. However, tasks can now checkpoint their computation state at their own pace without waiting for each other. After the computation of N timesteps, each task dumps its state to the checkpoint container at epoch LHE+1, flushes the changes and calls a non-blocking barrier (e.g. MPI\_Ibarrier()) once done. Then after another N timesteps, the new checkpoint is written with epoch LHE+2 and so on. For each checkpoint, the epoch number is incremented.

Moreover, each task regularly calls MPI\_Test() to check for barrier completion which allows them to recycle the MPI\_Request. Upon barrier completion, one designated task (typically rank 0) also commits the associated epoch number. All epochs are guaranteed to be committed in sequence and each committed epoch is a new consistent checkpoint to restart from. On failure, checkpointed states that have been written by individual tasks, but not committed, are automatically rolled back.

### Producer/Consumer

In the previous figure, we have two examples of producer/consumer. The down-sample job consumes raw timesteps generated by the simulation job and produces sampled timesteps analyzed by the post-process job. The DAOS stack provides specific mechanims for producer/consumer workflow which even allows the consumer to dumps the result of its analysis into the same container as the producer.

Private Container

The down-sample job opens the sampled timesteps container, fetches the current global HCE, obtains an epoch hold and writes new sampled data to this container at epoch LHE. While this is occurring, the post process job opens the container storing analyzed data for write, checks for the latest analyzed timesteps and obtains an epoch hold on this container. It then opens the sampled timesteps container for read, and checks whether the next time-step to be consumed is ready. If not, it waits for a new global HCE to be committed (notified by asynchronous event completion on the event queue) and checks again. When the requested time-step is available, the down-sample job processes input data for this new time-step, dumps the results in its own container and updates the latest analyzed time-step in its metadata. It then commits updates to its output container and waits again for a new epoch to be committed and repeats the same process.

Another approach is for the producer job to create explicit snapshots for epochs of interest and have the analysis job waiting and processing snapshots. This avoid processing every single committed epoch.

Shared Container

We now assume that the container storing the sampled timesteps and the one storing the analyzed data are a single container. In other words, the down-sample job consumes input data and writes output data to the same container.

The down-sample job opens the shared container, obtains an hold and dumps new sampled timesteps to the container. As before, the post-process job also opens the container, fetches the latest analyzed timestep, but does not obtain an epoch hold until a new global HCE is ready. Once the post-process job is notified of a new global HCE, it can analyze the new sampled timesteps, obtain an hold and write its analyzed data to the same container. Once this is done, the post-process job flushes its updates, commits the held epoch and releases the held epoch. At that point, it can wait again for a new global HCE to be generated by the down-sample job.

### Concurrent Producers

In the previous section, we consider a producer and a consumer job concurrently reading and writing into the same container, but in disjoint objects. We now consider a workflow composed of concurrent producer jobs modifying the same container in a conflicting and uncoordinated manner. This effectively means that the two producers can update the same key of the same KV object or document store or overlapping extents of the same byte array. This model requires the implementation of a concurrency-control mechanism (not part of DAOS) to coordinate conflicting accesses. This section presents an example of such a mechanism based on locking, but alternative approaches can also be considered.

A workflow is composed of two applications using a distributed lock manager to serialize contended accesses to DAOS objects. Each application individually opens the same container and grabs an epoch hold whenever it wants to modify some objects in the container. Prior to modifying an object, an application should acquire a write lock on the object. This lock carries a lock value block (LVB) storing the last epoch number in which this object was last modified and committed. Once the lock is acquired, the writer must:

* read from an epoch equal to the greatest of the epoch specified in the LVB and the handle LRE.
* submit new writes with an epoch higher than the one in the LVB and the currently held epoch.

After all the I/O operations have been completed, flushed, and committed by the application, the LVB is updated with the committed epoch in which the object was modified, and the lock can finally be released.

## Storage Node Failure and Resilvering

In this section, we consider a workflow connected to a DAOS pool and one storage node that suddenly fails. Both DAOS clients and servers communicating with the failed server experience RPC timeouts and inform the RAS system. Failing RPCs are resent repeatedly until the RAS system or the pool metadata service itself decides to declare the storage node dead and evicts it from the pool map. The pool map update, along with the new version, is propagated to all the storage nodes that lazily (in RPC replies) inform clients that a new pool map version is available. Both clients and servers are thus eventually informed of the failure and enter into recovery mode.

Server nodes will cooperate to restore redundancy on different servers for the impacted objects, whereas clients will enter in degraded mode and read from other replicas, or reconstruct data from erasure code. This rebuild process is executed online while the container is still being accessed and modified. Once redundancy has been restored for all objects, the poolmap is updated again to inform everyone that the system has recovered from the fault and the system can exit from degraded mode.