# Introduction to REEF

It is instructive to first remind us of how MapReduce works: Hadoop Map-Reduce schedules compute tasks on containers for executing map and reduce functions on record data. The basic structure of a Map-Reduce job is as follows. For each input block, schedule a map task that passes each internal record to a user-defined map function and materializes the output in key-sorted order. Further, assign a user-defined number of reduce tasks to non-overlapping portions of the key-space from the map output, then shuffle it across the network to where the respective reduce task is scheduled. For each reduce task, perform a global key-based sort on the shuffled data, group it by key and call the reduce function on each record group; storing the output in a durable fashion (i.e., HDFS).

From the perspective of the scheduler, a number of issues arise that must be appropriately handled in order to scale-out to massive datasets. First, each map task should be scheduled close to where the input block resides; ideally on the same machine or rack. Second, failures can occur at the task level at any step; requiring backup tasks to be scheduled or the job being aborted. Third, performance bottlenecks can cause an imbalance in the task-level progress. The scheduler must react to these stragglers by scheduling clones and incorporating the logical task that crosses the finish line first.

Anyone of these issues can limit the scale-out degree of a Map-Reduce job. In what follows, we prescribe a scheduler framework that provides task life-cycle management mechanisms. Using this framework, we developed a complete version of the Map-Reduce runtime that addresses the above issues. Our framework is designed around three components.

1. Driver: captures the client code embodying the scheduler
2. Evaluator: provides a runtime environment on a container.
3. Task: encapsulates the task-level client code to be executed in an Evaluator.

Below, we describe the client facing interfaces to these components. The core REEF control flow design is based on the reactive extensions (Rx), which enforce asynchronous message-passing method signatures. In Rx terms, interfaces are based on an observer pattern, which expose methods that accept messages from a (possibly remote) asynchronous caller.

**Features**

* **Centralized Control Flow:** Apache REEF turns the chaos of a distributed application into events in a single machine, the Job Driver. Events include container allocation, Task launch, completion and failure. For failures, Apache REEF makes every effort of making the actual Exception thrown by the Task available to the Driver.
* **Task runtime:** Apache REEF provides a Task runtime called Evaluator. Evaluators are instantiated in every container of a REEF application. Evaluators can keep data in memory in between Tasks, which enables efficient pipelines on REEF.
* **Support for multiple resource managers:** Apache REEF applications are portable to any supported resource manager with minimal effort. Further, new resource managers are easy to support in REEF.
* **.NET and Java API:** Apache REEF is the only API to write YARN or Mesos applications in .NET. Further, a single REEF application is free to mix and match Tasks written for .NET or Java.
* **Plugins:** Apache REEF allows for plugins (called “Services”) to augment its feature set without adding bloat to the core. REEF includes many Services, such as a name-based communications between Tasks, MPI-inspired group communications (Broadcast, Reduce, Gather, …) and data ingress.

# Running HelloREEF with no client

#### The difference between HelloREEF and HelloREEFNoClient

 The HelloREEF application has multiple versions that all service different needs; one of these applications, HelloREEFNoClient, allows the creation of the Driver and Evaluators without a Client. In many scenarios involving a cluster of machines one Client will access multiple Drivers, so not every Driver needs to create a Client and that is where the HelloREEFNoClient application shines.

Running HelloREEFNoClient is nearly identical to running HelloREEF:

|  |
| --- |
| > java -cp lang/java/reef-examples/target/reef-examples-{$REEF\_VERSION}-shaded.jar org.apache.reef.examples.hello.HelloREEFNoClient |

The output should be the same to HelloREEF, with evaluator.stdout containing the “Hello, REEF!” message.

# Running HelloREEF on YARN

REEF applications can be run on multiple runtime environments. Using HelloREEFYarn, we will see how to configure and launch REEF applications on YARN.

### Prerequisites

[You have compiled REEF locally](https://cwiki.apache.org/confluence/display/REEF/Compiling+REEF), and have [YARN](http://hadoop.apache.org/docs/current/hadoop-yarn/hadoop-yarn-site/YARN.html) installed and correctly configured.

### How to configure REEF on YARN

The only difference between running a REEF application on YARN vs locally is the runtime configuration:

|  |
| --- |
| final LauncherStatus status = DriverLauncher          .getLauncher(YarnClientConfiguration.CONF.build())          .run(getDriverConfiguration(), JOB\_TIMEOUT); |

### How to launch HelloREEFYarn

Running HelloREEFYarn is very similar to running HelloREEF:

|  |
| --- |
| > yarn jar lang/java/reef-examples/target/reef-examples-{$REEF\_VERSION}-shaded.jar org.apache.reef.examples.hello.HelloREEFYarn |

You can see how REEF applications work on YARN environments in [Introduction to REEF](http://reef.incubator.apache.org/introduction.html).