Introductory Tutorial for the *Task Distribution Framework* Fachbereich 20: Informatik, TU Darmstadt

Jan-Michael Heller

January 7, 2015

Contents

1 Concept

The *Task Distribution Framework* is used to define various tasks that are executed by a set of client machines. An example application would be a web crawler which needs multiple machines to crawl websites and uses the framework to automatically distribute the tasks to the clients and receive results.

The main hub of the framework is a *REDIS* database, where all information are stored. A set of commands is provided to add tasks, receive results and perform various maintenance operations. Each client computer runs one (or multiple) clients, that each listen for new tasks. To divide different projects, namespaces are used, so that tasks can be added to a specific namespace. Clients on the other hand can listen to specific namespaces, if assigning ressources to specific machines is wanted. The only communication between clients and the commands is done via the database. Tasks specification and different inputs for the commands are written in *JSON* to achieve a high interoperability.
The executables for tasks are stored on either a webserver or a file ressource accessible by all clients.

2 Installation

2.1 Compilation and Packaging

You should have received a copy of the projects source code. To compile the project, you have to install the build system *maven*.
Compiling the project is then done, by changing into the java directory in the source directory and issuing:

```
$ mvn package -Dmaven.test.skip=true
```

If maven fails, you probably need to add additional repositories to your maven configuration. Check the error log for needed packages and look for repositories on the internet to add them.

Now you should check the configurations `java/common/target/cmd.properties` for the commands and `java/client/target/client.properties` for the clients and set the database adress accordingly. If you want your clients to listen for specific namespaces only, add them to the variable in the clients properties accordingly.

3 Usage

3.1 Packaging workers

To prepare a *worker*, that clients can retreive and run, a standard *zip*-file has to be created.
It needs to contain at least a script called `setup.sh` and `run.sh`. The setup script is used to perform additional tasks before the worker is run (such as downloading additional files) and needs to return 0 on success. If the setup completed successfully the run script is run. It also needs to return 0 on success.
Keep in mind, that the same worker, if defined in several tasks is not downloaded again but the same working directory is then used for the new task.

Both scripts are called from the framework with three command line parameters:

```
$ {setup.sh, run.sh} [input file] [output file] [temporary directory]
```

The *input file* can be opened and contains information passed by the task definition. The *output file* can be used to store information by the script that will be exported to the database after the task has finished. The temporary directory can be used to store files; it is common for all worker instances of the same type.

`STDOUT` of the worker is used for logging and `STDERR` can be used for error messages.

3.2 Adding tasks

Before adding tasks, you have to create a namespace. Here we create an exemplatory namespace for our later task:

```
$ echo '{"name" : "crawlCat"}' | ./AddNamespace
```

TDF provides two different ways of adding tasks. Clients always work with task lists, so you can either directly provide the command `AddTaskList` with a list of tasks (or a list of a single task) or you add multiple tasks with the `AddTask` command and combine them into task lists afterwards with the `QueueSingleTasks` command.

To add a single task, simply call the `AddTask` command and provide it with a task definition given by the following example JSON:

```
{
"namespace": "crawlCat",
"session": "Students-2021-4-25",
"worker": "file:///tmp/worker.zip",
"input": "",
"timeout": 100000,
```

```
"waitAfterSetupError":100,
"waitAfterRunError":100,
"waitAfterSuccess":10,
"runBefore": "2017-11-24T21:46:48.424+01:00"
}
```

The field *worker* contains the path to the worker. *input* is passed to the run script as `input.txt` (see 3.1). *timeout* specifies a timeout in milliseconds, after which the process is killed and the task is treated as failed. The field *runBefore* specifies a DateTime after which the task will not be run anymore. If the task is added successfully, you are provided with the key under which the task is stored.

After you have added a bunch of tasks, you have to call `QueueSingleTasks` command to combine them into task lists, e.g.:

```
$ ./QueueSingleTasks -n crawlCat -k 10 -e
```

Here the parameter `-n` specifies the namespace, `-k` the size of the list and `-e` can be added if you do not want a last list, which may contain much less tasks than the others, but create the lists possibly of equal length as given by a heuristic.
Please remember that if multiple programs work on the same namespace, calling `QueueSingleTasks` from both of them may cause interferance, because there is only a single list, where newly added single tasks are stored.

Adding task lists simply works by adding tasks as specified above but combined into a JSON list with the command `AddTaskList`. You may as well add a list containing a single task to circumvent additional calls of `QueueSingleTasks` if you only want the clients to process each task seperately.

3.3 Managing results

Issueing the command `ExportProcessed` will provide you with a list of JSON-tasks that have been processed by the clients. They will contain additional fields like *started* and *finished*, which contain a DateTime, which specify when the task has been started or stopped, *output* which specifies the output the script generated and *error* which contains a non-empty string if something went wrong during execution.

If you want to reschedule tasks, that have failed, you can just call the command `Requeue` to readd tasks. Those tasks are added as single task lists to the head of the queue of to be executed tasks.

4 Documentation

4.1 Queues

Each namespace has a node in the imaginary REDIS tree, eg. for an exemplatory namespace called *namespace*: `tdf:namespace`. Under this node all data belonging to the namespace are stored. This regards the tasks and task lists as well as queues, in wich the same are assorted, which will be described here. The queues are all generated as lists and contain keys of the corresponding tasks or task lists. Normally new items are added on the left, which represents the tail of the list and the oldest item is on the right side (the head). This way the lists are often used as queues, s.t. tasks and task lists are processed in a FIFO order. The only exception are task lists which are generated out of failed tasks by the command `Requeue` (see 5.9), which are added to the head of the list `queueingTaskLists`.

The queues are all assorted under the namespaces node, for example `tdf:namespace:queueingTaskLists`.

4.1.1 Description of queues

`failed` This queue contains the database key of all tasks, that have failed in the order of which they have been added.

`newlyProcessed` This lists contains the database key of tasks that have been processed (successful and unsuccessful) and is flushed each time the command `ExportProcessed` is called with parameter **`-u`** (see 5.7).

`newlySuccessful` The same as `newlyProcessed` but only for successfully ran tasks.

`processed` This list contains the database key of all tasks which have been already processed by a client.

`queueingTaskLists` In this queue the database keys of all task lists which are ready to be run are queued.

`successful` This list contains all successfully worked off tasks.

`unmergedTasks` This queue is used to collect single tasks, which have been added by `AddTask` (see 5.2), to combine them into task lists lateron by calling `QueueSingleTasks` (see 5.8).

5 Command reference

5.1 AddNamespace

```
$ AddNamespace
```

5.1.1 Input

Reads a JSON document with the following keys:

name Name of the namespace do add

5.1.2 Description

Creates a namespace into which tasks and task lists can be inserted.

5.2 AddTask

```
$ AddTask
```

5.2.1 Input

A single task definition given by a JSON document like the following:

```
{
 "namespace": "crawlCat",
 "session": "Students-2021-4-25",
 "worker": "file:///tmp/worker.zip",
 "input": "",
 "timeout": 100000,
 "waitAfterSetupError":100,
 "waitAfterRunError":100,
 "waitAfterSuccess":10,
 "runBefore": "2017-11-24T21:46:48.424+01:00"
}
```

namespace defines the namespace the task is added to

session is a session name to recognise the task lateron

worker URL (http or filesystem) to the worker package, see section 3.1 for information on how to pack a worker

input parameters which will be handed to the run script of the worker

timeout in *ms* after which the worker is killed

runBefore DateTime after which the task becomes invalid and is not run anymore

5.2.2 Output

Returns the database key of the newly added task.

5.2.3 Description

Adds a single task to the list of unmerged tasks (**tdf:*namespace*:unmergedTasks**, so it can be collected into a task list by calling `QueueSingleTasks` lateron (see 5.8).
If you simply want the clients to run single tasks, it is most convenient to add task lists with a single entry via `AddTaskList`, see 5.3.

5.3 AddTaskList

```
$ AddTaskList
```

5.3.1 Input

A JSON-List of multiple task definitions, e.g.:

```
[
 {
  "namespace": "crawlCat",
  "session": "Students-2021-4-25",
  "worker": "file:///tmp/worker.zip",
  "input": "arm",
  "timeout": 100000,
  "waitAfterSetupError":100,
  "waitAfterRunError":100,
  "waitAfterSuccess":10,
  "runBefore": "2017-11-24T21:46:48.424+01:00"
 },
 {
  "namespace": "crawlCat",
  "session": "Students-2021-4-26",
  "worker": "file:///tmp/worker.zip",
  "input": "belly",
  "timeout": 100000,
  "waitAfterSetupError":100,
  "waitAfterRunError":100,
  "waitAfterSuccess":10,
  "runBefore": "2017-11-24T21:46:48.424+01:00"
 }
]
```

5.3.2 Output

Returns the database key of the newly added task list.

5.3.3 Description

Add a list of tasks to the given namespace. If tasks of the list contain different namespaces, the namespace of the last task is considered.
The task list is directly added to the queue **tdf:*namespace*:queueingTaskLists** and will be ran, as soon as it hits the tail.

5.4 DeleteNamespace

```
$ DeleteNamespace
```

5.4.1 Input

Reads a JSON document with the following keys:

name Name of the namespace do delete

5.4.2 Description

Deletes a namespace.

Warning! All tasks in the namespace will be deleted, so be sure to export data you still need before (eg. use `ExportProcessed`, see 5.7)!

5.5 DeleteTask

```
$ DeleteTask <taskKey>
```

5.5.1 Parameters

<taskKey> the database key of the task to delete

5.5.2 Description

Removes task from the list of unmerged tasks. Does not work anymore, if task has been already merged by use of `QueueSingleTasks` (see 5.8).

Has lineary runtime.

5.6 DeleteTaskList

```
$ DeleteTaskList <tasklistKey>
```

5.6.1 Parameters

<taskKey> the database key of the task list to delete

5.6.2 Description

Removes task list from the queue of waiting task lists. Only possible until task list is fetched by a client.

Has lineary runtime.

5.7 ExportProcessed

```
$ ExportProcessed [-n <namespace>] [[-u | -f | -s] [-r]]
```

5.7.1 Parameters

-n *<namespace>* only export processed tasks of a specific namespace

-u return every new task only once (useful for batch processing)

-f return all failed tasks

-s return new tasks only once and only show successful tasks

5.7.2 Output

A list of JSON tasks.

A document might look like this:

```
{
 "input" : "",
 "waitAfterRunError" : "100",
 "error" : "run.sh did not return 0\nhave this stderr:\n\n",
 "client" : ""client-0"",
 "session" : "Students-2021-4-25",
 "waitAfterSetupError" : "100",
 "waitAfterSuccess" : "10",
 "worker" : "file:///tmp/worker-bashful-1.zip",
 "finished" : "2014-11-24T21:46:48.424+01:00",
 "started" : "2014-11-24T21:46:17.201+01:00",
 "timeout" : "60000",
 "namespace" : "bashful"
}
```

Compared to the input format (see 5.2) it contains following additional fields:

error empty if successful, contains error messages otherwise (newlines are replaced with
n to fit JSON format)

started time when job started (DateTime, read from clients local clock)

finished time when job finished (DateTime, read from clients local clock)

client is the client which last worked on the task (interally also important for timeout handling)

5.7.3 Description

Export all already processed tasks as a JSON list. Normally contains Tasks of all namespaces but can be limited to a specific namespace by parameter.

If **[**-u] or **[**-s] is set, newly finished tasks are returned, in a manner, such that tasks are only exported once. At best only use one of these options, as the lists for these options are both flushed after calling the command with one of the parameters them.

To keep space in the database it is generally a good idea to call `DeleteTask` after tasks are not needed anymore (see 5.5).

5.8 QueueSingleTasks

```
$ QueueSingleTasks -n <namespace> -k <size> [-e]
```

5.8.1 Parameters

-n *<namespace>* namespace of which tasks shall be merged

-k *<size>* desired list size

-e Try to build equally great list, no sparse last list

5.8.2 Description

Queues tasks that have been added with `AddTask` to the list of unmerged tasks. If **-e** is provided, lists with possibly less than the given number, but of equal size will be created, so that the lists are nearly the same size. Without the parameter, lists of *<size>* will be created until less than *<size>* number of tasks are in the list of unmerged tasks, which will then be merged to a smaller list.

5.9 Requeue

```
$ Requeue -n <namespace> -k <size> [-e]
```

5.9.1 Parameters

-n *<namespace>* specific namespace of which tasks shall be requeued

-k *<size>* desired list size

-e Try to build equally great list, no sparse last list

5.9.2 Output

Linewise database keys of the requeued tasks.

5.9.3 Description

Requeue failed tasks. The tasks will be splitted to task lists similiar to `QueueSingleTasks` (see 5.8). If no namespace is provided, tasks of all namespaces will be requeued.

5.10 RetrieveClientLogs

```
$ RetrieveClientLogs {-c <client> [-b]} | -l
```

5.10.1 Parameters

-c *<**client**>* client of which log messages shall be received

-l list clients that have logs available

-b receive logs blockingly

5.10.2 Output

They output will be linewise. Log messages have the following format:

<unix timestamp>:*<message type>*:*<content (optional)>*

Message types available can be found in the class `e.tuda.p2p.tdf.common.databaseObjects.LogM`

5.10.3 Description

Retrieves client log messages from the database. Log messages are deleted on retrieval.

5.11 Show

```
$ Show <database key>
```

5.11.1 Output

The content of the requested database key. Is either a line-by-line list (most likely of database keys) or a JSON document, representing the contained hashset.

5.11.2 Description

Can be used to view content of certain database keys. It is for example useful, to view generated task lists after creation with eg. `QueueSingleTasks` or to view different queues (see for example 4.1).

5.12 Timeout

```
$ Timeout -n <namespace> [-f|-g <tasklist>]
```

5.12.1 Parameters

-n <***namespace***> specific namespace of which tasklists shall be evaluated

-g <***tasklist***> fail all tasks which have not been run of a specific tasklist

-f fail all tasks of timed out tasklists which have not been run yet

5.12.2 Output

Line-by-line list of tasks which have been marked as failed because client took too long.

5.12.3 Description

Lists tasklists which have timed out. May be used to fail all tasks in a task list which have not been run to requeue them using the `Requeue` command (see 5.9).

WARNING: To use this command, you need meaningful synchronised clocks between the server and all clients. Elsewise behaviour of this command can become odd.