System Design Document - Draft 2 (10/31)

About This Documntation

Status

This is an draft version 2 for design documentation.

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

- Communicatons are asusmed to be done using messaging interface, but in implementation RPC (Remote-Procedure Call) will be used.
- “Synchronization” step needs to be specified more in detail.
- Scheduler currently does take remaining disk space of each machines into consideration.

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

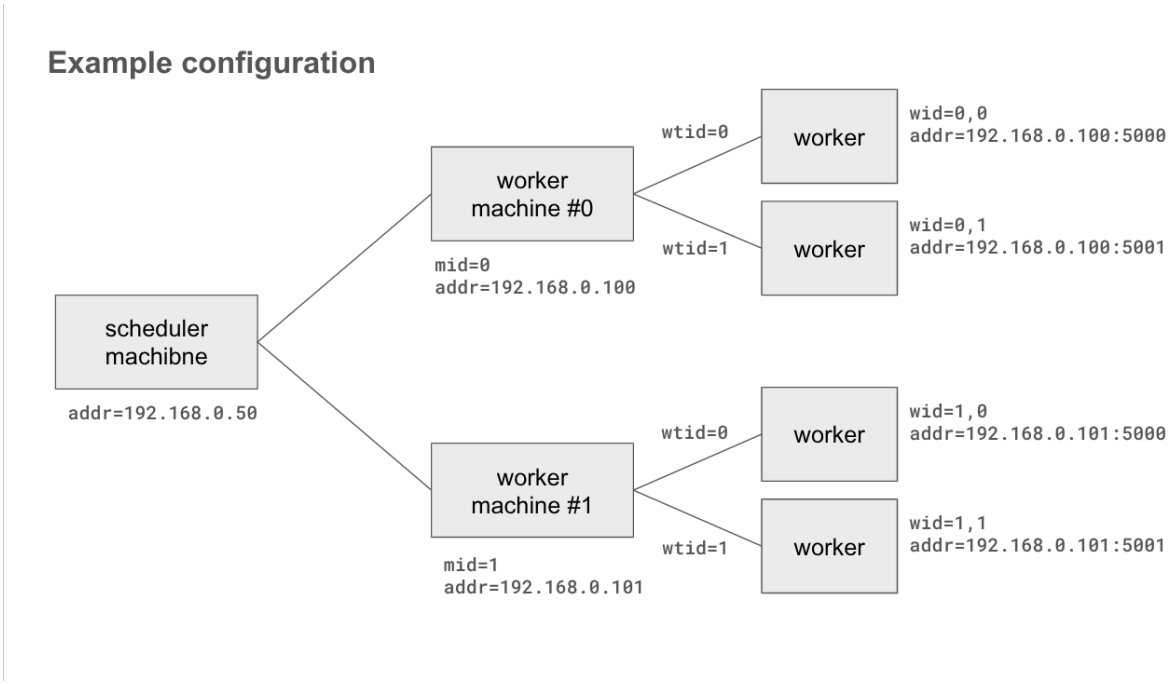
A cluster consists of multiple slave machines and a single master machine. Slave machines run multiple worker programs and are thus called worker machines. A master machine runs scheduler programs and is called the scheduler machine.

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.



Initialization

Worker sends a `WorkerHello` message to scheduler periodically until it gets `SchedulerHello` back. Worker reports its worker thread ID. If `wtid` is 0, worker also sends list of available input files and remaining storage capacity.

```
struct WorkerHello {
  wtid: Int                    // Worker thread ID.
  storageInfo: Option[StorageInfo]
}

struct StorageInfo {
    input_files: List[FileEntry]   // File entries in /input directory.
  remainingStorage: Int          // Remaining storage of machine as bytes.
}

struct FileEntry {
  name: String                 // Local path to a file.
  size: Int                    // Size of a file.
}
```

When scheduler is initially started, it creates two data structures: **worker status table** and **global file directory**. Worker status table is a table of `WorkerRecord`s which contains status of a worker plus some informations like worker ID or network address. Global file directory indexes all files in worker machines using machine ID as a key.

Scheduler knows number of workers and machines, so it can populate worker status table before receiving WorkerHello. Each `WorkerRecord` is initialized with `wid` and `addr` to pre-assigned worker ID and worker address, `status` as DOWN, and `initialized` as false.

```
type Wid = (Int, Int)              // Worker ID is a tuple of (mid, wtid)
type NetAddr = (String, Int)       // Tuple of (IPv4 address, port number)

let workerStatusTable:
    Map[Wid, WorkerRecord]

enum WorkerStatus {                // Indicates whether worker is online.
  UP
  DOWN
}
```

```
struct WorkerRecord {
  wid: Wid
  addr: NetAddr
  status: WorkerStatus
  initialized: Bool        // Flag indicating whether WorkerHello was received
}

let globalFileDirectory:
    Map[Int, MachineFileDirectory]

struct MachineFileDirectory {
  mid: Int
  remainingStorage: Int
  records: List[FileRecord]
}

struct FileRecord {
  name: String
  replicas: List[Int]
  size: Int
}
```

Upon receiving `WorkerHello`, scheduler updates status of worker to UP, set `initialized` to true, and update global file directory. It responds back with `SchedulerHello` message.

```
struct SchedulerHello {
  wid: Wid
}
```

Job

Job is a atomic unit of a distributed program.

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

```
struct JobSpec {
    name: String                // Job name.
                                // Worker will invoke function associated with
                                // a job name.
    args: List[Any]             // List of arguments passed to job body function.
    input_spec: List[FileSpec]    // List of input files.
    output_spec: List[FileSpec]   // List of output files.
}

struct FileSpec {
    name: String                // File path.
    replicas: List[Int]         // List of machine ids the file is available on.
}
```

```
// Example: a job specification for a worker that sorts a file "/input/chunk.0"
and writes result to "/working/sort/0_0/chunk.0"
JobSpec {
    name: "sort",      // run a job named "sort"
    args: [],
    input_spec: [{
        name: "/input/chunk.0"
        replicas: [0]   // input file is only available on machine 0.
    }],
    output_spec: [{
        name: "/working/sort/0_0/chunk.0"
        replicas: [0, 1] // output file should be replicated to machine 0 and 1.
    }]
}
```

`FileSpec` object represents a single input or output file entry.

- `name` field is a path to a file. The file can be located at either local or remote machine. This must be unique among all workers since files can be replicated to other machines.
- `replicas` field is a list of machine ids on which file exists (input) or will be replicated (output).

- When resolving input files, all machine ids are consider in order. Input files not present in local storage will be pulled from one of the other machines. Machine ID of a worker running the task can be NOT present here.
- When replicating output files, they are replicated to all machines specified without a specific order. Machine ID of a worker running the task MUST be present here as the first element.

Worker Behavior

A scheduler sends job request messages with a `JobSpec` object to each worker. Upon receiving a job, the worker performs the following operations:

1. Check whether all input files are present in local storage. If not, it pulls the missing file from one of the machines specified in the `replicas` field.
2. Execute a job body function with job spec as argument.
3. Replicate all output files to machines specified in `replicas` field.

If all of these steps were successful, the worker sends back a success message with return value of a name. If any of steps fail, error message is sent back.

Additionally, worker periodically sends heartbeat message to scheduler.

Workers should be “stupid” as possible. Here “stupid” equals “stateless” - they do exactly what was requested, and maintains no or minimal state, similar to a pure function.
This makes designing fault tolerant system much more easier since a job can be rerun in case of machine fault.
Also, design of a system is greatly simplified because scheduler handles all states and no synchronization between machines is required. (Synchronizing states in a distributed system is surprisingly hard: see why GNU Hurd was abondoned.)

Scheduler Behavior

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

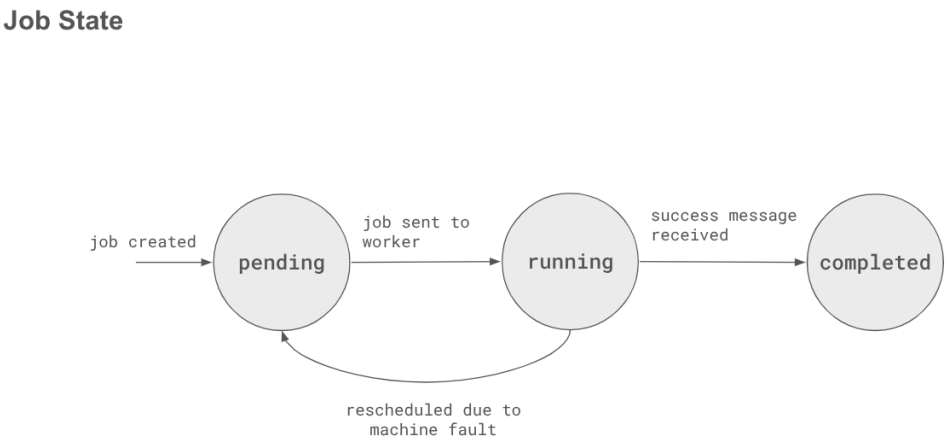
```
struct Job {
    state: JobState
    ttl: Int
    spec: JobSpec
}

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`.

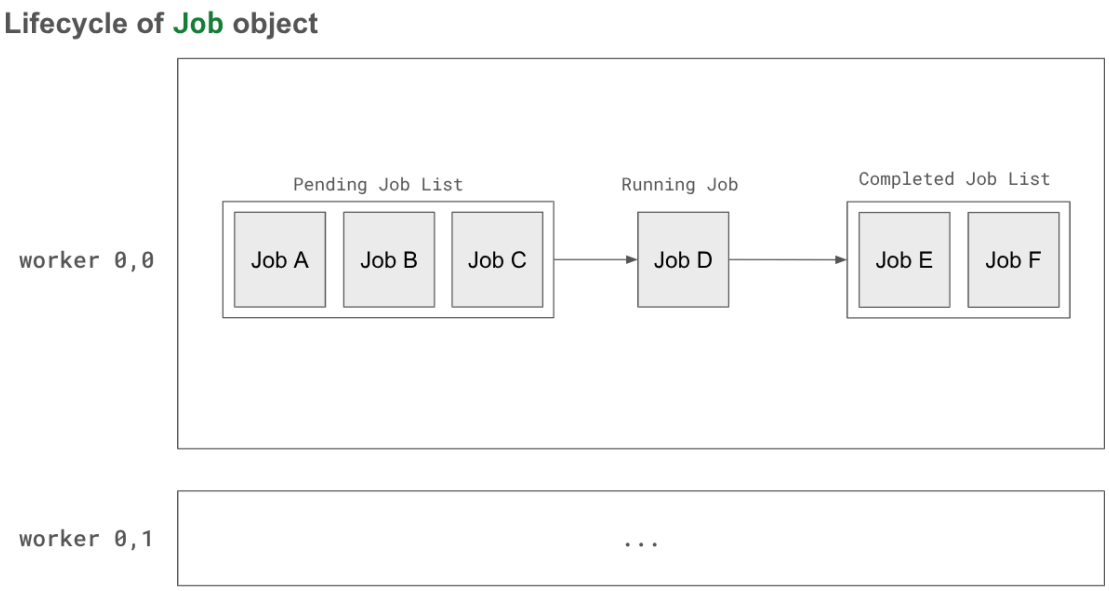


Scheduler internally maintains list of `Job`s per worker. For each worker, `Job` is stored in distinct locations by its state. `pending` jobs are stored in pending job list, `completed` jobs in completed job list, and `running` job is stored as a variable.

Job is popped from pending job list and submitted to a worker. 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 three classes: failures caused directly by machine fault, indirect failures caused by machine fault, and programming errors. Failures caused directly by by machine fault happens when a machine faults while the job was running. In this case the failure is not reported directly but rather using an external mechanisms. (Heartbeat messages and duplicate `WorkerHello`) Remaining two classes are reported directly from a running worker. Failures caused indirectly by machine fault happens only if output file fails to be replicated to the faulted machine. In this case, status of workers running on the faulting machine is updated to DOWN, then the job is submitted to the same worker again. `downMachine` property of `JobRequest` will prevent replication to the faulted machine.

All other errors are considered as programming errors. If `ttl` is > 0, then the job is submitted again until `ttl` reaches 0. Otherwise an exception is thrown.



Scheduler selects on which worker the job will be run as follows:

1. **Retrieve a preferred machine ID**. First element of `replicas` of output file specs must be same and thus designates a machine to run the job. That machine ID is called **preferred machine ID**.
   (TODO: explain why output file spec rather than input files specifies preferred machine ID)
2. **Select a worker**.
   a. Select all workers whose mid matches preferred machine ID and status is UP. If such workers exists, select worker with least pending jobs.
   b. If no such workers exists or preferred machine ID does not exists, first select machine with highest frequency in `replicas` field of input file specs, then select worker with least pending jobs among those workers. Note that the job can be scheduled to worker with status DOWN if input files are not sufficiently replicated.
3. Put `Job` to a pending job list of a selected worker.

Reschedule can happen to distribute jobs to workers evenly. It is triggered in following situations:

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

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.

Scheduler tracks workers' status for this functionality using heartbeat message. Worker periodically sends heartbeat message to scheduler. If scheduler does not receive heartbeat for 10 seconds, status of worker is set to DOWN. It is set back to UP once heartbeat is received again. `WorkerHello` is also considered as a heartbeat message.

After scheduling, `JobRequest` is sent to each worker. It specifies `downMachines`, a list of currently DOWN machines in addition to job specification. Worker will not consider replication failure (either pull or push) from or to these machines as

error. Without this field, a successful job might try to replicate its output to currently DOWN worker.

```
struct JobRequest {
  spec: JobSpec
  downMachines: List[Int]
}
```

Job can fail due to various reasons that is NOT machine fault, notably programming errors. In this case scheduler can receive *halt* message from any source (which might not be worker). For example, if worker thread crashes, then a worker machine can send halt message to scheduler to stop the entire program. Scheduler should immediately halt all workers and log the error for debugging.

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

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 receives push or pull request from a fixed, known port. Workers in the same machine request file replication using following API.

push(file, dst): Push local *file* to machine with ID *dst*, make sure the file is written on the disk, then return.

pull(file, src): Pull remote *file* available at machine with ID *src* to current machine, make sure the file is written on the disk, then return.

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.

Synchronization

Additional “synchronization” step is required after partitioning and merging steps to 1) reclaim disk space and 2) to replicate files to expected machines.

TODO: specify semantics of synchornization step

Fault Tolerance

Fault Detection

Scheduler detects machine fault using three mechanisms.

First, worker sends `WorkerHello` when it starts. If scheduler receives `WorkerHello` but the worker was already initialized, then we can know that the worker was restarted.

Secondly, workers periodically send heartbeat message to scheduler. If scheduler did not received heartbeat message from a specific worker for 10 seconds, its status will change to DOWN and we can know that the worker has faulted.

Note that machine fault can be always detected when duplicate `WorkerHello` was received, but heartbeat mechanism does not guarantee this. The machine might restart all of workers faster than 10 seconds. Nevertheless, the heartbeat mechanism is required for efficiency reason: by tracking whether individual worker is UP or DOWN, assigned to faulting workers can be rescheduled to another workers if possible.

Lastly, an indirect job failure caused by machine fault also signals machine fault. This happens when replicating output files to the faulted machine fails. Strictly, this mechanism is also not required for correctness, but it helps fast fault detection and recovery.

In the following sections, we discuss how system reacts to machine fault in various scenarios. We assume N = 2 and C = 1 in these scenarios, so there are two workers: worker (0,0) and (1,0). We call these workers A and B respectively for convenience. Machine fault will happen in a machine running worker A.

Scenario 1: Fault while Initializing

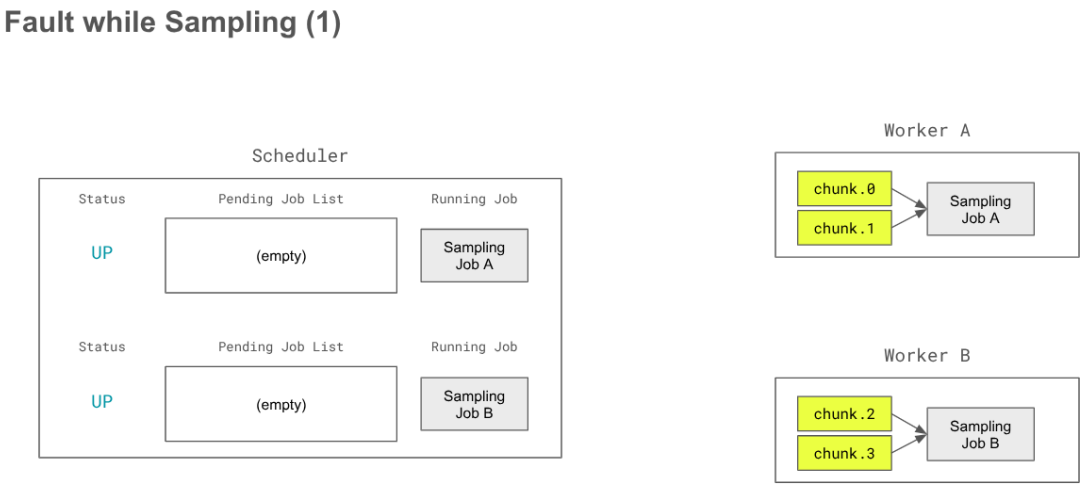
We can further divide this scenario into two cases: 1) scheduler does not receive `WorkerHello`, 2) scheduler receives `WorkerHello` but `SchedulerHello` was not delivered.

In the first case, scheduler will wait indefinitely until the `WorkerHello` is received from worker A. In scheduler's perspective, this is no different than worker A starting slowly.

In the second case, scheduler resets record of worker A in worker status table back to its initial value and wait for `WorkerHello` from worker A indefinitely. Everything proceeds normally after `WorkerHello` is received again from worker A.

Scenario 2: Fault while Sampling

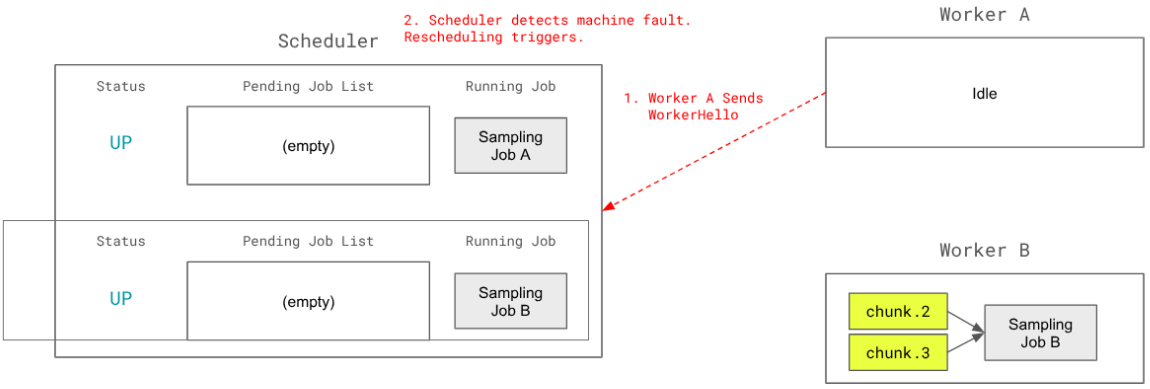
Suppose workers were running sampling jobs just before fault happens, as shown in the following figure.



We can divide this scenario into following two cases, regarding how long the worker A goes DOWN: 1) machine restarts so fast that heartbeat mechanism does not detect fault, and 2) heartbeat mechanism detects fault.

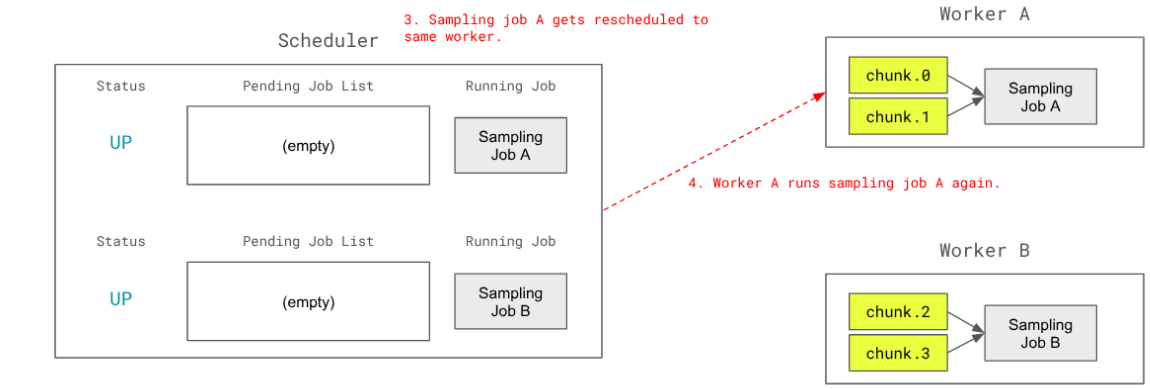
In the first case, worker A sends `WorkerHello` after restart. By receiving this message, scheduler can know that worker A has been restarted even though its status was UP.

Fault while Sampling (2)



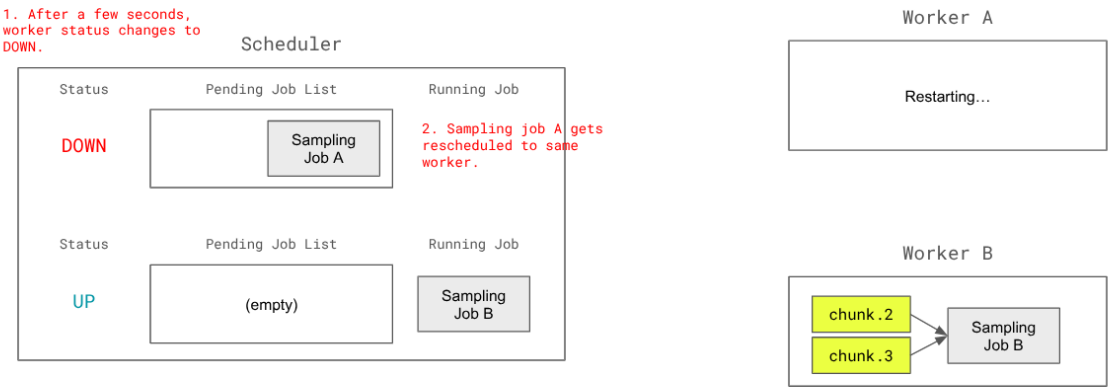
Sampling job A gets rescheduled, but because it does not have output files, and all of input files reside in machine 0, the job gets scheduled to same worker. Sampling job A runs again on the worker A. The system has recovered from fault successfully.

Fault while Sampling (3)



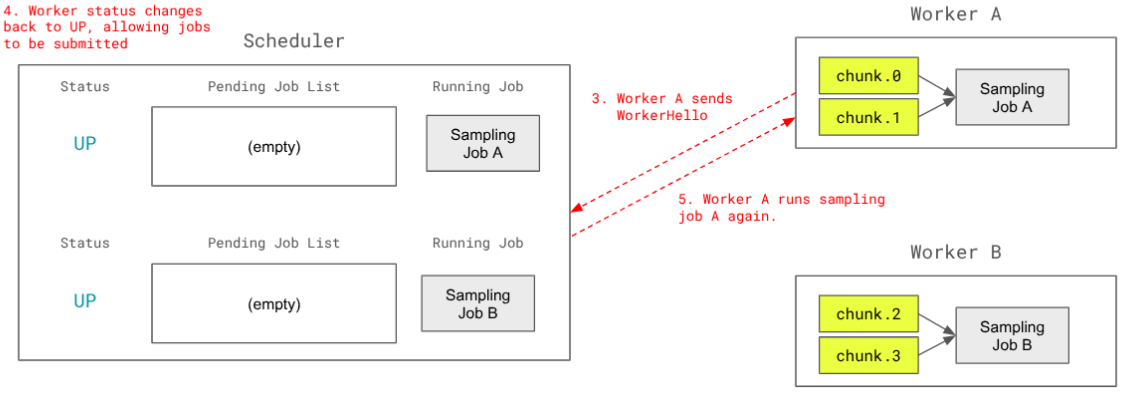
In the second case, after a few seconds status of worker A changes to DOWN, which also triggers reschedule. Sampling job A gets rescheduled to worker A, but because it is currently DOWN no jobs are sent to the worker.

Fault while Sampling (4)



When worker A is restarted it sends `WorkerHello` to scheduler. Because `WorkerHello` is also considered as a heartbeat message, status of worker A changes back to UP. This triggers reschedule again but the position of the job does not change. Finally, job A is sent to worker A, running the job again. The system has recovered from fault successfully.

Fault while Sampling (5)

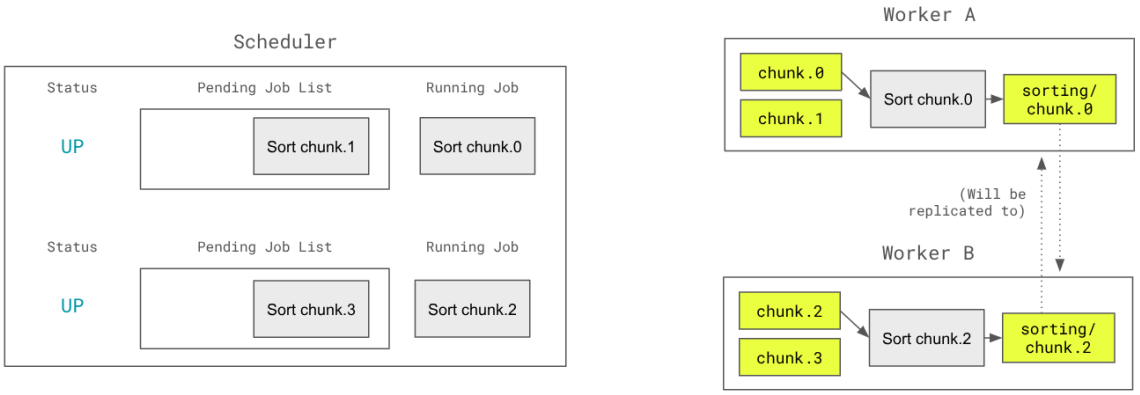


Scenario 3: Fault while Sorting

Similar to scenario 2 (fault while sampling), the job must be run on a specific machine, but additionally job writes output files which needs to be replicated. This means job running on worker B can fail because it couldn't replicate output files to worker A.

Suppose a scenario where each workers are running sorting jobs as shown following figure. Worker A is sorting file `chunk.0` and B is sorting `chunk.2`.

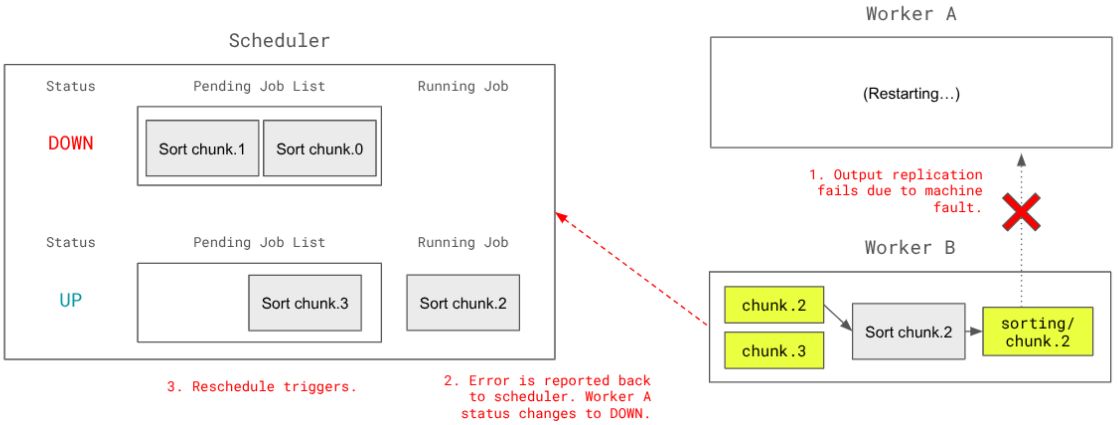
Fault while Sorting (1)



Fault on worker A might cause replication failure on worker B if A is DOWN when B tries output file replication. This failure can arrive either before scheduler detects machine fault or after.

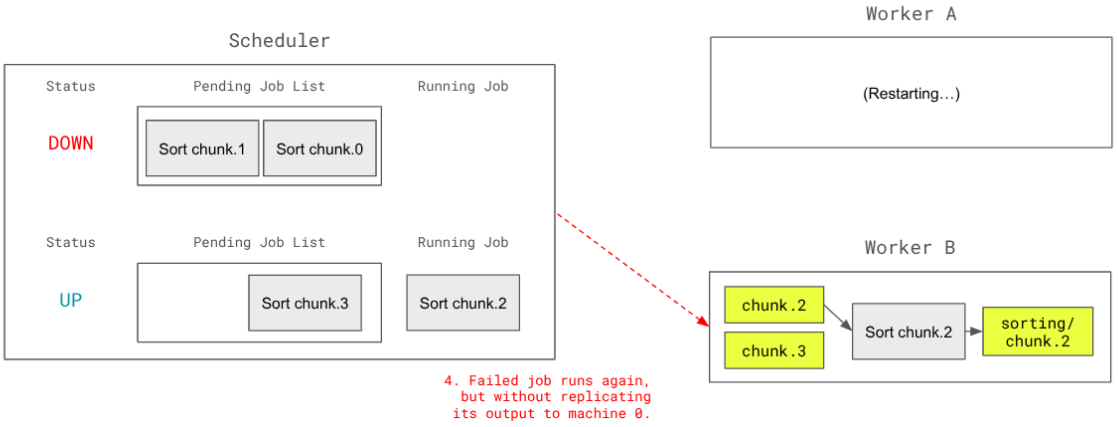
Consider the first case. The failure in worker B will arrive before scheduler detects machine fault. Then, the job failure also act as a fault detection mechanism, changing status of worker A to DOWN and triggering job reschedule.

Fault while Sorting (2)



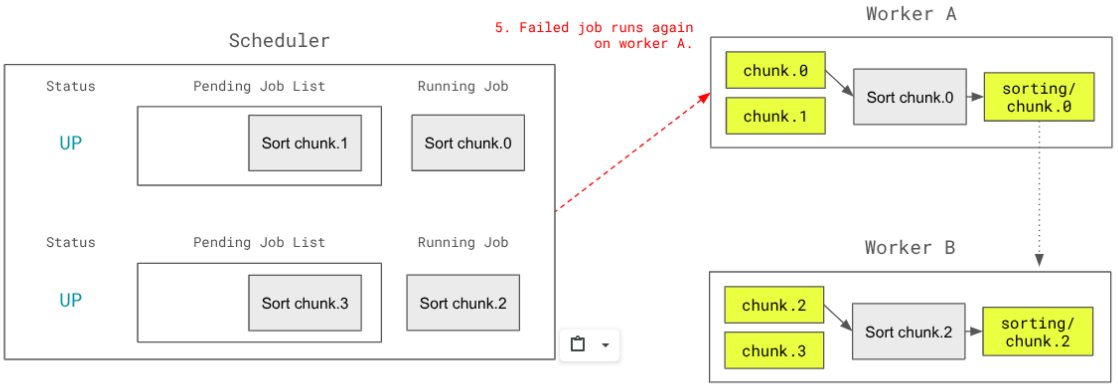
Failed job runs again on worker B, but the worker is notified that the worker A is down via `downMachines` property of `JobRequest`. It will still try replicating output files to worker A, but will not consider failure to do so as an error.

Fault while Sorting (3)



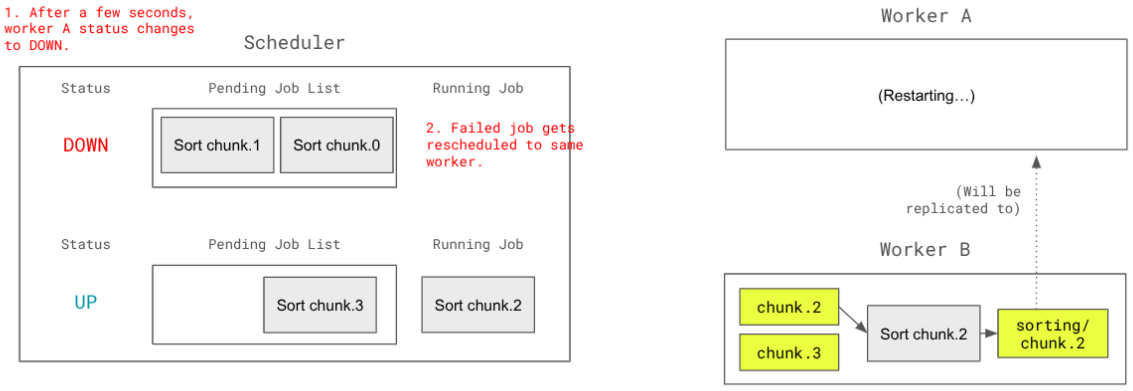
When worker A is reinitialized, failed job will run again on worker A.

Fault while Sorting (4)



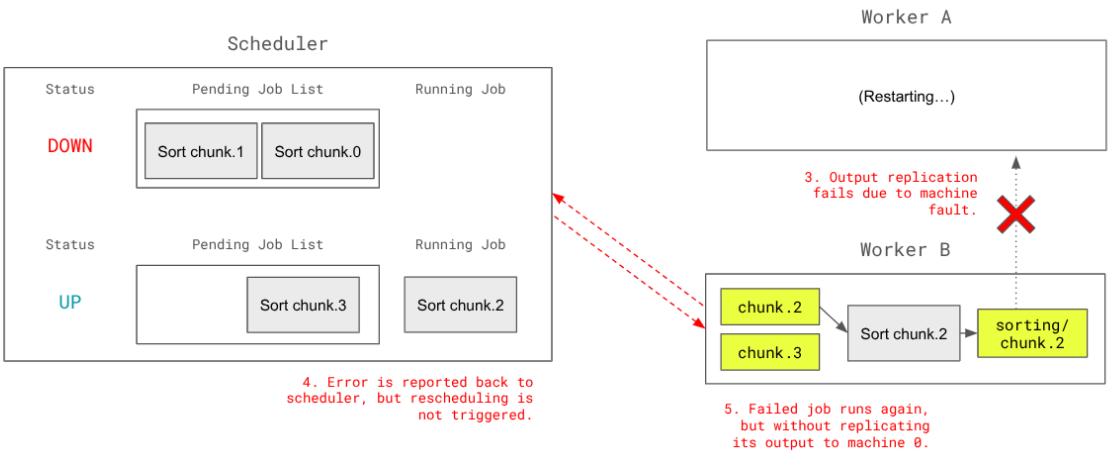
Now, let's see what happens if worker B reports failure after machine fault has been detected. When machine failure is detected jobs jobs will be rescheduled as seen in the previous scenario.

Fault while Sorting (5)



Then indirect failure due to machine fault is reported back to scheduler, but this does not trigger reschedule since status of worker A is already DOWN. Failing job is sent again to worker B with `downMachines` set to `[0]`.

Fault while Sorting (6)



What happens next is same with the previous case.

Scenario 4: Fault while Partitioning/Merging

Jobs in these stages are different from those in previous stages since input files are available in multiple machines. This allows jobs failed directly due to machine fault to be rescheduled on another workers.

All jobs submitted to worker A has preferred machine ID of 0, and also available to worker B. So when worker A goes DOWN, all jobs on worker A will be rescheduled to worker B. But this will produce output files only on machine 1 while `replicas` field of output file specs of those jobs are `[0, 1]`. This discrepency is resolved by “finalization” stage which happens after partitioning step. This step will delete intermediate files that are no longer needed and replicate missing files.

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.

Appendix

All figures are drawn using google slides, available here.