RTDroid Alarm Manager

Design Document

Vaibhav Dwivedi (vdwivedi@buffalo.edu)
Siddharth Sinha (ssinha4@buffalo.edu)
Imran Bijapuri (imransay@buffalo.edu)
Abhijit Pattanaik (abhijitp@buffalo.edu)

PROBLEM STATEMENT

In order for an alarm to provide real time guarantees it should provide the mechanisms to do the following:

- Prioritize alarms based on the priority of the thread, task or application that registers the alarm.
- Handle scheduling of many low priority alarms without missing deadline of incoming high priority alarm.
- Prevent registration of an alarm that triggers after a long period.
- Implement an efficient and concurrent data structure for registering, deregistering and retrieving alarms.
- Provide bounds on memory that can be used for alarm allocation.
- Provide bounds on the number of handlers that are released for handling of alarms.
- Prevent the delays case due to Garbage Collection by using immortal and scoped memory.

Thus, the alarm manager must perform optimally even in the worst case scenarios and provide real time guarantees under all conditions. So, we continued to analyze the design proposed in a quest to find better guarantees and bounds. We considered the memory occupied by each data structure and the time involved for each operation from a worst case perspective and tried to find a solution for each shortcoming.

INTRODUCTION

In this documentation, we will look into the things that have been implemented as proposed in the checkpoint 1 and 2. We will also look into the deviations that have been taken from our original proposal to make the implementation more efficient.

ALARM TREE

In our previous implementations a doubly nested TreeMap was being used as a container to store the registered alarms. Though it was an efficient and fast mechanism for retrieval for alarm objects it also had many disadvantages such as -

- Unnecessary burden on memory due to the generic architecture of TreeMap.
- Extra memory needed to store timestamp and priority.
- Delays in instantiation of subtree and linked list containers to store alarm objects.

To solve this problem we decided to implement our own implementation of tree that would solve the above mentioned problems. Since we used the data types specific to the alarm objects the problem of generic architecture was avoided.

To avoid storing keys for insertion, deletion and retrieval of alarms we decided to represent alarms as keys. This was made possible by introducing four new links in the RealtimeAlarm class which refer to other RealtimeAlarm objects. These links are leftLink, rightLink, downLink and backLink.

LeftLink and RightLink – These links are used to store the left and right nodes in the alarm tree.

DownLink – In order to bring a doubly nested structure to our alarm tree, this link was introduced. When an alarm with a previously inserted timestamp but a different priority is inserted into a tree, it follows the downLink of the alarm node in the upper level tree to reach the lower subtree where it is then stored on the basis of priority ordering. This provides a mechanism to create a doubly nested subtree without the using a TreeMap.

BackLink – When an alarm is inserted in the tree which has a priority and timestamp similar to previously inserted alarm object, then a linked list like structure is needed to store the new alarm. This mechanism is provided by using the backlink, where alarms with similar timestamp and priority are appended to each other using the backlink to form a list like structure.

ALARM POOL

When a new alarm has to be instantiated, a new object has to be instantiated. This may induce some delays in the program. To avoid this issue an alarm pool was created to store a previously instantiated alarm objects which then returns an alarm when requested. In older implementations the alarm pool was stored as a node of the TreeMap, which was part of the main tree. But since the tree is locked when an alarm is requested from the pool, in the latest implementation, the pool is created as a separate node independent from the main tree. The pool has a list like structure by using the backlinks of the alarm objects.

Now, when an alarm is requested and the pool has no more alarms to offer, it must then traverse the tree to find an alarm with lowest priority and farthest timestamp and then reset it with the newest parameters and then reinsert it into the tree. Traversing is a costly operation and hence an auxiliary tree was introduces to solve this problem.

AUXILIARY TREE

Auxiliary tree is an inverted main tree in which the higher level tree is stored on the basis or priorities and the lower level tree is stored on the basis of timestamps. This allows us to easily traverse a low priority alarm and then find the farthest timestamp in its substructure. In the older implementations, a new doubly nested TreeMap was used to represent this auxiliary tree. But this leads to unnecessary memory overhead. To solve this issue the four links in the Alarm Class are replicated for the Auxiliary tree as well. Thus the alarm object has four links which are part of the main tree and another four links which are part of the auxiliary tree.

SCHEDULING MECHANISM

In the existing implementations of the RTDroid scheduler, the scheduler pulls the next timestamp from the tree and then schedules it by assigning its pending intent to the handler which is then passed to an event. They are wrapped under a real time thread which sleeps for the time equivalent to the alarm trigger time, after which the event is triggered.

This creates a problem in which we have too many independent handler threads in the memory at a given time even though they may be triggered at a very later stage. To solve this problem, we pull the next timestamp that has to be scheduled from the tree and then find the difference between the trigger time of that alarm and the current time and make the scheduler sleep for that given amount of time. Once the scheduler wakes up it assigns the alarm to a handler thread which then executes the pending intent.

A problem that may arise because of it is that when a new alarm is registered with a trigger timestamp less than the timestamp of the alarm for which the scheduler is sleeping for, the new alarm may never trigger. Thus every time a new alarm is registered it sends an interrupt to the sleeping thread which leads to the scheduler looking for a next timestamp again thereby allowing it to find the newly registered alarm. When the alarms have been assigned their handlers the timestamp is removed from both the schedule tree and auxiliary tree and then the next timestamp is fetched from the tree. If there are no more timestamp to be pulled, the scheduler goes in the waiting state which is then notified when a new alarm is registered.

SCOPED MEMORY

One of the main weaknesses with standard Java, from a real-time perspective, is that threads can be arbitrarily delayed by the action of the garbage collector. GC may have significant impact on the response time of a RT thread. RTSJ solves this problem by allowing objects to be created in memory areas other than the heap. These areas are not subjected to GC.

The memory areas free from garbage collection are classified into two types

- Immortal Memory (objects with unbounded lifetimes).
- Scoped Memory (objects that must be garbage collected).

For an efficient scope memory implementation of RTDroid Alarm Manager, we took into consideration the four types of memory in RTSJ and analyzed the benefits of using each one of them with respect to our new RTDroid code base. We primarily focused on the regions that can be put into immortal memory and scope memory that could provide some benefits in terms of memory. We also considered various other significant properties of these memory types such as forbidden access and dangling references.

Alarms in Immortal Memory

When the alarms are instantiated in the pool, they are instantiated in the immortal memory. The primary advantage of Immortal memory is that objects created in any immortal memory area are unexceptionally referenceable from all Java threads, and all schedulable objects, and the allocation and use of objects in immortal memory is never subject to garbage collection delays. This allows for an uninterrupted and delay free access to all alarm objects from all points in the program at any given time.

Events and Handlers in Scope Memory

- Events and handlers, in the existing implementation are created in the heap memory. Since they get collected by the GC once the alarm has been triggered, it may induce delays in the code.
- In the latest implementation the event and handlers are instantiated in the scope memory.
- The scope memory consists of LTMemory with the size proportional to the no of alarms in the pool.
- The events and handlers are wrapped in a wrapper which when receives the alarm, instantiates new event and handler with corresponding priority, timestamp and runnable logic specific to the received alarm and then enters it to the LTMemory.
- When the alarms have been triggered, the event and handlers that were used to trigger the alarm are no more required and thus the reference count of scope reaches zero making it available for garbage collection.

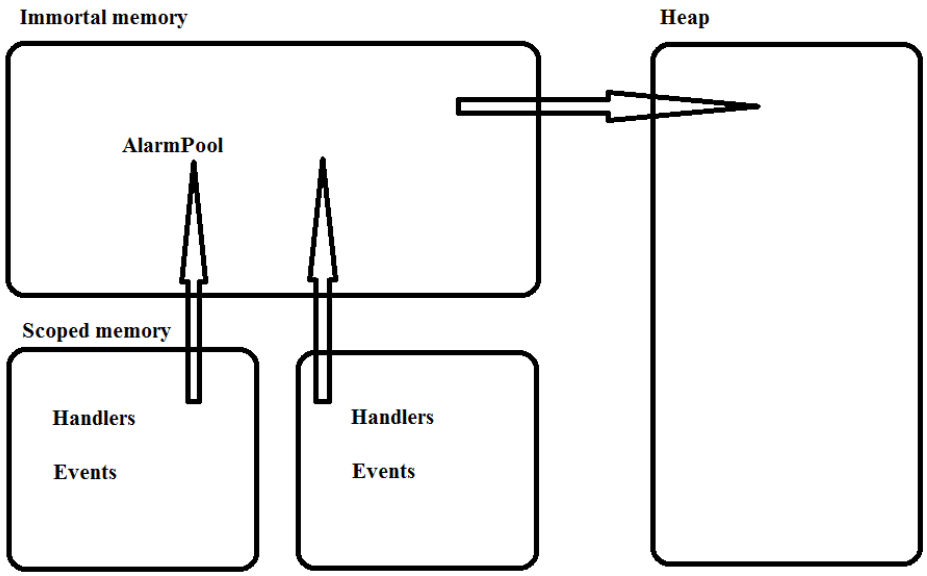

Figure 1 - The alarm pool is created in the immortal memory. The alarms are referenced by the events and handlers present in the scope memory and the threads present in the heap memory.

BENCHMARKING

We carried out rigorous latency testing and analysis to ensure correctness of the Realtime System. The latency tests were configured in the following way:

1. Running an iteration of tests in which, the number of alarms registered were equal to the number of iteration. Therefore, in the first iteration a single alarm was registered. In the second iteration, two alarms were registered and so forth.

2. We allowed for ample recovery time between each iteration such that all alarms were executed and the system didn't incur any latency overhead pending from the previous iteration.

3. To capture the latency values each "alarm" writes its values to a separate file. The benefits of writing to a separate file are that it prevents the latency to be affected by other alarms in case of a single data structure or single file approach. The synchronization overhead incurred is not deterministic. Whereas using a separate file for each alarm only adds a constant overhead for each file write operation that is deterministic and not affected by the presence of other alarms.

4. The various latency values written to these file are:
   - Time at which the alarm is registered.
   - Time of alarm fire.
   - Time taken for alarm to be registered.
   - Time at which the alarm was actually fired.

5. These values from several alarms from each iterations are then consolidated in a file and the delay in registering each alarm is calculated. We then select the maximum latency value (worst case analysis) from each iteration and plot the latency curve.

6. This method of testing was carried out on all forms of implementation including the TreeMap, Alarm-as-a-Tree and Alarm-as-a-Tree with scoped memory approach.

NOTE: Please refer to the README file on how to configure and run the tests.

GRAPHS AND OBSERVATIONS:

The following graphs represent the worst case analysis of our different implementations under different stress tests.

1. TreeMap

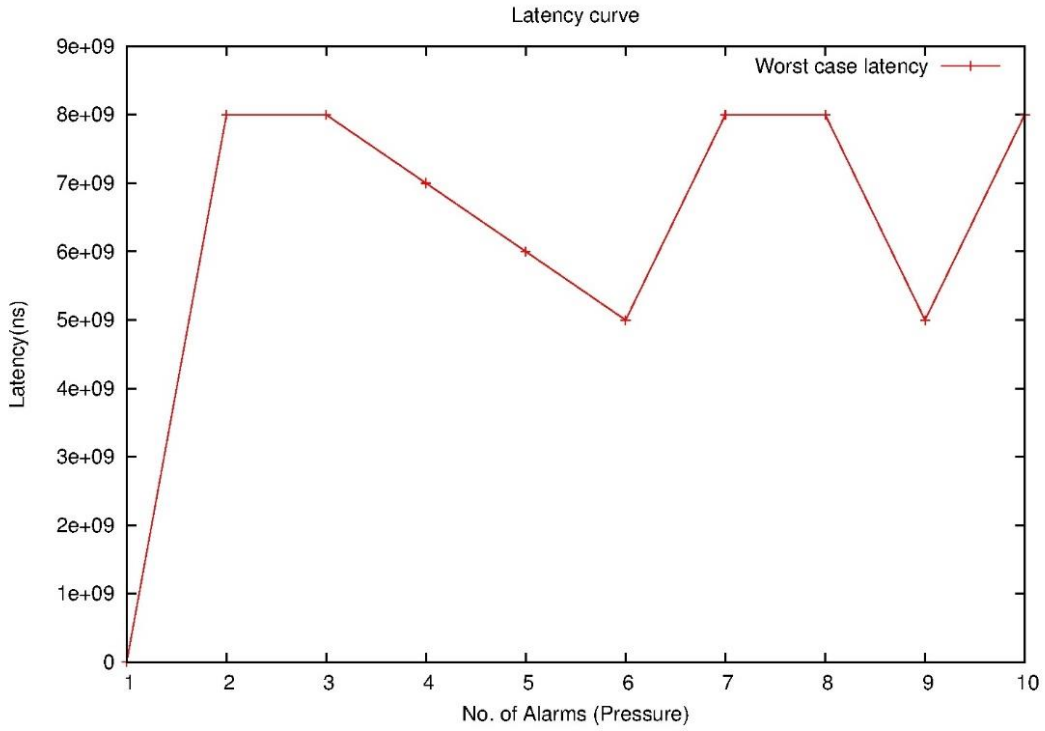

FIGURE 2 – Latency of TreeMap for multiple low priority and high priority alarms.

Here we can observe that the latency values fluctuate drastically. This can be attributed to the instantiations of the TreeMap and LinkedList containers to store the alarm objects. However the latency values do not increase linearly with pressure and there is a maximum bound for the latency. Hence, this shows that the implementation adheres to the Realtime specifications and thus can be used in a Realtime setting.

2. Alarm Tree

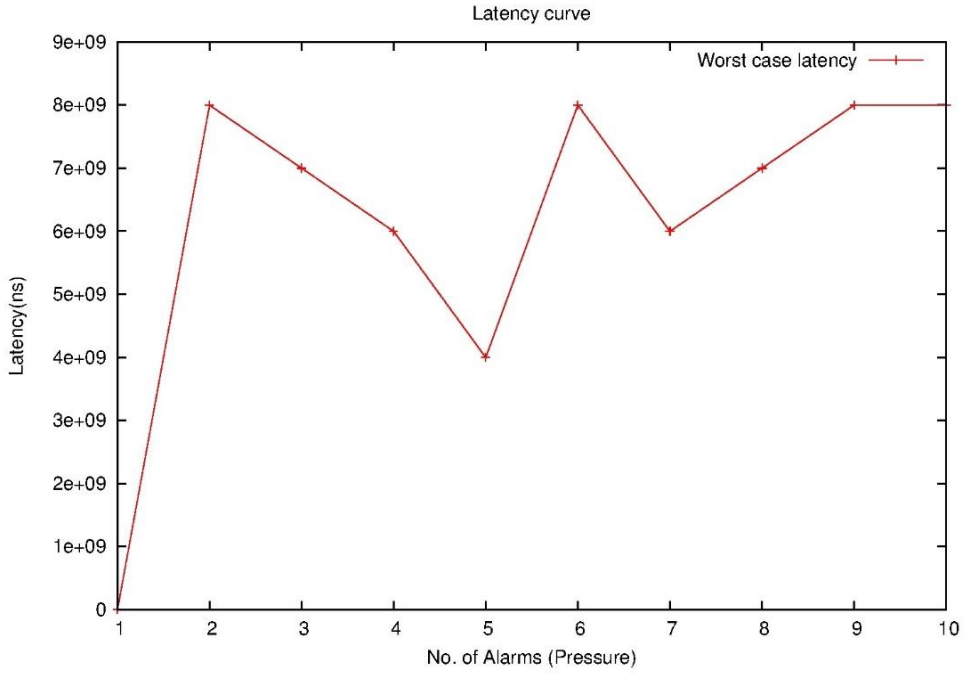

FIGURE 3 – Latency of Alarm Tree for multiple low priority and high priority alarms.

As compared to the previous graph, here the average latency values observed are lower. This can be attributed to the improved performance of using the alarm as a tree. Though we can observe that the number of crests representing the worst case latencies of the alarms remain nearly identical. This can be attributed to the GC delays induced by the collection of Event and Handler objects after the alarm has been triggered.

3. Scope

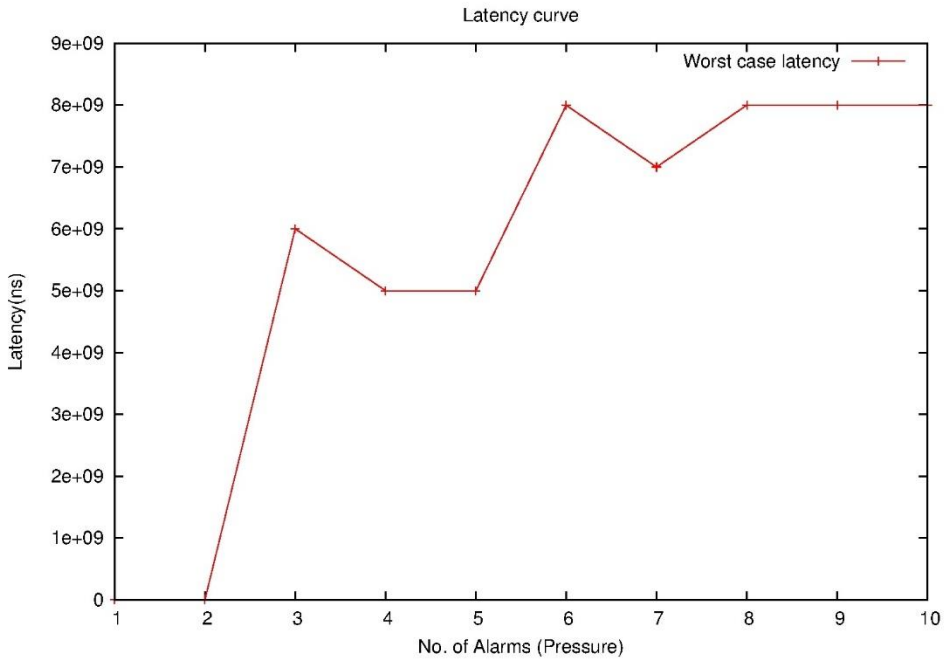

FIGURE 4 – Latency of Alarm Tree with Scope memory for multiple low priority and high priority alarms.

The values observed here are far more consistent than both the above implementations and there is also a reduction in the number of crests representing the worst case latency of the alarm. This can be attributed to the usage of the immortal and the scope memory to store the alarm and handler objects respectively.

4. TreeMap - Stress test of high priority alarm

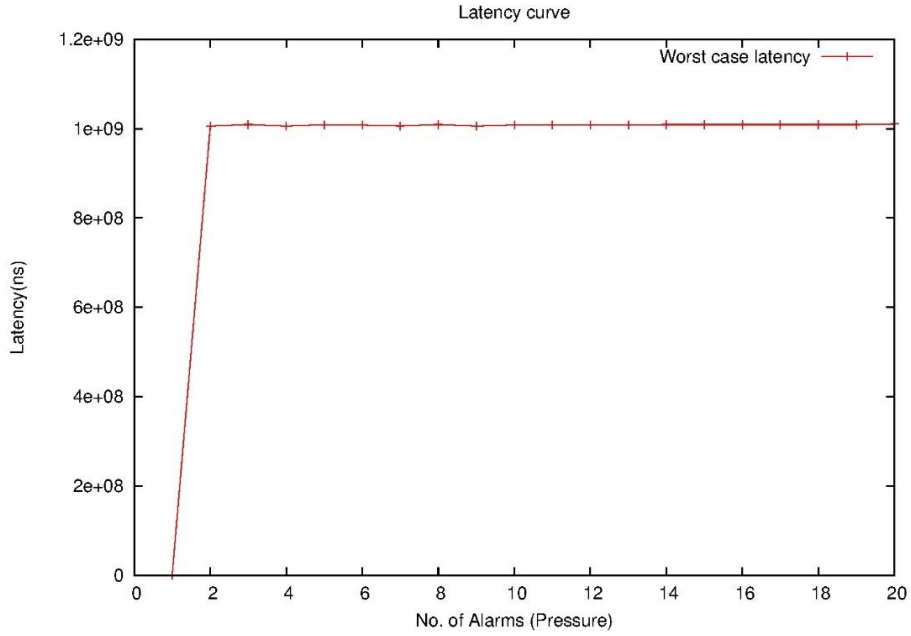

FIGURE 5 – Latency of Alarm Tree for multiple low priority and one high priority alarms measured in a small time interval.

This graph measures the latency of the high priority is alarm when it is triggered along with multiple low priority alarms. Here we can clearly see that the number of low priority alarm does not affect the latency of the high priority alarm thus indicating strict adherence to Realtime specification of scheduling of high priority alarm.

5. Alarm Tree - Stress test of high priority alarm

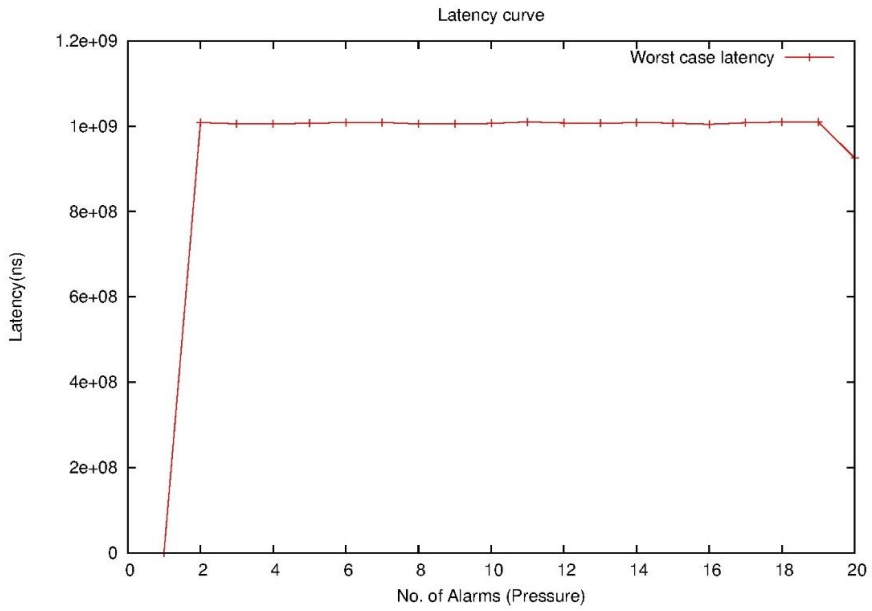

FIGURE 6 – Latency of Alarm Tree for multiple low priority and one high priority alarms measured in a small time interval.

As we can clearly see from the above graph that the alarm tree is equally as efficient as the TreeMap implementations when it comes to scheduling a high priority alarm on time when subjected a stress test of multiple low priority alarms.

CONCULSION

Thus based on above observations we can conclude that our Alarm tree implementation provides an optimum structure for scheduling of alarms under different conditions. We can also conclude that storage of alarms in immortal memory and handlers in the scope memory free the program of GC delays thereby improving the overall performance of the system.

LITERATURE

RTDroid: http://rtdroid.cse.buffalo.edu/

RTDroid provides real-time guarantees with Android's like programming model. To provide timing guarantees, RTDroid leverages an existing real-time kernel with a real-time Java Virtual Machine, and redesigns Android's core message passing components and a subset of its system services with priority awareness.

FijiVM: http://fiji-systems.com/

The Fiji VM™ is a real-time virtual machine providing the features of a modern Java virtual machine with predictability suitable for hard real-time environments. It offers unparalleled flexibility, from per-thread choice between real-time garbage collection with the Schism™ patent-pending collector, GC-less allocation with scoped memory, or a combination of both, to advanced multi-VM features with predictable time and space partitioning.

Perl: https://www.perl.org/

Perl was used for writing the scripts to perform data aggregation and calculations.

GNUplot: www.gnuplot.info/

GNUplot was used for computing the latency curves from the obtained data.

README

```
README : RTDROID ALARM MANAGER USING SCOPED MEMORY
================================================================================
AUTHOR(S): HAKUNA MATATA
================================================================================
STEP 1: COMPILATION
--------------------------------------------------------------------------------
1.1 Change build.properties, set your local fijivm root
1.2 Change fivmc_compile, change to {your local fijivm root}/bin/fivmc
1.3 Change directory to current directory (./rtdroid)
1.4 Run the 'fivmc_compile' script:

$> ./fivmc_compile
--------------------------------------------------------------------------------
STEP 2: RUN THE EXECUTABLE
--------------------------------------------------------------------------------
2.1 The compilation would have created a 'rtdroid-app-demo.exe' file.
Execute it with sudo permissions. Wait for the app to exit after it
completes several iterations.

$> sudo ./rtdroid-app-demo.exe
--------------------------------------------------------------------------------
STEP 3: RUN THE MASTER SCRIPT
--------------------------------------------------------------------------------
3.1 The app would have written several data files with '*.time' format.
This contains the various time and latency values for several iterations
of alarms.
Run the master script to aggregate these values and compute worst case
latency and plot the graph.
*Please refer to masterscript usage for additional details.

$> ./masterscript.pl
================================================================================
FILES:
--------------------------------------------------------------------------------
1. masterscript.pl     : The perl script to collect data and compute
                         worst case latency and plot the latency curve.
2. Graph.ps            : This is the final latency curve obtained after
                         running the master script.
3. *.time              : These are the various files into which the
                         latency is written into by the RTDROID
                         application.
4. latency.data        : This contains the final data points that the
                         gnuplot script uses to plot the latency curve.
5. plot.gp             : The gnuplot script written to plot the latency
                         curve.
6. consolidatedFile.tmp : Temporary files which contains the consolidated
      & latencyFile.tmp   latency values from the several *.time files.

================================================================================
TWEAKS AND CONFIGURATION:
--------------------------------------------------------------------------------
```

1. Number of iterations.
   To change number of iterations and number of alarms to be registered please change the values of related variables in Bootstrap.java class.

** Be sure to clear out the various *.time and *.tmp files when changing the number of iterations. Also copy the graph to another location else it will be overwritten.

==========

MASTER SCRIPT USAGE AND DATA FILES:

Please ensure that the following files are in the same location:

1. *.time
2. plot.gp
3. masterscript.pl

The masterscript can be run from any location as long as the above mentioned files are in the same directory.

==========

SOURCE FOLDERS:

We have included various source folders in the submission. Any of the particular implemenation can be executed by renaming the current 'src' to something else and changing the required implementation as 'src'. Thereafter following the steps given previously. (Be sure to compile!)

1. src
   : Alarm-as-a-tree and Scoped memory implementation,testing with multiple low priority alarms and single high priority alarm.
2. src_scope
   : Alarm-as-a-tree and Scoped memory implementation with multiple low priority alarms and multiple high priority alarms.
3. src_tree
   : Alarm-as-a-tree implemenatation with multiple low priority alarms and multiple high priority alarms.
4. src_treemap
   : Treemap implementation with multiple low priority alarms and multiple high priority alarms.
5: src_high_priority_treemap
   : Treemap implementation with multiple low priority alarms and single high priority alarms.

==========

DATA DUMP FOLDERS:

We have included various data dump folders in the submission. They contain the data generated as the following:

1. dump
   : Alarm-as-a-tree and Scoped memory implementation, testing with multiple low priority alarms and single high priority alarm.
2. dump_scope
   : Alarm-as-a-tree and Scoped memory implementation with multiple low priority alarms and multiple high priority alarms.
3. dump_tree
   : Alarm-as-a-tree implemenatation with multiple low priority alarms and multiple high priority alarms.

4. dump_treemap
   : Treemap implementation with multiple low priority alarms and multiple high priority alarms.
5. dump_high_priority_treemap
   : Treemap implementation with multiple low priority alarms and single high priority alarms.