03/05/2018

COMPARATIVE ANALYSIS OF TWO SOFTWARE PROCESSeS

# PRoposal

The project works on how we can model and observe how close the two software processes are by basically generating the UML Model State Machine Diagrams from the code or behavioral aspect of the software application to be considered. From the same we can observe how the two software processes are similar to each other by borrowing the base from each other or borrowing the base and building the base of the other.

The Two Software Projects Considered are:

QZMQ Task Distribution Framework

Zookeeper Multiple Processes Handling Using Queues.

# Software PRocesses Taken in consideration:

Task Distribution Framework by QZMQ.

Zookeeper Queues.(C Implementation ZQC model).

Zookeeper Distributed Queues(Java Implementation ZKDJ model)

## Tools Used

### Eclipse IDE.

### Papyrus UML.

### State Machine Diagrams.

### GitHub Source Code.

# SOURCE OF CODE AND BEHAVIOURAL ASPECT OF THE MODELs

The source for the Zookeeper Queues and Distributed Queues Implementation is from the GitHub Repository of Apache Zookeeper. Links for the same are as:

<https://github.com/apache/zookeeper/blob/master/src/recipes/queue/src/c/src/zoo_queue.c>

<https://github.com/apache/zookeeper/blob/master/src/recipes/queue/src/java/org/apache/zookeeper/recipes/queue/DistributedQueue.java>

The Behavioral Aspects of the QZMQ framework have been taken from the State Machine Diagrams given in the paper about the Task Distribution Framework(TDF).

# NOTATIONS USED IN THE STATE MACHINE DIAGRAMS DESIGNED

The following notations are useful in understanding how the notations for how:



# Results of the comparison modelling and the behavioral model given:

The state transition if fully understandable based on the variations available in the source code, and on a similar note the state machine diagrams(SMD) are designed.

The SMD are complex in some ways and the use of multiple states in a single SMD are designed. And on a similar note when needed the State Machine Diagrams are single and atomic.

The initial observation shows the QMZQ TDF is not exactly implemented in the same manner as the original Zookeeper Framework of deploying the Distributed Framework.

But the second observation that can be made is that QZMQ is a subset of the original Zookeeper framework concept of blocking and waiting the processes/nodes in the queues. The areas of variation for the same will be elaborated in detail.

From the observations made on the TDF state machines which are developed based on the fact that it uses the concept of wait till Queue is empty and then proceeds only when the queue is empty but uses its own mechanism we can understand the QMZQ TDF is indeed a subset of the original Distributed Queue Framework initially laid out by Apache Zookeeper & improved on later by open source developments.

# VARIATIONS OF THE QZMQ TDF from APACHE ZOOKEEPER IMPLEMENTATION:

WHEN ASSIGNING THE TASK: The important thing to consider here is how do we detect that there is a new task in the system. Zookeeper does not give us the option to get the last created child. Hence, every time that a new task is created, we have to take the entire input\_task children list and compare it with the assigned\_task. If we treat the lists as sets of elements, u\_list (unassigned\_list) contains a list of unassigned tasks.

Many things may happen to a task:

* A new task is submitted by a client in input\_task
* The task has been executed and hence deleted from both the  input\_task and the assigned\_task znode
* A worker crashes while executing a task. A different worker has to execute it again.  If the system detects that there are no unassigned\_tasks, it has to wait until something happens. However, Zookeeper Child Watches do not give us too much information. They just notify that something has happened to the children. This is very inconvenient! Every time that we receive a watch, we have to get the entire list of input\_tasks and assigned\_tasks.
* In a very busy environment with lots of tasks, this fact would create a scalability problem. Our system can work in small-medium architectures, but not in very large systems, since the message payload increases with the number of input tasks and the number of workers. In fact, we can detect the same problem while trying to find our predecessor in the worker\_queue.

WHEN EXECUTING THE TASK: The most important detail is how do we have to submit the task. We have to do three different things:

* A: Delete the task from the input\_task list
* B: Delete the task from the assigned\_task list
* C: Submit the task in the proper client queue
* Zookeeper can fail at any state.In general, the computer may crash also at any moment. If we execute these operations one by one, and the system fails, the system would remain in an inconsistent state.
* To implement a distributed queue in ZooKeeper, first designate a znode to hold the queue, the queue node. The distributed clients put something into the queue by calling create() with a pathname ending in "queue-", with the sequence and ephemeral flags in the create() call set to true. Because the sequence flag is set, the new pathnames will have the form \_path-to-queue-node\_/queue-X, where X is a monotonic increasing number.
* A client that wants to be removed from the queue calls ZooKeeper's getChildren( ) function, with watch set to true on the queue node, and begins processing nodes with the lowest number. The client does not need to issue another getChildren( ) until it exhausts the list obtained from the first getChildren( ) call. If there are are no children in the queue node, the reader waits for a watch notification to check the queue again.

On a concluding note from the above we can observe that the QZMQ is a subset of the implementation of Zookeeper but does improve on the base set by Zookeeper Distributed Queues Implementation of the same.