Performance Considerations of Persistent State

This page discusses a few critical points to optimize the run-time performance of a Persistent State model.

> If a service never processes more than a few dozen objects in parallel, only triggers a state transition every few seconds and does not rely on response times below one second, there is no need to fine-tune model and configuration for performance. Other criteria like readability of the state machine diagram, robustness of the service and maintainability of the persisted data are much more important.

> Activating **asynchronous trace** for a certain persistent state class will have a **major performance impact** on its processing! However, non-traced object within the same service will behave normal, assuming there are enough system resources to handle the additional load.

Related Pages:

- Persistent States Concept
- Persistent State Components
- License for Running xUML Services

Configuration

Storage Medium

Even though the data storage configuration has a major impact on performance, the choice of a storage medium must mainly consider requirements like data safety upon system failure and scalability in distributed systems. See also chapter **Databases** for a detailed discussion of technical details.
Roughly, the performance of the different storage media is as follows (Desktop computer, Intel, 3GHz, 1 GB RAM, external database on a remote system over 100MBit LAN, 5 workers, 10 concurrent persisted objects):

- **memory/volatile** ~1200 state transitions per second
- **local** ~200 state transitions per second
- **database** ~80 state transitions per second

These numbers represent the raw processing power of the state machine. They were measured with a very simple state machine diagram (1 regular state, 1 choice), a basic activity per transition (decreasing a counter) and small persistent state classes (4 simple attributes, one primary key, one search key). In reality, performance will be lower depending on amount of data and complexity of processes and activities.

Workers

The number of workers defines how many transitions can be processed at the same time. The optimal number is dependent on three factors:

- For models that generate a high CPU load by complex activities or processing of large amounts of data, a low number of concurrent workers are beneficial as it reduces the overhead of the operating system for task switching.
- For models that include some long-running activities, a higher number of concurrent workers improve the response time as multiple shorter activities may be processed parallel to the long running. This is especially valid for long-running transactions that are not CPU bound, e.g. when waiting for an external system to reply.
- Additionally the number of available license concurrency limits the useful number of workers.

  > Each active worker requires one license slot (concurrent connection) to process activities.
  >
  > We recommend to use not more than half of the license slots for persistent state workers. If the workers consume all license slots, the persistent state service in question will be severely afflicted by errors.
  > For more information on concurrent connections and Bridge licensing refer to License for Running xUML Services.

  As well, the required memory to process data, the number of database connections and other external resources used in the activities should be considered. In practice, all these factors limit the number of useful workers to a few dozen (see also xUML Runtime Resources further below).

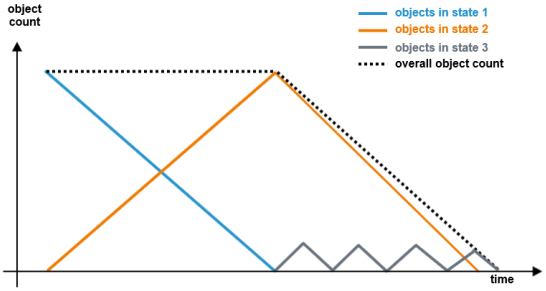
We consider the default of 5 workers is a reasonable compromise. Task switching between 5 concurrent threads is neglectable, even with fully CPU bound activities. Still a certain amount of concurrency is possible, should some of the workers be busy with long-running tasks.
Decrease this number if you want to limit resources, or if concurrency is not a requirement (in which case 1 worker is sufficient). Increase this number if you want less latency for short-running transitions and your system resources are not exhausted yet.

Event Selection Algorithm

Runtime 2019.2 Builder 7.4.0 The event selection algorithm defines in which order the xUML Runtime processes the incoming persistent state events:

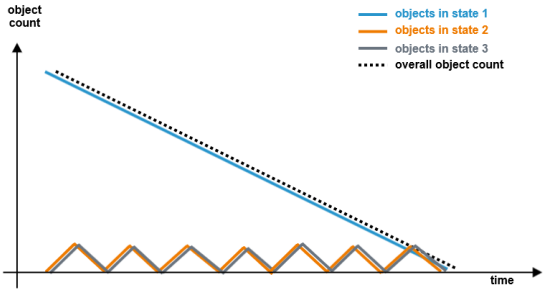
- **Favour Signals**
  The Runtime processes events according to signal appearance. Signals that have arrived earlier - independent which object is concerned - are processed before newer signals.

  The following figure shows a schematic overview on the object/signal processing in this case:



- **Favour Objects**
  The Runtime processes events according to object age. Signals of older objects are processed in advance of signals of newer objects.

  The following figure shows a schematic overview on the object/signal processing in this case:



The overall throughput should be the same whatever algorithm you use, but:

- Comparing the figures above, you can see that with **Favour Objects** objects get deleted earlier. This reduces load on the persistent state engine.
- You should look at your process and decide what is important in your dedicated case: the arrival on a certain state (**Favour States**) or that objects get to the end state fast (**Favour Objects**).
- **Favour Objects** can help if you have states that do not run very well in parallel.
- In general, these considerations are relevant only for persistent state services with a huge amount of events, especially if the count of workers is less than the count of events. This algorithm only is invoked if events are queued (for a definition of event refer to Persistent States Concept).

Note, however, that these considerations are very abstract. How the persistent state engine processes the events in a given scenario is dependent on a variety of factors, e.g.

- what activities are implemented to the particular states and how long these will take
- how many consecutive states are implemented to the state machine

Model

The model has an immediate impact on the performance of the resulting run-time service.

Size and Complexity of Persisted Objects

The persistency layer of the run-time stores persistent state object data (and attributes of signals) as XML into the database. This data is read and parsed with each access to the object, including the processing of transitions. After transitions, the data is serialized and written back to the database. This has multiple implications on the performance:

- Complex persistent state classes increase the time to serialize and de-serialize object data.

- Large amounts of data increase the size of the XML and therefore increase the time to write and read it to/from the database.
- Both require a large amount of memory resources, which might limit the efficiency of in the case of increased concurrency.

Given these parameters, it is clear that for optimal performance, persistent state classes should only include the attributes required to process an object. Attributes that have no functionality in process control and are rarely accessed during the process, should be stored separately.
E.g. for a persistent state service routing large and complex SAP IDOC's, it makes sense to only map IDOC headers into a persistent state class and keep the IDOC itself as file or in a separate database.

Number of Search Keys

To allow efficient queries in scenarios with many persisted objects, attributes that are stereotyped as search keys are duplicated as single entries in a separate table. These entries are updated each time a persistent state object is serialized and written back to the database, e.g. on creation and after every transition.
Even though the impact of one additional search keys is small (a few percentage points), only attributes that are actually used in queries should be marked as search key. Also occasionally filtering the result set of a wider query is often more efficient than the internal overhead to maintain the search index.
Attributes of the primary key stereotype are only accessed to calculate a unique key. The number of primary keys has only a neglectable impact on overall performance.

Few Complex States vs. Many Simple States

The processing of each transition involves loading, de-serializing, serializing and storing the data of a persisted object. Additionally other internal actions like updating the object state, discarding and creating time-out events and the dispatching of state transitions itself take place.
This overhead can be reduced by limiting the number of states in a process to the states in which an object has to wait for external input. All steps that can be executed in a synchronous manner are consolidated into fewer but more complex activity diagrams.
However, there are drawbacks to such a consolidation of states. First, the state machine diagram might no longer resemble all the logical steps of an object's given business life cycle. Second, the complex activities will take longer to complete and therefore may reduce the responsiveness of the state machine, especially with a limited number of workers.

Fork and Join

After fork, the run-time internally creates a new token and the persistent state object has two active states, adding slight overhead in dispatching events. When a token reaches a join state, an internal event is generated and upon its dispatching the join condition is tested (did all required tokens arrive?).
Due to this complexity, it is recommended to avoid parallel states if there is no true parallelism in execution.
A typical case for true parallelism is accessing of two slow external systems at the same time to improve latency. However, if latency is not an issue, the access could as well be serialized.

Do Activities

To allow asynchronous processing, internally start and completion of do activities will create events. These events take a certain time to be dispatched and processed. This may have a noticeable performance impact in latency and system load.
Therefore, it is recommended to use do activities only for activities that block a persistent state object for too much time. Be aware that in practice blocking, long-running transitions are only an issue in two cases:

- The object has other work to do in a parallel state.
- The activity takes place in the first state right after object creation.

The first case is a valid use of do activities, actually the reason for its implementation. The second case can be avoided by an interim state and a completion transition (transition without triggering event, executed asynchronously as soon as possible) – however, the performance of such a construct has no advantages over a do activity.

Composite States

Similar to a fork, entering a composite state creates a new token, resulting in a minor overhead during event dispatching. While exiting a composite state is not as expensive as a join, still a few steps are required to clean up active sub states. This overhead is larger when the sub state machine uses fork/join or additional composite states.
In most real-word cases, a refacturing of composite states will not have a discernible performance impact.

xUML Runtime Resources

At run-time, the processing of persistent state objects needs system resources. Even though CPU time is finite, an overloaded system will slow down but not fail with errors. However, breaking the given limits of memory and external resources like database connections might lead to system failure.
The required amount of memory is dependent on the following factors:

- Number of events being concurrently processed, limited by the number of configured workers.
- Size of objects being processed.
- Moreover, if memory is configured as data storage, the number and size of persisted objects and pending events.

The number of workers limits the required number of database connections (or any other external system accessed in state transition activities).
As of now, the Bridge will only use one database connection per deployed service to access the persistent state database.