ELEMENTS SPECIFICATION

# DEFINITIONS

The process of progressing a matrix from the introductory state to solved state is called a **solution**.

A **solution** is composed of several **waves.**

A **wave** is composed of **tasks** which reference **packages**.

**Packages** are recorded separately, identified by **solution** and **wave**.

# SERVER BEHAVIOR

The server is a persistent entity, one per processing cloud. The server can fulfill several tasks and has triggers. The tasks and signals the server performs and recognized are as follows.

# SERVER SIGNALS

1. **Task REQUEST\_TASK**(node\_auth)

Server will check for available tasks, choose possibly the oldest, and send it to the requesting client. State of the task will be changed to TASK\_SENT. If the sent Task object has an ID of zero it means that server has failed or refused to provide a task.

1. **file\_stream REQUEST\_TASKFILE**(task\_id, node\_auth)

Server will prepare and send the part file for the provided task. A zero-length response means that the server has failed or refused to provide a task.

1. **file\_stream REQUEST\_PARTFILE**(wave\_id, part\_id, node\_auth)

Server will prepare and send the part file for the matrix part in the provided wave. A zero-length response means that the server has failed or refused to provide a task.

1. **file\_stream REQUEST\_META**(task\_id, node\_auth)

Server will prepare and send the metafile for the provided task. A zero-length response means that the server has failed or refused to provide a task.

1. **integer RETURN\_TASK**(task\_id, task\_file\_length, has\_meta, meta\_file\_length, node\_auth, task\_file, meta\_file)

Server will overwrite the appropriate part file on the drive with the contents of the file stream, mark the task as TASK\_COMPLETE, and decrement tasks\_num of the appropriate wave. If meta was attached, it will be stored on drive and attached to the task appropriately.

1. **Integer SPLIT\_MATRIX**(file\_stream) // WIP

This should behave like the splitter described below, in section called “splitter”.

# SERVER TRIGGERS

1. **wave->task\_num decremented**

Whenever the number of tasks in a wave is decremented, a check is performed whether the number of tasks has reached 0. If it has, the following happens:

* 1. If the wave comprised tasks of type TASK\_PART\_SOLUTION, the metafiles provided with the parts in the previous wave are used to create an aggregate map. A new wave of tasks of type TASK\_PART\_REDUCE are created, parts referring to previous wave, metafile pointing to the aggregate map.
  2. If the wave comprised tasks of type TASK\_PART\_SOLUTION, returned parts are scanned for number of operations performed. If it was above 0, a new wave of TASK\_PART\_SOLUTION is created, otherwise a final solution is assembled on the server (using the last aggregate map).

1. **periodic trigger**

To be resilient to errors the database must be periodically scanned for tasks that have been out too long (which might mean that the client has crashed). These tasks must be either reset or deemed broken.

# CLIENT BEHAVIOR

The client will comprise two main communication threads beyond the processing threads. On multicore machines it will be wise to dedicate one core to those threads as they will read and write FS often.

# CLIENT PROVISIONER

The provisioner periodically gets another task from the server following these steps:

1. Send signal REQUEST\_TASK
2. **If** got a task:
   1. Write it to local DB with state TASK\_RECEIVED
   2. Send signal REQUEST\_TASKFILE, write taskfile to HDD
   3. **If** task is of type TASK\_PART\_REDUCE:
      1. Send signal REQUEST\_META
   4. Change local DB state to TASK\_PROVISIONED
3. **Else** defer next request

# CLIENT UPLOADER

The uploader periodically checks local database for tasks with state TASK\_PROCESSED, following these steps:

1. Check local DB
2. **If** found a task in state TASK\_PROCESSED
   1. Send signal RETURN\_TASK
   2. **If** signal succeeds:
      1. Remove task from local database
   3. **Else:**
      1. Defer further uploads

# CLIENT PROCESSOR

The processor will periodically check local database for tasks in state TASK\_PROVISIONED. Then it will follow these steps:

1. If task is of type TASK\_PART\_SOLUTION, it will generate a part solution and mark task as TASK\_PROCESSED
2. If the task is of type TASK\_PART\_REDUCE, it will perform a dry run against metafile, infer which parts are required, fetch these parts to HDD, then perform reductions against them. Finally it will mark the task as TASK\_PROCESSED
3. If during processing any tasks any unrecoverable errors happen, the task will be marked as TASK\_BROKEN to be resolved by hand by a system operator.

# SPLITTER (OUTDATED)

The working assumption for now is that a precomposed matrix in MTX format will be preloaded into a splitting system, which will reside on a server. This is reasonable since matrix splitting is considered to be an unparallelizable task.

The splitter is a separate binary which lives on the server, is compiled with the same DAO objects as the server and has physical access to the database. As such, it is considered to be secured by the OS and does not require separate authentication.

The sequence of tasks will be as follows:

1. MTX file handle is opened
2. Matrix dimensions are read
   1. If matrix is not square, file is rejected – appropriate error message is produced
3. Matrix is conceptually divided into parts of configurable heights
4. File handles for the parts are opened
5. MTX is read and entries are passed on to appropriate files
6. MTX file handle is closed
7. Part file handles are closed
8. **FUTURE** Files are encrypted
9. Files are compressed
10. A transaction is opened with the database if applicable
11. A solution is registered in the database
12. A wave is registered in the database
13. Matrix parts are registered into tasks
14. Wave is published
15. Solution is published
16. Transaction is commited
17. Splitter exits

# DATA MODELS

# SOLUTION

Solution will contain fields:

1. ID
2. state (see states below)
3. current\_wave (numeric ID)
4. name (text)
5. solution (text, file reference)
6. created (timestamp)
7. updated (timestamp)
8. completed (timestamp)

Solution has states:

**0** SOLUTION\_UNPUBLISHED

**1** SOLUTION\_AVAILABLE

**2** SOLUTION\_COMPLETE

**9** SOLUTION\_BROKEN

# WAVE

Wave will contain fields:

1. ID
2. solution\_id (numeric id)
3. seq (numeric)
4. tasks (numeric)
5. state (see states below)
6. created (timestamp)
7. updated (timestamp)
8. completed (timestamp)

Wave has states:

**0** WAVE\_UNPUBLISHED

**1** WAVE\_AVAILABLE

**2** WAVE\_COMPLETE

**3** WAVE\_BROKEN

# TASK

Task will contain fields:

1. ID
2. wave\_id (numeric id)
3. name
4. type (numeric)
5. state (numeric)
6. part\_num (numeric)
7. metafile (text, file reference)
8. node (text)
9. created (timestamp)
10. updated (timestamp)
11. completed (timestamp)

Task has types:

**0** TASK\_UNDEFINED (reserved for broken tasks)

**1** TASK\_PART\_SOLUTION

**2** TASK\_PART\_REDUCE

Task has states:

**0** TASK\_NOT\_READY

**1** TASK\_READY

**2** TASK\_SENT

**3** TASK\_RECEIVED

**4** TASK\_PROVISIONED

**5** TASK\_STARTED

**6** TASK\_PROCESSED

**7** TASK\_COMPLETE

**9** TASK\_BROKEN