6CS030 Lecture 5

Introduction to Hadoop

Hadoop and Map Reduce

Hadoop

- Open-source software framework used for distributed storage and processing of big datasets
- Can be set up over a cluster of computers built from normal, commodity hardware
- Many vendors offer their implementation of a Hadoop stack (e.g. Amazon, Cloudera, Dell, Oracle, IBM, Microsoft)

History of Hadoop

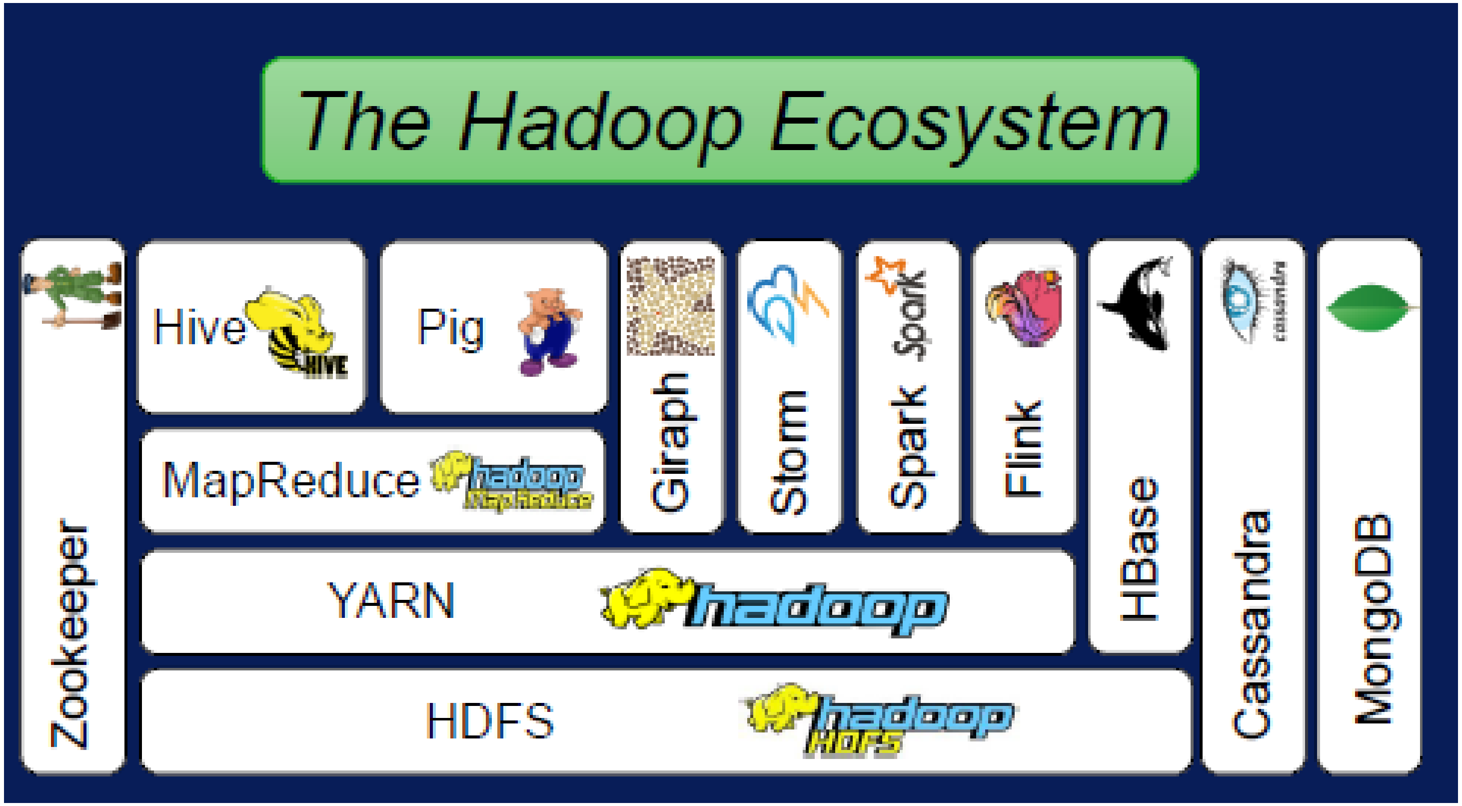
- Key building blocks:
  - Google File System: a file system that could be easily distributed across commodity hardware, whilst providing fault tolerance
  - Google MapReduce: a programming paradigm to write programs that can be automatically parallelized and executed across a cluster of different computers
- Nutch web crawler prototype developed by Doug Cutting
  - Later renamed to Hadoop
- In 2008, Yahoo! open-sourced Hadoop as "Apache Hadoop"

The Hadoop Stack

- Four modules:
  - Hadoop Common: a set of shared programming libraries used by the other modules
  - Hadoop Distributed File System (HDFS): a Java-based file system to store data across multiple machines
  - MapReduce framework: a programming model to process large sets of data in parallel
  - YARN (Yet Another Resource Negotiator): handles the management and scheduling of resource requests in a distributed environment

Hadoop Ecosystem

- Lots of applications associated with Hadoop



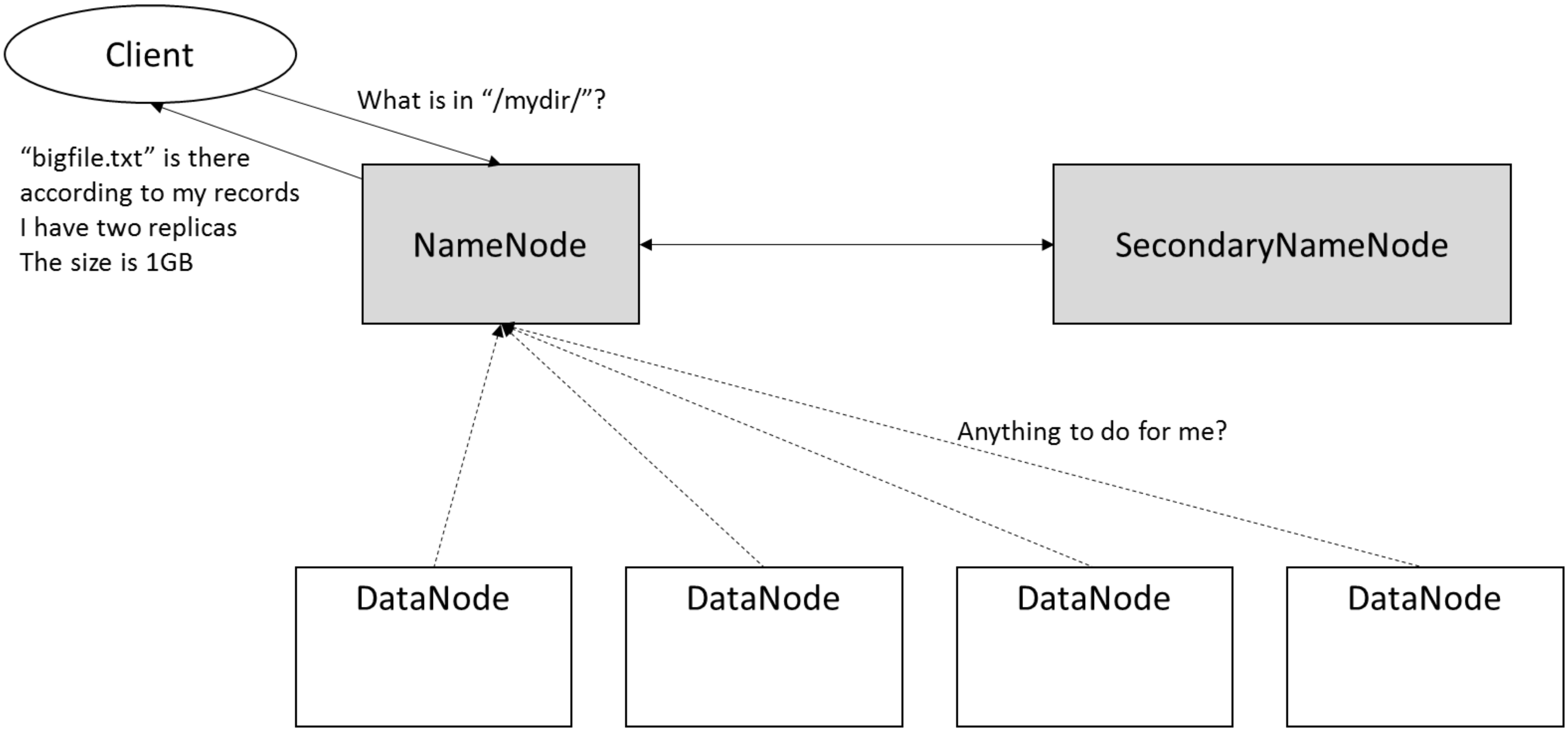
Hadoop Distributed File System (HDFS)

- Distributed file system to store data across a cluster of commodity machines
- High emphasis on fault-tolerance
- HDFS cluster is composed of a NameNode and various DataNodes

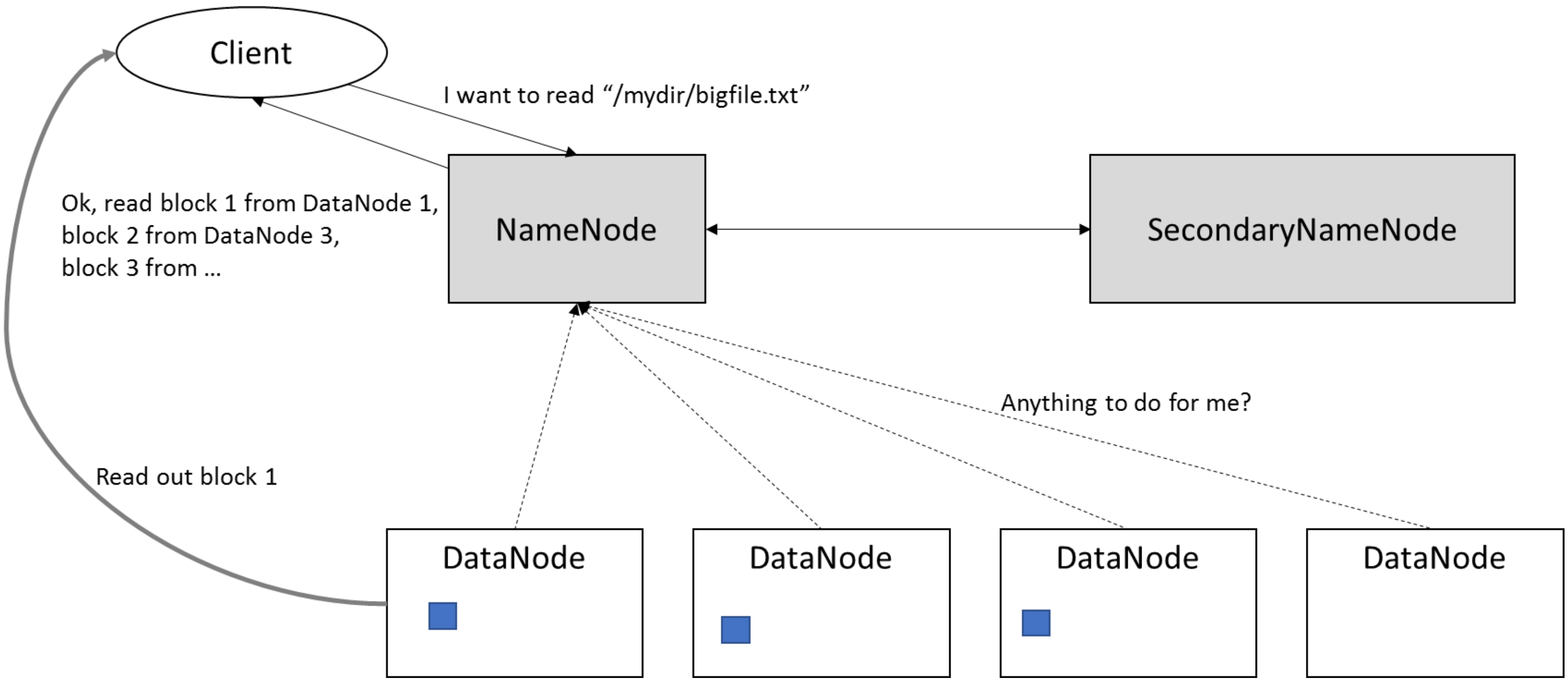
Hadoop Distributed File System (HDFS)

- NameNode
  - a server which holds all the metadata regarding the stored files
  - manages incoming file system operations
  - maps data blocks (parts of files) to DataNodes
- DataNode
  - handles file read and write requests
  - create, delete and replicate data blocks amongst their disk drives
  - continuously loop, asking the NameNode for instructions.
- Note: size of 1 data block is typically 64 megabytes

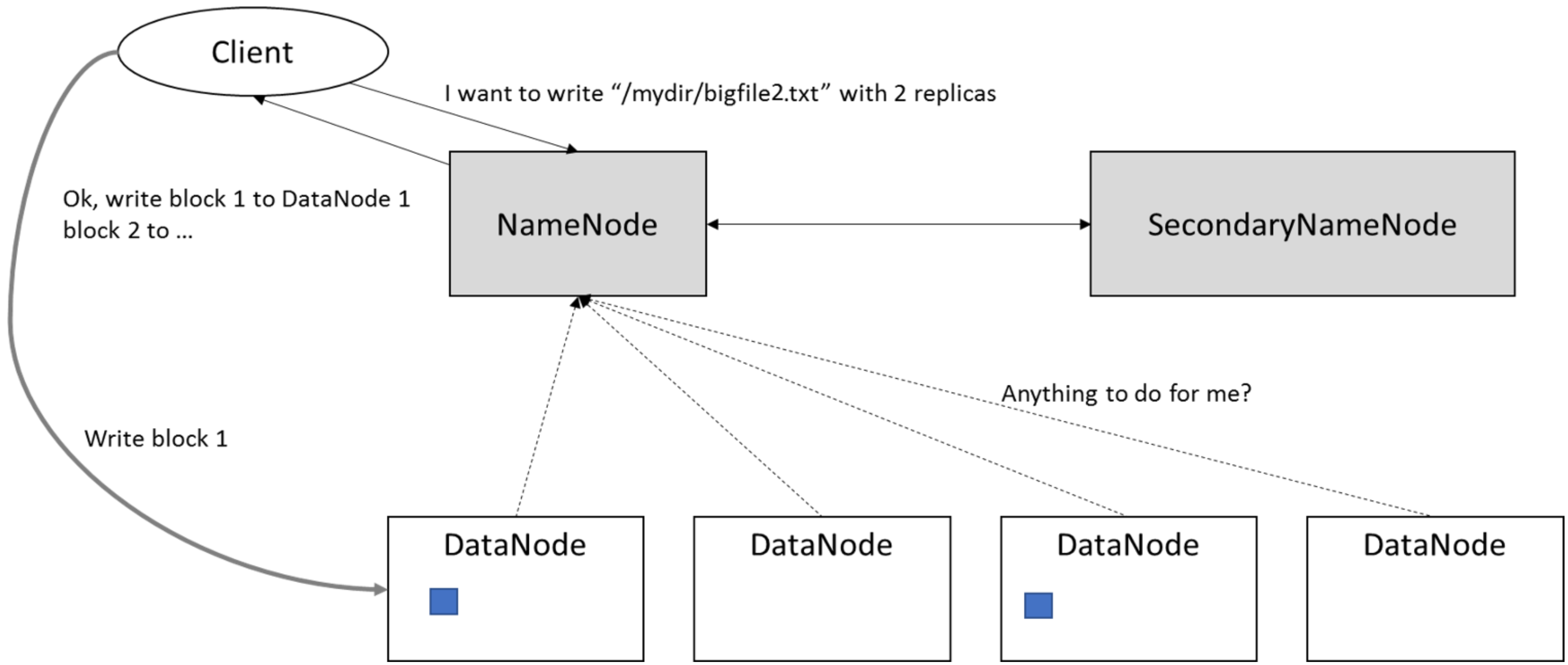
Hadoop Distributed File System (HDFS)



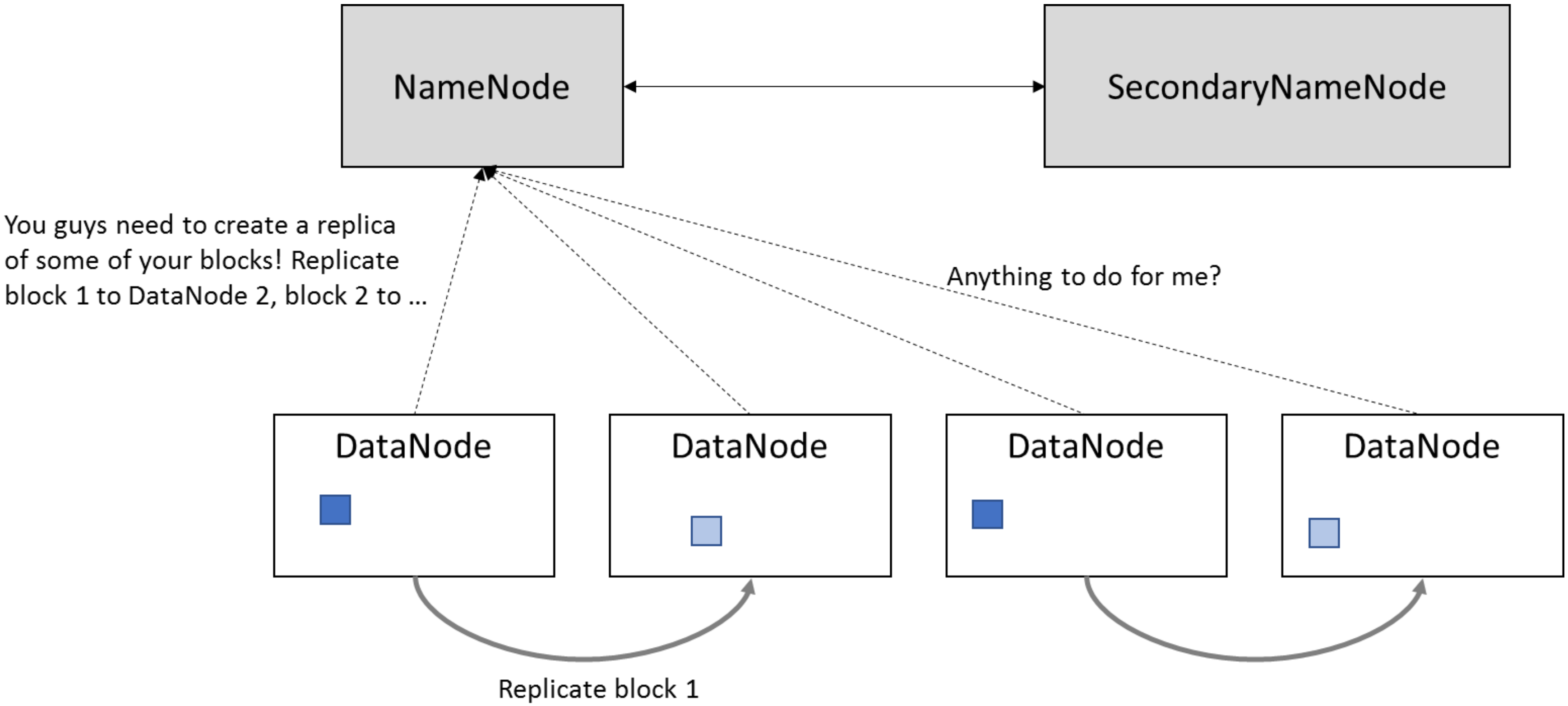
Hadoop Distributed File System (HDFS)



Hadoop Distributed File System (HDFS)



Hadoop Distributed File System (HDFS)



Hadoop Distributed File System (HDFS)

- HDFS provides a native Java API to allow for writing Java programs that can interface with HDFS

```
String filePath = “/data/all_my_customers.csv”;
Configuration config = new Configuration();
org.apache.hadoop.fs.FileSystem hdfs =
        org.apache.hadoop.fs.FileSystem.get(config);
org.apache.hadoop.fs.Path path = new
        org.apache.hadoop.fs.Path(filePath);
org.apache.hadoop.fs.FSDataInputStream inputStream = hdfs.open(path);
byte[] received = new byte[inputStream.available()];
inputStream.readFully(received);


// ...
org.apache.hadoop.fs.FSDataInputStream inputStream = hdfs.open(path);
byte[] buffer=new byte[1024]; // Only handle 1KB at once
int bytesRead;
while ((bytesRead = in.read(buffer)) > 0) {
    // Do something with the buffered block here
}
```

Hadoop Distributed File System (HDFS)

`hdfs dfs -mkdir mydir`	Create a directory on HDFS
`hdfs dfs -ls`	List files and directories on HDFS
`hdfs dfs -cat myfile`	View a file's content
`hdfs dfs -put myfile mydir`	Store a file on HDFS
`hdfs dfs -rm myfile`	Delete a file on HDFS
`hdfs dfs -touchz myfile`	Create an empty file on HDFS
`hdfs dfs -stat myfile`	Check the status of a file (file size, owner, …)
`hdfs dfs -test -e myfile`	Check if file exists on HDFS
`hdfs dfs  -test -z myfile`	Check if file is empty on HDFS
`hdfs dfs  -test -d myfile`	Check if myfile is a directory on HDFS
`hdfs dfs –du`	Check disk space usage on HDFS

The commands in blue are the key ones you will most likely use

MapReduce

- Programming paradigm made popular by Google and subsequently implemented by Apache Hadoop
- Focus on scalability and fault tolerance
- A map-reduce pipeline starts from a series of values and maps each value to an output using a given mapper function

MapReduce

- A MapReduce pipeline in Hadoop starts from a list of key-value pairs, and maps each pair to one or more output elements
- The output elements are also key-value pairs
- Next, the output entries are grouped so all output entries belonging to the same key are assigned to the same worker (e.g. physical machine)
- These workers then apply the reduce function to each group, producing a new list of key-value pairs
- The resulting, final outputs can then be sorted

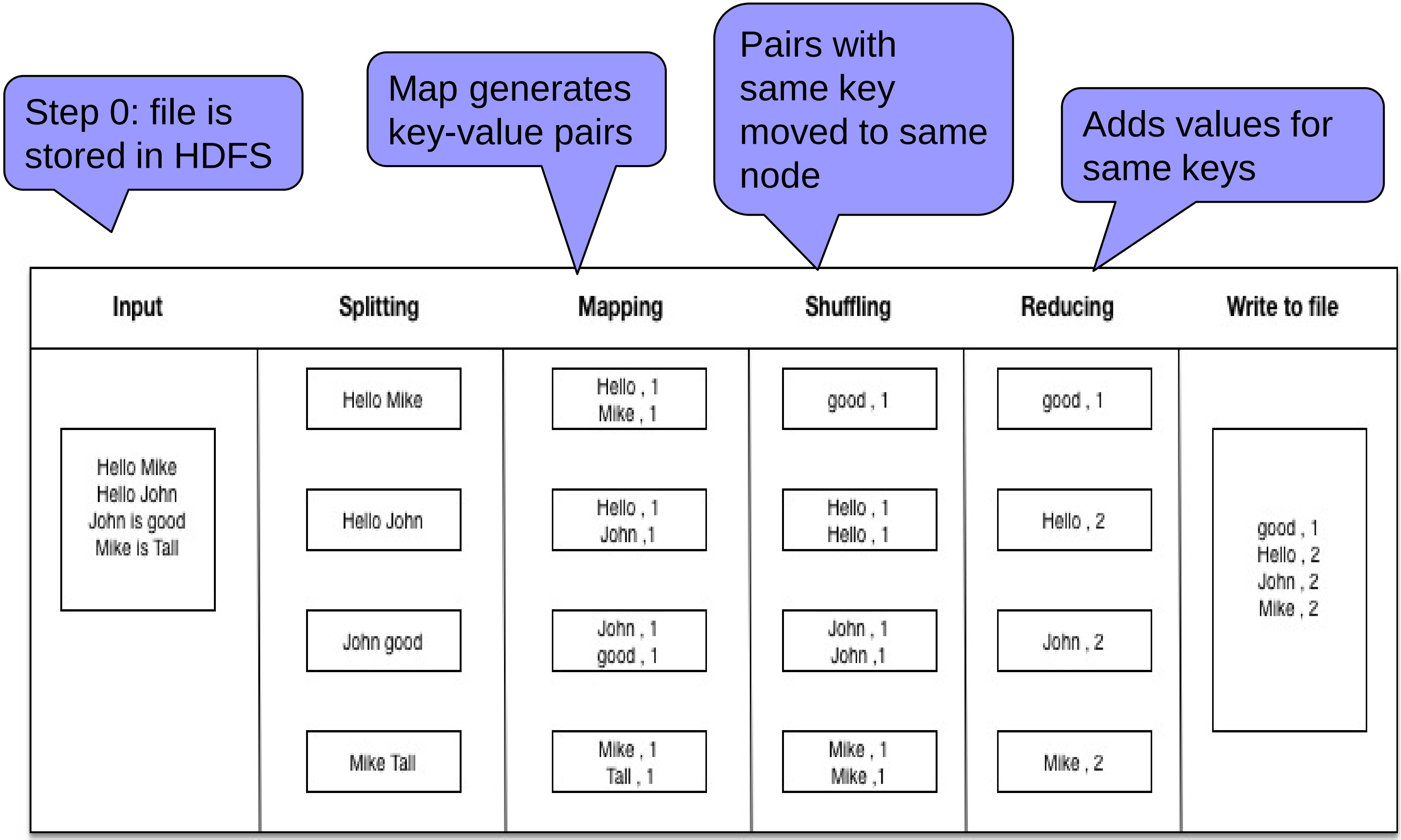
MapReduce

- Reduce-workers can already get started on their work even although not all mapping operations have finished yet
- Implications:
  - the reduce function should output the same key-value structure as the one emitted by the map function
  - the reduce function itself should be built in such a way so it provides correct results, even if called multiple times

Map Reduce Algorithm

- Map–reduce: a programming pattern for analysing streams or sets of data
- The computation takes a set of input key/value pairs, and produces a set of output key/value pairs. The user of the MapReduce library expresses the computation as two functions: Map and Reduce.
- Map takes an input pair and produces a set of intermediate key/value pairs. The MapReduce library groups together all intermediate values associated with the same intermediate key I and passes them to the Reduce function.
- Reduce accepts an intermediate key I and a set of values for that key.
  - It merges together these values to form a possibly smaller set of values.
  - Typically just zero or one output value is produced per Reduce invocation.
  - The intermediate values are supplied to the user's reduce function via an iterator.
  - This allows us to handle lists of values that are too large to manage in memory.

Map Reduce Count Example



MapReduce

- In Hadoop, MapReduce tasks are written in Java
  - Can also use Python, but is converted to Java
- To run a MapReduce task, a Java program is packaged as a JAR archive and launched as:

```
hadoop jar myJarFile.jar myJavaClass [args...]
```

- You first need to compile the Java file:

```
javac -classpath $(hadoop classpath) -d myClassDir
myJavaClass.java
```

- Then produce the Jar file:

```
cd myClassDir    # cd to where the class files are
jar cf myJarFile.jar classesRequired*.class # note there
can be a number of classes produced
```

MapReduce Example – Word Count

- Word Count is the "Hello World" of Hadoop!
- This Java example counts the appearance of a word in a file:

```
import java.io.IOException;
import java.util.StringTokenizer;

import org.apache.hadoop.conf.Configuration;
import org.apache.hadoop.fs.Path;
import org.apache.hadoop.io.IntWritable;
import org.apache.hadoop.io.Text;
import org.apache.hadoop.mapreduce.Job;
import org.apache.hadoop.mapreduce.Mapper;
import org.apache.hadoop.mapreduce.Reducer;
import org.apache.hadoop.mapreduce.lib.input.FileInputFormat;
import org.apache.hadoop.mapreduce.lib.output.FileOutputFormat;

public class WordCount {
  // Following fragments will be added here
}
```

https://hadoop.apache.org/docs/stable/hadoop-mapreduce-client/hadoop-mapreduce-client-core/MapReduceTutorial.html#Inputs_and_Outputs

MapReduce - Mapper

- Define mapper function as a class extending the built-in mapper class:

```
Mapper<KeyIn, ValueIn, KeyOut,
ValueOut>
```

- Need to indicate which type of key-value input pair we expect and which type of key-value output pair our mapper will emit

MapReduce

```
public class WordCount {
        public static class TokenizerMapper
          extends Mapper<Object, Text, Text, IntWritable> {
        private final static IntWritable one = new IntWritable(1);
        private Text word = new Text();
```

Can’t use standard Java types such as String, so use Text for String type data

"context" is used to emit output values

```
        public void map(Object key, Text value, Context context )
          throws IOException, InterruptedException {
                StringTokenizer itr = new StringTokenizer(value.toString());
                while (itr.hasMoreTokens()) {
                        word.set(itr.nextToken());
                        context.write(word, one);
                }
        }
}
```

MapReduce

Input key-value pairs	
Key <Object>	Value <Text>
0	This is the first line
23	And this is the second line, and this is all

Mapped key-value pairs	
Key <Text>	Value <IntWritable>
this	1
is	1
the	1
first	1
line	1
and	1
…	…

MapReduce

- The reducer function is specified as a class extending the built-in class:

```
Reducer<KeyIn, ValueIn, KeyOut,
ValueOut>
```

MapReduce

```
public static class IntSumReducer extends Reducer
   <Text, IntWritable,Text,IntWritable> {
  private IntWritable result = new IntWritable();

  public void reduce(Text key,
          Iterable<IntWritable> values, Context context )
          throws IOException, InterruptedException {
                int sum = 0;
                for (IntWritable val : values)  {
                        sum += val.get(); }
                result.set(sum);
                context.write(key, result);
} }
```

IntWritable used for numeric types

Summarises the results so far

Outputs a (word, sum) pair

MapReduce

Mapped key-value pairs	
Key <Text>	Value <IntWritable>
this	1
is	1
the	1
first	1
line	1
and	1
this	1
is	1

Mapped key-value pairs for “this”	
Key <Text>	Value <IntWritable>
this	1
this	1

Reduced key-value pairs for “this”	
Key <Text>	Value <IntWritable>
this	1 + 1 = 2

MapReduce

```
public static void main(String[] args) throws Exception {
        Configuration conf = new Configuration();
        Job job = Job.getInstance(conf, "word count");

        job.setJarByClass(WordCount.class);
        job.setMapperClass(TokenizerMapper.class);
        job.setCombinerClass(IntSumReducer.class);
        job.setReducerClass(IntSumReducer.class);
        job.setOutputKeyClass(Text.class);
        job.setOutputValueClass(IntWritable.class);

        FileInputFormat.addInputPath(job, new Path(args[0]));
        FileOutputFormat.setOutputPath(job, new Path(args[1]));
        System.exit(job.waitForCompletion(true) ? 0 : 1);
}
```

Sets up Map Reduce job with a short name

Tells Hadoop which JAR it needs to distribute to workers

Sets Mapper class

Sets Reducer class

Sets output classes

The program expects two arguments, the first one is the input directory on HDFS and second the output directory

MapReduce

- Before the program can be run you need to compile it first:

```
javac -classpath $(hadoop classpath) -d classDir WordCount.java
```

- Then produce the Jar file:

```
cd classDir # cd to where the class files are
jar cf wordcount.jar Word*.class
```

- Use vi or nano to create 2 testfiles:

  testfile1:

  A long time ago in a galaxy far far away

  testfile2:

  Another episode of Star Wars

- Put these in the hdfs input area.
- By default you have an input directory already created. Use –put to save the files there:

  hdfs dfs –put testfile? /user/yourStudentNo/input

- The output directory must not exist already. If you have already run the program it can be deleted first using –rm:

  hdfs dfs –rm –R /user/yourStudentNo/output_dir

MapReduce

Needs to contain the input files

Must **not** exist beforehand

`hadoop jar wordcount.jar WordCount /user/myDir/input_wc /user/myDir/output_wc`

```
testuser2@sml:~/java/classDir$ hadoop jar wordcount.jar WordCount /user/testuser2/input_wc /user/testuser2/output_wc
2019-03-04 17:39:00,884 INFO client.RMProxy: Connecting to ResourceManager at localhost/127.0.0.1:8050
2019-03-04 17:39:01,364 WARN mapreduce.JobResourceUploader: Hadoop command-line option parsing not performed. Implement the Tool interface an
d execute your application with ToolRunner to remedy this.
2019-03-04 17:39:01,376 INFO mapreduce.JobResourceUploader: Disabling Erasure Coding for path: /tmp/hadoop-yarn/staging/testuser2/.staging/jo
b_1551235767797_0068
2019-03-04 17:39:01,565 INFO input.FileInputFormat: Total input files to process : 2
2019-03-04 17:39:01,606 INFO mapreduce.JobSubmitter: number of splits:2
2019-03-04 17:39:01,731 INFO mapreduce.JobSubmitter: Submitting tokens for job: job_1551235767797_0068
2019-03-04 17:39:01,733 INFO mapreduce.JobSubmitter: Executing with tokens: []
2019-03-04 17:39:01,897 INFO conf.Configuration: resource-types.xml not found
2019-03-04 17:39:01,898 INFO resource.ResourceUtils: Unable to find 'resource-types.xml'.
2019-03-04 17:39:01,958 INFO impl.YarnClientImpl: Submitted application application_1551235767797_0068
2019-03-04 17:39:01,992 INFO mapreduce.Job: The url to track the job: http://sml:8088/proxy/application_1551235767797_0068/
2019-03-04 17:39:01,993 INFO mapreduce.Job: Running job: job_1551235767797_0068
2019-03-04 17:39:08,084 INFO mapreduce.Job: Job job_1551235767797_0068 running in uber mode : false
2019-03-04 17:39:08,085 INFO mapreduce.Job:  map 0% reduce 0%
2019-03-04 17:39:12,142 INFO mapreduce.Job:  map 100% reduce 0%
2019-03-04 17:39:17,175 INFO mapreduce.Job:  map 100% reduce 100%
2019-03-04 17:39:17,187 INFO mapreduce.Job: Job job_1551235767797_0068 completed successfully
2019-03-04 17:39:17,285 INFO mapreduce.Job: Counters: 53
        File System Counters
                FILE: Number of bytes read=156
                FILE: Number of bytes written=646457
                FILE: Number of read operations=0
                FILE: Number of large read operations=0
                FILE: Number of write operations=0
                HDFS: Number of bytes read=310
                HDFS: Number of bytes written=94
                HDFS: Number of read operations=11
                HDFS: Number of large read operations=0
                HDFS: Number of write operations=2
```

If successful, will see above, plus many more lines of output….

MapReduce

- To see what is in your output directory:

  hdfs dfs -ls /user/testuser2/output_wc

- Should output:

  Found 2 items

  -rw-r--r--   1 testuser2 hadoop   0 2019-03-04 17:39 /user/testuser2/output_wc/_SUCCESS

  -rw-r--r--   1 testuser2 hadoop   94 2019-03-04 17:39 /user/testuser2/output_wc/part-r-00000

- To see what is in the output file:

  hdfs dfs -cat /user/testuser2/output_wc/part-r-00000

- Sample output:

A	1
Another	1
Star	1
Wars	1
a	1
ago	1
away	1
episode	1
far	2
galaxy	1
in	1
long	1
of	1
time	1

- Sample input:

  A long time ago in a galaxy far far away

  Another episode of Star Wars

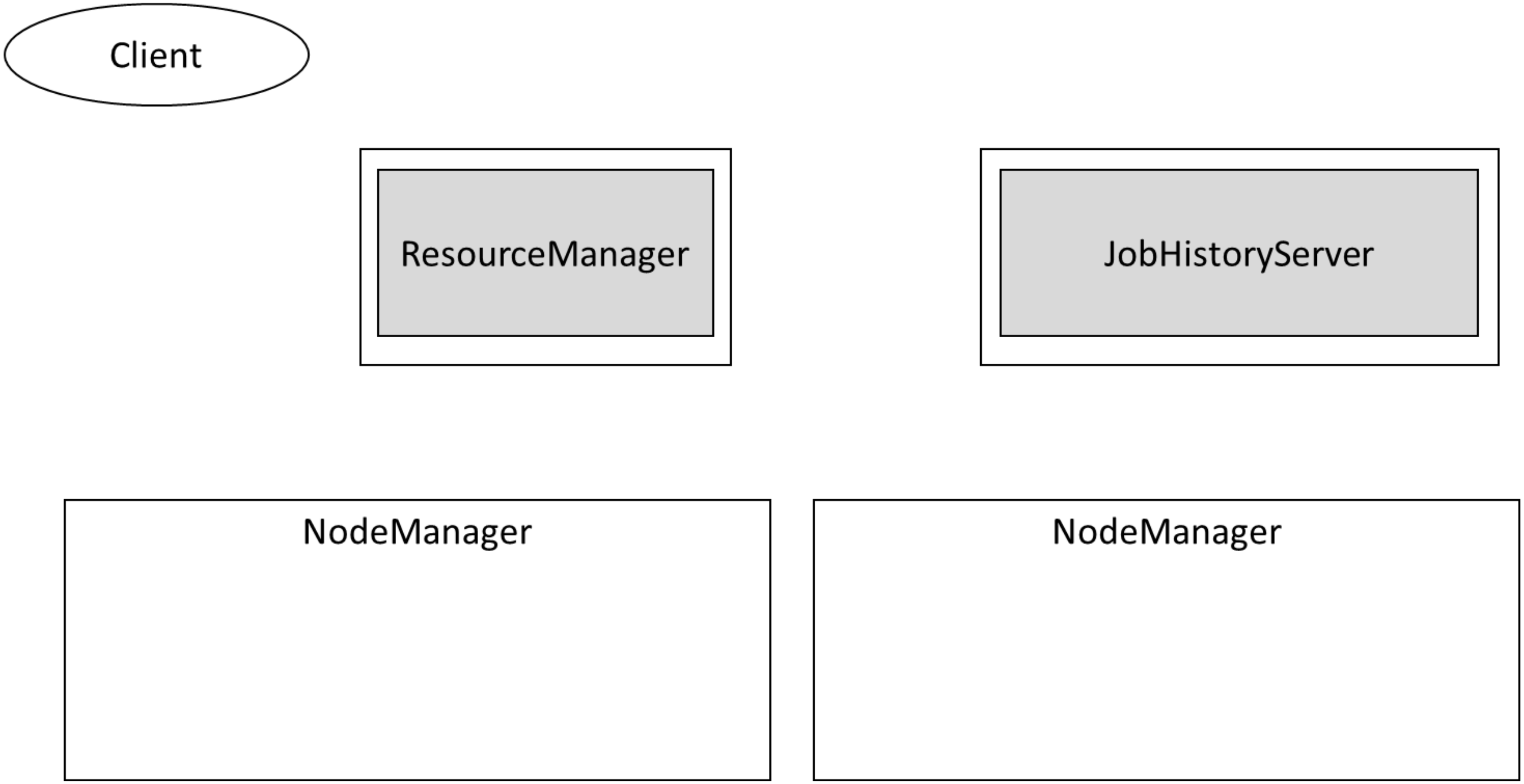
MapReduce

- Constructing MapReduce programs requires a certain skillset in terms of programming
- Tradeoffs in terms of speed, memory consumption, and scalability

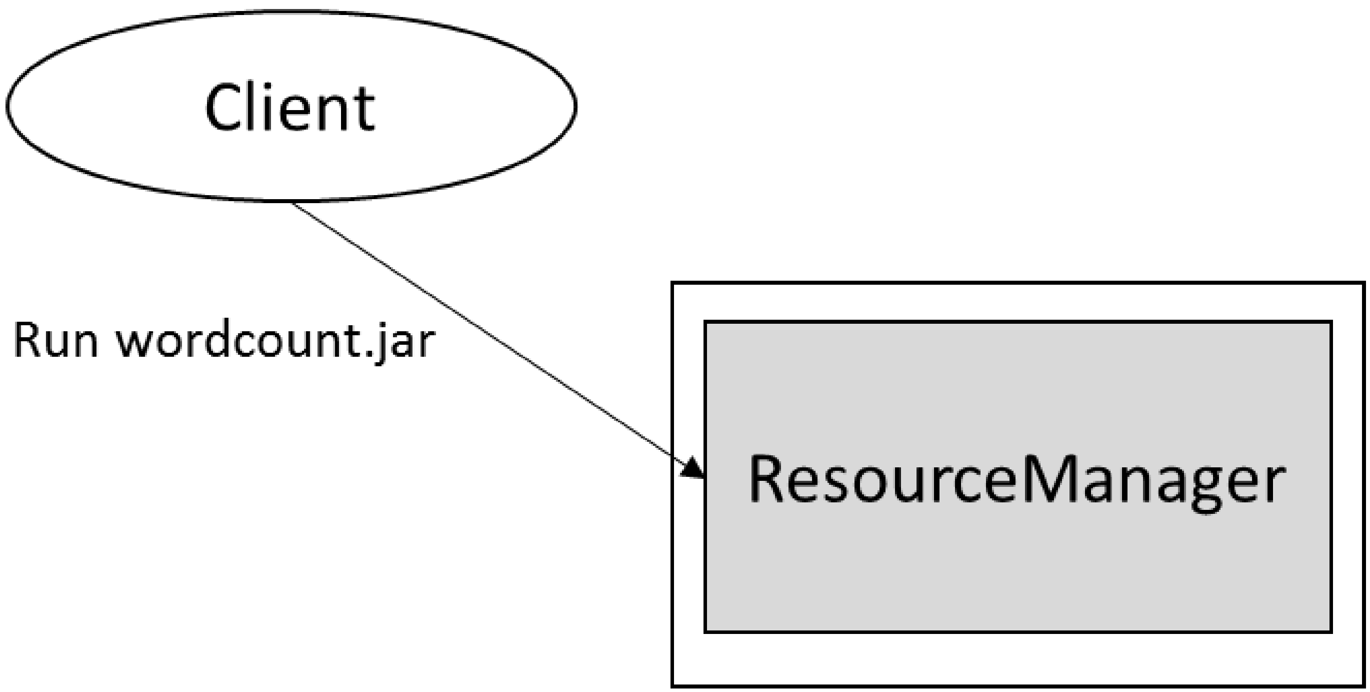
Yet Another Resource Negotiator (YARN)

- Yet Another Resource Negotiator (YARN) distributes a MapReduce program across different nodes and takes care of coordination
- Three important services
  - ResourceManager: a global YARN service that receives and runs applications (e.g., a MapReduce job) on the cluster
  - JobHistoryServer: keeps a log of all finished jobs
  - NodeManager: responsible to oversee resource consumption on a node

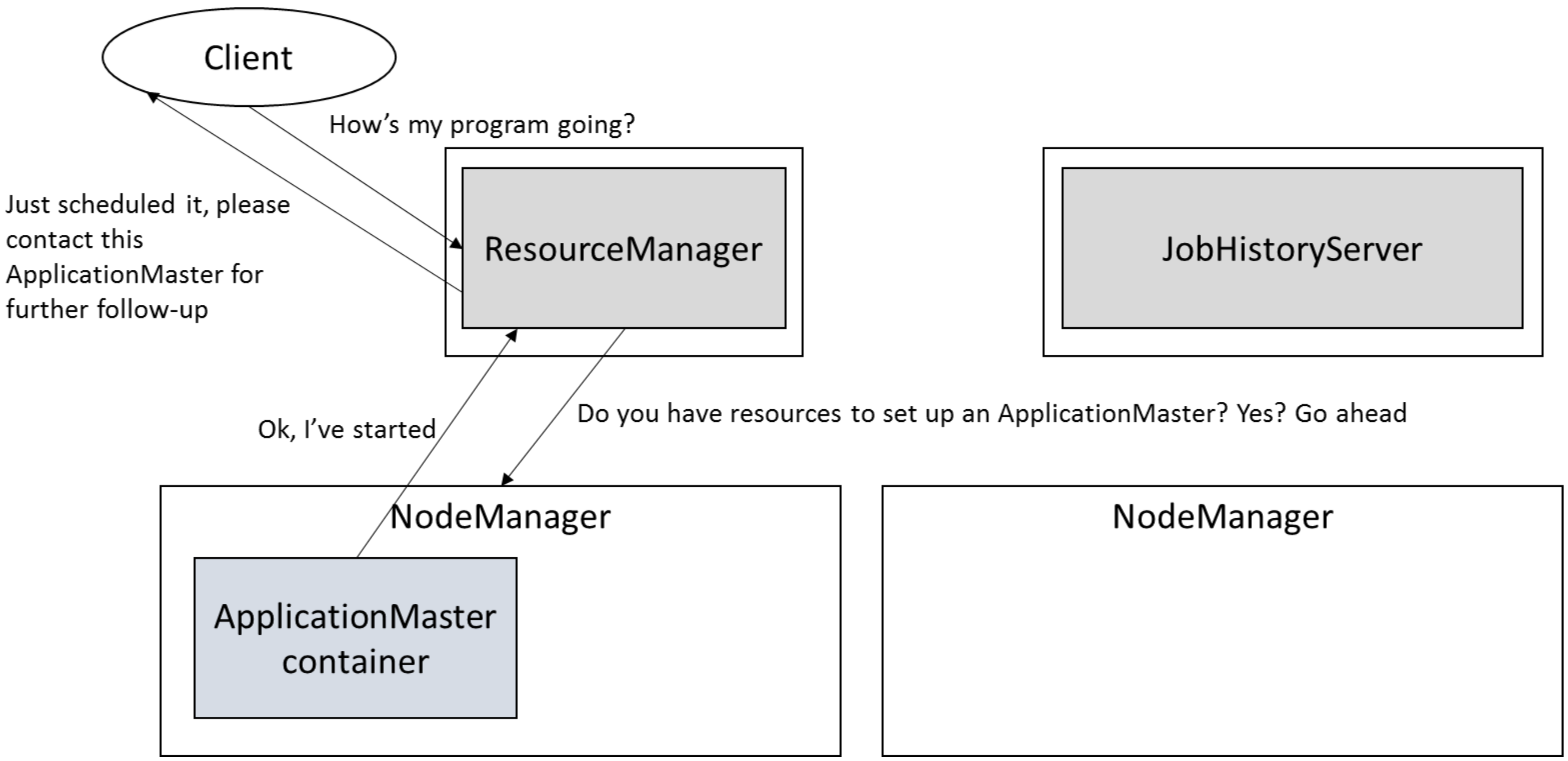
Yet Another Resource Negotiator (YARN)



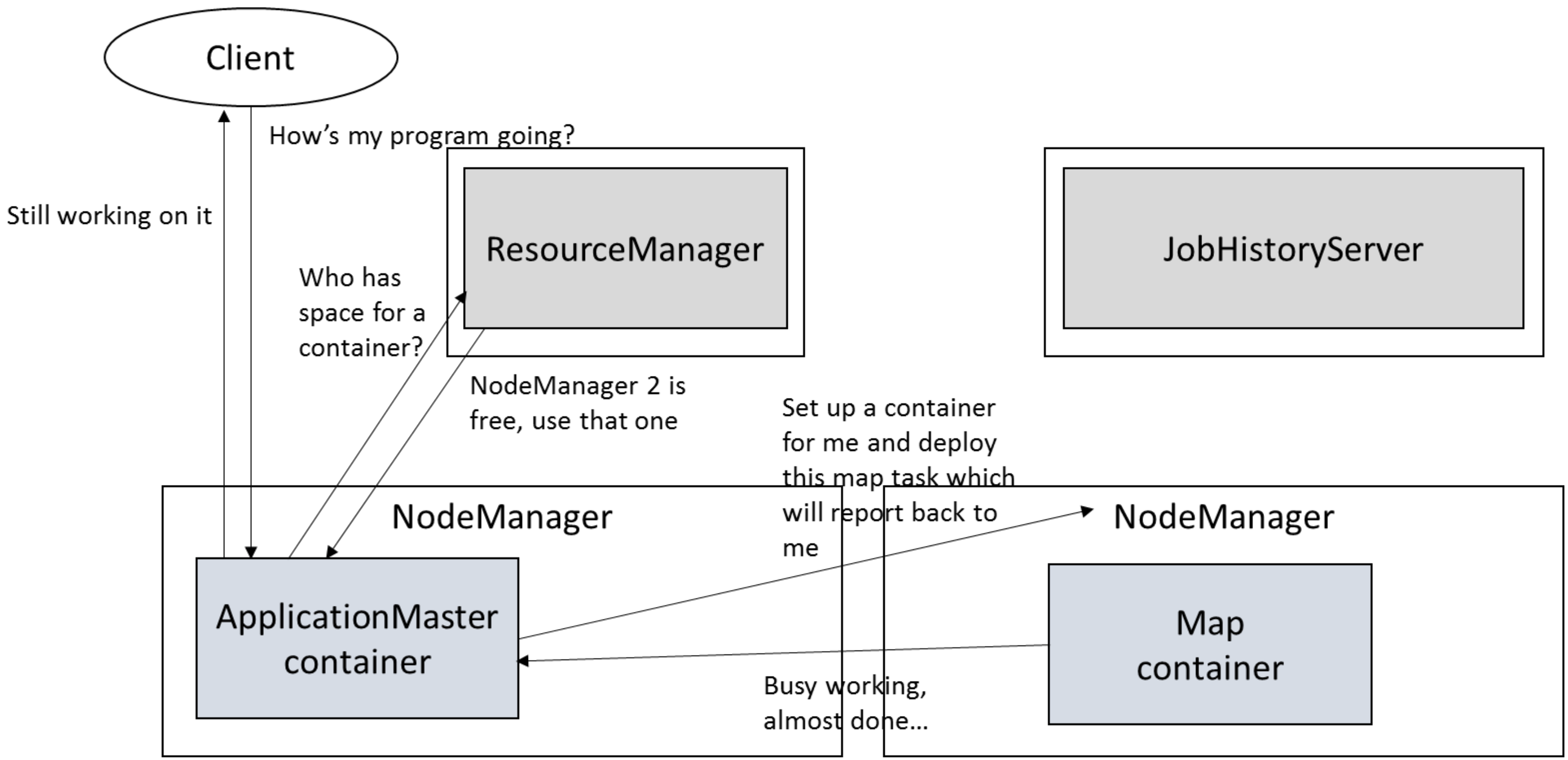
Yet Another Resource Negotiator (YARN)



Yet Another Resource Negotiator (YARN)



Yet Another Resource Negotiator (YARN)



Yet Another Resource Negotiator (YARN)

- Complex setup
- Allows to run programs and applications other than MapReduce

Conclusion

- This lecture has:
  - Introduced Hadoop
  - Looked at using Map Reduce with Hadoop
  - See the Workshop material for some examples using Java and Python
- Next week will look at:
  - SQL on Hadoop
  - Apache Spark