Draft 3 (11/9)

About This Documntation

Status

This is an draft version 3 for design documentation.

This document is not complete. Following points must be addressed before finalization.

- Job specs created by each stages of distributed sorting, shown in “Distributed Sorting” section, need to be changed to conform to updated `JobSpec`.
- “Fault Tolerance” section is missing.

Changes in This Revision

This revision introduces following changes.

- All communications between machines are now formulated as a RPC services rather than messages.
- `replicas` field of job files spec is now of `Option` type. If this field of output file spec set to `None`, scheduler will decide where to put the file.
- Scheduler takes remaining disk space of each machines into consideration. It calculates space requirements for each jobs and schedule it to worker with sufficient storage.
- Worker does not consider output replication failure as an error. This also obsoletes the notion of “indirect error” by machine fault.
- Scheduler runs “file synchronization” jobs at the end of each stages, which will delete no longer needed intermediate files and fix replication failures.

Purpose and Scope

The goal of this documentation is to specify a high level system architecture of fault-tolerant distributed sorting program.

This document covers following subjects:

- Key abstractions to enable distributed and fault-tolerance system.
- Important data structures and algorithms.

- Components and their responsibilities.

This document does NOT cover following subjects.

- Classes, traits, and other kinds of code-level architecture.
- Which libraries are used.
- Network connectivity between machines.

Notations

Structures are formalized using a hypothethical programming language similar to Rust and Scala. All data structures are algebraic data types: `struct` represents product type and `enum` represents sum type.

Objectives and Assumptions

The goal is build a correct and robust system that sorts data distributed across machines.

Input and Output

We are given a cluster of machines. There are N worker machines and a one master machine. physical processors are available to each machines. All machines are interconnected using a commodity network.

Input files are stored in each storage of worker machines. We assume there are total *N* chunks and are stored as `/input/chunk.{n}` where *n* is a increasing sequence of integers from 0 to *N-1*. Each chunk contains sequence of records. Each record is a 100 bytes long. Its first 10 bytes is a key and trailing 90 bytes is a value. Key distribution can be either even or skewed. There can be an nonunique, overlapping key.

Output files should be stored as `/output` directory of each machines. Each machine creates files `/output/partition.{n}` where *n* can be any integer. Data must be sorted in ascending order when all output files are merged in ascending order of file names.

Assumptions on Faults

We define **machien fault** as an unexpected restart of one of workers while sorting is in progress. Upon restart, worker program is restarted but file system gets reverted back to

original state, which means all intermediate files are gone. Duration of a machine fault is unknown, thus the system should handle the fault regardless of how long the worker machine is down.

We assume other types of fault does not happen. This includes disk failure or restart of a master machine.

Assumptions on Network

We assume there is a reliable and ordered network channel between any pairs of machines available unless one of machine is down.

Assumptions on Storage

Let combined size of all input files be *M* and combined size of local storage size of all worker machines be *N*. We assume *N* to be at least twice as large as *M.* In other words, it must be possible to replicate all input files at least once. No assumptions are made regarding available storage of each worker machines.

Job Scheduling System

System Configuration

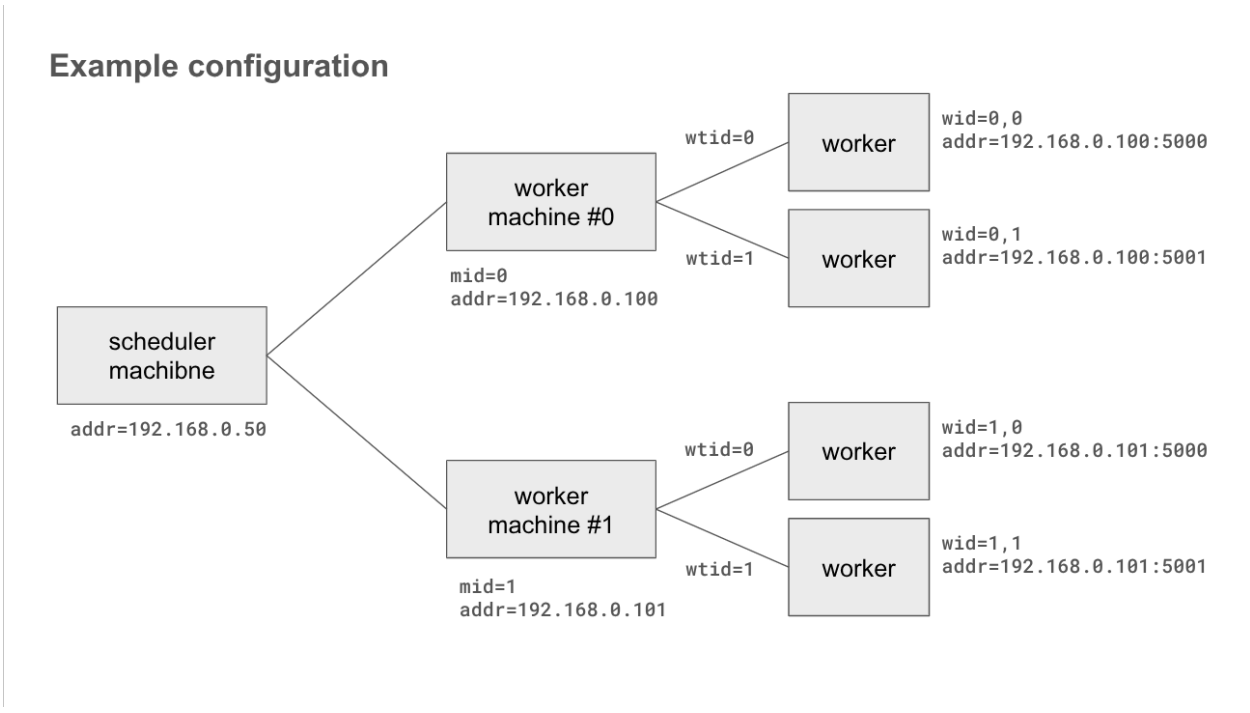
A cluster consists of multiple slave machines and a single master machine. Slave machines run multiple worker programs and are thus called worker machines. A master machine runs scheduler programs and is called the scheduler machine.

A worker is a program that executes a job upon request from a scheduler. Each worker receives a job request from a scheduler and returns the result using a network. Each worker is assigned a unique IP:PORT pair.

Each worker is assigned a **worker ID** or **wid**. Worker ID is a tuple (mid, wtid) where **mid** (machine ID) identifies a physical machine and **wtid** (worker thread ID) identifies each thread in a machine.

A scheduler is a program that assigns jobs to workers. Its primary role is to distribute jobs so as to minimize communication overhead among workers.

The following figure shows an example system configuration with two worker machines each running two workers as threads.



Scheduler and workers interact using RPC. Methods exposed by scheduler and worker are described in the following sections.

Initialization

Worker calls `RegisterWorker` method of scheduler upon start or restart (possibly due to machine fault). An argument to this method, `WorkerHello`, contains wtid of worker. If wtid is 0, worker also sends lit of available input files and remaining storage capacity.

When scheduler is initially started, it creates two data structures: **worker status table** and **global file directory**.

- Worker status table is a table of `WorkerInfo`s which contains status of a worker plus some informations like worker ID or network address. Each `WorkerInfo` is initialized with `status` field set to DOWN and `initialized` set to false.
- Global file directory indexes all files in worker machines.

```
let workerStatusTable:
    Map[Wid, WorkerInfo]
```

```
struct WorkerInfo {
  wid: Wid
  addr: NetAddr
  status: WorkerStatus
  initialized: Bool         // Flag indicating whether Worker
Hello was received
}

enum WorkerStatus {                  // Indicates whether wor
ker is online.
  UP
  DOWN
}

let globalFileDirectory:
    Map[Int, LocalStorageInfo]
```

Scheduler updates status of worker to UP, set `initialized` to true, and update global file directory. It responds back with `SchedulerHello`.

Job

Job is a atomic unit of a distributed program.

A single job consists of input file specifications, job name, and output file specifications.

```
// Example: a job specification sent to some worker that sort
s a file "/input/chunk.0" and writes result to "/working/sor
t/0_0/chunk.0"
JobSpec {
    name: "sort",
    args: [],
    inputs: [{
        name: "/input/chunk.0",
        size: 12345678,
```

```
        replicas: Some([0])   // input file is only available
on machine 0.
    }],
    outputs: [{
        name: "/working/sort/0_0/chunk.0",
        size: 0,              // Output file size is unknown.
        replicas: None        // Scheduler will determine where
to replicate
                              // the output file.
    }]
}
```

`FileEntry` of input and output files are subject to following restrictions.

- `path` field must be a globally unique, since files can be replicated to multiple machines.
- `replicas` field:
  - For input file specs this must be `Some`. If input file is not available in local storage of a machine running a job, it will be replicated from one of machines specified by this field.
  - For output file specs this can be either `Some` or `None`. If `None`, scheduler will decide where to replicate each files.
- `size` field:
  - For input file specs this must be actual size of input file.
  - For output file specs this field must be 0.

Worker Behavior

Worker RPC Methods

Worker exposes following RPC methods to scheduler.

RunJob(spec: JobSpec) → JobResult

Run a job specified by `spec`.

1. Replicate input files that are not present in the local machine.
2. Execute job body function with `spec` as argument.
3. Replicate all output files to machine specified in `replicas` field of output file specs.

`success` field of `JobResult` indicates whether job execution was successful. If all of these steps were done without failure, a successful `JobResult` is returned. If error happens, all changes made to disks are reverted, and failing `JobResult` is returned with `error` field set to source of error. One exception is output replication failure - in this case `success` is set to true, but `error` field is set to `Some(OutputReplicationError))`.

Returns error when worker is running a job. In this case `error` field is set to `Some(WorkerBusy))`.

Halt(reason: String) → None

Forcefully halts the worker. Invoked when `HaltOnError` of scheduler was called, or unexpected error was raised in scheduler.

Heartbeat

Worker also periodically invoke `NotifyUp` RPC method of scheduler every 3 seconds.

Scheduler Behavior

Job Lifecycle

Scheduler receives list of `JobSpec` from distributed program. These jobs must not have dependencies and thus parallelizable. It creates a `Job` object which is a `JobSpec` plus informations required to track job status.

```
struct Job {
    state: JobState
    ttl: Int
    spec: JobSpec
}
```

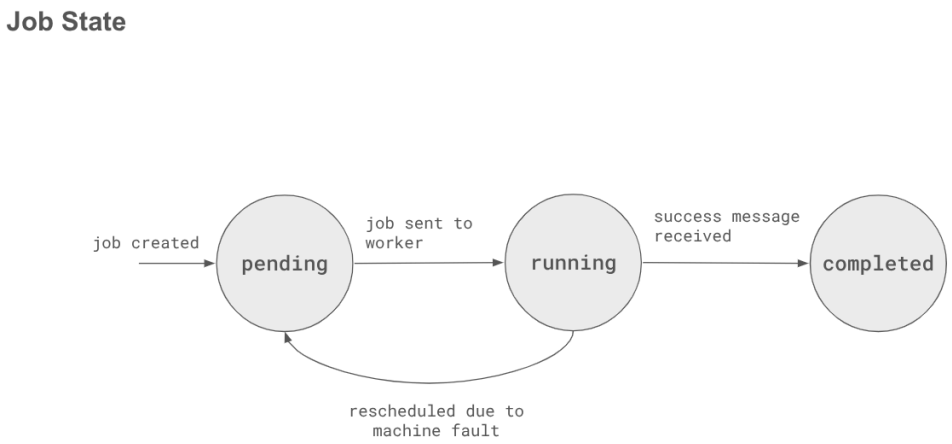
```
enum JobState {
    Pending
    Running
    Completed
}
```

Meaning of each `state` is as follows:

- `Pending`: Job is waiting to be sent to worker. This is initial state. A job in this state is either fresh or re-schedule due to machine fault.
- `Running`: Job is sent to worker and being run by worker.
- `Completed`: Job completed without error.

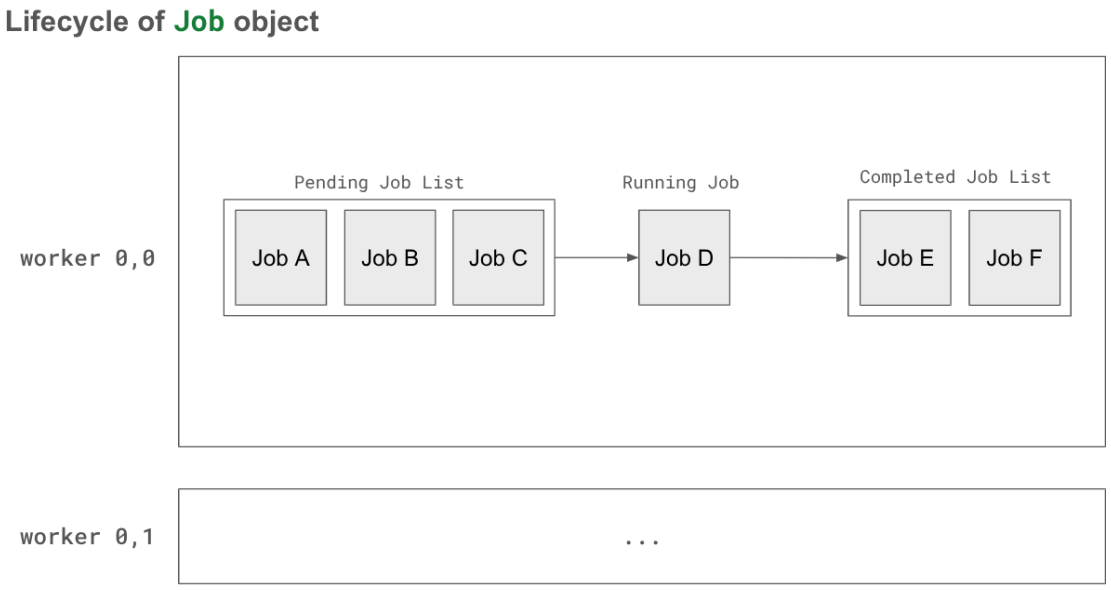
`ttl` indicates how much this `Job` can be retried. If job fails and `ttl` is > 0, then it is decremented by 1 then re-submitted to same worker. If `ttl` is 0, exception is thrown.

Below is a state diagram of `Job`.



Scheduler internally maintains list of `Job`s per worker. For each worker, `Job` is stored in distinct locations by its state. `pending` jobs are stored in pending job list, `completed` jobs in completed job list. `running` job is stored seperately. Scheduler perform **scheduling** procedure to fill pending job list of each workers. Job is popped from pending job list

and submitted to a worker with status ON. If running job succeeds, it is moved to completed job list, and another job is popped from pending job list.



Job failure can be divided into two classes: job error and worker error.

- Job error happens when worker is behaving normally but job execution was failed. In this case, worker returns failing `JobResult` to scheduler. Scheduler decides whether to rerun the failing job using `ttl` field. If `ttl` is > 0, then it is decremented and run again on the same worker. If `ttl` reaches 0, then error is raised to distributed program.
- Worker error happens when worker is malfunctioning, for example RPC request failing to be delivered to worker. This indicates machine fault, so status of workers in the same machine will change to DOWN, triggering reschedule.

Scheduler returns once all jobs are completed, and all workers are UP.

Scheduling

Scheduler choose on which worker to run each job. It must do it correctly and efficienctly.

Scheduler must schedule jobs so that following **correctness** conditions are met.

1. **Disk Space Requirement**: Worker that runs a job must have enough spaces to store input and output files.
2. **Input File Availability**: All input files must be accessible either locally or remotely. In other words, in presence of inaccessible input files (stored only in machine that is currently DOWN), the job must NOT be scheduled to machines other than the machine that stores that file.
3. **Output Redundancy**: All output files must be *eventually* replicated to multiple machines. The output file might not be replicated across multiple machines at the time it is created, but it must be replicated after all jobs are completed.
4. **Fault Tolerance**: One or more UP workers can go DOWN at any time. When one or more workers become DOWN, jobs scheduled for that worker should block until status changes back to UP or should be run by other machines (if possible).

Also, scheduler should schedule jobs to maximize **efficiency**, which are defined by following conditions.

1. **Computation Follows Data**: Scheduler should prefer to run jobs on machine in which input files are available locally.
2. **Time Efficiency**: jobs should be scheduled so as to minimize total running time of all jobs.
3. **Worker Revival**: One or more DOWN workers can go UP at any time. When one or more workers become UP, that worker should also receive jobs to run.

For each pair of jobs and machines, it evaluates whether jobs are runnable in the worker, and if true, how much time and spaces are required. Evaluation result is represented by `SchedulingEvaluation` enum.

```
struct SchedulingEvaluation {
  NotRunnable                     // Job is not runnable on t
his machine.
  Runnable(time: Int, space: Int) // Job is runnable on this
machine. `time` and
                                  // `space` field represents
estimated time cost
```

```
                                        // and space requirements r
espectively.
}
```

Evaluation is done in following steps.

1. **Calculate Space Requirement**: calculates how much disk spaces are required to store input and output files. If this value exceeds remaining disk space of the machine, then return `NotRunnable`.
    a. Input files:
        i. If all files exist locally, then add 0.
        ii. If some of files are in remote machine,
            1. If net input file size is under 500MB, then add 0. Input files are stored in memory.
            2. Otherwise, those files replicated to local disk, thus add net size of all these input files.
            (We don't consider storing only some of input files in memory for simplicity)
    b. Output files: Add `outputSize` field of job spec.
    (We don't consider storing output file in memory and replicating it to other machine for simplicity.)
2. **Determine Output Replica**: For each output files,
    a. If `replica` field is `None`, then set machine IDs of following machines as its value.
        i. Current machine.
        ii. Any machine with sufficient storage. If such machine does not exists, panic.
    b. If `replica` field is `Some`, skip.
3. **Check Input File Availability**
    a. For each input files, determine availability by taking logical OR of followings: for each value `r` of `replica` field,
        i. If `r` matches machine ID of worker, then return true.

    ii. Otherwise, return true if machine status of `r` is UP and false if DOWN.
  b. If there exists unavilable file, return `NotRunnable`.
4. **Estimate Time Requirement**: calculate time cost of running job on the machine.
  a. Let X be network cost per byte, and Y be calculation cost.
  b. Input File Cost:
    i. If all files exist locally, then add 0.
    ii. If some of files are in remote machine, add X * (sum of remote files)
  c. Calculation Cost:
    i. If worker is UP, then add Y.
    ii. If worker is DOWN, then add +Inf.
  d. Output File Cost: add X * (`outputSize` field)

For each machine on which evaluation result is `Runnable`, calculate total time cost and choose worker to run the job as follows.

1. Prepare a zero-initialized list `timeCosts` with length equal to number of workers. This list is reused for all jobs.
2. For each worker on each machine, add estimated time cost to `timeCosts[wid]` and take maximum of `timeCosts` as a total time costs.
3. Choose worker that minimizes total time cost.
  a. If such machine does not exists, panic.
  b. If there are multiple workers that minimizes total time cost to same value, choose random one.

The scheduling procedure described above meets all correctness conditions except fault tolerance (which is addressed by rescheduling mechanism). But efficient scheduling requires network cost per byte X and calculation cost Y to be known, but in practice they are hard to know without actually running the job. So, scheduler determines X and Y by using `JobResult.diagnostics` field for first 10 samples.

Reschedule

Reschedule can happen to redistribute jobs to workers. This mechanism enables jobs to be scheduled correctly and efficiently in presence of machine fault. It is triggered in following situations:

- One of pending job list becomes empty.
- One of worker status changes to UP or DOWN.

In those cases, all jobs in pending job list and running job of a worker with status DOWN are scheduled again according to scheduling procedure described earlier.

Synchronization

After all jobs are completed, scheduler additionally queues “synchronization” jobs to one of worker of each machine. Argument of job contains file entries of `/working` directory. Worker will sync local directory with these entries by removing no longer needed files and replicating missing files with `pull` method of file replication service.

This step is implemented as a job rather than RPC method in order to be fault tolerant.

Scheduler RPC Methods

Scheduler exposes following RPC methods to workers.

RegisterWorker(hello: WorkerHello) → SchedulerHello

Register new worker. Called by when new worker is spawned or is restarted by machine fault.

Worker reports its wtid and optionally local storage information (input files and remaining storage). Local storage infomration is only reported by worker with wtid 0.

Scheduler updates global file directory and changes worker status to UP. It responds with `SchedulerHello` which contains wid assigned to the worker.

NotifyUp() → None

Heartbeat mechanism. Worker should call this method for every 5 seconds. Worker that did not called this method more than 10 seconds is considered DOWN.

HaltOnError(err: JobSystemError) → None

Forcefully halts entire job system. Use this method to report unexpected errors raised in worker.

File Replication Service

Each workers runs a file replication service, which is a program that can **push** a file to another machine or **pull** a file from another machine. The service runs as a seperate thread or process in each machine. It exposes two RPC service, one for local workers and one for remote file replication services.

Following RPC methods are available to local workers.

Push(path: String, dst: Int) → ReplicationDiagnostics: Push local file with given `path` to machine with ID `dst`, make sure the file is written on the disk of remote machine, then return.

Pull(path: String, src: Int) → ReplicationDiagnostics: Pull remote file avilable at machine with ID `src` to current machne, make sure the file is written on the disk, then return.

Following RPC methods are available for remote file replication services.

Read(path: String) → Stream[Bytes]: Return contents of local file as a stream of bytes.

Write(path: String, data: Stream[Bytes]) → None: Write data stream to specified path. Path must NOT be present in local storage, i.e. no overwrite is allowed.

Distributed Sorting

We can write a distributed sorting program by using the job scheduling system described above.

The procedure is divided into 4 parts: **sampling**, **sorting**, **partitioning**, and **merging**. At each step, the program splits the step into multiple parallelizable jobs, pass it to scheduler, and wait for scheduler to finish.

We assume the cluster consists of N machines and C threads in each machine. There are in total N x C workers. In actual environment, N = 20, and C = 4.

Sampling

For each machine id *mid*, find a *path* of randomly chosen chunk file that is available on that machine. Then, create and schedule the following jobs.

- Input:
  - name: `<path>`
- Arguments: None.
- Output: None.
- Return value: List of ~1MB keys samples from input file.

Scheduler will schedule each job to one of worker with machine ID *mid*.

Upon completion, master aggregates and sorts samples to find N x C + 1 pivot points.

Based on the pivot points, each worker (`<mid>_<wtid>`) gets assigned a range and an index for this range (to follow the output guidelines {partition.<n>, partition.<n+1>, … for some n} — project presentation, slide 23).

Sorting

Let *path* be a local path of a file `/input/chunk.<i>` where *i* ranges from 0 to *S*. Let *mid* be a machine ID storing the file. Create and schedule the following jobs for each input file:

- Input:
  - name: `<path>`
- Output:
  - name: `/working/sorting/chunk.<i>`
  - replicas: `[<mid>, <any machine ID other than mid>]`
- Arguments: None.
- Return value: None.

This will create a sorted chunk file on each machine.

Partitioning

For each machine ID *mid*, create and schedule the following jobs for each sorted chunk file (files `/working/sorting/<mid>/chunk.<i>` with *i* in {*0*, …, *S-1*}).

- Input:
  - name: `/working/sorting/chunk.<i>`
- Output:
  - name: files of the form `/working/partitioning/<mid>/chunk.<_mid>_<_wtid>.<i>` where *_mid* and *_wtid* specify which range (determined during **Sampling**) the contents belong to (and `.<i>` is used to specify which sorted chunk the data came from to ensure uniqueness of the output file path)
  - replicas: `[<mid>, <any machine ID other than mid>]`
- Arguments: mapping of pivot points (from **Sampling**) to corresponding `<_mid>` and `<_wtid>`.
- Return value: None.

This will prepare all the needed files for merging.

Merging

For every worker with *mid* and *wtid*, create and schedule the following jobs.

- Input:
  - name: all files matching the pattern `/working/partitioning/{_mid}/chunk.<mid>_<wtid>.{chunk}` where *_mid* and *chunk* match any sequence of characters.
- Output:
  - name: `/output/partition.<n>` where `<n>` is the index assigned to this `<mid>_<wtid>` in **Sampling**
  - replicas: `[<mid>]`
- Arguments: None.
- Return value: None.

Fault Tolerance

TODO

Appendix

Glossary

Scheduler: Program that schedules job.

Scheduler Machine: Machine on which scheduler runs. Usually the master machine of a cluster.

Worker: Program that executes job assigned by scheduler.

Worker Machine: Machine on which workers run. Usually slave machines of a cluster.

Worker ID or **wid**: Integer ID of a worker program.

Worker Thread ID or **wtid**: Integer ID of a worker thread in a machine.

Machine ID or **mid**: Integer ID of a worker machine.

Job: Atomic unit of a distributed program.

Job Body function: Function associated with a specific job name.

Common Structures

This section documents common data types and structures shared by scheduler and worker RPC services.

```
// --------------- Common ---------------
struct Wid {                          // Worker ID.
  mid: Int
  wtid: Int
}

struct NetAddr {                      // Network address of a w
orker.
  ip: String
  port: Int
}
```

```
// --------------- Storage ---------------
struct LocalStorageInfo {
  mid: Option[Int]
    entries: Map[String, FileEntry]   // List of file entrie
s, indexed by path.
    remainingStorage: Int             // Remaining storage of
machine as bytes.
}

struct FileEntry {
  path: String                    // Local path to a file.
  size: Int                       // Size of a file.
  replicas: Option[List[Int]]     // List of mids this file is
replicated to.
}

// --------------- Job ---------------
struct JobSpec {
    name: String                   // Job name.
                                   // Worker will invoke funct
ion associated with
                                   // a job name.
    args: List[Any]                // List of arguments passed
to job body function.
    inputs: List[FileEntry]        // List of input files.
    outputs: List[FileEntry]       // List of output files.
    outputSize: Int                // Total size of output fil
es.
}

struct JobResult<T> {
  success: Bool                  // Indicate whether job execu
tion was successful.
  retval: Option[T]              // Return value of job, only
set
```

```
                                  // if job succeeded.
  error: Option[WorkerError]    // Error happened during exec
ution of job.
  outputs: List[FileEntry]      // List of output files with
actual sizes.
  diagnostics: {
    inputReplications: List[ReplicationDiagnostics]
    outputReplications: List[ReplicationDiagnostics]
    calculationTime: Int
  }
}

struct WorkerError {
    kind: WorkerErrorKind
    message: String
}

enum WorkerErrorKind {
  InputReplicationError(entry: FileEntry)
  BodyFuncError
  OutputReplicationError(entry: FileEntry)
  WorkerBusy
}

struct ReplicationDiagnostics {
  entry: FileEntry
  src: Int
  dst: Int
  start: Float
  end: Float
}

// --------------- Initialization ---------------
struct WorkerHello {
  wtid: Int                               // Worker thread I
D.
```

```
  storageInfo: Option[LocalStorageInfo]
}

struct SchedulerHello {
  wid: Wid
}
```

Additional Resources

All figures are drawn using google slides, available here.