Part 1 Introduction to distributed system

What is a distributed system? The core of it is a set of cooperating computers that are communicating with each other over network to get some coherence tasks done. It is an infrastructure for applications. We'd like to build be able to build an abstraction that hides the nature of the distributed system.

Examples/case study: storage for big websites(GFS), big data computation(MapReduce), p2p file shared.

The reasons people are driven to use distributed systems: parallelism(high-performance), fault tolerance, physical reasons(geographical divide, for example, bank transfers), security(isolate your computer from suspicious codes).

Basic challenges: concurrency, partial failure(Basic challenge), performance(needs to take careful design), scalability.

Kinds of infrastructure: storage, communications, computation.

Implementations topics: RPC(remote procedure call, masks the fact that they are communicating over an unreliable network), threads, concurrency control(locks…).

Topic in all papers: performance(speedup, scalable speedup).

Fault tolerance flavours(a particular type of sth, especially computer software), availability(under certain failures, the system is still available), recoverability(the process would stop until being fixed. After the repair occurs, the system could be able to continue as if nothing bad had gone wrong. it is weaker than availability.) A good available system would sort of be recoverable as well, since there are too many failures occur, they would stop answering, but then it would continue correctly after that.

Two important thing(tool): non-volatile storage(corresponding to recoverability; checkpoints, logs), replication(corresponding to availability; however, replicas may drift out of sync). Writing on NV storage is expensive, so high performance systems should avoid write on it too often.

Last topic, consistency(across multiple replicas)(risking visit old data)(strong consistency, which guarantee to provide the latest date, requires a lot of communication).

Part 2 Introduction to MapReduce

The goal of MapReduce is to allow the application designer, the consumer of this kind of distributed computation, just be able to write a simple map and a simple reduce function don't know anything about distribution, and the MapReduce framework.

**Abstracts:**

MapReduce is a programming model and an associated implementation for processing and generating large data sets. Users specify a map function that processes a key/value pair to generate a set of intermediate key/value pairs, and a reduce function that merges all intermediate values associated with the same intermediate key. Many real world tasks are expressible in this model.

Programs written in this functional style are automatically parallelized and executed on a large cluster of commodity machines. The run-time system takes care of the details of partitioning the input data, scheduling the program’s execution across a set of machines, handling machine failures, and managing the required inter-machine communication.

**Implementations:**

User 
Program 
(l) fork 
(I) foik 
Master 
. assign 
map 
f!) fork 
(2) 
åssjgn 
reduee. 
split 0 
split 1 
split 2 
split 3 
split 4 
Input 
files 
(3) read 
worker 
worker 
worker 
Map 
phase 
(5) remote read 
(4) local write 
Intermediate files 
(on local disks) 
Figure 1: Execution overview 
worker 
worker 
Reduce 
phase 
(6) write 
output 
file O 
output 
file 1 
Output 
files 

When the user program calls the MapReduce function, the following sequence of actions occurs:

1. The MapReduce library in the user program first splits the input files into M pieces of typically 16 megabytes to 64 megabytes (MB) per piece (controllable by the user via an optional parameter). It then starts up many copies of the program on a **cluster of machines**.
2. One of the copies of the program is special – the master. The rest are workers that are assigned work by the master. There are M map tasks and R reduce tasks to assign. The master picks idle workers and assigns each one a map task or a reduce task. (The total work is called a job, and each map/reduce function is called a task.)
3. A worker who is assigned a map task reads the contents of the corresponding input split. It parses key/value pairs out of the input data and passes each pair to the user-defined Map function. The intermediate key/value pairs produced by the Map function are buffered in memory.
4. Periodically, the buffered pairs are written to local disk, partitioned into R regions by the partitioning function. The locations of these buffered pairs on the local disk are passed back to the master, who is responsible for forwarding these locations to the reduce workers.
5. When a reduce worker is notified by the master about these locations, it uses remote procedure calls to read the buffered data from the local disks of the map workers. When a reduce worker has **read all intermediate data**, it **sort**s it by the intermediate keys so that all occurrences of the same key are grouped together. The sorting is needed because typically many different keys map to the same reduce task. If the amount of intermediate data is too large to fit in memory, an external sort is used. (all intermediate data whose key value falls into its space)
6. The reduce worker iterates over the sorted intermediate data and for each unique intermediate key encountered, it passes the key and the corresponding set of intermediate values to the user’s Reduce function. The output of the Reduce function is appended to a final output file for this reduce partition.
7. When all map tasks and reduce tasks have been completed, the master wakes up the user program. At this point, the MapReduce call in the user pro- gram returns back to the user code.

After successful completion, the output of the mapreduce execution is available in the R output files (one per reduce task, with file names as specified by the user).

Master data structures:

The master keeps several data structures. For each map task and reduce task, it stores the **state** (**idle, in-progress, or completed**), and the identity of the worker machine (for non-idle tasks).

Therefore, for each completed map task, the master stores the locations and sizes of the R intermediate file regions produced by the map task. Updates to this location and size information are received as map tasks are completed. The information is pushed incrementally to workers that have in-progress reduce tasks.

Fault tolerance:

The master pings every worker periodically. If no response is received from a worker in a certain amount of time, the master marks the worker as failed.

Any map tasks completed by the worker are reset back to their initial idle state, and therefore become eligible for scheduling on other workers. Similarly, any map task or reduce task in progress on a failed worker is also reset to idle and becomes eligible for rescheduling.

Completed **map tasks** are re-executed on a failure because their **output** is stored on the **local disk(s)** of the failed machine and is therefore inaccessible. Completed **reduce tasks** do not need to be re-executed since their output is stored in a **global file system**.

When a map task is executed first by worker A and then later executed by worker B (because A failed), all workers executing reduce tasks are notified of the re- execution. Any reduce task that has not already read the data from worker A will read the data from worker B.

(Even worker A has already finished the task, the task has to be done completely by worker B, as "reduce workers" may have not read and can not read worker A's data.)

MapReduce is resilient to large-scale worker failures. For example, during one MapReduce operation, network maintenance on a running cluster was causing groups of 80 machines at a time to become unreachable for several minutes. The MapReduce master simply re-executed the work done by the unreachable worker machines, and continued to make forward progress, eventually completing the MapReduce operation.

Given that there is only a single master, its failure is unlikely; therefore our current implementation aborts the MapReduce computation if the master fails.

Semantics in the presence of failure:

When the user-supplied map and reduce operators are deterministic functions of their input values, our distributed implementation produces the same output as would have been produced by a non-faulting sequential execution of the entire program.

We rely on atomic commits of map and reduce task outputs to achieve this property. Each in-progress task writes its output to **private** temporary files. A reduce task produces one such file, and a **map task** produces **R such files** (one per reduce task).

Network bandwidth is a relatively scarce resource in our computing environment. We conserve network band-width by taking advantage of the fact that the input data (managed by GFS [8]) is stored on the local disks of the machines that make up our cluster. GFS divides each file into 64 MB blocks, and stores several copies of each block (typically 3 copies) on different machines. The MapReduce master takes the location information of the input files into account and attempts to schedule a map task on a machine that contains a replica of the corresponding input data. Failing that, it attempts to schedule a map task near a replica of that task’s input data (e.g., on a worker machine that is on the same network switch as the machine containing the data).

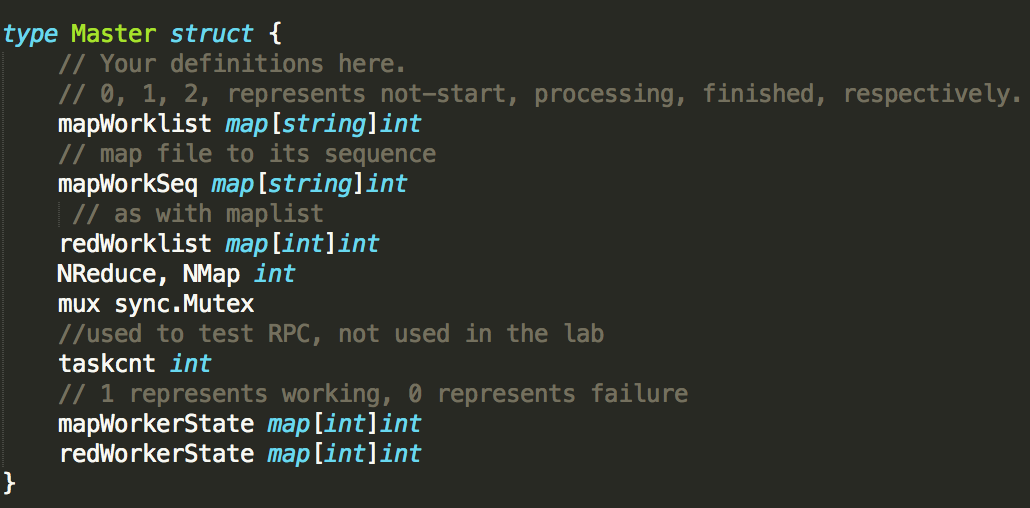
We subdivide the map phase into M pieces and the reduce phase into R pieces, as described above. Ideally, M and R should be much larger than the number of worker machines(finer granularity). Having each worker perform many different tasks improves dynamic load balancing, and also speeds up recovery when a worker fails: the many map tasks it has completed can be spread out across all the other worker machines.

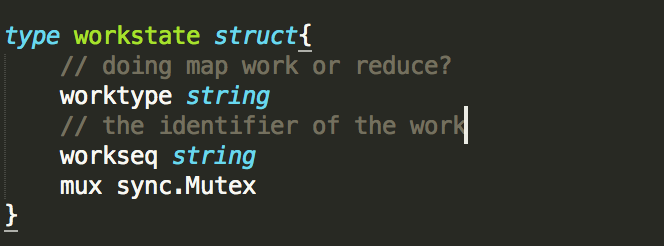
One of the common causes that lengthens the total time taken for a MapReduce operation is a “straggler”: a machine that takes an unusually long time to complete one of the last few map or reduce tasks in the computation. Stragglers can arise for a whole host of reasons. For ex- ample, a machine with a bad disk may experience frequent correctable errors that slow its read performance.

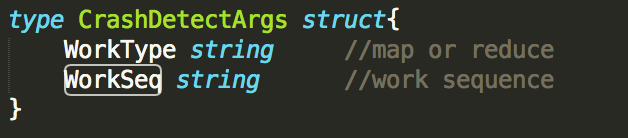
We have a general mechanism to alleviate the problem of stragglers. When a MapReduce operation is close to completion, the master schedules backup executions of the remaining in-progress tasks. The task is marked as completed whenever either the primary or the backup execution completes.

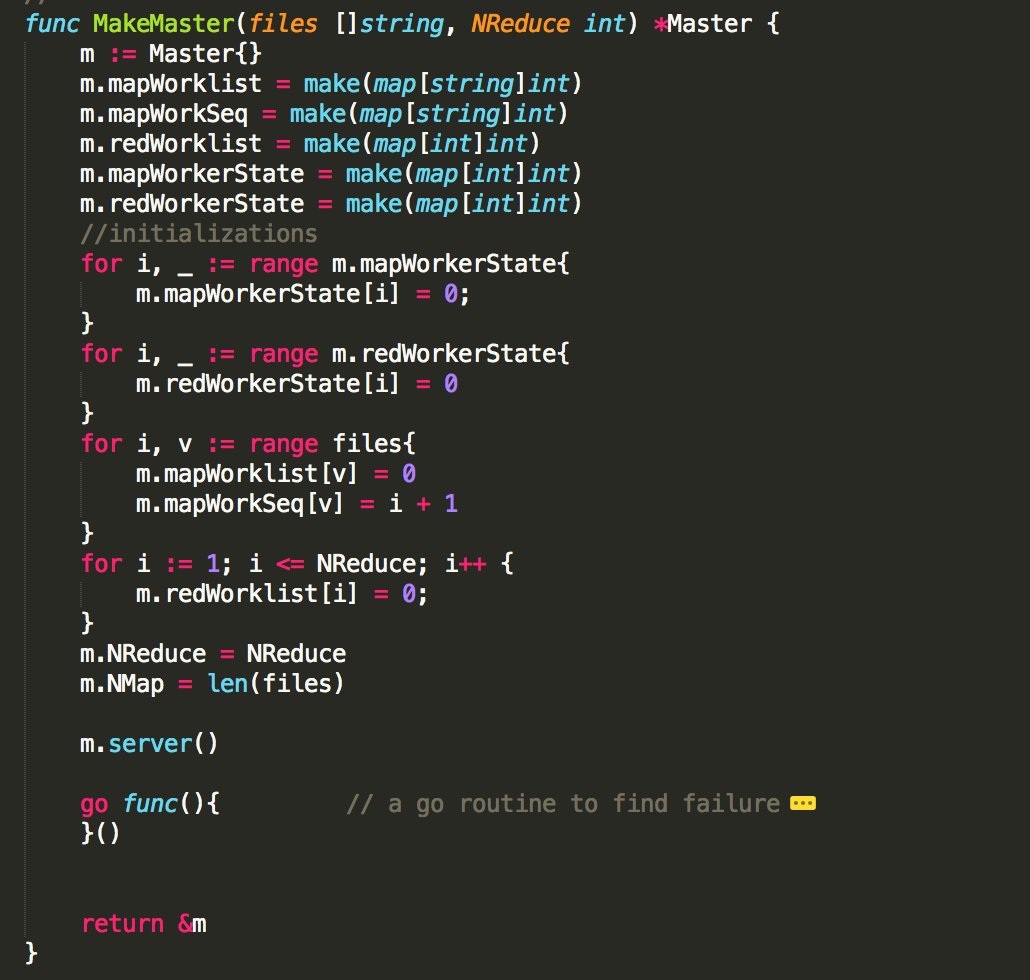
Part 3 MapReduce Lab

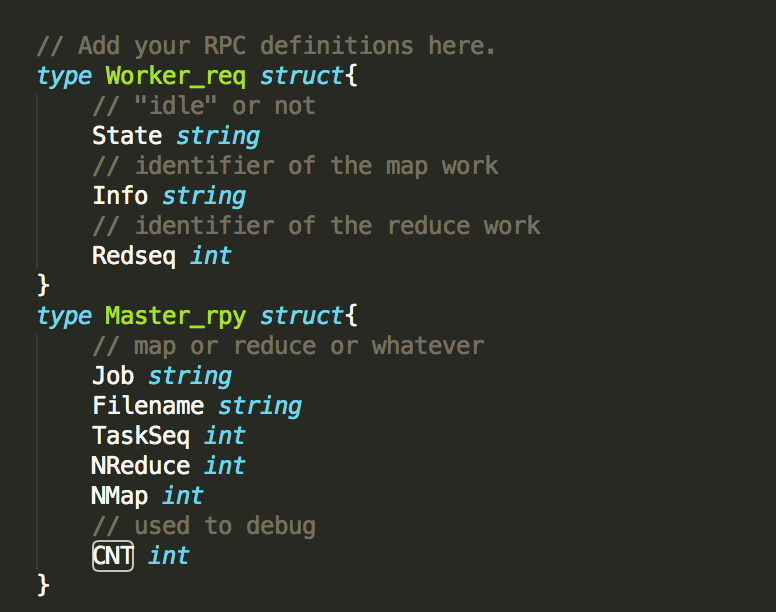
1) Data structures:











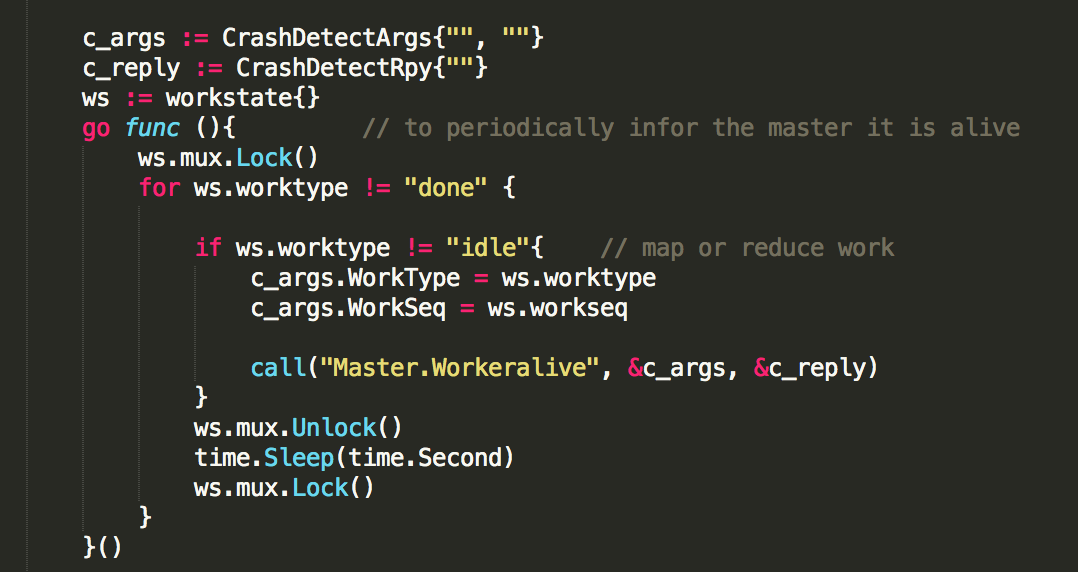
2) Failure detection:

Master side





Worker side



3) Distribute work

Work call master to distribute work for it.

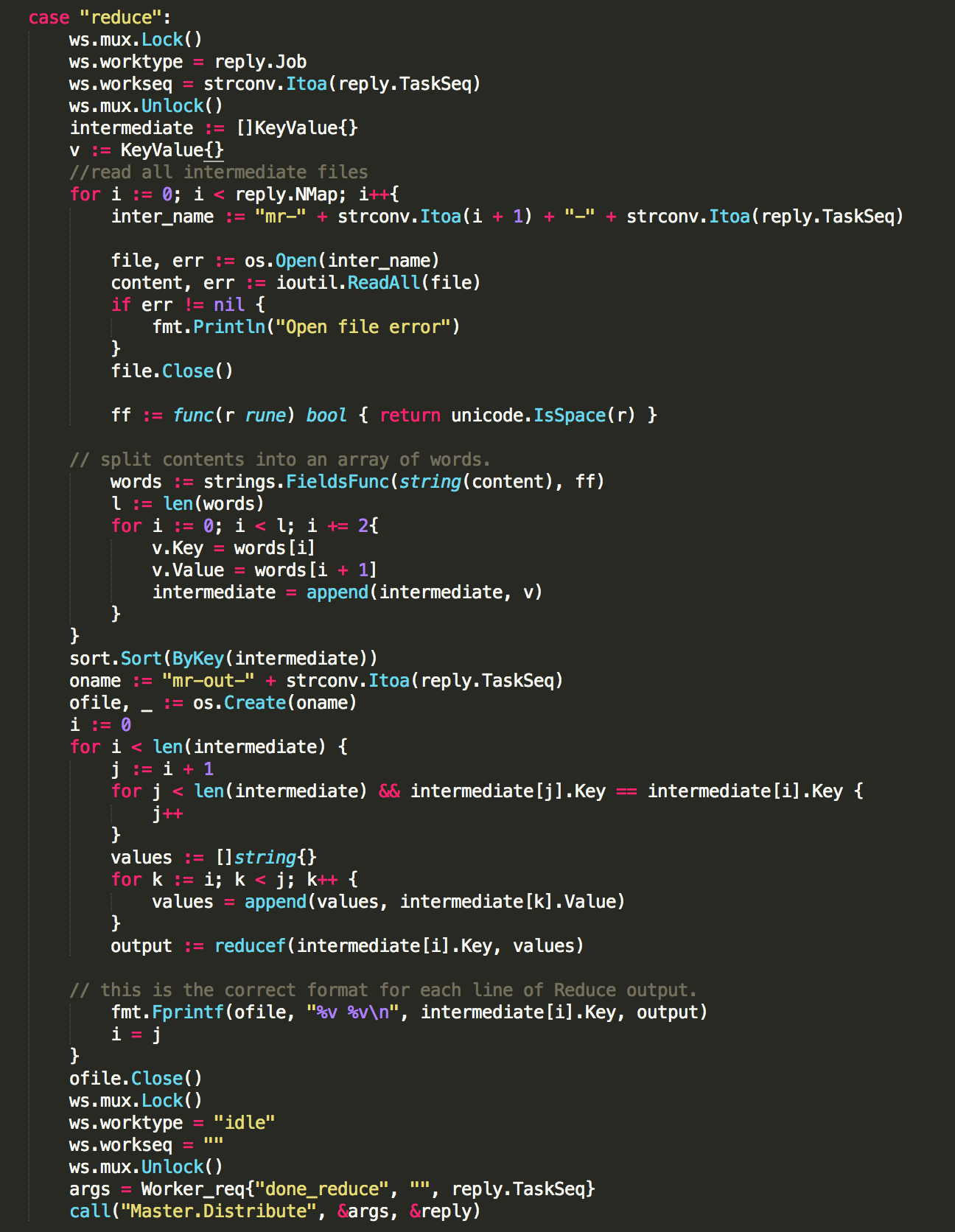


Master reply:



Worker’s work:





When finish the work, the worker will call master again.

