Message and Architecture description

On the following page you can find a compilation of Message Sequence Charts. The messages required for our communication protocol can be put into two different categories: job-related and mgmt-related messages.

- job related messages
  - cancelJob / cancelJobAck
  - queryStatus / replyStatus
  - jobFinished / jobFinishedAck
  - jobFailed / jobFailedAck

- mgmt related messages
  - submitJob / submitJobAck
  - deleteJob / deleteJobAck
  - requestJob
  - configRequest / configReply
  - lifeSignal

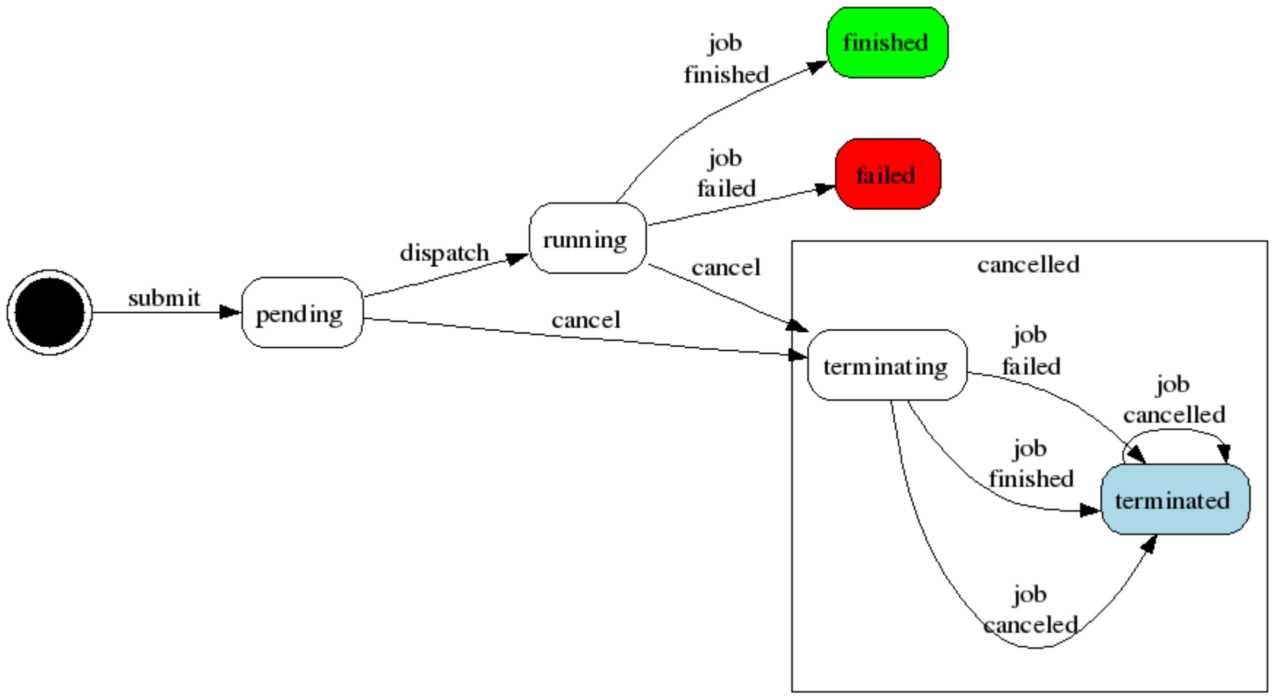
The `deleteJob` message can be seen as both, job related and management, but I decided to put into the latter category because it is more or less the counter part to the `submitJob`.

Job state model

We are using the Job-State model defined by the OGSA-BES group. It is an extendable model where a job can only be in five different states:

- Pending - submitted but not yet scheduled to some resource
  - on the orchestrator: not yet sent to an aggregator
  - on the aggregator: not handled yet
- Running - scheduled and executing
  - O: sent to the aggregator
  - A: workflow-engine handles the workflow and issues new executable activities
- Finished
  - O: the aggregator notified the end of execution
  - A: the workflow engine notified the end of execution
- Failed
  - same as finished but with an unsuccessful outcome
- Cancelled
  - the user requested the cancellation of the task
  - this state may also be entered in case the system was required to cancel the job
    - for example if the orchestrator does not have any aggregator to send jobs to, jobs have to be cancelled by the system

We introduce several new substates to the basic states described above. For example, the `cancel` request issued by an user is "immediate" which means that it will return as soon as possible. But since the execution is distributed we need to inform each involved component about that request.



Job State Model

Message and Protocol Definition of User-Orchestrator communication

The Orchestrator is more or less passive and waits for requests from a user or for notifications sent by the Aggregator component. Basically the user can issue five requests:

- submit, cancel, query, results and delete

Each message is replied to with either an acknowledge or an error indicating the reason for the error which occured. There is currently no kind of security embodied into this part of the protocol.

- `submitJob` (U -> O)
  - job-id (ignored by the orchestrator, see below)
  - contains workflow description and initial tokens
  - contains a flag for simulation
  - parse the workflow (syntax checking) using the builder pattern
    - creates an object-representation of the workflow
  - if something is wrong send an `error` message
  - create a new `Job` object and assign a unique job-id
  - put the job into the job-map
  - send an `submitJobAck` back

- `submitJobAck` (O -> U)
  - contains job-id

- `cancelJob` (U -> O)
  - job-id of the job that shall be cancelled
  - lookup job in the job-map
  - execute the terminate transition
    - send the cancel down to the aggregator
  - move into `cancelling` state waiting for the aggregator
  - send `cancelJobAck` to the user
    - the cancel is acknowledged as soon as possible
    - the user can query the cancellation progress
    - `sdpa cancel` returns immediately, the user does not care when the cancellation is done
    - the client can still query and "wait" until the job reaches cancelled (i.e. `sdpa cancel -w`)
- `cancelJobAck` (O -> U)
  - job-id of job that is to be cancelled

- `queryStatus` (U -> O)
  - job-id of the job whose status shall be retrieved
  - orchestrator looks up the job and returns is current status
  - maybe ask the aggregator for more details?
  - send a `statusReply` message back
- `statusReply` (O -> U)
  - job-id
  - status as an integer
  - status as a text-string
  - additional information
    - maybe some statistics like executed transitions, tokens generated/consumed, tasks waiting in queue

- `retrieveResults` (U -> O)
  - job-id
  - we keep the information about a job around until the user explicitly deletes it
  - compare this behaviour to a unix `wait(pid)` system call
  - however, there must be a timeout after which we just delete the entry and maybe send an email/notification to the administrator stating that a job had been removed etc.
  - send a `results` message back
- `results` (O -> U)
  - job-id
  - output token descriptions
  - with the output token descriptions the user client can figure out where the output data has been generated and stored

- `delete` (U -> O)
  - job-id
  - the user-client can remove the job entry after the results have successfully been retrieved
  - the orchestrator sends an `deleteAck` or an error back
  - this operation is only successfull when the job is in one of the following states:
    - Cancelled, Failed, Finished
- `deleteAck` (O -> U)
  - job-id

- `error` (O -> U)
  - this is a generic error message, containing an `error-code` and a human readable `error-message` indicating the kind of error
  - additional data maybe attached as well

On the following page you can see some details on the implementation details.

Message and Protocol Definition of Orchestrator <-> Aggregator communication

The Orchestrator manages a high-level view on submitted jobs from a user. It just keeps track of the current state of a given job and sends it down to the aggregator for execution. The aggregator then manages the execution of the complete workflow. The basic procedure of job-execution is based on polling, maybe we add a push later, but since the NRE will use polling, too, we can reuse some code parts.

- `lifeSign` (A -> O)
  - timestamp, load, other probably useful information
  - last_job_id the id of the last received job identification
  - the aggregator first sends a request for configuration to its orchestrator
  - the orchestrator allocates an internal data structure to keep track of the state of the aggregator
  - this datastructure is being updated everytime a message is received
  - an aggregator is supposed to be unavailable when no messages have been received for a (configurable) period of time

The following three messages are closely related to each other. The orchestrator sends a new job down after it has received a request (polling) from an aggregator. With this model and the attached 'last_job_id' (also in the lifeSign) we can make sure that every of those messages can be lost at any time and still have the same view on both sides.

- `requestJob` (A -> O)
  - the aggregator requests new executable jobs
  - this message is sent in regular frequencies depending on the load of the aggregator
  - this message can be seen as the trigger for a submitJob
  - it contains the id of the last job that has been received
  - the orchestrator answers to this message with a `submitJob`
- `submitJob` (former: `pollReply`) (O -> A)
  - the submitJob message contains an executable entity, i.e. a workflow description
  - job-id
- `submitJobAck` (A->O)

  - contains the job-id
  - Orchestrator can remove the job from the pending queue of the scheduler

- `jobFinished` (A -> O)
  - job-id and the generated output tokens
  - when a task completes execution, the orchestrator is notified about that
  - completion is indicated by the workflow engine
  - Orchestrator replies with an ack or an error (unknown job for example)
- `jobFailed` (A->O)
  - job-id and zero or more token describing the failure?
  - Orchestrator replies with an ack or an error

- `configRequest` (A->O)
  - on startup the aggregator tries to retrieve a configuration from its orchestrator
- `configReply` (O->A)
  - the reply message contains the configuration data for the requesting aggregator
  - TODO: what is contained in the Configuration?

Message and Protocol Definition of Aggregator <-> NRE communication

The communication between an NRE and the Aggregator is nearly the same as the communication between Aggregator and Orchestrator.

- `configRequest` (NRE->A)
  - on startup the NRE tries to get the configuration from its aggregator
- `configReply` (A->NRE)
  - the reply message contains the configuration data for the requesting NRE
  - TODO: what is contained in the Configuration?

- `lifeSignal` (NRE->A)

- `poll` (NRE->A)
- `pollReply` (A->NRE)

- `jobFinished` (NRE->A)
  - job-id
  - generated tokens
  - maybe we can improve this by sending one finished for a bunch of jobs
- `jobFailed` (NRE->A)
  - job-id

User client

The user client has a very simple command line interface:

- `sdpac submit <path>`
  - prints the job-id and exits with exit-code 0
  - prints the error message and code and exits with non-0 exit-code
- `sdpa simulate <path>`
  - simulate the given workflow
  - behaviour is the same as `submit` but an additional flag in the submit message is set

- `sdpac cancel #id`
  - sends the cancellation request to the orchestrator
- `sdpa cancel -w #id`
  - sends the cancellation request to the orchestrator and waits until the job enters state `Cancelled`

- `sdpac status #id`
  - request the status of the given job
- `sdpac status -d #id`
  - request a detailed status of the job (involving aggregator query)

- `sdpac results #id`
  - request the result retrieval
- `sdpac delete #id`
  - delete all information about the job

Overview of messages and their data

SubmitJob

- workflow description
  - serialized Workflow (type: string)

- job-id (only allowed for A->O or A->N communication)

SubmitJobAck

CancelJob

CancelJobAck

DeleteJob

DeleteJobAck

JobFailed

JobFailedAck

JobFinished

JobFinishedAck

QueryJobStatus

JobStatusReply

RetrieveJobResults

JobResultsReply

ConfigRequest

ConfigReply

LifeSign

RequestJob

!Error

Life-Cycle of a Workflow, Fault-Tolerance and bootstrapping (randomly ordered collection of thoughts)

I would like to discuss the topic of fault-tolerance on the lowest level with respect to simulation and acutal execution. We have this process container concept thingy around and i am not 100% sure how we are supposed to use it (say in a simulation environment on my laptop). So i'll just write up my current ideas on that.

I start up all the components on my laptop by hand:

```
$ orchestrator --location="orchestrator:127.0.0.1:5000" &
$ aggregator --location="aggregator:127.0.0.1:5001" --master="orchestrator:127.0.0.1:5000" &
$ nre --num-workers=1 --worker-binary="/opt/sdpa/sdpa/bin/nre_worker"
--master="aggregator:127.0.0.1:5001" &
```

What i didn't cover here are the configuration details, the aggregator for example needs to know the exact "location" of the orchestrator - i.e. the parameters for the communication channel whatsoever (IP/port pair for example).

The single parameter to the NRE defines the number of worker processes, the nre should use. Right, i am speaking of "worker processes". Let me explain why: The worker process can be a very very simple program with nearly no dependencies, i.e. it should be easy to compile it with different compilers (i.e. no intel/boost etc. required). This worker process will be started by the NRE either by fork/exec or via the process container mechanism. The worker process can also be the only process having access to the FVM. It can crash and it will not affect the NRE process - it will affect the memory though but that's another story. We also buy us a way to decouple everything, i know that it will be damn slow, but it is safe, which also means, that we can provide a "demo-on-laptop-worker-process" which does everything on its own without the FVM. The worker could also speak to processing elements on Cell, the NRE just doesn't care.

Back to the topic, how does a life-cycle look like?

Me as an user submits a Workflow-Description (sequence of bytes) to an orchestrator. The orchestrator analyses this sequence and attempts to create a workflow-object out of it. If everything goes well, the orchestrator replies with a job-id, if something fails an error is returned.

This workflow is now passed to the WFE for 'execution' - the WFE submits 'activities' back to the Orchestrator. The

Orchestrator hands those activities to (one of) its aggregator (as a workflow). The aggregator parses the workflow and hands it over to its own WFE which submits activities and so on until we reach the NRE level. On this level we do something different: when the WFE submits an activity, we transform it to something manageable, i.e. an object describing exactly what to execute: module, function, input parameters, output parameters. This object is passed to one of the worker processes which takes the object and calls the real function. Results are handed back to the NRE and from there up the chain.

What about fault-tolerance? Lets start on the lowest level:

- function crash:
  - when a function crashes, it will usually mean that the process who made a call to that function terminates (i.e. killed by the kernel)
  - the process-container / child process of the NRE dies
  - the NRE can simply restart it by using the defined restart-strategy (process container / fork-exec)
    - TODO: what happened to the memory, can we provide some mechanism to "check out" memory of the FVM, modify it, check it back in?
    - this would reduce the 'failure-window' - it should be pretty uncommon, that one of 'check out' 'check in' crashes, the modify however is bound to fail (so to say)
  - after restarting the worker, any pending activity will be re-executed, again: in what (f*****) state are we - the WFE cannot help us out here...

- NRE crash
  - the aggregator will discover the loss of an NRE. We have to decide whether it is: node-failure or process-failure:
  - process-failure:
    - restart the NRE and resubmit all activities
    - in which state is/was the memory
  - node-failure:
    - restart-strategy for node
    - what memory regions/data was affected?
      - scan through all tokens referring to memory and check if it was on that node
    - which activities are affected?
      - scan through all current activities and check if input or output tokens refer memory on that node
      - cancel these activities (if already running, they will fail anyways)
    - "rebuild" the memory and resubmit all activities with "new" memory locations

- Aggregator crash
  - Orchestrator will discover that an aggregator crashed
    - simply restart the whole segment (first iteration)
    - later on: only restart the aggregator and resume execution using the persisted state information

- Orchestrator crash
  - see aggregator
  - but: who discovers orchestrator failure? who is responsible for restarting it?

Request for comments!

Interface Definition for WFE<->SDPA

Interface to the WFE

Please refer also to

- the version of Alexander:
  - source:trunk/sdpa/trunk/sdpa/wf/SDPA_to_WFE.hpp
  - source:trunk/sdpa/trunk/sdpa/wf/WFE_to_SDPA.hpp
- the modified version of Andreas included in the GWES:
  - source:trunk/gwes/trunk/gwes_cpp/include/gwes/Sdpa2Gwes.h
  - source:trunk/gwes/trunk/gwes_cpp/include/gwes/Gwes2Sdpa.h

An example of how to use this interface from perspective of the SDPA is available at source:trunk/gwes/trunk/tests/gwes/SdpaDummy.cpp

```
class WFE_to_SDPA_Interface;
class SDPA_to_WFE_Interface {
public:
   // transition from pending to running
   virtual void activityDispatched(const workflow_id_t &wid,
                                   const activity_id_t &aid) = 0;

   // transition from running to failed
   virtual void activityFailed(const workflow_id_t &wid,
                               const activity_id_t &aid,
                               const parameter_list_t &output) = 0;

   // transition from running to finished
   virtual void activityFinished(const workflow_id_t &wid,
                                 const activity_id_t &aid,
                                 const token_list_t &output) = 0;

   // transition from * to cancelled
   virtual void activityCancelled(const workflow_id_t &wid,
                                  const activity_id_t &aid) = 0;


   virtual void registerHandler(WFE_to_SDPA_Interface *sdpa) = 0;

   // corresponds to WFE_to_SDPA::workflowFinished/Failed
   virtual void submitWorkflow(workflow::ptr_t wf) = 0;

   virtual void cancelWorkflow(workflow::ptr_t wf) = 0;
};

class WFE_to_SDPA_Interface {
public:
   virtual void submitWorkflow(const workflow_t &workflow) = 0; // (sub-)workflow
   virtual void submitActivity(const activity_t &activity) = 0; // atomic workflow
                                                               // see Activity.hpp
   // parent job has to cancel all children
   virtual void cancelWorkflow(const workflow_t &workflow) = 0;
   virtual void cancelActivity(const activity_t &activity) = 0;

   virtual void workflowFinished(const workflow_t &workflow) = 0;
   virtual void workflowFailed(const workflow_t &workflow) = 0;
   virtual void workflowCancelled(const workflow_t &workflow) = 0;
};

SDPA_to_WFE_Interface *gwes;

// orchestrator/aggregator/nre initialization (somewhere)
gwes->registerHandler((WFE_to_SDPA_Interface*)this);
// ...
// job submission:
gwes::Workflow::ptr_t wf = WFBuilder.build("<xml></xml>");
gwes->startWorkflow(gwes::Workflow::ptr_t wf);
```