WritingYarnApps

Hadoop MapReduce Next Generation - Writing YARN Applications

Purpose

This document describes, at a high-level, the way to implement new Applications for YARN.

Concepts and Flow

The general concept is that an 'Application Submission Client' submits an 'Application' to the YARN Resource Manager. The client communicates with the ResourceManager using the 'ClientRMProtocol' to first acquire a new 'ApplicationId' if needed via ClientRMProtocol#getNewApplication and then submit the 'Application' to be run via ClientRMProtocol#submitApplication. As part of the ClientRMProtocol#submitApplication call, the client needs to provide sufficient information to the ResourceManager to 'launch' the application's first container i.e. the ApplicationMaster. You need to provide information such as the details about the local files/jars that need to be available for your application to run, the actual command that needs to be executed (with the necessary command line arguments), any Unix environment settings (optional), etc. Effectively, you need to describe the Unix process(es) that needs to be launched for your ApplicationMaster.

The YARN ResourceManager will then launch the ApplicationMaster (as specified) on an allocated container. The ApplicationMaster is then expected to communicate with the ResourceManager using the 'AMRMProtocol'. Firstly, the ApplicationMaster needs to register itself with the ResourceManager. To complete the task assigned to it, the ApplicationMaster can then request for and receive containers via AMRMProtocol#allocate. After a container is allocated to it, the ApplicationMaster communicates with the NodeManager using ContainerManager#startContainer to launch the container for its task. As part of launching this container, the ApplicationMaster has to specify the ContainerLaunchContext which, similar to the ApplicationSubmissionContext, has the launch information such as command line specification, environment, etc. Once the task is completed, the ApplicationMaster has to signal the ResourceManager of its completion via the AMRMProtocol#finishApplicationMaster.

Meanwhile, the client can monitor the application's status by querying the ResourceManager or by directly querying the ApplicationMaster if it supports such a service. If needed, it can also kill the application via ClientRMProtocol#forceKillApplication.

Interfaces

The interfaces you'd most like be concerned with are:

- ClientRMProtocol - Client\<--\>ResourceManager\
  The protocol for a client that wishes to communicate with the ResourceManager to launch a new application (i.e. the ApplicationMaster), check on the status of the application or kill the application. For example, a job-client (a job launching program from the gateway) would use this protocol.
- AMRMProtocol - ApplicationMaster\<--\>ResourceManager\
  The protocol used by the ApplicationMaster to register/unregister itself to/from the ResourceManager as well as to request for resources from the Scheduler to complete its tasks.
- ContainerManager - ApplicationMaster\<--\>NodeManager\
  The protocol used by the ApplicationMaster to talk to the NodeManager to start/stop containers and get status updates on the containers if needed.

Writing a Simple Yarn Application

Writing a simple Client

- The first step that a client needs to do is to connect to the ResourceManager or to be more specific, the ApplicationsManager (AsM) interface of the ResourceManager.

```
ClientRMProtocol applicationsManager;
YarnConfiguration yarnConf = new YarnConfiguration(conf);
InetSocketAddress rmAddress =
    NetUtils.createSocketAddr(yarnConf.get(
        YarnConfiguration.RM_ADDRESS,
        YarnConfiguration.DEFAULT_RM_ADDRESS));
LOG.info("Connecting to ResourceManager at " + rmAddress);
configuration appsManagerServerConf = new Configuration(conf);
appsManagerServerConf.setClass(
    YarnConfiguration.YARN_SECURITY_INFO,
    ClientRMSecurityInfo.class, SecurityInfo.class);
applicationsManager = ((ClientRMProtocol) rpc.getProxy(
    ClientRMProtocol.class, rmAddress, appsManagerServerConf));
```

- Once a handle is obtained to the ASM, the client needs to request the ResourceManager for a new ApplicationId.

```
GetNewApplicationRequest request =
    Records.newRecord(GetNewApplicationRequest.class);
GetNewApplicationResponse response =
    applicationsManager.getNewApplication(request);
LOG.info("Got new ApplicationId=" + response.getApplicationId());
```

- The response from the ASM for a new application also contains information about the cluster such as the minimum/maximum resource capabilities of the cluster. This is required so that to ensure that you can correctly set the specifications of the container in which the ApplicationMaster would be launched. Please refer to GetNewApplicationResponse for more details.
- The main crux of a client is to setup the ApplicationSubmissionContext which defines all the information needed by the ResourceManager to launch the ApplicationMaster. A client needs to set the following into the context:
    - Application Info: id, name
    - Queue, Priority info: Queue to which the application will be submitted, the priority to be assigned for the application.
    - User: The user submitting the application
    - ContainerLaunchContext: The information defining the container in which the ApplicationMaster will be launched and run. The ContainerLaunchContext, as mentioned previously, defines all the required information needed to run the ApplicationMaster such as the local resources (binaries, jars, files etc.), security tokens, environment settings (CLASSPATH etc.) and the command to be executed.

```
// Create a new ApplicationSubmissionContext
ApplicationSubmissionContext appContext =
    Records.newRecord(ApplicationSubmissionContext.class);
// set the ApplicationId
appContext.setApplicationId(appId);
// set the application name
appContext.setApplicationName(appName);

// Create a new container launch context for the AM's container
ContainerLaunchContext amContainer =
    Records.newRecord(ContainerLaunchContext.class);

// Define the local resources required
Map<String, LocalResource> localResources =
    new HashMap<String, LocalResource>();
// Lets assume the jar we need for our ApplicationMaster is available in
// HDFS at a certain known path to us and we want to make it available to
// the ApplicationMaster in the launched container
Path jarPath; // <- known path to jar file
FileStatus jarStatus = fs.getFileStatus(jarPath);
LocalResource amJarRsrc = Records.newRecord(LocalResource.class);
// Set the type of resource - file or archive
// archives are untarred at the destination by the framework
amJarRsrc.setType(LocalResourceType.FILE);
// Set visibility of the resource
// Setting to most private option i.e. this file will only
// be visible to this instance of the running application
amJarRsrc.setVisibility(LocalResourceVisibility.APPLICATION);
// Set the location of resource to be copied over into the
// working directory
amJarRsrc.setResource(ConverterUtils.getYarnUrlFromPath(jarPath));
```

```
// Set timestamp and length of file so that the framework
// can do basic sanity checks for the local resource
// after it has been copied over to ensure it is the same
// resource the client intended to use with the application
amJarRsrc.setTimestamp(jarStatus.getModificationTime());
amJarRsrc.setSize(jarStatus.getLen());
// The framework will create a symlink called AppMaster.jar in the
// working directory that will be linked back to the actual file.
// The ApplicationMaster, if needs to reference the jar file, would
// need to use the symlink filename.
localResources.put("AppMaster.jar",  amJarRsrc);
// Set the local resources into the launch context
amContainer.setLocalResources(localResources);

// Set up the environment needed for the launch context
Map<String, String> env = new HashMap<String, String>();
// For example, we could setup the classpath needed.
// Assuming our classes or jars are available as local resources in the
// working directory from which the command will be run, we need to append
// "." to the path.
// By default, all the hadoop specific classpaths will already be available
// in $CLASSPATH, so we should be careful not to overwrite it.
String classPathEnv = "$CLASSPATH:./*:";
env.put("CLASSPATH", classPathEnv);
amContainer.setEnvironment(env);

// Construct the command to be executed on the launched container
String command =
    "${JAVA_HOME}" + /bin/java" +
    " MyAppMaster" +
    " arg1 arg2 arg3" +
    " 1>" + ApplicationConstants.LOG_DIR_EXPANSION_VAR + "/stdout" +
    " 2>" + ApplicationConstants.LOG_DIR_EXPANSION_VAR + "/stderr";

List<String> commands = new ArrayList<String>();
commands.add(command);
// add additional commands if needed

// Set the command array into the container spec
amContainer.setCommands(commands);

// Define the resource requirements for the container
// For now, YARN only supports memory so we set the memory
// requirements.
// If the process takes more than its allocated memory, it will
// be killed by the framework.
// Memory being requested for should be less than max capability
// of the cluster and all asks should be a multiple of the min capability.
Resource capability = Records.newRecord(Resource.class);
capability.setMemory(amMemory);
amContainer.setResource(capability);

// Set the container launch content into the ApplicationSubmissionContext
appContext.setAMContainerSpec(amContainer);
```

- After the setup process is complete, the client is finally ready to submit the application to the ASM.

```
// Create the request to send to the ApplicationsManager
SubmitApplicationRequest appRequest =
    Records.newRecord(SubmitApplicationRequest.class);
appRequest.setApplicationSubmissionContext(appContext);

// Submit the application to the ApplicationsManager
// Ignore the response as either a valid response object is returned on
// success or an exception thrown to denote the failure
applicationsManager.submitApplication(appRequest);
```

- At this point, the ResourceManager will have accepted the application and in the background, will go through the process of allocating a container with the required specifications and then eventually setting up and launching the ApplicationMaster on the allocated container.
- There are multiple ways a client can track progress of the actual task.
    - It can communicate with the ResourceManager and request for a report of the application via ClientRMProtocol#getApplicationReport.

```
GetApplicationReportRequest reportRequest =
    Records.newRecord(GetApplicationReportRequest.class);
reportRequest.setApplicationId(appId);
GetApplicationReportResponse reportResponse =
    applicationsManager.getApplicationReport(reportRequest);
ApplicationReport report = reportResponse.getApplicationReport();
```

The ApplicationReport received from the ResourceManager consists of the following:

- General application information: ApplicationId, queue to which the application was submitted, user who submitted the application and the start time for the application.
- ApplicationMaster details: the host on which the ApplicationMaster is running, the rpc port (if any) on which it is listening for requests from clients and a token that the client needs to communicate with the ApplicationMaster.
- Application tracking information: If the application supports some form of progress tracking, it can set a tracking url which is available via ApplicationReport#getTrackingUrl that a client can look at to monitor progress.
- ApplicationStatus: The state of the application as seen by the ResourceManager is available via ApplicationReport#getYarnApplicationState. If the YarnApplicationState is set to FINISHED, the client should refer to ApplicationReport#getFinalApplicationStatus to check for the actual success/failure of the application task itself. In case of failures, ApplicationReport#getDiagnostics may be useful to shed some more light on the the failure.

- If the ApplicationMaster supports it, a client can directly query the ApplicationMaster itself for progress updates via the host:rpcport information obtained from the ApplicationReport. It can also use the tracking url obtained from the report if available.
    - In certain situations, if the application is taking too long or due to other factors, the client may wish to kill the application. The ClientRMProtocol supports the forceKillApplication call that allows a client to send a kill signal to the ApplicationMaster via the Resource Manager. An ApplicationMaster if so designed may also support an abort call via its rpc layer that a client may be able to leverage.

```
KillApplicationRequest killRequest =
    Records.newRecord(KillApplicationRequest.class);
killRequest.setApplicationId(appId);
applicationsManager.forceKillApplication(killRequest);
```

Writing an ApplicationMaster

- The ApplicationMaster is the actual owner of the job. It will be launched by the ResourceManager and via the client will be provided all the necessary information and resources about the job that it has been tasked with to oversee and complete.
- As the ApplicationMaster is launched within a container that may (likely will) be sharing a physical host with other containers, given the multi-tenancy nature, amongst other issues, it cannot make any assumptions of things like pre-configured ports that it can listen on.
- All interactions with the ResourceManager require an ApplicationAttemptId (there can be multiple attempts per application in case of failures). When the ApplicationMaster starts up, the ApplicationAttemptId associated with this particular instance will be set in the environment. There are helper apis to convert the value obtained from the environment into an ApplicationAttemptId object.

```
Map<String, String> envs = System.getenv();
ApplicationAttemptId appAttemptID =
    Records.newRecord(ApplicationAttemptId.class);
if (!envs.containsKey(ApplicationConstants.APPLICATION_ATTEMPT_ID_ENV)) {
  // app attempt id should always be set in the env by the framework
  throw new IllegalArgumentException(
      "ApplicationAttemptId not set in the environment");
}
appAttemptID =
    ConverterUtils.toApplicationAttemptId(
        envs.get(ApplicationConstants.APPLICATION_ATTEMPT_ID_ENV));
```

- After an ApplicationMaster has initialized itself completely, it needs to register with the ResourceManager via AMRMProtocol#registerApplicationMaster. The ApplicationMaster always communicate via the Scheduler interface of the ResourceManager.

```
// Connect to the Scheduler of the ResourceManager.
YarnConfiguration yarnConf = new YarnConfiguration(conf);
InetSocketAddress rmAddress =
    NetUtils.createSocketAddr(yarnConf.get(
        YarnConfiguration.RM_SCHEDULER_ADDRESS,
        YarnConfiguration.DEFAULT_RM_SCHEDULER_ADDRESS));
LOG.info("Connecting to ResourceManager at " + rmAddress);
AMRMProtocol resourceManager =
    (AMRMProtocol) rpc.getProxy(AMRMProtocol.class, rmAddress, conf);

// Register the AM with the RM
// Set the required info into the registration request:
// ApplicationAttemptId,
// host on which the app master is running
// rpc port on which the app master accepts requests from the client
// tracking url for the client to track app master progress
RegisterApplicationMasterRequest appMasterRequest =
    Records.newRecord(RegisterApplicationMasterRequest.class);
appMasterRequest.setApplicationAttemptId(appAttemptID);
appMasterRequest.setHost(appMasterHostname);
appMasterRequest.setRpcPort(appMasterRpcPort);
appMasterRequest.setTrackingUrl(appMasterTrackingUrl);


// The registration response is useful as it provides information about the
// cluster.
// Similar to the GetNewApplicationResponse in the client, it provides
// information about the min/mx resource capabilities of the cluster that
// would be needed by the ApplicationMaster when requesting for containers.
RegisterApplicationMasterResponse response =
    resourceManager.registerApplicationMaster(appMasterRequest);
```

- The ApplicationMaster has to emit heartbeats to the ResourceManager to keep it informed that the ApplicationMaster is alive and still running. The timeout expiry interval at the ResourceManager is defined by a config setting accessible via YarnConfiguration. RM_AM_EXPIRY_INTERVAL_MS with the default being defined by YarnConfiguration.DEFAULT_RM_AM_EXPIRY_INTERVAL_MS. The AMRMProtocol#allocate calls to the ResourceManager count as heartbeats as it also supports sending progress update information. Therefore, an allocate call with no containers requested and progress information updated if any is a valid way for making heartbeat calls to the ResourceManager.
- Based on the task requirements, the ApplicationMaster can ask for a set of containers to run its tasks on. The ApplicationMaster has to use the ResourceRequest class to define the following container specifications:
    - Hostname: If containers are required to be hosted on a particular rack or a specific host. '*' is a special value that implies any host will do.
    - Resource capability: Currently, YARN only supports memory based resource requirements so the request should define how much memory is needed. The value is defined in MB and has to less than the max capability of the cluster and an exact multiple of the min capability.
    - Priority: When asking for sets of containers, an ApplicationMaster may define different priorities to each set. For example, the Map-Reduce ApplicationMaster may assign a higher priority to containers needed for the Map tasks and a lower priority for the Reduce tasks' containers.

```
// Resource Request
ResourceRequest rsrcRequest = Records.newRecord(ResourceRequest.class);

// setup requirements for hosts
// whether a particular rack/host is needed
// useful for applications that are sensitive
// to data locality
rsrcRequest.setHostName("*");

// set the priority for the request
Priority pri = Records.newRecord(Priority.class);
pri.setPriority(requestPriority);
rsrcRequest.setPriority(pri);

// Set up resource type requirements
// For now, only memory is supported so we set memory requirements
Resource capability = Records.newRecord(Resource.class);
capability.setMemory(containerMemory);
rsrcRequest.setCapability(capability);

// set no. of containers needed
// matching the specifications
rsrcRequest.setNumContainers(numContainers);
```

- After defining the container requirements, the ApplicationMaster has to construct an AllocateRequest to send to the ResourceManager. The AllocateRequest consists of:
  - Requested containers: The container specifications and the no. of containers being requested for by the ApplicationMaster from the ResourceManager.
  - Released containers: There may be situations when the ApplicationMaster may have requested for more containers that it needs or due to failure issues, decide to use other containers allocated to it. In all such situations, it is beneficial to the cluster if the ApplicationMaster releases these containers back to the ResourceManager so that they can be re-allocated to other applications.
  - ResponseId: The response id that will be sent back in the response from the allocate call.
  - Progress update information: The ApplicationMaster can send its progress update (range between to 0 to 1) to the ResourceManager.

```
List<ResourceRequest> requestedContainers;
List<ContainerId> releasedContainers
AllocateRequest req = Records.newRecord(AllocateRequest.class);

// The response id set in the request will be sent back in
// the response so that the ApplicationMaster can
// match it to its original ask and act appropriately.
req.setResponseId(rmRequestID);

// Set ApplicationAttemptId
req.setApplicationAttemptId(appAttemptID);

// Add the list of containers being asked for
req.addAllAsks(requestedContainers);

// If the ApplicationMaster has no need for certain
// containers due to over-allocation or for any other
// reason, it can release them back to the ResourceManager
req.addAllReleases(releasedContainers);

// Assuming the ApplicationMaster can track its progress
req.setProgress(currentProgress);

AllocateResponse allocateResponse = resourceManager.allocate(req);
```

- The AllocateResponse sent back from the ResourceManager provides the following information via the AMResponse object:
  - Reboot flag: For scenarios when the ApplicationMaster may get out of sync with the ResourceManager.
  - Allocated containers: The containers that have been allocated to the ApplicationMaster.
  - Headroom: Headroom for resources in the cluster. Based on this information and knowing its needs, an ApplicationMaster can make intelligent decisions such as re-prioritizing sub-tasks to take advantage of currently allocated containers, bailing out faster if resources are not becoming available etc.

- Completed containers: Once an ApplicationMaster triggers a launch an allocated container, it will receive an update from the ResourceManager when the container completes. The ApplicationMaster can look into the status of the completed container and take appropriate actions such as re-trying a particular sub-task in case of a failure.
  One thing to note is that containers will not be immediately allocated to the ApplicationMaster. This does not imply that the ApplicationMaster should keep on asking the pending count of required containers. Once an allocate request has been sent, the ApplicationMaster will eventually be allocated the containers based on cluster capacity, priorities and the scheduling policy in place. The ApplicationMaster should only request for containers again if and only if its original estimate changed and it needs additional containers.

```
// Get AMResponse from AllocateResponse
AMResponse amResp = allocateResponse.getAMResponse();

// Retrieve list of allocated containers from the response
// and on each allocated container, lets assume we are launching
// the same job.
List<Container> allocatedContainers = amResp.getAllocatedContainers();
for (Container allocatedContainer : allocatedContainers) {
  LOG.info("Launching shell command on a new container."
      + ", containerId=" + allocatedContainer.getId()
      + ", containerNode=" + allocatedContainer.getNodeId().getHost()
      + ":" + allocatedContainer.getNodeId().getPort()
      + ", containerNodeURI=" + allocatedContainer.getNodeHttpAddress()
      + ", containerState" + allocatedContainer.getState()
      + ", containerResourceMemory"
      + allocatedContainer.getResource().getMemory());


  // Launch and start the container on a separate thread to keep the main
  // thread unblocked as all containers may not be allocated at one go.
  LaunchContainerRunnable runnableLaunchContainer =
      new LaunchContainerRunnable(allocatedContainer);
  Thread launchThread = new Thread(runnableLaunchContainer);
  launchThreads.add(launchThread);
  launchThread.start();
}

// Check what the current available resources in the cluster are
Resource availableResources = amResp.getAvailableResources();
// Based on this information, an ApplicationMaster can make appropriate
// decisions

// Check the completed containers
// Let's assume we are keeping a count of total completed containers,
// containers that failed and ones that completed successfully.
List<ContainerStatus> completedContainers =
    amResp.getCompletedContainersStatuses();
for (ContainerStatus containerStatus : completedContainers) {
  LOG.info("Got container status for containerID= "
      + containerStatus.getContainerId()
      + ", state=" + containerStatus.getState()
      + ", exitStatus=" + containerStatus.getExitStatus()
      + ", diagnostics=" + containerStatus.getDiagnostics());


  int exitStatus = containerStatus.getExitStatus();
  if (0 != exitStatus) {
    // container failed
    // -100 is a special case where the container
    // was aborted/pre-empted for some reason
    if (-100 != exitStatus) {
      // application job on container returned a non-zero exit code
      // counts as completed
      numCompletedContainers.incrementAndGet();
      numFailedContainers.incrementAndGet();
    }
    else {
      // something else bad happened
      // app job did not complete for some reason
      // we should re-try as the container was lost for some reason
      // decrementing the requested count so that we ask for an
      // additional one in the next allocate call.
```

```
      numRequestedContainers.decrementAndGet();
      // we do not need to release the container as that has already
      // been done by the ResourceManager/NodeManager.
    }
  }
  else {
    // nothing to do
    // container completed successfully
    numCompletedContainers.incrementAndGet();
    numSuccessfulContainers.incrementAndGet();
  }
}
```

- After a container has been allocated to the ApplicationMaster, it needs to follow a similar process that the Client followed in setting up the ContainerLaunchContext for the eventual task that is going to be running on the allocated Container. Once the ContainerLaunchContext is defined, the ApplicationMaster can then communicate with the ContainerManager to start its allocated container.

```
//Assuming an allocated Container obtained from AMResponse
Container container;
// Connect to ContainerManager on the allocated container
String cmIpPortStr = container.getNodeId().getHost() + ":"
    + container.getNodeId().getPort();
InetSocketAddress cmAddress = NetUtils.createSocketAddr(cmIpPortStr);
ContainerManager cm =
    (ContainerManager)rpc.getProxy(ContainerManager.class, cmAddress, conf);


// Now we setup a ContainerLaunchContext
ContainerLaunchContext ctx =
    Records.newRecord(ContainerLaunchContext.class);


ctx.setContainerId(container.getId());
ctx.setResource(container.getResource());


try {
  ctx.setUser(UserGroupInformation.getCurrentUser().getShortUserName());
} catch (IOException e) {
  LOG.info(
      "Getting current user failed when trying to launch the container",
      + e.getMessage());
}


// Set the environment
Map<String, String> unixEnv;
// Setup the required env.
// Please note that the launched container does not inherit
// the environment of the ApplicationMaster so all the
// necessary environment settings will need to be re-setup
// for this allocated container.
ctx.setEnvironment(unixEnv);


// Set the local resources
Map<String, LocalResource> localResources =
    new HashMap<String, LocalResource>();
// Again, the local resources from the ApplicationMaster is not copied over
// by default to the allocated container. Thus, it is the responsibility
       // of the ApplicationMaster to setup all the necessary local resources
       // needed by the job that will be executed on the allocated container.

// Assume that we are executing a shell script on the allocated container
// and the shell script's location in the filesystem is known to us.
Path shellScriptPath;
LocalResource shellRsrc = Records.newRecord(LocalResource.class);
shellRsrc.setType(LocalResourceType.FILE);
```

```
shellRsrc.setVisibility(LocalResourceVisibility.APPLICATION);
shellRsrc.setResource(
    ConverterUtils.getYarnUrlFromURI(new URI(shellScriptPath)));
shellRsrc.setTimestamp(shellScriptPathTimestamp);
shellRsrc.setSize(shellScriptPathLen);
localResources.put("MyExecShell.sh", shellRsrc);


ctx.setLocalResources(localResources);


// Set the necessary command to execute on the allocated container
String command = "/bin/sh ./MyExecShell.sh"
    + " 1>" + ApplicationConstants.LOG_DIR_EXPANSION_VAR + "/stdout"
    + " 2>" + ApplicationConstants.LOG_DIR_EXPANSION_VAR + "/stderr";


List<String> commands = new ArrayList<String>();
commands.add(command);
ctx.setCommands(commands);


// Send the start request to the ContainerManager
StartContainerRequest startReq = Records.newRecord(StartContainerRequest.class);
startReq.setContainerLaunchContext(ctx);
cm.startContainer(startReq);
```

- The ApplicationMaster, as mentioned previously, will get updates of completed containers as part of the response from the AMRMProtocol#allocate calls. It can also monitor its launched containers pro-actively by querying the ContainerManager for the status.

```
GetContainerStatusRequest statusReq =
    Records.newRecord(GetContainerStatusRequest.class);
statusReq.setContainerId(container.getId());
GetContainerStatusResponse statusResp = cm.getContainerStatus(statusReq);
LOG.info("Container Status"
    + ", id=" + container.getId()
    + ", status=" + statusResp.getStatus());
```

FAQ

How can I distribute my application's jars to all of the nodes in the YARN cluster that need it?

You can use the LocalResource to add resources to your application request. This will cause YARN to distribute the resource to the ApplicationMaster node. If the resource is a tgz, zip, or jar - you can have YARN unzip it. Then, all you need to do is add the unzipped folder to your classpath. For example, when creating your application request:

```
File packageFile = new File(packagePath);
Url packageUrl = ConverterUtils.getYarnUrlFromPath(
    FileContext.getFileContext.makeQualified(new Path(packagePath)));

packageResource.setResource(packageUrl);
packageResource.setSize(packageFile.length());
packageResource.setTimestamp(packageFile.lastModified());
packageResource.setType(LocalResourceType.ARCHIVE);
packageResource.setVisibility(LocalResourceVisibility.APPLICATION);


resource.setMemory(memory)
containerCtx.setResource(resource)
containerCtx.setCommands(ImmutableList.of(
    "java -cp './package/*' some.class.to.Run "
    + "1>" + ApplicationConstants.LOG_DIR_EXPANSION_VAR + "/stdout "
    + "2>" + ApplicationConstants.LOG_DIR_EXPANSION_VAR + "/stderr"))
```

```
    containerCtx.setLocalResources(
        Collections.singletonMap("package", packageResource))
    appCtx.setApplicationId(appId)
    appCtx.setUser(user.getShortUserName)
    appCtx.setAMContainerSpec(containerCtx)
    request.setApplicationSubmissionContext(appCtx)
    applicationsManager.submitApplication(request)
```

As you can see, the setLocalResources command takes a map of names to resources. The name becomes a sym link in your application's cwd, so you can just refer to the artifacts inside by using ./package/*. Note: Java's classpath (cp) argument is VERY sensitive. Make sure you get the syntax EXACTLY correct.

Once your package is distributed to your ApplicationMaster, you'll need to follow the same process whenever your ApplicationMaster starts a new container (assuming you want the resources to be sent to your container). The code for this is the same. You just need to make sure that you give your ApplicationMaster the package path (either HDFS, or local), so that it can send the resource URL along with the container ctx.

How do I get the ApplicationMaster's ApplicationAttemptId?

The ApplicationAttemptId will be passed to the ApplicationMaster via the environment and the value from the environment can be converted into an ApplicationAttemptId object via the ConverterUtils helper function.

My container is being killed by the Node Manager

This is likely due to high memory usage exceeding your requested container memory size. There are a number of reasons that can cause this. First, look at the process tree that the node manager dumps when it kills your container. The two things you're interested in are physical memory and virtual memory. If you have exceeded physical memory limits your app is using too much physical memory. If you're running a Java app, you can use -hprof to look at what is taking up space in the heap. If you have exceeded virtual memory, things are slightly more complicated.

How can my ApplicationMaster kill a container? Releasing it via AMRMProtocol#allocate does not seem to work.

A container can only be released back to the ResourceManager if it has not been launched. To kill a launched container, the ApplicationMaster can send a stop command to the container via ContainerManager#stopContainer(StopContainerRequest request). This will trigger a kill event to the launched container and this container will eventually be part of the list of completed Containers in the RM's response to the AM on an AMRMProtocol#allocate call.

Useful Links

- Map Reduce Next Generation Architecture
- Map Reduce Next Generation Scheduler