TensorFlow on YARN

tslam75, shariat

This document describes the design of a tool to launch TensorFlow application on a Apache Hadoop YARN cluster. It is designed for distributed TensorFlow job, but can also be used for single TensorFlow job. It supports native execution on a YARN compute node, or run within a docker image. TensorBoard integration can be enabled for visualization while the application is running.

This tool is implemented as a native YARN application; which take user's resource requirement and TensorFlow program, and allocates/starts the corresponding tasks.

Design

YARN is a cluster manager that supports multi-tenancy, it provides resource allocation for multiple tenants and multiple applications. TensorFlow can be one of these application that request and consume resources from YARN. Multiple tenants and multiple instance of TensorFlow can run on a single YARN cluster concurrently.

Distributed TensorFlow tasks are required to know about the *ClusterSpec*, which specifies where (i.e.: host and network port) all the tasks are started. While YARN can handle resource allocation, it does not allocate network ports. The following design makes use of a component called *Task Starter* and *Registry* to dynamically allocate network ports and provides a way to synchronize the information between all the tasks.

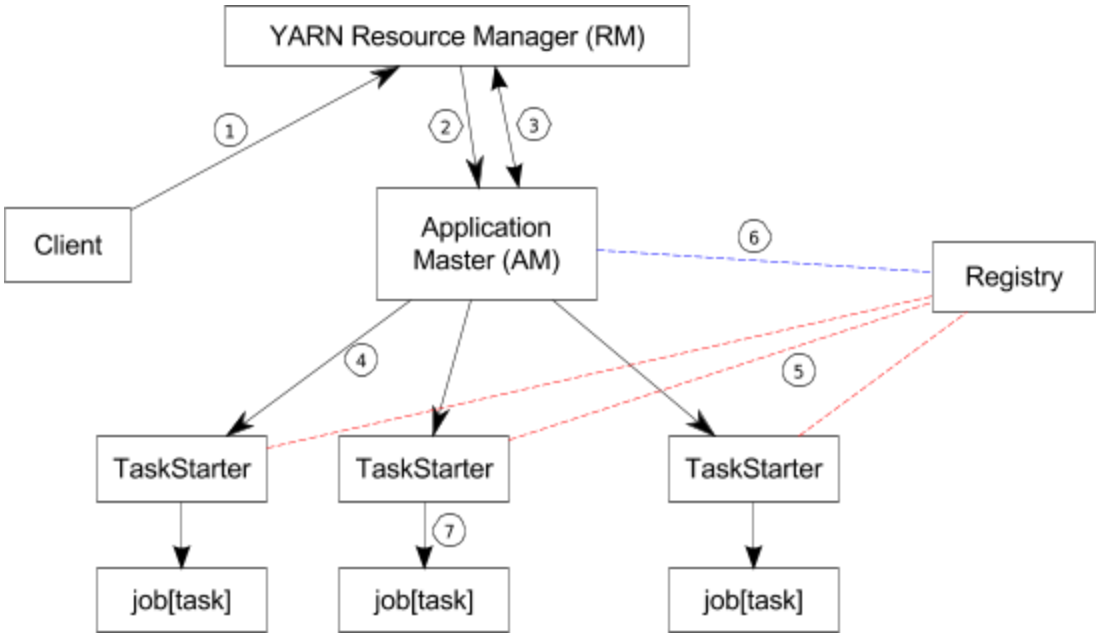

Figure 1: Design and execution flow

1. User submits TF job via the *Client* program to *YARN Resource Manager (RM)*.
2. *RM* schedules and starts the *Application Master (AM)*.
3. *AM* requests resources from *RM*, and waits for RM to reply with allocation.
4. *AM* start a *Task Starter* for each task via YARN's Node Manager (not shown)
5. *Task Starter* allocates a port dynamically on the YARN node, and reports it to the *Registry*; while waits for *AM* to publish the cluster specification. See step 7.
6. *AM* monitors the *Registry* and waits for all the *Task Starters* to report back; it publishes the information equivalent to the *ClusterSpec* back to the *Registry*.
7. *Task Starter* gets the information equivalent to the *ClusterSpec* from *Registry*; then sets up a few environment variables for task, then starts the real task.

NOTE: Apache Hadoop Distributed File System (HDFS) is also used as underlying component but is not shown in the diagram.

Client

Typical to a YARN application, the client program submits the Application Master (AM) to YARN Resource Manager (RM) to be scheduled. As part of the submission, *Client* tells *RM* the resource requirement (ie: vcores, memory) required for running the *AM* itself, as well as the user submission parameters for the TF jobs as options to *AM*. Part of the information may also be written to *HDFS*, and it is where the *AM* can read from.

See also *Client-mode*.

Application Master

Application Master (AM) is the coordinator for the TF job. First, based on user's submission parameters determine what resources are required to run the job, and make the resource request to YARN. Upon allocation, calls YARN to start the tasks (via *TaskStarter*). Collect information published by all the *TaskStarter* from the Registry, and publish the *ClusterSpec*, which all the *TaskStarter* will use to start the real task.

After the tasks are started, the *AM* keeps track of exit status of tasks, and determines the job's overall status. See also Life cycle.

Life cycle

A typical distributed TensorFlow job has Parameter Server (PS) and Worker tasks. PS tasks are for parameter update/retrieve; while workers does the computations.

The life cycle of a job ends when one of the two scenarios happen:

	Scenario	Status
1	All workers finishes successfully (exit status is zero(0))	SUCCESS
2	One of the worker finishes unsuccessfully (exit status is non-zero)	FAIL

In case of #1, AM set status to SUCCESS; then exit, during this process terminate the PS task(s).
In case of #2, AM set status to FAIL; then exit, during this process terminate the PS and possibility other reminding worker task(s).

Registry

Current implementation uses *Apache Zookeeper (ZK)* as the *Registry* component.

The following path convention is used for *AM* and *Task Starter* to set information. Although it was originally designed for a different purpose, this convention follows YARN service registry notation.

AM

ZK path:
{yarnRegistry}/users/{username}/{application_name}/{instance}

yarnRegistry:	YARN defined registry root. See *hadoop.registry.zk.root. D*efault */registry*.
username:	name of submission user.
application_name:	name of application. Default: *TensorFlow*.
instance:	YARN application instance ID.

Data format:
Use for AM to publish *ClusterSpec.*

```
{
  ...
  “DTF_PS_HOSTS”: “host1:2222,host2:2222”,
  “DTF_WORKER_HOSTS”: “host3:2222,host3:3333,host4:2222,host4:3333”,
}
```

Task Starter

ZK path:
{yarnRegistry}/users/{username}/{application_name}/{instance}/components/{componetname}

componetname:YARN container ID.

Data format:
Use for Task Starter to write dynamically allocated port back to registry.

```
{
  ...
  “task_job_name”: “worker”,
  “task_job_index”: “0”,
  “task_port”: “2222”
}
```

Alternative implementation

An alternative implementation for *Registry* without a *ZK* dependency, the *AM* can act as the *Registry* by exposing a API (potentially REST) for *Task Starter* to report dynamically allocated port, and also retrieve cluster specification once it is ready.

Task Starter

Task Starter is the component responsible for a) dynamically allocate network port for the task, b) report port number back to *AM*, c) wait for *ClusterSpec* from *AM*, d) setup environment and start the task (either natively or via Docker).

The *Task Starter* is provided with the following information via environment variables:

DTF_TASK_PROGRAM
User specify command line to execute the task. See also *task_command* in *ytf-submit*.

DTF_TASK_JOB_NAME
Name of job this task is assigned to (i.e.: *ps* or *worker*) by AM. This information is pass on to the task execution environment.

DTF_TASK_INDEX
Index of the job this task is assigned to by AM. This information is pass on to the task execution environment.

DTF_INPUT_PATH
User specified input path. See *-i*, *--input* in *ytf-submit*. This information is pass on to the task execution environment.

DTF_OUTPUT_PATH
User specified input path. See *-o*, *--output* in *ytf-submit*. This information is pass on to the task execution environment.

DTF_DOCKER_IMAGE

Name of Docker image specify by user. See *--docker_image* in *ytf-submit.*

TensorBoard

TensorBoard is part of TensorFlow which provides visualization via a web server.

TensorBoard integration can be enable as part of submission option; if enabled, it is also required to specify the output directory (aka checkpoint directory). The integration is done by invoke an internally generated TensorBoard task; and uses the dynamically allocated network port as its listening port.

The URL to TensorBoard interface can be found following *Tracking URL* from *YARN* application instance details page.

This task does not impact the life cycle of the instance, and will be terminated the same way as PS task(s).

Docker

Docker image provides a way to encapsulate application required packages (i.e.: TensorFlow binaries, Python, …).

When docker integration is enabled, the user is required to provide a docker image name to the submission script. The image name is required to be accessible on the execution host, either pre-deployed or accessible in a Docker registry server. Furthermore, the submission user has the correct permission to start a Docker container.

Instead of executing the task natively, a docker container will be launched. In addition to environment variables in Task Execution Environment, the following paths are mounted in the container to the host.

HADOOP_HOME, *HADOOP_CONF_DIR*, *JAVA_HOME*

DTF_INPUT_PATH and *DTF_OUT_PATH* if they are not HDFS path.

Client-mode (not implemented yet)

The default mode is to execute all the tasks in YARN, and the *Client* program will exit after submission is done. This mode is good for batch like training and no interaction is required from a user.

For *client-mode* the idea is to have the client program runs on the submission environment, and potentially interactive use cases can be handled.

*****This feature is not implemented yet*****

User interface

Submission Interface

Jobs are submitted using a command line script ***ytf-submit***.

> *ytf-submit* [OPTIONS] -r <*cluster_requirement*> <*task_command*>

task_command is the command to be execute for each of the task of the session.
The two environment variables, ***DTF_TASK_JOB_NAME*** and ***DTF_TASK_INDEX***, will be set before the task is executed.

cluster_requirement is a comma separated list of job names and the number of the instances for that job, with this format:

> <job_name1>:<num_tasks1>,<job_name2>:<num_task2>, …

the syntax is kept generic as per TensorFlow ClusterSpec. However, typically, this would be something like:

> ps:2,worker:4

For full options, see APPENDEX for full command line help.

Task Execution Environment

The user specified ***task_command*** will be executed as a YARN container. The following environment variables will be set for the `task_command` to consume.

> **DTF_{JOBNAME}_HOSTS**
>
> Variable with a list of host (and port) allocated to the job with name *JOBNAME*.

Format: "host1:port1,host2:port2,..."

The number of *host:port* in the list should match one specified in *cluster_requirement*. For example, *DTF_PS_HOSTS* and *DTF_WORKER_HOSTS* would be commonly used for PS and WORKER jobs.

DTF_TASK_JOB_NAME

Name of job this task is assigned to (i.e.: *ps* or *worker*). See also *DTF_TASK_INDEX*.

DTF_TASK_INDEX

Index of the job this task is assigned to. The tuple of *DTF_TASK_JOB_NAME*, and *DTF_TASK_INDEX* can also be used to cross reference with *DTF_{JOBNAME}_HOSTS*. For example, to get the dynamic port allocated to this task.

DTF_TASK_SCRIPT

Name of file which contains the content of the *script_file* specified during submission. Available only when “-s, --script” option is used.

DTF_INPUT_PATH

Input path specified during submission. Available only when “-i, --input” option is used.

DTF_OUTPUT_PATH

Output path specified during submission. Available when “-o, --output” option is used.

Examples

Example: Simple task submission

Let's execute a session with 2 x **Parameter Servers (ps)** and 4 x **Workers**.

Assume task program, input data, and output train, all reside in */home/user1/mnist* and is accessible on every node.

```
$ ytf-submit -r "ps:1,worker:4" \
'python /home/user1/mnist/mnist.py \
--job_name ${DTF_TASK_JOB_NAME} --task_index ${DTF_TASK_INDEX} \
--ps_hosts ${DTF_PS_HOSTS} --worker_hosts ${DTF_WORKER_HOSTS} \
--data_dir /home/user1/mnist/data --train_dir /home/user1/mnist/train'
```

Example: Enabling TensorBoard

TensorBoard is enabled by *--tensorboard* or *-t*. The address of TensorBoard is available at ***Tracking URL*** section of the submitted application in Apache YARN Resource Manager web interface. For using TensorBoard, output path must be specified by *--output* or *-o*. *DTF_OUTPUT_PATH* environment variable will be set and can be used in *task_command*. Similarly, input path can be passed to *ytf-submit* and will be available as *DTF_INPUT_PATH*.

```
$ ytf-submit --tensorboard \
-i /home/user1/mnist/data -o /home/user1/mnist/train10 -r "ps:1,worker:2" \
'python /home/user1/mnist/mnist.py \
--job_name ${DTF_TASK_JOB_NAME} --task_index ${DTF_TASK_INDEX} \
--ps_hosts ${DTF_PS_HOSTS} --worker_hosts ${DTF_WORKER_HOSTS} \
--data_dir ${DTF_INPUT_PATH} --train_dir ${DTF_OUTPUT_PATH}'
```

Example: Passing the script file

The training code itself can be passed to *ytf-submit*. The code will be copied to HDFS and will be available at execution time. The path to the training code will be available as *DTF_TASK_SCRIPT* environment variable.

```
$ ytf-submit --tensorboard \
-i /home/user1/mnist/data -o /home/user1/mnist/train10 -r "ps:1,worker:2" \
-s /home/user1/mnist/mnist.py \
'python ${DTF_TASK_SCRIPT} \
--job_name ${DTF_TASK_JOB_NAME} --task_index ${DTF_TASK_INDEX} \
```

*--ps_hosts ${DTF_PS_HOSTS} --worker_hosts ${DTF_WORKER_HOSTS} *

--data_dir ${DTF_INPUT_PATH} --train_dir ${DTF_OUTPUT_PATH}'

Example: Using HDFS paths

Input and output paths can be HDFS paths.

*$ ytf-submit --tensorboard *

-i hdfs://users/user1/mnist/data -o hdfs://users/user1/mnist/train10

*-r "ps:1,worker:2" -s /home/user1/mnist/mnist.py *

*'python ${DTF_TASK_SCRIPT} *

*--job_name ${DTF_TASK_JOB_NAME} --task_index ${DTF_TASK_INDEX} *

*--ps_hosts ${DTF_PS_HOSTS} --worker_hosts ${DTF_WORKER_HOSTS} *

--data_dir ${DTF_INPUT_PATH} --train_dir ${DTF_OUTPUT_PATH}'

Example: Using Docker

To execute the tasks as a Docker container, pass the Docker image name using

--docker_image <image_name>. The docker image is required to be accessible on

the execution host; and submission user has the correct permission to launch a Docker container.

Future Consideration and discussion

Considerations

- Feature: implement job-level resource requirement
- Feature: implement task-level resource requirement
- Feature: client-mode
- Feature: additional user files (directory?) to be transferred to task execution environment
- Feature: support Windows
- Feature: node label expression

Remove ZK dependency, AM implement REST API for Task Starter to call
Verification: Test under secure HDFS
Verification: Test under secure registry (ie: ZK)

Discussion

GPU support?

- There are a few open JIRA open in the Hadoop YARN community related to support from GPU (and other resources other than CPU and MEMORY), but did not find official release containing support for it yet. Keeping an eye out for YARN's support.
- *YARN-4122: Add support for GPU as a resource*
- *YARN-5517: Add GPU as a resource type for scheduling*
- *YARN-3926: Extend the YARN resource model for easier resource-type management and profiles*

Better fault handling for life cycle management?

- Current design terminate the job when any of the worker task fails, and does not attempt to restart/retry within the job. It keeps the design simple, and jobs can utilize check-pointed data and rerun the job. Is there any benefit to do fine gain fault handling within the job?

APPENDIX

```
% ytf-submit -h
NAME
   ytf-submit - Submit a TensorFlow session to Apache Hadoop YARN

   This tool submits a YARN application master, responsible to allocate
   required resources, and execute corresponding tasks.

SYNOPSIS
   Usage: ytf-submit [OPTIONS] -r <cluster_requirement>  <task_command>

DESCRIPTION
   task_command
      The command to be execute for each of the task of the session. The two
      environment variables DTF_TASK_JOB_NAME and DTF_TASK_INDEX will be set
      before the task is executed.
      See also TASK EXECUTION ENVIRONMENT

   -r, --cluster_requirement <requirement>
```

Specify cluster requirement for the session.
Format: <job_name1>:<num_tasks1>,<job_name2>:<num_task2>,...
Example: "ps:2,worker:4"
See also TASK EXECUTION ENVIRONMENT

Additional options:

-c, --task_vcores <vcores>
General form to specify number of vcores required by each of the task.
DEFAULT=1

-c, --task_vcores <job_name>:<vcores>
NOT IMPLEMENTED YET
Job-level form to specify number of vcores required by tasks in specific
job. Overrides "general" form.

-c, --task_vcores <job_name>[<task_index>]:<vcores>
NOT IMPLEMENTED YET
Task-level form to specify number of vcores required by a specific task.
Overrides both "job-level" and "general" form.

-m, --task_memory <memory>
General form to specify amount of memory required by each of task; with
unit in MB. DEFAULT=8192

-m, --task_memory <job_name>:<memory>
NOT IMPLEMENTED YET
Job-level form to specify amount of memory required by tasks in specific
job. Overrides "general" form.

-m, --task_memory <job_name>[<task_index]:<memory>
NOT IMPLEMENTED YET
Task-level form to specify amount of memory required by a specific task.
Overrides both "job-level" and "general" form.

-i, --input input_path
Input path, this variable is not interpreted by YARN-DTF at the
moment, it serve as a convenience. Its value will be set as
environment variable {DTF_INPUT_PATH} in tasks execution environment.
DEFAULT=

-o, --output <output_path>
Output path, this variable is not interpreted by YARN-DTF at the
moment, it serve as a convenience. Its value will be set as
environment variable {DTF_OUTPUT_PATH} in tasks execution environment.

However, when TensorBoard integration is enabled, this option becomes
mandatory. See also --tensorboard option.

Its value will be set as environment variable {DTF_OUTPUT_PATH} in tasks
execution environment.

-s, --script <script_file>
A local script file to be transfer to tasks execution environment, where
a file named by variable {DTF_TASK_SCRIPT} will contain the content of
the script file. For example, if the script is a Python script,
the execution command can be written as "python ${DTF_TASK_SCRIPT} ..."

-t, --tensorboard
Enable TensorBoard integration. When enabled, YARN-DTF will start an
additional YARN container as tensorboard with output path specified in
--output option. DEFAULT=disabled

--docker_image <image_name>
Enable tasks to be executed as a docker container. The docker image is
required to be accessible on the execution host. In addition to variables
in TASK EXECUTION ENVIRONMENT, the following paths are mounted in
container to the execution host.

HADOOP_HOME, HADOOP_CONF_DIR, JAVA_HOME.
DTF_INPUT_PATH and DTF_OUT_PATH if they are not hdfs path.

-q, --queue
Specify which YARN queue to submit this session to.
DEFAULT=default

-n, --name
Name of this session, will be used as name of YARN application.
DEFAULT=TensorFlow

--client
NOT IMPLEMENTED YET
Specify if an additional task should be started on locally. This
would be useful if user interaction is required.

This task will same execution environment as the rest of the tasks,
and will be assigned with DTF_TASK_JOB_NAME=client and DTF_TASK_INDEX=0;
however, will not be part of the TensorFlow cluster and dynamic port
allocation would not apply.

TASK EXECUTION ENVIRONMENT

The user specified 'task_command' will be executed as a YARN container allocated to the session. The following environment variables will be set for the 'task_command' to consume.

DTF_TASK_SCRIPT:
Name of file which contains the content of the 'script_file' specified during submission.

DTF_INPUT_PATH:
Input path specified during submission.

DTF_OUTPUT_PATH:
Output path specified during submission.

DTF_{JOBNAME}_HOSTS:
Variable with a list of host (and port) allocated to the job with name {JOBNAME}.
Format: "host1:port1,host2:port2,..."
The number of host:port in the list should match one specified in "cluster-requirement". For example, DTF_PS_HOSTS and DTF_WORKER_HOSTS would be commonly used for PS and WORKER jobs.

DTF_TASK_JOB_NAME:
Name of job this task is assigned to. See also DTF_TASK_INDEX.

DTF_TASK_INDEX
Index of the job this task is assigned to.
The tuple of DTF_TASK_JOB_NAME, and DTF_TASK_INDEX can also be used to cross reference with DTF_{JOBNAME}_HOSTS. For example, to get the dynamic port allocated to this task.