A Simple Command Orchestrator

It is distributed architecture orchestrating multiple commands with dependency on each other to be executed by its dependency rule.

The orchestrator will accept multiple dependent/independent command from an external application and convert all of them as an ordered work flow depending on the command dependency. Then the entire work flow will be executed in external work flow execution server. The orchestrator will chose a work flow execution server (micro service) and request the work flow execution in the execution server.

Workflow Engine

Orchestrator

Here each Application will orchestrate set of commands from application and create a ordered work flow depending on command dependencies. Then send those commands to the work flow execution server one by one. Each command execution status will be responded back by the work flow engiene

Design

With this design we will first create a Cli application which will send multiple commands to the orchestrator. These commands will be sent in asynchronous way. i.e. Cli can send commands as and when it wants. Once all commands are sent from Cli Application can query for status and display it with this small design. However it can be further enhanced to aggregate the result and provide the same to Order Service Application, i.e there can be an order service application entire cli application to execute a order.

Low Level Design

1. Create a connection socket with Orchestrator
2. Wait for user/app input
3. Convert command input in to Orchestrator understandable format
4. Send the command to Orchestrator wait for response on successful dispatch of command
5. A show command will be send which will display Orchestrator response on work flow status

Orchestrator Design

Orchestrator will be central unit. It will accept the commands from the Cli session. It will look if the command is a simple Orchestrator known command OR a request to parse XML files containing command. In case of simple command it will build “command\_data” using inbuilt API and in case of XML it will parse the corresponding XML file to build “command\_data” .

The “command\_data” will be then checked against dependency with other in progress “command\_data”. When the “command\_data” become independent (free from its dependency on other command), it will be delivered to “Execution Server” for execution. Orchestrator will wait for each execution status and record them

Each “command\_data” will be dispatched with a separate thread context and the thread will be responsible for dispatching the “command\_data”, waiting for response and recording the response

Low Level Design

1. Initialize Orchestrator
   1. Create 3 queues “RUN-QUEUE”, “PENDING- QUEUE” and “COMPLETE- QUEUE”
   2. Each of these QUEUE are a linked-list to store “command\_data”
   3. “RUN-QUEUE” will store the command which are dispatched and waiting for response
   4. “PENDING- QUEUE” will store commands which are not dispatch ablecurrently
   5. “COMPLETE- QUEUE” will store commands which are completed by “execution server”
2. Initialize a Worker Thread for serving “PENDING- QUEUE”
   1. Whenever “command\_data” in the “PENDING- QUEUE” become ready the Worker Thread will wake up and server the “command\_data” from “PENDING- QUEUE”
   2. Once wake up Worker Thread will go through all the items in the “PENDING- QUEUE” schedule those for dispatch
   3. It will create dispatch thread for all runnable “command\_data” and remove those from “PENDING- QUEUE” which will be put into “RUN-QUEUE”, by dispatch thread
3. Create connection to the “Execution Server”
   1. It has ability to simultaneously connect to multiple “Execution Server”, however its under development
4. Created a Cli Server Thread
   1. This thread is responsible for serving multiple Cli sessions
   2. This is a server instance which will wait for Cli client connection and serve them
   3. This is the thread where Cli connects to and do all transaction
   4. It accepts the command and parses the command (simple/XML) and build “command\_data”
   5. Once “command\_data” is made it dispatches it Dispatch Thread
5. Dispatch Thread (Dynamic Threads)
   1. Dispatch Thread are the Dynamic Threads which carry out major responsibility and multiple such threads run concurrently
      1. Each thread handle one single “command\_data”
      2. It checks the execution eligibility of the “command\_data”
      3. Eligibility computing is done by comparing current “command\_data” dependency with all “command\_data” , which is there in “PENDING- QUEUE”
      4. If a “command\_data” RUNNABLE, it is put in to “RUN-QUEUE”, and immediately sent to “Execution Server” for execution of command
      5. If “command\_data” is not RUNNABLE, it is put in to “PENDING- QUEUE”
      6. It sends the command to “Execution Server” and wait for status
      7. Once it receives the response from “Execution Server” the “command\_data” is removed from “RUN-QUEUE” and put into “COMPLETE- QUEUE”
      8. Upon completion of “command\_data” execution the thread signals the Worker Thread to do Eligibility computing of items in “PENDING- QUEUE”
6. Simple Command Parser
   1. It is a API which parses the Orchestrator known command cli commands and generate “command\_data” for orchestration
7. XML parser
   1. It’s a “libxml2” based API parses the XML file and generate “command\_data” for orchestration
8. QUEUE Management
   1. Each queue will have lock
   2. Eligibility computing for “command\_data” is done by taking those locks
   3. Multiple locks can be taken
   4. Each “command\_data” will be distributed across “RUN-QUEUE”, “PENDING- QUEUE” and “COMPLETE- QUEUE” based in their eligibility
   5. “RUN-QUEUE” should only hold “command\_data” which are in progress and already sent to “Execution Server”
   6. “PENDING- QUEUE” should only hold “command\_data” which are not RUNNABLE, i.e it has dependency on currently running “command\_data” which are on “RUN-QUEUE”,
   7. “COMPLETE- QUEUE” should only hold “command\_data” which are executed successfully by “Execution Server”
   8. “COMPLETE- QUEUE” is solely for display and work flow status calculation
   9. All queue should be made per session, However it is under development now

Execution Server Design

This is the service create multiple Execution Server as a micro service. Each server just accepts the “command\_data” from Orchestrator, execute them and send the execution status to Orchestrator.

Low Level Design

1. Create multiple server running on different ports
2. Each server waits for a connection from Orchestrator
3. Upon Orchestrator connection it creates a session and wait in the client socket for accepting command and executing it

Build Procedure

Clone the repository

Go to each module and do make

How To Test

Run all the application in separate terminal

$ ./ action-handler

$ ./ orchestrator

$ ./cli

CLI> help

Command Description

---------------- -----------------------------------

help Command Help

quit Quit Cli Session

showr Show Running List

showp Show Pending List

showc Show Completed List

show Show Progress

cleanup Cleanup Completed List

exec0 Execute Command 0 ()

exec1 Execute Command 1 ()

exec2 Execute Command 2 ()

exec3 Execute Command 3 ()

exec4 Execute Command 4 ()

exec5 Execute Command 5 ()

CLI> exec4

Dispatched Cmd: exec1

exec1

CLI> exec5

Dispatched Cmd: exec4

exec4

CLI> exec3

Dispatched Cmd: exec5

exec3

CLI> exec3

Dispatched Cmd: exec3

exec3

CLI> show

=================STATUS===========

RUNNING:

CMD [ 2 | pwd | Present Working Directory ]

CMD [ 1 | ls | List ]

CMD [ 3 | date | Print Date ]

PENDING:

CMD [ 4 | ps | Process Status ]

COMPLETED:

CMD [ 0 | ls -altr | List Time Ascending ]

=================STATUS END===========

show

CLI> show

=================STATUS===========

RUNNING:

CMD [ 2 | pwd | Present Working Directory ]

CMD [ 1 | ls | List ]

CMD [ 3 | date | Print Date ]

PENDING:

CMD [ 4 | ps | Process Status ]

COMPLETED:

CMD [ 0 | ls -altr | List Time Ascending ]

=================STATUS END===========

show

CLI> show

=================STATUS===========

RUNNING:

CMD [ 4 | ps | Process Status ]

CMD [ 2 | pwd | Present Working Directory ]

CMD [ 1 | ls | List ]

PENDING:

COMPLETED:

CMD [ 3 | date | Print Date ]

CMD [ 0 | ls -altr | List Time Ascending ]

=================STATUS END===========

Show

CLI> show

=================STATUS===========

RUNNING:

CMD [ 3 | date | Print Date ]

CMD [ 0 | ls -altr | List Time Ascending ]

CMD [ 4 | ps | Process Status ]

PENDING:

COMPLETED:

CMD [ 1 | ls | List ]

CMD [ 4 | ps | Process Status ]

CMD [ 2 | pwd | Present Working Directory ]

CMD [ 1 | ls | List ]

CMD [ 3 | date | Print Date ]

CMD [ 0 | ls -altr | List Time Ascending ]

=================STATUS END===========

Show

CLI> show

=================STATUS===========

RUNNING:

CMD [ 3 | date | Print Date ]

PENDING:

COMPLETED:

CMD [ 0 | ls -altr | List Time Ascending ]

CMD [ 4 | ps | Process Status ]

CMD [ 1 | ls | List ]

CMD [ 4 | ps | Process Status ]

CMD [ 2 | pwd | Present Working Directory ]

CMD [ 1 | ls | List ]

CMD [ 3 | date | Print Date ]

CMD [ 0 | ls -altr | List Time Ascending ]

=================STATUS END===========

show

CLI> show

=================STATUS===========

RUNNING:

PENDING:

COMPLETED:

CMD [ 3 | date | Print Date ]

CMD [ 0 | ls -altr | List Time Ascending ]

CMD [ 4 | ps | Process Status ]

CMD [ 1 | ls | List ]

CMD [ 4 | ps | Process Status ]

CMD [ 2 | pwd | Present Working Directory ]

CMD [ 1 | ls | List ]

CMD [ 3 | date | Print Date ]

CMD [ 0 | ls -altr | List Time Ascending ]

=================STATUS END===========

Show

Caveats