CS 423 Fall 2011

MP4

Group 6: Burrough, Busjaeger, Decker

Index

Illustration Index

Index of Tables

1 Programming Model

Our solution is a partial implementation of the MapReduce programming model described in [1]. Users implement map, reduce, and related functions as Java classes packaged into Jar files. For each job a jar file along with the input data is submitted to the runtime for processing. The Jar file must declare the implementation classes via attributes in the Jar manifest file. Table 1 lists the set of supported attributes.

Attribute Name	Mandatory	Attribute Value
`MapperClass`	Yes	Fully qualified name of the class implementing the map function. The class must extend `edu.illinois.cs.mapreduce.api.Mapper`
`CombinerClass`	No	Fully qualified name of the class implementing the combiner function. Depending on the job, this may be the same as the reduce function. The class must extend `edu.illinois.cs.mapreduce.api.Reducer`
`ReducerClass`	Yes	Fully qualified name of the class implementing the reduce function. The class must extend `edu.illinois.cs.mapreduce.api.Reducer`
`InputFormatClass`	Yes	Fully qualified name of the class implementing the input format. The class must extend `edu.illinois.cs.mapreduce.api.InputFormat`
`OutputFormatClass`	Yes	Fully qualified name of the class implementing the output format. The class must extend `edu.illinois.cs.mapreduce.api.OutputFormat`

Table 1: Manifest Attributes

An `InputFormat` implements a specific input format, such as a text or image files, by means of a `Splitter` and `RecordReader`. The splitter splits a given input file into a set of splits based on some format-specific algorithm. For example, for text files the splitter may split by a combination of byte and line size. The record reader is traversed by the runtime to produce key/value pairs for a given input split.

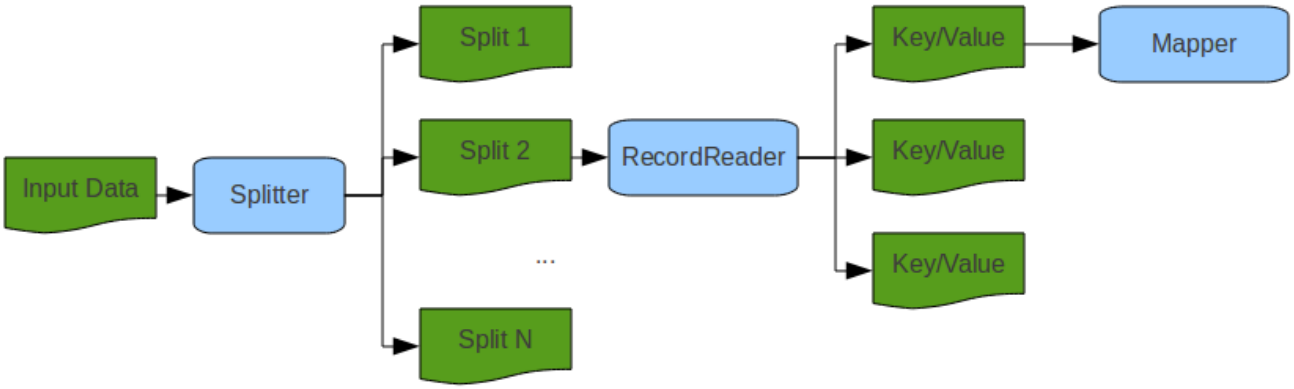

Illustration 1: Input Format Function

The `Mapper` transforms the key/value pair into any number of key/value pairs that are written to a `Context` object passed to it, which sorts the output by key and stores it in an intermediate file once the split has been processed by the map. If a `Combiner` is specified, it is called with the list of values for each key so that it can reduce the intermediate file to a smaller size. This is mainly done to reduce network traffic. Our implementation does not support partitioning, because it always uses a single reducer.

The `Reducer` is called with the list of values for each key produced by the map/combine phase. It stores the output key/value pairs in the context, which uses the `RecordWriter` returned by the `OutputFormat`

to store the pairs in the output file.

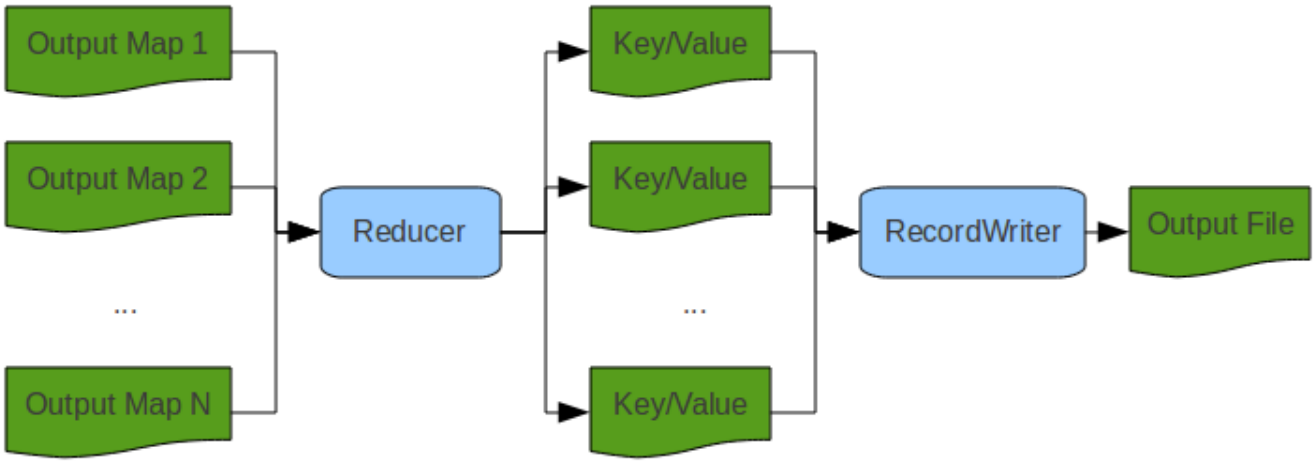

Illustration 2: Output Format Function

We have implemented library implementation classes for text-based input and output formats called `TextInputFormat` and `TextOutputFormat` that can be used directly by users. The `TextSplitter` splits text files into a configurable number of lines. The `LineRecordReader` returns key/value pairs for the form line number/line. The `LineRecordWriter` outputs key/value pairs one line at a time with a configurable separator.

2 User Interface

We have implemented two interfaces that users can use to submit jobs, retrieve status for their jobs, retrieve the job output upon completion, throttle a node, and stop node. One is a command line interface which parses commands typed by the user. Another is a Swing-based graphical user interface, which supports the same functions as the console, but in a more consumable fashion. The GUI is depicted in Illustration 3. Both user interfaces are able to connect to any node in the cluster.

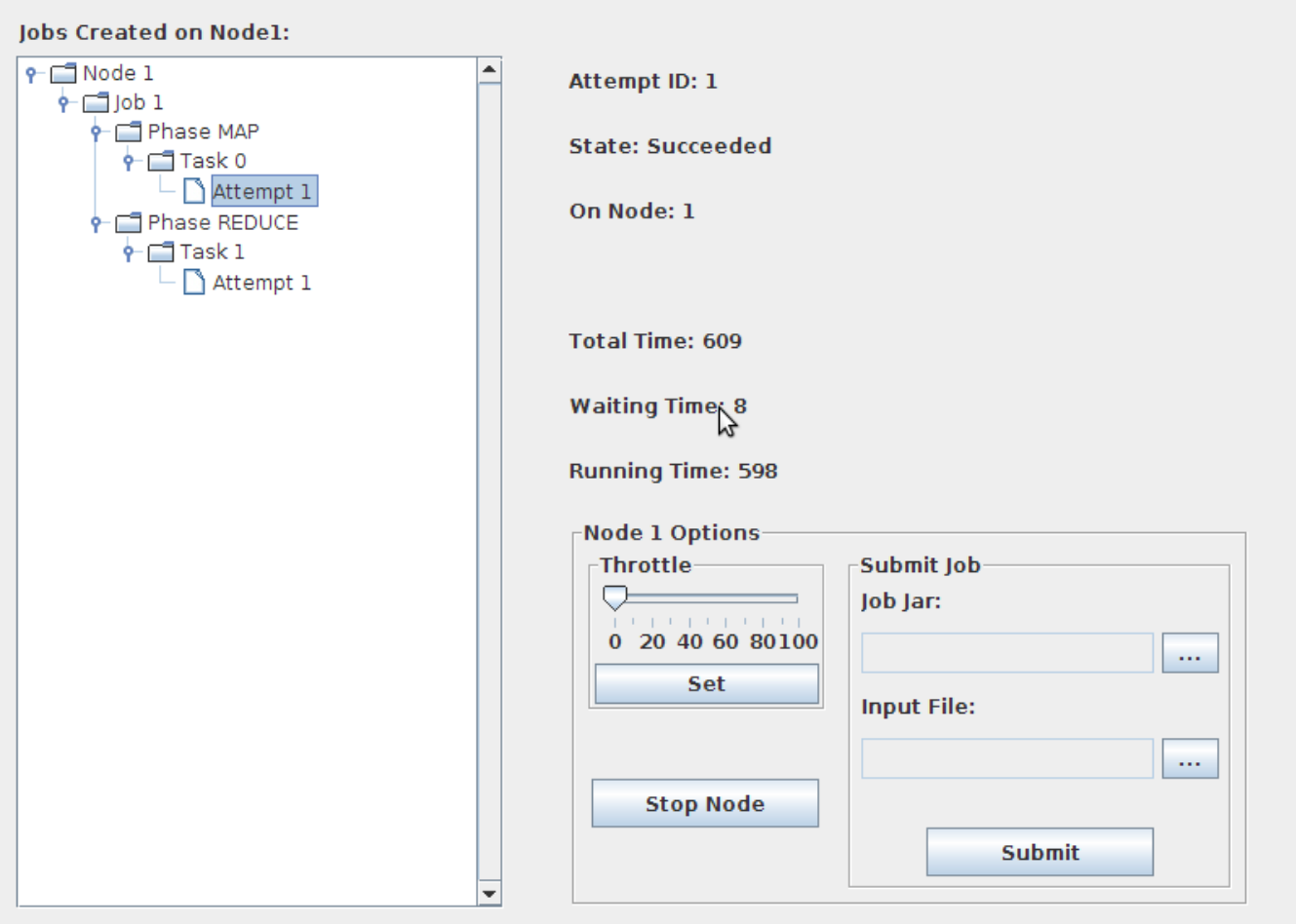

Illustration 3: Graphical User Interface

3 Runtime Design

3.1 Nodes

The runtime consists of one more nodes that collaborate to process jobs submitted by users to different nodes. By default each node is started in a separate VM, but it is possible to run multiple nodes in the same VM. Each node is started for a given configuration captured in a properties file. The configuration contains a unique numerical ID for the node, the port the node should listen on, and the list of remote nodes the should interact with (in the form id:host:port). The configuration also contains properties for the node services that will be discussed in subsequent sections.

3.1.1 Node Startup

Nodes are started via the shell script `bin/start-node.sh`, which calls the main method of the Node class. The main method creates a new node from the default or user-supplied configuration file. The node contains two thread pools: a cached thread pool for system threads such as RPC invocations and asynchronous operations, and a scheduled thread pool for periodic tasks. When a node starts, it first starts an RPC server at the configured port and then starts the individual services provided by the node. The last service to start is the load balancer. This service blocks the startup thread until all remote nodes specified in the node configuration have contacted it with an initial status update. Once the last node

has registered, startup finishes by blocking the main thread until the node is stopped. If a user submits jobs to a node before it has completed startup, the threads are blocked until startup has completed.

3.1.2 Node Shutdown

Nodes are stopped via the shell script `bin/stop-node.sh`, which calls the `stop` method on the RPC interface of the node. By default it uses the node configuration host and port defaults, but the user can specify an alternate host to connect to. When the node receives the request to stop, it stops the node services in reverse order they were started in, stops the RPC server[1], and shuts down the thread pools.

3.2 Services

Each node exposes a set of services that are called by the user interfaces or other nodes to interact with the node. The services are bound to the node's configured port and invocable via the custom RPC protocol (see section TODO). Each node holds client stubs for each other node int the cluster indexed by node ID. Services running on a particular node access this map to look up and invoke services running on other nodes. The node services consist of a file system service, a job management service, a task execution service, and a load balancing service.

3.2.1 File System

The `FileSystemService` is the lowest-level service used by other services to create, read, and delete files on remote nodes. Illustration 4 shows the service interface as well as the `Path` class used as input to several of the offered methods.[2] The path object is a simple serializable wrapper around a string array used to specify file locations (The methods of this class are not shown).

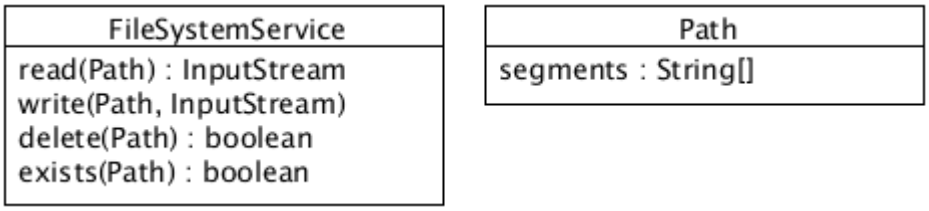

Illustration 4: File System Service

The implementation of this service directly invokes the Java file system implementation, which in turn issues calls against the local file system services. Paths are resolved against a root directory, which is configurable for each node through the configuration properties. No synchronization is implemented as this is not needed given the use of this service within the runtime.

3.2.2 Task Execution

The `TaskExecutorService` is responsible for managing the execution of map and reduce tasks. Illustration 5 shows the methods provided by this service and the input classes (the methods on the input classes are not shown). The job manager service discussed in the next section uses this service to execute tasks for input splits. The files pointed to by the paths in the data structure must be present on

[1] Note that the RPC thread does not actually exit until the current invocation has been processed

[2] Note that the RPC implementation supports passing (non-serializable) InputStream objects as input and output paramters

the executor's local node file system. The executor also offers a method to cancel a task previously submitted. This method takes a timeout to prevent blocking the caller indefinitely in case the task cannot complete. Finally, the service exposes a method to adjust the throttle of the worker threads.

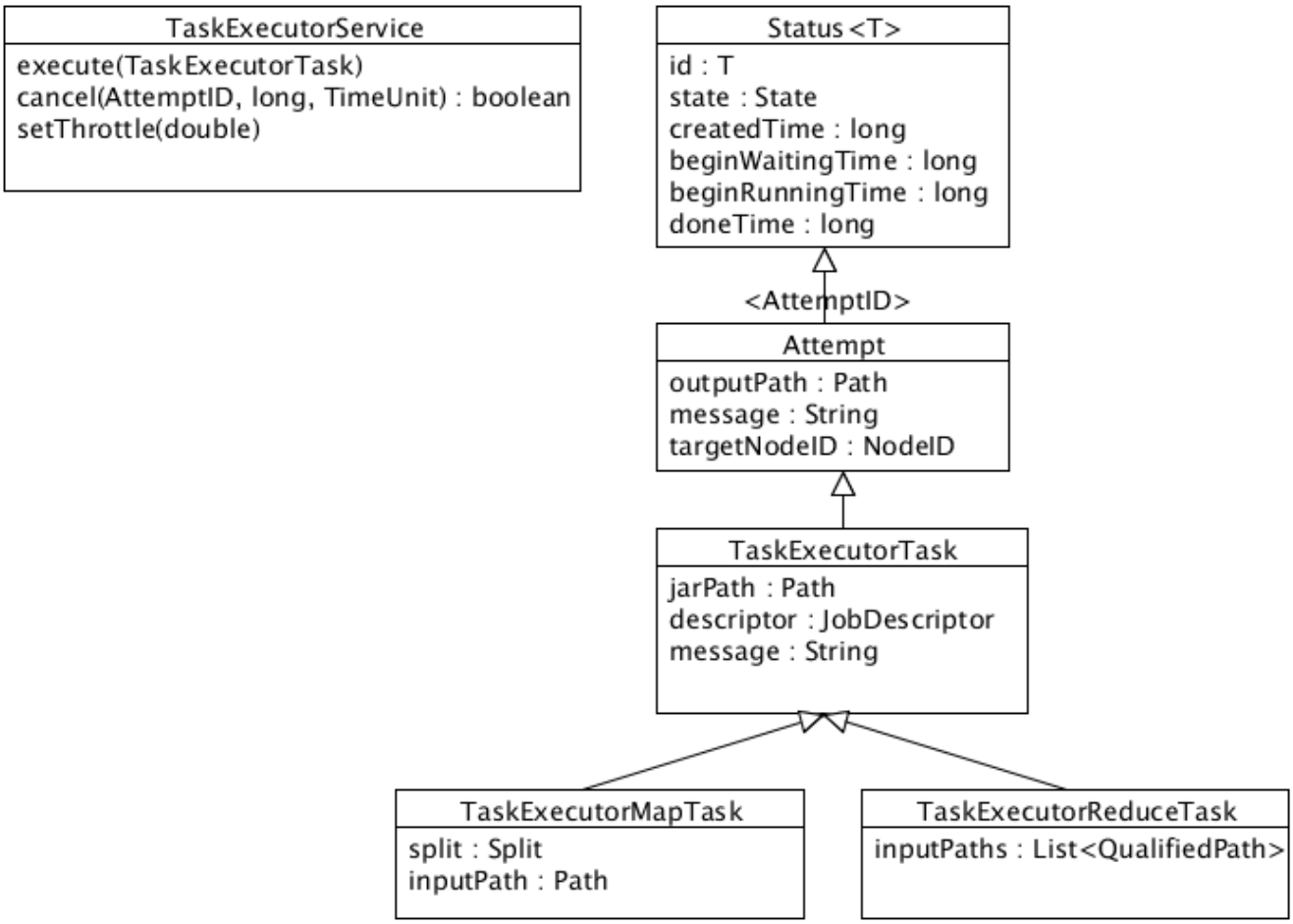

Illustration 5: TaskExecutorService

The implementation of this service uses a fixed-sized thread pool to run tasks. The size of this thread pool is configurable via the node properties. Pending tasks are submitted to a blocking queue maintained by the thread pool. When a task is submitted to the thread pool, the executor service stores a handle to the Future and completion semaphore used to implement the cancel function.

The executor also keeps track of the average running time of the ten most recent tasks to impose throttling on the worker threads, by making them sleep for a given percentage of their running time.

Finally, the executor runs a periodic status update thread, which calls back each job manager in the cluster with the status of the tasks it has submitted to the executor. The rational for this is to avoid chatty network traffic caused by callbacks for each individual task status change. The period for the update thread is adjustable through the node configuration.

The executor synchronizes access to its internal mutable state in order to perform in a thread-safe manner.

3.2.3 Job Management

The `JobManagerService` manages jobs submitted to the node it is running on. Job managers on different nodes are unaware of each other's jobs. Jobs are created and started through the submitJob method, which takes the file paths to the jar and input file. The method returns the JobID, which consists of the node ID and a numeric job id, which together make the JobID unique across the cluster. The getJobStatus method returns an immutable view of the current state of the Job and the subtasks created for it. Once a job has completed, writeOutput can be used to copy the output file to a specific location in the local file system. Finally, updateStatus is used by the TaskExecutor as previously mentioned to update the job manager with the status of the tasks running on the executor on behalf of this job manager. The array is sorted by attempt ID to facilitate processing. The interface to the job manager is depicted in Illustration 6.

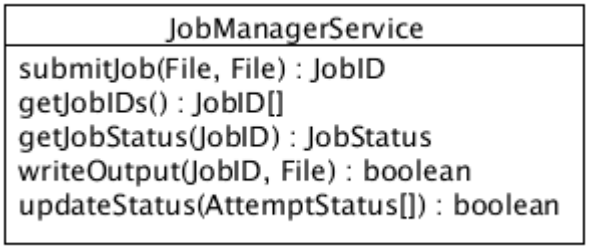

Illustration 6: JobManagerService

The implementation of the submit method returns as soon as it has created the new ID and spawned a thread to distribute the work. The spawned thread creates tasks in a loop until the splitter has reached the end of the input file. Each iteration starts by asking the load balancer to pick a target node for the next task which is then created an registered with the job. Next, the splitter is called to create the split which is streamed directly to the target node using piped input/output streams. The job jar is also copied to the target node if it has not already been copied to it in a previous iteration. Finally, a new task attempt is it created, registered with the task and submitted. The objects created the submitJob method are illustrated below. The links between model objects are implemented using TreeMaps to provide fast random and sequenced access.

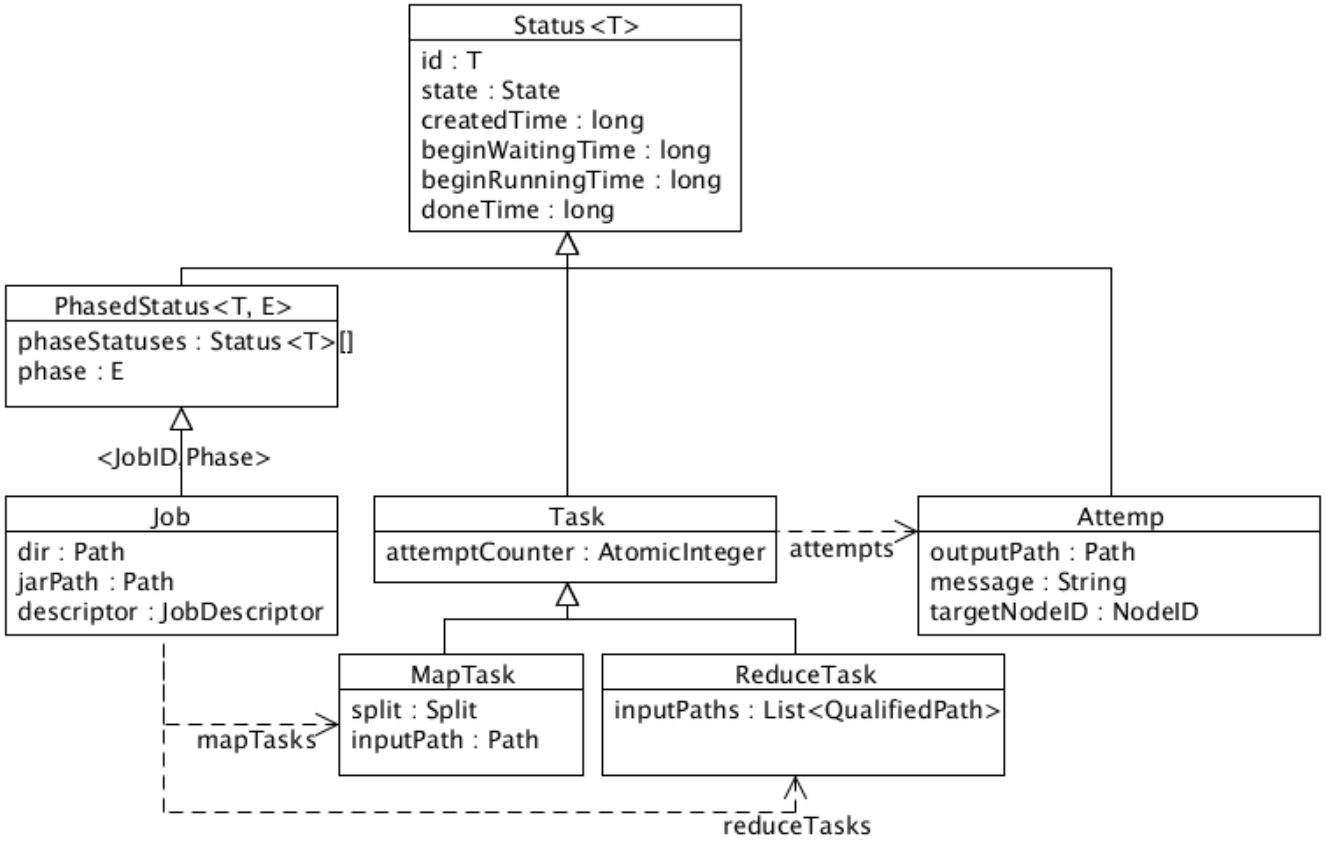

Illustration 7: Jobs, Tasks, and Attempts

When an attempt is submitted to the task executor, the necessary information from the job, task, and attempt is aggregated into a task executor task, which avoids serializing the whole object tree for each attempt. The JobStatus object hierarchy returned to the user looks similar to the diagram shown above, except that all non-status fields are omitted and all fields are immutable to avoid race conditions while traversing or accessing the data structure.

When the job manager receives an attempt status update, it updates the status fields from bottom to top in the hierarchy. The status at the task and job level is derived from the status at the next lower level using specific algorithms. If all tasks in the map phase have succeeded, the update call schedules a thread to create and submit a reduce task.

The reduce task holds the qualified paths, i.e. node ID/path, pairs of all map output files. When run, it merges all sorted map output files into a single sorted file using a multi-way stream sort. The reducer is always scheduled on the job manager node and writes the output file into the local file system.

The job manager also implements a method to migrate a task from one to another node. It does this by canceling the task attempt on the source node and creating/submitting a new task attempt on the target node. The method also copies the jar file and the input file if they are not already present on the target node. The load balancer calls this method if it has determined that a transfer is needed.

3.2.4 Load Balancer

The `LoadBalancerService` exchanges node status across the different nodes in the cluster by calling the updateStatus method on remote nodes in the cluster. The node status contains CPU utilization and stats related to the executor service, including queue length, throttle, worker thread pool size, and

number of active worker threads.

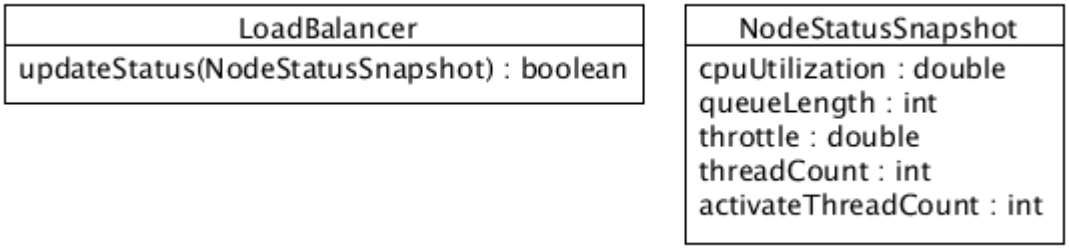

Illustration 8: LoadBalancerService

The load balancer implementation contains a CPU profiler that reads information from /proc/stat upon request and computes the utilization by comparing it to the last read values. It also runs a periodic update thread, which gathers information to create a node status snapshot and sends it off to all other nodes in the cluster. Upon receiving status updates from remote nodes, the load balancer creates or updates a local node status object, which maintains the last five snapshots to compute load averages. The initial blocking of the node startup thread is implemented in the load balancer using Java's built in monitor.

The policies used to decide whether rebalancing is needed and if so how, are configurable via SPI policy implementations. Interfaces are defined for transfer, location, node selection, and selection policy as shown in the illustration below. The transfer policy is invoked every time a status update is received unless a transfer is in progress to decide whether a transfer is needed. The location policy is used to determine the source and target node of the transfer. Only if the source node is the local node does the load balancer initiate a transfer, i.e. the transfer decision is sender initiated. The selection policy selects a task running on the source node to migrate to the target node. Finally, the node selection policy is used during the bootstrap phase to decide which node to send splits to.

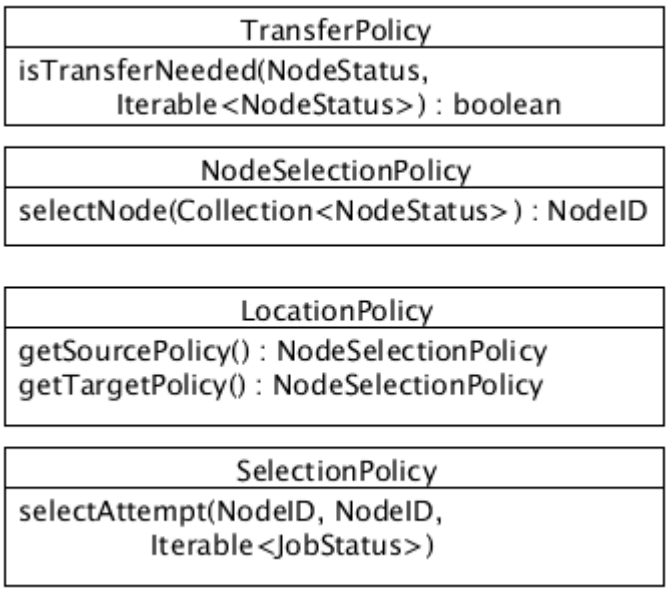

Illustration 9: Policy Interfaces

We have provided default implementations for all policies. Our transfer policy only returns true if the updated node has an empty queue. Our location policy selects a task with a waiting attempt on the local node if that task doesn't already have a non-failed/canceled attempt running on another node. The location policy iterates over the tasks in reverse order to increase the chance of picking a waiting task that will not run soon. Our bootstrap node selection policy simply distributes task in a round-robin fashion. We have also implemented a bootstrap policy that schedules all tasks on the local node to test our load balancing solution.

4 Measurement and Configuration

4.1 Test Configuration

A number of tests were conducted using various configuration options and using different input file sizes and different VM clusters. Except where noted in the table, each test was run with these default values:

- Throttle set to 0
- 1 worker thread per node
- 2000 ms task executor status update interval
- 2000 ms load balancer status update interval
- Round robin bootstrap selection policy
- Idle transfer policy
- Score based location policy
- Waiting task selection policy

- 1,000,000 lines per task
- All tests initiated on Node 1.

The two different VM clusters used were:

a. The provided UIUC CS lab VMs (green column results in table 2)

b. Student's personal server (blue columns below). This setup consisted of a machine running VMWare Workstation on an Intel Core i7 CPU with 4 cores @ 3.07 GHz w/ Hyper-Threading (8 logical cores) and 12 GB of RAM. The VMs on this machine were configured as:
VM Node 1: Fedora 15 on a dual vCPU VM, 1.5 GB RAM, 1 Gbps vNIC
VM Node 2: Fedora 15 on a single vCPU VM, 2.0 GB RAM, 1 Gbps vNIC

Test configuration	70 MB	400 MB	2.3 GB	400 MB	800 MB
1. Baseline – all defaults used	13	67	413	89	169
2. Node 1 @ 50 throttle, node 2 @ 0 throttle	12	69	427	98	187
3. Node 1 @ 100 throttle, node 2 @ 0 throttle	12	57	466	91	168
4. Both nodes 2 worker threads, 0 throttle	14	57	422	91	166
5. Both nodes 5 worker threads, 0 throttle	14	62	463	91	166
6. LocalNodeSelectionPolicy, transfers allowed	18	65	406	94	170
7. LocalNodeSelectionPolicy, never transfer	18	57	354	91	165
8. CPU profiling and task executor update set to 10 sec	19	69	376	101	170
9. IdlestNodeSelectionPolicy	12	60	349	91	165
10. BusiestNodeSelectionPolicy	14	71	314	90	165
11. Lines per task set to 100,000	36	203	-	-	-
12. Lines per task set to 5,000,000	13	387	-	-	-
13. Lines per task set to 10,000,000	20	-	-	-	-

Table 2: Test Result completion times. All times in seconds.

5 Summary of Improvements Implemented

We implemented the following improvements over what was required for this MP:

1. support for user-defined map reduce functions
2. support for custom input formats for splitting and reading input files
3. support for more than 2 nodes
4. ability to submit arbitrary number of jobs concurrently to any node in the cluster
5. configurable runtime properties, e.g. number of threads, status update interval, port, throttle
6. configurable job properties, e.g. input split size, output key/value separator
7. command line interface to start, stop, and execute commands on any node (local or remote)

8. graphical user interface to submit jobs, view status, retrieve output, set throttle, stop nodes
9. report runtime metrics (total time, wait time, running time) for jobs, task, and attempt s
10. bootstrap policy controlling selection of node for initial task distribution
11. pluggable policy framework

6 Resources

[1] Jeffrey Dean and Sanjay Ghemawat. Mapreduce: simplified data pro- cessing on large clusters. In *OSDI'04: Proceedings of the 6th conference on Symposium on Opearting Systems Design & Implementation*, pages 10–10, Berkeley, CA, USA, 2004. USENIX Association.