Item 129
git_comments:

git_commits:

1. **summary:** YARN-4398. Remove unnecessary synchronization in RMStateStore. Contributed by Ning Ding
   message: YARN-4398. Remove unnecessary synchronization in RMStateStore. Contributed by Ning Ding (cherry picked from commit 6b9a5beb2b2f9589ef86670f2d763e8488ee5e90)

github_issues:

github_issues_comments:

github_pulls:

github_pulls_comments:

github_pulls_reviews:

jira_issues:

1. **summary:** Yarn recover functionality causes the cluster running slowly and the cluster usage rate is far below 100
   description: In my hadoop cluster, the resourceManager recover functionality is enabled with FileSystemRMStateStore. I found this cause the yarn cluster running slowly and cluster usage rate is just 50 even there are many pending Apps. The scenario is below. In thread A, the RMAppImpl$RMAppNewlySavingTransition is calling storeNewApplication method defined in RMStateStore. This storeNewApplication method is synchronized. {code:title=RMAppImpl.java|borderStyle=solid} private static final class RMAppNewlySavingTransition extends RMAppTransition { @Override public void transition(RMAppImpl app, RMAppEvent event) { // If recovery is enabled then store the application information in a // non-blocking call so make sure that RM has stored the information // needed to restart the AM after RM restart without further client // communication LOG.info("Storing application with id " + app.applicationId); app.rmContext.getStateStore().storeNewApplication(app); } } {code} {code:title=RMStateStore.java|borderStyle=solid} public synchronized void storeNewApplication(RMApp app) { ApplicationSubmissionContext context = app .getApplicationSubmissionContext(); assert context instanceof ApplicationSubmissionContextPBImpl; ApplicationStateData appState = ApplicationStateData.newInstance(app.getSubmitTime(), app.getStartTime(), context, app.getUser()); dispatcher.getEventHandler().handle(new RMStateStoreAppEvent(appState)); } {code} In thread B, the FileSystemRMStateStore is calling storeApplicationStateInternal method. It's also synchronized. This storeApplicationStateInternal method saves an ApplicationStateData into HDFS and it normally costs 90~300 milliseconds in my hadoop cluster. {code:title=FileSystemRMStateStore.java|borderStyle=solid} public synchronized void storeApplicationStateInternal(ApplicationId appId, ApplicationStateData appStateDataPB) throws Exception { Path appDirPath = getAppDir(rmAppRoot, appId); mkdirsWithRetries(appDirPath); Path nodeCreatePath = getNodePath(appDirPath, appId.toString()); LOG.info("Storing info for app: " + appId + " at: " + nodeCreatePath); byte[] appStateData = appStateDataPB.getProto().toByteArray(); try { // currently throw all exceptions. May need to respond differently for HA // based on whether we have lost the right to write to FS writeFileWithRetries(nodeCreatePath, appStateData, true); } catch (Exception e) { LOG.info("Error storing info for app: " + appId, e); throw e; } } {code} Think thread B firstly comes into FileSystemRMStateStore.storeApplicationStateInternal method, then thread A will be blocked for a while because of synchronization. In ResourceManager there is only one RMStateStore instance. In my cluster it's FileSystemRMStateStore type. Debug the RMAppNewlySavingTransition.transition method, the thread stack shows it's called form AsyncDispatcher.dispatch method. This method code is as below. {code:title=AsyncDispatcher.java|borderStyle=solid} protected void dispatch(Event event) { //all events go thru this loop if (LOG.isDebugEnabled()) { LOG.debug("Dispatching the event " + event.getClass().getName() + "." + event.toString()); } Class<? extends Enum> type = event.getType().getDeclaringClass(); try{ EventHandler handler = eventDispatchers.get(type); if(handler != null) { handler.handle(event); } else { throw new Exception("No handler for registered for " + type); } } catch (Throwable t) { //TODO Maybe log the state of the queue LOG.fatal("Error in dispatcher thread", t);

// If serviceStop is called, we should exit this thread gracefully. if (exitOnDispatchException && (ShutdownHookManager.get().isShutdownInProgress()) == false && stopped == false) { Thread shutDownThread = new Thread(createShutDownThread()); shutDownThread.setName("AsyncDispatcher ShutDown handler"); shutDownThread.start(); } } } {code} Above code shows AsyncDispatcher.dispatch method can process different type events. In fact this AsyncDispatcher instance is just ResourceManager.rmDispatcher created in ResourceManager.serviceInit method. You can find many eventTypes and handlers are registered in ResourceManager.rmDispatcher. In above scenario thread B blocks thread A, then many following events processing are blocked. In my testing cluster, there is only one queue and the client submits 1000 applications concurrently, the yarn cluster usage rate is 50. Many apps are pending. If I disable resourceManager recover functionality, the cluster usage can be 100. To solve this issue, I removed synchronized modifier on some methods defined in RMStateStore. Instead, in these methods I defined a dedicated lock object before calling dispatcher.getEventHandler().handle. In this way, the yarn cluster usage rate can be 100 and the whole cluster is good running. Please see my attached patch.

2. **summary:** Yarn recover functionality causes the cluster running slowly and the cluster usage rate is far below 100
description: In my hadoop cluster, the resourceManager recover functionality is enabled with FileSystemRMStateStore. I found this cause the yarn cluster running slowly and cluster usage rate is just 50 even there are many pending Apps. The scenario is below. In thread A, the RMAppImpl$RMAppNewlySavingTransition is calling storeNewApplication method defined in RMStateStore. This storeNewApplication method is synchronized. {code:title=RMAppImpl.java|borderStyle=solid} private static final class RMAppNewlySavingTransition extends RMAppTransition { @Override public void transition(RMAppImpl app, RMAppEvent event) { // If recovery is enabled then store the application information in a // non-blocking call so make sure that RM has stored the information // needed to restart the AM after RM restart without further client // communication LOG.info("Storing application with id " + app.applicationId); app.rmContext.getStateStore().storeNewApplication(app); } } {code} {code:title=RMStateStore.java|borderStyle=solid} public synchronized void storeNewApplication(RMApp app) { ApplicationSubmissionContext context = app .getApplicationSubmissionContext(); assert context instanceof ApplicationSubmissionContextPBImpl; ApplicationStateData appState = ApplicationStateData.newInstance(app.getSubmitTime(), app.getStartTime(), context, app.getUser()); dispatcher.getEventHandler().handle(new RMStateStoreAppEvent(appState)); } {code} In thread B, the FileSystemRMStateStore is calling storeApplicationStateInternal method. It's also synchronized. This storeApplicationStateInternal method saves an ApplicationStateData into HDFS and it normally costs 90~300 milliseconds in my hadoop cluster. {code:title=FileSystemRMStateStore.java|borderStyle=solid} public synchronized void storeApplicationStateInternal(ApplicationId appId, ApplicationStateData appStateDataPB) throws Exception { Path appDirPath = getAppDir(rmAppRoot, appId); mkdirsWithRetries(appDirPath); Path nodeCreatePath = getNodePath(appDirPath, appId.toString()); LOG.info("Storing info for app: " + appId + " at: " + nodeCreatePath); byte[] appStateData = appStateDataPB.getProto().toByteArray(); try { // currently throw all exceptions. May need to respond differently for HA // based on whether we have lost the right to write to FS writeFileWithRetries(nodeCreatePath, appStateData, true); } catch (Exception e) { LOG.info("Error storing info for app: " + appId, e); throw e; } } {code} Think thread B firstly comes into FileSystemRMStateStore.storeApplicationStateInternal method, then thread A will be blocked for a while because of synchronization. In ResourceManager there is only one RMStateStore instance. In my cluster it's FileSystemRMStateStore type. Debug the RMAppNewlySavingTransition.transition method, the thread stack shows it's called form AsyncDispatcher.dispatch method. This method code is as below. {code:title=AsyncDispatcher.java|borderStyle=solid} protected void dispatch(Event event) { //all events go thru this loop if (LOG.isDebugEnabled()) { LOG.debug("Dispatching the event " + event.getClass().getName() + "." + event.toString()); } Class<? extends Enum> type = event.getType().getDeclaringClass(); try{ EventHandler handler = eventDispatchers.get(type); if(handler != null) { handler.handle(event); } else { throw new Exception("No handler for registered for " + type); } } catch (Throwable t) { //TODO Maybe log the state of the queue LOG.fatal("Error in dispatcher thread", t); // If serviceStop is called, we should exit this thread gracefully. if (exitOnDispatchException && (ShutdownHookManager.get().isShutdownInProgress()) == false && stopped == false) { Thread shutDownThread = new Thread(createShutDownThread()); shutDownThread.setName("AsyncDispatcher ShutDown handler"); shutDownThread.start(); } } } {code} Above code shows AsyncDispatcher.dispatch method can process different type events. In fact this AsyncDispatcher instance is just ResourceManager.rmDispatcher created in ResourceManager.serviceInit method. You can find many eventTypes and handlers are registered in ResourceManager.rmDispatcher. In above scenario thread B blocks thread A, then many following events processing are blocked. In my testing cluster, there is only one

queue and the client submits 1000 applications concurrently, the yarn cluster usage rate is 50. Many apps are pending. If I disable resourceManager recover functionality, the cluster usage can be 100. To solve this issue, I removed synchronized modifier on some methods defined in RMStateStore. Instead, in these methods I defined a dedicated lock object before calling dispatcher.getEventHandler().handle. In this way, the yarn cluster usage rate can be 100 and the whole cluster is good running. Please see my attached patch.
label: requirement

3. **summary:** Yarn recover functionality causes the cluster running slowly and the cluster usage rate is far below 100
description: In my hadoop cluster, the resourceManager recover functionality is enabled with FileSystemRMStateStore. I found this cause the yarn cluster running slowly and cluster usage rate is just 50 even there are many pending Apps. The scenario is below. In thread A, the RMAppImpl$RMAppNewlySavingTransition is calling storeNewApplication method defined in RMStateStore. This storeNewApplication method is synchronized. {code:title=RMAppImpl.java|borderStyle=solid} private static final class RMAppNewlySavingTransition extends RMAppTransition { @Override public void transition(RMAppImpl app, RMAppEvent event) { // If recovery is enabled then store the application information in a // non-blocking call so make sure that RM has stored the information // needed to restart the AM after RM restart without further client // communication LOG.info("Storing application with id " + app.applicationId); app.rmContext.getStateStore().storeNewApplication(app); } } {code} {code:title=RMStateStore.java|borderStyle=solid} public synchronized void storeNewApplication(RMApp app) { ApplicationSubmissionContext context = app .getApplicationSubmissionContext(); assert context instanceof ApplicationSubmissionContextPBImpl; ApplicationStateData appState = ApplicationStateData.newInstance(app.getSubmitTime(), app.getStartTime(), context, app.getUser()); dispatcher.getEventHandler().handle(new RMStateStoreAppEvent(appState)); } {code} In thread B, the FileSystemRMStateStore is calling storeApplicationStateInternal method. It's also synchronized. This storeApplicationStateInternal method saves an ApplicationStateData into HDFS and it normally costs 90~300 milliseconds in my hadoop cluster. {code:title=FileSystemRMStateStore.java|borderStyle=solid} public synchronized void storeApplicationStateInternal(ApplicationId appId, ApplicationStateData appStateDataPB) throws Exception { Path appDirPath = getAppDir(rmAppRoot, appId); mkdirsWithRetries(appDirPath); Path nodeCreatePath = getNodePath(appDirPath, appId.toString()); LOG.info("Storing info for app: " + appId + " at: " + nodeCreatePath); byte[] appStateData = appStateDataPB.getProto().toByteArray(); try { // currently throw all exceptions. May need to respond differently for HA // based on whether we have lost the right to write to FS writeFileWithRetries(nodeCreatePath, appStateData, true); } catch (Exception e) { LOG.info("Error storing info for app: " + appId, e); throw e; } } {code} Think thread B firstly comes into FileSystemRMStateStore.storeApplicationStateInternal method, then thread A will be blocked for a while because of synchronization. In ResourceManager there is only one RMStateStore instance. In my cluster it's FileSystemRMStateStore type. Debug the RMAppNewlySavingTransition.transition method, the thread stack shows it's called form AsyncDispatcher.dispatch method. This method code is as below. {code:title=AsyncDispatcher.java|borderStyle=solid} protected void dispatch(Event event) { //all events go thru this loop if (LOG.isDebugEnabled()) { LOG.debug("Dispatching the event " + event.getClass().getName() + "." + event.toString()); } Class<? extends Enum> type = event.getType().getDeclaringClass(); try{ EventHandler handler = eventDispatchers.get(type); if(handler != null) { handler.handle(event); } else { throw new Exception("No handler for registered for " + type); } } catch (Throwable t) { //TODO Maybe log the state of the queue LOG.fatal("Error in dispatcher thread", t); // If serviceStop is called, we should exit this thread gracefully. if (exitOnDispatchException && (ShutdownHookManager.get().isShutdownInProgress()) == false && stopped == false) { Thread shutDownThread = new Thread(createShutDownThread()); shutDownThread.setName("AsyncDispatcher ShutDown handler"); shutDownThread.start(); } } } {code} Above code shows AsyncDispatcher.dispatch method can process different type events. In fact this AsyncDispatcher instance is just ResourceManager.rmDispatcher created in ResourceManager.serviceInit method. You can find many eventTypes and handlers are registered in ResourceManager.rmDispatcher. In above scenario thread B blocks thread A, then many following events processing are blocked. In my testing cluster, there is only one queue and the client submits 1000 applications concurrently, the yarn cluster usage rate is 50. Many apps are pending. If I disable resourceManager recover functionality, the cluster usage can be 100. To solve this issue, I removed synchronized modifier on some methods defined in RMStateStore. Instead, in these methods I defined a dedicated lock object before calling dispatcher.getEventHandler().handle. In this way, the yarn cluster usage rate can be 100 and the whole cluster is good running. Please see my attached patch.
4. **summary:** Yarn recover functionality causes the cluster running slowly and the cluster usage rate is far below 100

description: In my hadoop cluster, the resourceManager recover functionality is enabled with FileSystemRMStateStore. I found this cause the yarn cluster running slowly and cluster usage rate is just 50 even there are many pending Apps. The scenario is below. In thread A, the RMAppImpl$RMAppNewlySavingTransition is calling storeNewApplication method defined in RMStateStore. This storeNewApplication method is synchronized. {code:title=RMAppImpl.java|borderStyle=solid} private static final class RMAppNewlySavingTransition extends RMAppTransition { @Override public void transition(RMAppImpl app, RMAppEvent event) { // If recovery is enabled then store the application information in a // non-blocking call so make sure that RM has stored the information // needed to restart the AM after RM restart without further client // communication LOG.info("Storing application with id " + app.applicationId); app.rmContext.getStateStore().storeNewApplication(app); } } {code} {code:title=RMStateStore.java|borderStyle=solid} public synchronized void storeNewApplication(RMApp app) { ApplicationSubmissionContext context = app .getApplicationSubmissionContext(); assert context instanceof ApplicationSubmissionContextPBImpl; ApplicationStateData appState = ApplicationStateData.newInstance(app.getSubmitTime(), app.getStartTime(), context, app.getUser()); dispatcher.getEventHandler().handle(new RMStateStoreAppEvent(appState)); } {code} In thread B, the FileSystemRMStateStore is calling storeApplicationStateInternal method. It's also synchronized. This storeApplicationStateInternal method saves an ApplicationStateData into HDFS and it normally costs 90~300 milliseconds in my hadoop cluster. {code:title=FileSystemRMStateStore.java|borderStyle=solid} public synchronized void storeApplicationStateInternal(ApplicationId appId, ApplicationStateData appStateDataPB) throws Exception { Path appDirPath = getAppDir(rmAppRoot, appId); mkdirsWithRetries(appDirPath); Path nodeCreatePath = getNodePath(appDirPath, appId.toString()); LOG.info("Storing info for app: " + appId + " at: " + nodeCreatePath); byte[] appStateData = appStateDataPB.getProto().toByteArray(); try { // currently throw all exceptions. May need to respond differently for HA // based on whether we have lost the right to write to FS writeFileWithRetries(nodeCreatePath, appStateData, true); } catch (Exception e) { LOG.info("Error storing info for app: " + appId, e); throw e; } } {code} Think thread B firstly comes into FileSystemRMStateStore.storeApplicationStateInternal method, then thread A will be blocked for a while because of synchronization. In ResourceManager there is only one RMStateStore instance. In my cluster it's FileSystemRMStateStore type. Debug the RMAppNewlySavingTransition.transition method, the thread stack shows it's called form AsyncDispatcher.dispatch method. This method code is as below. {code:title=AsyncDispatcher.java|borderStyle=solid} protected void dispatch(Event event) { //all events go thru this loop if (LOG.isDebugEnabled()) { LOG.debug("Dispatching the event " + event.getClass().getName() + "." + event.toString()); } Class<? extends Enum> type = event.getType().getDeclaringClass(); try{ EventHandler handler = eventDispatchers.get(type); if(handler != null) { handler.handle(event); } else { throw new Exception("No handler for registered for " + type); } } catch (Throwable t) { //TODO Maybe log the state of the queue LOG.fatal("Error in dispatcher thread", t); // If serviceStop is called, we should exit this thread gracefully. if (exitOnDispatchException && (ShutdownHookManager.get().isShutdownInProgress()) == false && stopped == false) { Thread shutDownThread = new Thread(createShutDownThread()); shutDownThread.setName("AsyncDispatcher ShutDown handler"); shutDownThread.start(); } } } {code} Above code shows AsyncDispatcher.dispatch method can process different type events. In fact this AsyncDispatcher instance is just ResourceManager.rmDispatcher created in ResourceManager.serviceInit method. You can find many eventTypes and handlers are registered in ResourceManager.rmDispatcher. In above scenario thread B blocks thread A, then many following events processing are blocked. In my testing cluster, there is only one queue and the client submits 1000 applications concurrently, the yarn cluster usage rate is 50. Many apps are pending. If I disable resourceManager recover functionality, the cluster usage can be 100. To solve this issue, I removed synchronized modifier on some methods defined in RMStateStore. Instead, in these methods I defined a dedicated lock object before calling dispatcher.getEventHandler().handle. In this way, the yarn cluster usage rate can be 100 and the whole cluster is good running. Please see my attached patch.
label: code-design

5. **summary:** Yarn recover functionality causes the cluster running slowly and the cluster usage rate is far below 100
description: In my hadoop cluster, the resourceManager recover functionality is enabled with FileSystemRMStateStore. I found this cause the yarn cluster running slowly and cluster usage rate is just 50 even there are many pending Apps. The scenario is below. In thread A, the RMAppImpl$RMAppNewlySavingTransition is calling storeNewApplication method defined in RMStateStore. This storeNewApplication method is synchronized. {code:title=RMAppImpl.java|borderStyle=solid} private static final class RMAppNewlySavingTransition extends RMAppTransition { @Override public void transition(RMAppImpl app, RMAppEvent event) { //

If recovery is enabled then store the application information in a // non-blocking call so make sure that RM has stored the information // needed to restart the AM after RM restart without further client // communication LOG.info("Storing application with id " + app.applicationId); app.rmContext.getStateStore().storeNewApplication(app); } } {code} {code:title=RMStateStore.java|borderStyle=solid} public synchronized void storeNewApplication(RMApp app) { ApplicationSubmissionContext context = app .getApplicationSubmissionContext(); assert context instanceof ApplicationSubmissionContextPBImpl; ApplicationStateData appState = ApplicationStateData.newInstance(app.getSubmitTime(), app.getStartTime(), context, app.getUser()); dispatcher.getEventHandler().handle(new RMStateStoreAppEvent(appState)); } {code} In thread B, the FileSystemRMStateStore is calling storeApplicationStateInternal method. It's also synchronized. This storeApplicationStateInternal method saves an ApplicationStateData into HDFS and it normally costs 90~300 milliseconds in my hadoop cluster. {code:title=FileSystemRMStateStore.java|borderStyle=solid} public synchronized void storeApplicationStateInternal(ApplicationId appId, ApplicationStateData appStateDataPB) throws Exception { Path appDirPath = getAppDir(rmAppRoot, appId); mkdirsWithRetries(appDirPath); Path nodeCreatePath = getNodePath(appDirPath, appId.toString()); LOG.info("Storing info for app: " + appId + " at: " + nodeCreatePath); byte[] appStateData = appStateDataPB.getProto().toByteArray(); try { // currently throw all exceptions. May need to respond differently for HA // based on whether we have lost the right to write to FS writeFileWithRetries(nodeCreatePath, appStateData, true); } catch (Exception e) { LOG.info("Error storing info for app: " + appId, e); throw e; } } {code} Think thread B firstly comes into FileSystemRMStateStore.storeApplicationStateInternal method, then thread A will be blocked for a while because of synchronization. In ResourceManager there is only one RMStateStore instance. In my cluster it's FileSystemRMStateStore type. Debug the RMAppNewlySavingTransition.transition method, the thread stack shows it's called form AsyncDispatcher.dispatch method. This method code is as below. {code:title=AsyncDispatcher.java|borderStyle=solid} protected void dispatch(Event event) { //all events go thru this loop if (LOG.isDebugEnabled()) { LOG.debug("Dispatching the event " + event.getClass().getName() + "." + event.toString()); } Class<? extends Enum> type = event.getType().getDeclaringClass(); try{ EventHandler handler = eventDispatchers.get(type); if(handler != null) { handler.handle(event); } else { throw new Exception("No handler for registered for " + type); } } catch (Throwable t) { //TODO Maybe log the state of the queue LOG.fatal("Error in dispatcher thread", t); // If serviceStop is called, we should exit this thread gracefully. if (exitOnDispatchException && (ShutdownHookManager.get().isShutdownInProgress()) == false && stopped == false) { Thread shutDownThread = new Thread(createShutDownThread()); shutDownThread.setName("AsyncDispatcher ShutDown handler"); shutDownThread.start(); } } } {code} Above code shows AsyncDispatcher.dispatch method can process different type events. In fact this AsyncDispatcher instance is just ResourceManager.rmDispatcher created in ResourceManager.serviceInit method. You can find many eventTypes and handlers are registered in ResourceManager.rmDispatcher. In above scenario thread B blocks thread A, then many following events processing are blocked. In my testing cluster, there is only one queue and the client submits 1000 applications concurrently, the yarn cluster usage rate is 50. Many apps are pending. If I disable resourceManager recover functionality, the cluster usage can be 100. To solve this issue, I removed synchronized modifier on some methods defined in RMStateStore. Instead, in these methods I defined a dedicated lock object before calling dispatcher.getEventHandler().handle. In this way, the yarn cluster usage rate can be 100 and the whole cluster is good running. Please see my attached patch.

6. **summary:** Yarn recover functionality causes the cluster running slowly and the cluster usage rate is far below 100
   description: In my hadoop cluster, the resourceManager recover functionality is enabled with FileSystemRMStateStore. I found this cause the yarn cluster running slowly and cluster usage rate is just 50 even there are many pending Apps. The scenario is below. In thread A, the RMAppImpl$RMAppNewlySavingTransition is calling storeNewApplication method defined in RMStateStore. This storeNewApplication method is synchronized. {code:title=RMAppImpl.java|borderStyle=solid} private static final class RMAppNewlySavingTransition extends RMAppTransition { @Override public void transition(RMAppImpl app, RMAppEvent event) { // If recovery is enabled then store the application information in a // non-blocking call so make sure that RM has stored the information // needed to restart the AM after RM restart without further client // communication LOG.info("Storing application with id " + app.applicationId); app.rmContext.getStateStore().storeNewApplication(app); } } {code} {code:title=RMStateStore.java|borderStyle=solid} public synchronized void storeNewApplication(RMApp app) { ApplicationSubmissionContext context = app .getApplicationSubmissionContext(); assert context instanceof ApplicationSubmissionContextPBImpl; ApplicationStateData appState = ApplicationStateData.newInstance(app.getSubmitTime(), app.getStartTime(), context, app.getUser());

dispatcher.getEventHandler().handle(new RMStateStoreAppEvent(appState)); } {code} In thread B, the FileSystemRMStateStore is calling storeApplicationStateInternal method. It's also synchronized. This storeApplicationStateInternal method saves an ApplicationStateData into HDFS and it normally costs 90~300 milliseconds in my hadoop cluster. {code:title=FileSystemRMStateStore.java|borderStyle=solid} public synchronized void storeApplicationStateInternal(ApplicationId appId, ApplicationStateData appStateDataPB) throws Exception { Path appDirPath = getAppDir(rmAppRoot, appId); mkdirsWithRetries(appDirPath); Path nodeCreatePath = getNodePath(appDirPath, appId.toString()); LOG.info("Storing info for app: " + appId + " at: " + nodeCreatePath); byte[] appStateData = appStateDataPB.getProto().toByteArray(); try { // currently throw all exceptions. May need to respond differently for HA // based on whether we have lost the right to write to FS writeFileWithRetries(nodeCreatePath, appStateData, true); } catch (Exception e) { LOG.info("Error storing info for app: " + appId, e); throw e; } } {code} Think thread B firstly comes into FileSystemRMStateStore.storeApplicationStateInternal method, then thread A will be blocked for a while because of synchronization. In ResourceManager there is only one RMStateStore instance. In my cluster it's FileSystemRMStateStore type. Debug the RMAppNewlySavingTransition.transition method, the thread stack shows it's called form AsyncDispatcher.dispatch method. This method code is as below. {code:title=AsyncDispatcher.java|borderStyle=solid} protected void dispatch(Event event) { //all events go thru this loop if (LOG.isDebugEnabled()) { LOG.debug("Dispatching the event " + event.getClass().getName() + "." + event.toString()); } Class<? extends Enum> type = event.getType().getDeclaringClass(); try{ EventHandler handler = eventDispatchers.get(type); if(handler != null) { handler.handle(event); } else { throw new Exception("No handler for registered for " + type); } } catch (Throwable t) { //TODO Maybe log the state of the queue LOG.fatal("Error in dispatcher thread", t); // If serviceStop is called, we should exit this thread gracefully. if (exitOnDispatchException && (ShutdownHookManager.get().isShutdownInProgress()) == false && stopped == false) { Thread shutDownThread = new Thread(createShutDownThread()); shutDownThread.setName("AsyncDispatcher ShutDown handler"); shutDownThread.start(); } } } {code} Above code shows AsyncDispatcher.dispatch method can process different type events. In fact this AsyncDispatcher instance is just ResourceManager.rmDispatcher created in ResourceManager.serviceInit method. You can find many eventTypes and handlers are registered in ResourceManager.rmDispatcher. In above scenario thread B blocks thread A, then many following events processing are blocked. In my testing cluster, there is only one queue and the client submits 1000 applications concurrently, the yarn cluster usage rate is 50. Many apps are pending. If I disable resourceManager recover functionality, the cluster usage can be 100. To solve this issue, I removed synchronized modifier on some methods defined in RMStateStore. Instead, in these methods I defined a dedicated lock object before calling dispatcher.getEventHandler().handle. In this way, the yarn cluster usage rate can be 100 and the whole cluster is good running. Please see my attached patch.

7. **summary:** Yarn recover functionality causes the cluster running slowly and the cluster usage rate is far below 100
   description: In my hadoop cluster, the resourceManager recover functionality is enabled with FileSystemRMStateStore. I found this cause the yarn cluster running slowly and cluster usage rate is just 50 even there are many pending Apps. The scenario is below. In thread A, the RMAppImpl$RMAppNewlySavingTransition is calling storeNewApplication method defined in RMStateStore. This storeNewApplication method is synchronized. {code:title=RMAppImpl.java|borderStyle=solid} private static final class RMAppNewlySavingTransition extends RMAppTransition { @Override public void transition(RMAppImpl app, RMAppEvent event) { // If recovery is enabled then store the application information in a // non-blocking call so make sure that RM has stored the information // needed to restart the AM after RM restart without further client // communication LOG.info("Storing application with id " + app.applicationId); app.rmContext.getStateStore().storeNewApplication(app); } } {code} {code:title=RMStateStore.java|borderStyle=solid} public synchronized void storeNewApplication(RMApp app) { ApplicationSubmissionContext context = app .getApplicationSubmissionContext(); assert context instanceof ApplicationSubmissionContextPBImpl; ApplicationStateData appState = ApplicationStateData.newInstance(app.getSubmitTime(), app.getStartTime(), context, app.getUser()); dispatcher.getEventHandler().handle(new RMStateStoreAppEvent(appState)); } {code} In thread B, the FileSystemRMStateStore is calling storeApplicationStateInternal method. It's also synchronized. This storeApplicationStateInternal method saves an ApplicationStateData into HDFS and it normally costs 90~300 milliseconds in my hadoop cluster. {code:title=FileSystemRMStateStore.java|borderStyle=solid} public synchronized void storeApplicationStateInternal(ApplicationId appId, ApplicationStateData appStateDataPB) throws Exception { Path appDirPath = getAppDir(rmAppRoot, appId); mkdirsWithRetries(appDirPath); Path nodeCreatePath = getNodePath(appDirPath, appId.toString()); LOG.info("Storing info for app: " + appId + " at: " + nodeCreatePath); byte[] appStateData =

appStateDataPB.getProto().toByteArray(); try { // currently throw all exceptions. May need to respond differently for HA // based on whether we have lost the right to write to FS writeFileWithRetries(nodeCreatePath, appStateData, true); } catch (Exception e) { LOG.info("Error storing info for app: " + appId, e); throw e; } } {code} Think thread B firstly comes into FileSystemRMStateStore.storeApplicationStateInternal method, then thread A will be blocked for a while because of synchronization. In ResourceManager there is only one RMStateStore instance. In my cluster it's FileSystemRMStateStore type. Debug the RMAppNewlySavingTransition.transition method, the thread stack shows it's called form AsyncDispatcher.dispatch method. This method code is as below. {code:title=AsyncDispatcher.java|borderStyle=solid} protected void dispatch(Event event) { //all events go thru this loop if (LOG.isDebugEnabled()) { LOG.debug("Dispatching the event " + event.getClass().getName() + "." + event.toString()); } Class<? extends Enum> type = event.getType().getDeclaringClass(); try{ EventHandler handler = eventDispatchers.get(type); if(handler != null) { handler.handle(event); } else { throw new Exception("No handler for registered for " + type); } } catch (Throwable t) { //TODO Maybe log the state of the queue LOG.fatal("Error in dispatcher thread", t); // If serviceStop is called, we should exit this thread gracefully. if (exitOnDispatchException && (ShutdownHookManager.get().isShutdownInProgress()) == false && stopped == false) { Thread shutDownThread = new Thread(createShutDownThread()); shutDownThread.setName("AsyncDispatcher ShutDown handler"); shutDownThread.start(); } } } {code} Above code shows AsyncDispatcher.dispatch method can process different type events. In fact this AsyncDispatcher instance is just ResourceManager.rmDispatcher created in ResourceManager.serviceInit method. You can find many eventTypes and handlers are registered in ResourceManager.rmDispatcher. In above scenario thread B blocks thread A, then many following events processing are blocked. In my testing cluster, there is only one queue and the client submits 1000 applications concurrently, the yarn cluster usage rate is 50. Many apps are pending. If I disable resourceManager recover functionality, the cluster usage can be 100. To solve this issue, I removed synchronized modifier on some methods defined in RMStateStore. Instead, in these methods I defined a dedicated lock object before calling dispatcher.getEventHandler().handle. In this way, the yarn cluster usage rate can be 100 and the whole cluster is good running. Please see my attached patch.

8. **summary:** Yarn recover functionality causes the cluster running slowly and the cluster usage rate is far below 100
   description: In my hadoop cluster, the resourceManager recover functionality is enabled with FileSystemRMStateStore. I found this cause the yarn cluster running slowly and cluster usage rate is just 50 even there are many pending Apps. The scenario is below. In thread A, the RMAppImpl$RMAppNewlySavingTransition is calling storeNewApplication method defined in RMStateStore. This storeNewApplication method is synchronized. {code:title=RMAppImpl.java|borderStyle=solid} private static final class RMAppNewlySavingTransition extends RMAppTransition { @Override public void transition(RMAppImpl app, RMAppEvent event) { // If recovery is enabled then store the application information in a // non-blocking call so make sure that RM has stored the information // needed to restart the AM after RM restart without further client // communication LOG.info("Storing application with id " + app.applicationId); app.rmContext.getStateStore().storeNewApplication(app); } } {code} {code:title=RMStateStore.java|borderStyle=solid} public synchronized void storeNewApplication(RMApp app) { ApplicationSubmissionContext context = app .getApplicationSubmissionContext(); assert context instanceof ApplicationSubmissionContextPBImpl; ApplicationStateData appState = ApplicationStateData.newInstance(app.getSubmitTime(), app.getStartTime(), context, app.getUser()); dispatcher.getEventHandler().handle(new RMStateStoreAppEvent(appState)); } {code} In thread B, the FileSystemRMStateStore is calling storeApplicationStateInternal method. It's also synchronized. This storeApplicationStateInternal method saves an ApplicationStateData into HDFS and it normally costs 90~300 milliseconds in my hadoop cluster. {code:title=FileSystemRMStateStore.java|borderStyle=solid} public synchronized void storeApplicationStateInternal(ApplicationId appId, ApplicationStateData appStateDataPB) throws Exception { Path appDirPath = getAppDir(rmAppRoot, appId); mkdirsWithRetries(appDirPath); Path nodeCreatePath = getNodePath(appDirPath, appId.toString()); LOG.info("Storing info for app: " + appId + " at: " + nodeCreatePath); byte[] appStateData = appStateDataPB.getProto().toByteArray(); try { // currently throw all exceptions. May need to respond differently for HA // based on whether we have lost the right to write to FS writeFileWithRetries(nodeCreatePath, appStateData, true); } catch (Exception e) { LOG.info("Error storing info for app: " + appId, e); throw e; } } {code} Think thread B firstly comes into FileSystemRMStateStore.storeApplicationStateInternal method, then thread A will be blocked for a while because of synchronization. In ResourceManager there is only one RMStateStore instance. In my cluster it's FileSystemRMStateStore type. Debug the RMAppNewlySavingTransition.transition method, the thread stack shows it's called form AsyncDispatcher.dispatch method. This method code is as below.

{code:title=AsyncDispatcher.java|borderStyle=solid} protected void dispatch(Event event) { //all events go thru this loop if (LOG.isDebugEnabled()) { LOG.debug("Dispatching the event " + event.getClass().getName() + "." + event.toString()); } Class<? extends Enum> type = event.getType().getDeclaringClass(); try{ EventHandler handler = eventDispatchers.get(type); if(handler != null) { handler.handle(event); } else { throw new Exception("No handler for registered for " + type); } } catch (Throwable t) { //TODO Maybe log the state of the queue LOG.fatal("Error in dispatcher thread", t); // If serviceStop is called, we should exit this thread gracefully. if (exitOnDispatchException && (ShutdownHookManager.get().isShutdownInProgress()) == false && stopped == false) { Thread shutDownThread = new Thread(createShutDownThread()); shutDownThread.setName("AsyncDispatcher ShutDown handler"); shutDownThread.start(); } } } {code} Above code shows AsyncDispatcher.dispatch method can process different type events. In fact this AsyncDispatcher instance is just ResourceManager.rmDispatcher created in ResourceManager.serviceInit method. You can find many eventTypes and handlers are registered in ResourceManager.rmDispatcher. In above scenario thread B blocks thread A, then many following events processing are blocked. In my testing cluster, there is only one queue and the client submits 1000 applications concurrently, the yarn cluster usage rate is 50. Many apps are pending. If I disable resourceManager recover functionality, the cluster usage can be 100. To solve this issue, I removed synchronized modifier on some methods defined in RMStateStore. Instead, in these methods I defined a dedicated lock object before calling dispatcher.getEventHandler().handle. In this way, the yarn cluster usage rate can be 100 and the whole cluster is good running. Please see my attached patch.

9. **summary:** Yarn recover functionality causes the cluster running slowly and the cluster usage rate is far below 100
description: In my hadoop cluster, the resourceManager recover functionality is enabled with FileSystemRMStateStore. I found this cause the yarn cluster running slowly and cluster usage rate is just 50 even there are many pending Apps. The scenario is below. In thread A, the RMAppImpl$RMAppNewlySavingTransition is calling storeNewApplication method defined in RMStateStore. This storeNewApplication method is synchronized. {code:title=RMAppImpl.java|borderStyle=solid} private static final class RMAppNewlySavingTransition extends RMAppTransition { @Override public void transition(RMAppImpl app, RMAppEvent event) { // If recovery is enabled then store the application information in a // non-blocking call so make sure that RM has stored the information // needed to restart the AM after RM restart without further client // communication LOG.info("Storing application with id " + app.applicationId); app.rmContext.getStateStore().storeNewApplication(app); } } {code} {code:title=RMStateStore.java|borderStyle=solid} public synchronized void storeNewApplication(RMApp app) { ApplicationSubmissionContext context = app .getApplicationSubmissionContext(); assert context instanceof ApplicationSubmissionContextPBImpl; ApplicationStateData appState = ApplicationStateData.newInstance(app.getSubmitTime(), app.getStartTime(), context, app.getUser()); dispatcher.getEventHandler().handle(new RMStateStoreAppEvent(appState)); } {code} In thread B, the FileSystemRMStateStore is calling storeApplicationStateInternal method. It's also synchronized. This storeApplicationStateInternal method saves an ApplicationStateData into HDFS and it normally costs 90~300 milliseconds in my hadoop cluster. {code:title=FileSystemRMStateStore.java|borderStyle=solid} public synchronized void storeApplicationStateInternal(ApplicationId appId, ApplicationStateData appStateDataPB) throws Exception { Path appDirPath = getAppDir(rmAppRoot, appId); mkdirsWithRetries(appDirPath); Path nodeCreatePath = getNodePath(appDirPath, appId.toString()); LOG.info("Storing info for app: " + appId + " at: " + nodeCreatePath); byte[] appStateData = appStateDataPB.getProto().toByteArray(); try { // currently throw all exceptions. May need to respond differently for HA // based on whether we have lost the right to write to FS writeFileWithRetries(nodeCreatePath, appStateData, true); } catch (Exception e) { LOG.info("Error storing info for app: " + appId, e); throw e; } } {code} Think thread B firstly comes into FileSystemRMStateStore.storeApplicationStateInternal method, then thread A will be blocked for a while because of synchronization. In ResourceManager there is only one RMStateStore instance. In my cluster it's FileSystemRMStateStore type. Debug the RMAppNewlySavingTransition.transition method, the thread stack shows it's called form AsyncDispatcher.dispatch method. This method code is as below. {code:title=AsyncDispatcher.java|borderStyle=solid} protected void dispatch(Event event) { //all events go thru this loop if (LOG.isDebugEnabled()) { LOG.debug("Dispatching the event " + event.getClass().getName() + "." + event.toString()); } Class<? extends Enum> type = event.getType().getDeclaringClass(); try{ EventHandler handler = eventDispatchers.get(type); if(handler != null) { handler.handle(event); } else { throw new Exception("No handler for registered for " + type); } } catch (Throwable t) { //TODO Maybe log the state of the queue LOG.fatal("Error in dispatcher thread", t); // If serviceStop is called, we should exit this thread gracefully. if (exitOnDispatchException && (ShutdownHookManager.get().isShutdownInProgress()) == false && stopped == false) { Thread

shutDownThread = new Thread(createShutDownThread()); shutDownThread.setName("AsyncDispatcher ShutDown handler"); shutDownThread.start(); } } } {code} Above code shows AsyncDispatcher.dispatch method can process different type events. In fact this AsyncDispatcher instance is just ResourceManager.rmDispatcher created in ResourceManager.serviceInit method. You can find many eventTypes and handlers are registered in ResourceManager.rmDispatcher. In above scenario thread B blocks thread A, then many following events processing are blocked. In my testing cluster, there is only one queue and the client submits 1000 applications concurrently, the yarn cluster usage rate is 50. Many apps are pending. If I disable resourceManager recover functionality, the cluster usage can be 100. To solve this issue, I removed synchronized modifier on some methods defined in RMStateStore. Instead, in these methods I defined a dedicated lock object before calling dispatcher.getEventHandler().handle. In this way, the yarn cluster usage rate can be 100 and the whole cluster is good running. Please see my attached patch.

10. **summary:** Yarn recover functionality causes the cluster running slowly and the cluster usage rate is far below 100
description: In my hadoop cluster, the resourceManager recover functionality is enabled with FileSystemRMStateStore. I found this cause the yarn cluster running slowly and cluster usage rate is just 50 even there are many pending Apps. The scenario is below. In thread A, the RMAppImpl$RMAppNewlySavingTransition is calling storeNewApplication method defined in RMStateStore. This storeNewApplication method is synchronized. {code:title=RMAppImpl.java|borderStyle=solid} private static final class RMAppNewlySavingTransition extends RMAppTransition { @Override public void transition(RMAppImpl app, RMAppEvent event) { // If recovery is enabled then store the application information in a // non-blocking call so make sure that RM has stored the information // needed to restart the AM after RM restart without further client // communication LOG.info("Storing application with id " + app.applicationId); app.rmContext.getStateStore().storeNewApplication(app); } } {code} {code:title=RMStateStore.java|borderStyle=solid} public synchronized void storeNewApplication(RMApp app) { ApplicationSubmissionContext context = app .getApplicationSubmissionContext(); assert context instanceof ApplicationSubmissionContextPBImpl; ApplicationStateData appState = ApplicationStateData.newInstance(app.getSubmitTime(), app.getStartTime(), context, app.getUser()); dispatcher.getEventHandler().handle(new RMStateStoreAppEvent(appState)); } {code} In thread B, the FileSystemRMStateStore is calling storeApplicationStateInternal method. It's also synchronized. This storeApplicationStateInternal method saves an ApplicationStateData into HDFS and it normally costs 90~300 milliseconds in my hadoop cluster. {code:title=FileSystemRMStateStore.java|borderStyle=solid} public synchronized void storeApplicationStateInternal(ApplicationId appId, ApplicationStateData appStateDataPB) throws Exception { Path appDirPath = getAppDir(rmAppRoot, appId); mkdirsWithRetries(appDirPath); Path nodeCreatePath = getNodePath(appDirPath, appId.toString()); LOG.info("Storing info for app: " + appId + " at: " + nodeCreatePath); byte[] appStateData = appStateDataPB.getProto().toByteArray(); try { // currently throw all exceptions. May need to respond differently for HA // based on whether we have lost the right to write to FS writeFileWithRetries(nodeCreatePath, appStateData, true); } catch (Exception e) { LOG.info("Error storing info for app: " + appId, e); throw e; } } {code} Think thread B firstly comes into FileSystemRMStateStore.storeApplicationStateInternal method, then thread A will be blocked for a while because of synchronization. In ResourceManager there is only one RMStateStore instance. In my cluster it's FileSystemRMStateStore type. Debug the RMAppNewlySavingTransition.transition method, the thread stack shows it's called form AsyncDispatcher.dispatch method. This method code is as below. {code:title=AsyncDispatcher.java|borderStyle=solid} protected void dispatch(Event event) { //all events go thru this loop if (LOG.isDebugEnabled()) { LOG.debug("Dispatching the event " + event.getClass().getName() + "." + event.toString()); } Class<? extends Enum> type = event.getType().getDeclaringClass(); try{ EventHandler handler = eventDispatchers.get(type); if(handler != null) { handler.handle(event); } else { throw new Exception("No handler for registered for " + type); } } catch (Throwable t) { //TODO Maybe log the state of the queue LOG.fatal("Error in dispatcher thread", t); // If serviceStop is called, we should exit this thread gracefully. if (exitOnDispatchException && (ShutdownHookManager.get().isShutdownInProgress()) == false && stopped == false) { Thread shutDownThread = new Thread(createShutDownThread()); shutDownThread.setName("AsyncDispatcher ShutDown handler"); shutDownThread.start(); } } } {code} Above code shows AsyncDispatcher.dispatch method can process different type events. In fact this AsyncDispatcher instance is just ResourceManager.rmDispatcher created in ResourceManager.serviceInit method. You can find many eventTypes and handlers are registered in ResourceManager.rmDispatcher. In above scenario thread B blocks thread A, then many following events processing are blocked. In my testing cluster, there is only one queue and the client submits 1000 applications concurrently, the yarn cluster usage rate is 50. Many apps are pending. If I disable resourceManager recover functionality, the cluster usage can be 100. To solve this

issue, I removed synchronized modifier on some methods defined in RMStateStore. Instead, in these methods I defined a dedicated lock object before calling dispatcher.getEventHandler().handle. In this way, the yarn cluster usage rate can be 100 and the whole cluster is good running. Please see my attached patch.

11. **summary:** Yarn recover functionality causes the cluster running slowly and the cluster usage rate is far below 100
description: In my hadoop cluster, the resourceManager recover functionality is enabled with FileSystemRMStateStore. I found this cause the yarn cluster running slowly and cluster usage rate is just 50 even there are many pending Apps. The scenario is below. In thread A, the RMAppImpl$RMAppNewlySavingTransition is calling storeNewApplication method defined in RMStateStore. This storeNewApplication method is synchronized. {code:title=RMAppImpl.java|borderStyle=solid} private static final class RMAppNewlySavingTransition extends RMAppTransition { @Override public void transition(RMAppImpl app, RMAppEvent event) { // If recovery is enabled then store the application information in a // non-blocking call so make sure that RM has stored the information // needed to restart the AM after RM restart without further client // communication LOG.info("Storing application with id " + app.applicationId); app.rmContext.getStateStore().storeNewApplication(app); } } {code} {code:title=RMStateStore.java|borderStyle=solid} public synchronized void storeNewApplication(RMApp app) { ApplicationSubmissionContext context = app .getApplicationSubmissionContext(); assert context instanceof ApplicationSubmissionContextPBImpl; ApplicationStateData appState = ApplicationStateData.newInstance(app.getSubmitTime(), app.getStartTime(), context, app.getUser()); dispatcher.getEventHandler().handle(new RMStateStoreAppEvent(appState)); } {code} In thread B, the FileSystemRMStateStore is calling storeApplicationStateInternal method. It's also synchronized. This storeApplicationStateInternal method saves an ApplicationStateData into HDFS and it normally costs 90~300 milliseconds in my hadoop cluster. {code:title=FileSystemRMStateStore.java|borderStyle=solid} public synchronized void storeApplicationStateInternal(ApplicationId appId, ApplicationStateData appStateDataPB) throws Exception { Path appDirPath = getAppDir(rmAppRoot, appId); mkdirsWithRetries(appDirPath); Path nodeCreatePath = getNodePath(appDirPath, appId.toString()); LOG.info("Storing info for app: " + appId + " at: " + nodeCreatePath); byte[] appStateData = appStateDataPB.getProto().toByteArray(); try { // currently throw all exceptions. May need to respond differently for HA // based on whether we have lost the right to write to FS writeFileWithRetries(nodeCreatePath, appStateData, true); } catch (Exception e) { LOG.info("Error storing info for app: " + appId, e); throw e; } } {code} Think thread B firstly comes into FileSystemRMStateStore.storeApplicationStateInternal method, then thread A will be blocked for a while because of synchronization. In ResourceManager there is only one RMStateStore instance. In my cluster it's FileSystemRMStateStore type. Debug the RMAppNewlySavingTransition.transition method, the thread stack shows it's called form AsyncDispatcher.dispatch method. This method code is as below. {code:title=AsyncDispatcher.java|borderStyle=solid} protected void dispatch(Event event) { //all events go thru this loop if (LOG.isDebugEnabled()) { LOG.debug("Dispatching the event " + event.getClass().getName() + "." + event.toString()); } Class<? extends Enum> type = event.getType().getDeclaringClass(); try{ EventHandler handler = eventDispatchers.get(type); if(handler != null) { handler.handle(event); } else { throw new Exception("No handler for registered for " + type); } } catch (Throwable t) { //TODO Maybe log the state of the queue LOG.fatal("Error in dispatcher thread", t); // If serviceStop is called, we should exit this thread gracefully. if (exitOnDispatchException && (ShutdownHookManager.get().isShutdownInProgress()) == false && stopped == false) { Thread shutDownThread = new Thread(createShutDownThread()); shutDownThread.setName("AsyncDispatcher ShutDown handler"); shutDownThread.start(); } } } {code} Above code shows AsyncDispatcher.dispatch method can process different type events. In fact this AsyncDispatcher instance is just ResourceManager.rmDispatcher created in ResourceManager.serviceInit method. You can find many eventTypes and handlers are registered in ResourceManager.rmDispatcher. In above scenario thread B blocks thread A, then many following events processing are blocked. In my testing cluster, there is only one queue and the client submits 1000 applications concurrently, the yarn cluster usage rate is 50. Many apps are pending. If I disable resourceManager recover functionality, the cluster usage can be 100. To solve this issue, I removed synchronized modifier on some methods defined in RMStateStore. Instead, in these methods I defined a dedicated lock object before calling dispatcher.getEventHandler().handle. In this way, the yarn cluster usage rate can be 100 and the whole cluster is good running. Please see my attached patch.
label: code-design
12. **summary:** Yarn recover functionality causes the cluster running slowly and the cluster usage rate is far below 100
description: In my hadoop cluster, the resourceManager recover functionality is enabled with FileSystemRMStateStore. I found this cause the yarn cluster running slowly and cluster usage rate is just 50

even there are many pending Apps. The scenario is below. In thread A, the RMAppImpl$RMAppNewlySavingTransition is calling storeNewApplication method defined in RMStateStore. This storeNewApplication method is synchronized. {code:title=RMAppImpl.java|borderStyle=solid} private static final class RMAppNewlySavingTransition extends RMAppTransition { @Override public void transition(RMAppImpl app, RMAppEvent event) { // If recovery is enabled then store the application information in a // non-blocking call so make sure that RM has stored the information // needed to restart the AM after RM restart without further client // communication LOG.info("Storing application with id " + app.applicationId); app.rmContext.getStateStore().storeNewApplication(app); } } {code} {code:title=RMStateStore.java|borderStyle=solid} public synchronized void storeNewApplication(RMApp app) { ApplicationSubmissionContext context = app .getApplicationSubmissionContext(); assert context instanceof ApplicationSubmissionContextPBImpl; ApplicationStateData appState = ApplicationStateData.newInstance(app.getSubmitTime(), app.getStartTime(), context, app.getUser()); dispatcher.getEventHandler().handle(new RMStateStoreAppEvent(appState)); } {code} In thread B, the FileSystemRMStateStore is calling storeApplicationStateInternal method. It's also synchronized. This storeApplicationStateInternal method saves an ApplicationStateData into HDFS and it normally costs 90~300 milliseconds in my hadoop cluster. {code:title=FileSystemRMStateStore.java|borderStyle=solid} public synchronized void storeApplicationStateInternal(ApplicationId appId, ApplicationStateData appStateDataPB) throws Exception { Path appDirPath = getAppDir(rmAppRoot, appId); mkdirsWithRetries(appDirPath); Path nodeCreatePath = getNodePath(appDirPath, appId.toString()); LOG.info("Storing info for app: " + appId + " at: " + nodeCreatePath); byte[] appStateData = appStateDataPB.getProto().toByteArray(); try { // currently throw all exceptions. May need to respond differently for HA // based on whether we have lost the right to write to FS writeFileWithRetries(nodeCreatePath, appStateData, true); } catch (Exception e) { LOG.info("Error storing info for app: " + appId, e); throw e; } } {code} Think thread B firstly comes into FileSystemRMStateStore.storeApplicationStateInternal method, then thread A will be blocked for a while because of synchronization. In ResourceManager there is only one RMStateStore instance. In my cluster it's FileSystemRMStateStore type. Debug the RMAppNewlySavingTransition.transition method, the thread stack shows it's called form AsyncDispatcher.dispatch method. This method code is as below. {code:title=AsyncDispatcher.java|borderStyle=solid} protected void dispatch(Event event) { //all events go thru this loop if (LOG.isDebugEnabled()) { LOG.debug("Dispatching the event " + event.getClass().getName() + "." + event.toString()); } Class<? extends Enum> type = event.getType().getDeclaringClass(); try{ EventHandler handler = eventDispatchers.get(type); if(handler != null) { handler.handle(event); } else { throw new Exception("No handler for registered for " + type); } } catch (Throwable t) { //TODO Maybe log the state of the queue LOG.fatal("Error in dispatcher thread", t); // If serviceStop is called, we should exit this thread gracefully. if (exitOnDispatchException && (ShutdownHookManager.get().isShutdownInProgress()) == false && stopped == false) { Thread shutDownThread = new Thread(createShutDownThread()); shutDownThread.setName("AsyncDispatcher ShutDown handler"); shutDownThread.start(); } } } {code} Above code shows AsyncDispatcher.dispatch method can process different type events. In fact this AsyncDispatcher instance is just ResourceManager.rmDispatcher created in ResourceManager.serviceInit method. You can find many eventTypes and handlers are registered in ResourceManager.rmDispatcher. In above scenario thread B blocks thread A, then many following events processing are blocked. In my testing cluster, there is only one queue and the client submits 1000 applications concurrently, the yarn cluster usage rate is 50. Many apps are pending. If I disable resourceManager recover functionality, the cluster usage can be 100. To solve this issue, I removed synchronized modifier on some methods defined in RMStateStore. Instead, in these methods I defined a dedicated lock object before calling dispatcher.getEventHandler().handle. In this way, the yarn cluster usage rate can be 100 and the whole cluster is good running. Please see my attached patch.

13. **summary:** Yarn recover functionality causes the cluster running slowly and the cluster usage rate is far below 100
description: In my hadoop cluster, the resourceManager recover functionality is enabled with FileSystemRMStateStore. I found this cause the yarn cluster running slowly and cluster usage rate is just 50 even there are many pending Apps. The scenario is below. In thread A, the RMAppImpl$RMAppNewlySavingTransition is calling storeNewApplication method defined in RMStateStore. This storeNewApplication method is synchronized. {code:title=RMAppImpl.java|borderStyle=solid} private static final class RMAppNewlySavingTransition extends RMAppTransition { @Override public void transition(RMAppImpl app, RMAppEvent event) { // If recovery is enabled then store the application information in a // non-blocking call so make sure that RM has stored the information // needed to restart the AM after RM restart without further client // communication LOG.info("Storing application with id " + app.applicationId);

app.rmContext.getStateStore().storeNewApplication(app); } } {code} {code:title=RMStateStore.java|borderStyle=solid} public synchronized void storeNewApplication(RMApp app) { ApplicationSubmissionContext context = app .getApplicationSubmissionContext(); assert context instanceof ApplicationSubmissionContextPBImpl; ApplicationStateData appState = ApplicationStateData.newInstance(app.getSubmitTime(), app.getStartTime(), context, app.getUser()); dispatcher.getEventHandler().handle(new RMStateStoreAppEvent(appState)); } {code} In thread B, the FileSystemRMStateStore is calling storeApplicationStateInternal method. It's also synchronized. This storeApplicationStateInternal method saves an ApplicationStateData into HDFS and it normally costs 90~300 milliseconds in my hadoop cluster. {code:title=FileSystemRMStateStore.java|borderStyle=solid} public synchronized void storeApplicationStateInternal(ApplicationId appId, ApplicationStateData appStateDataPB) throws Exception { Path appDirPath = getAppDir(rmAppRoot, appId); mkdirsWithRetries(appDirPath); Path nodeCreatePath = getNodePath(appDirPath, appId.toString()); LOG.info("Storing info for app: " + appId + " at: " + nodeCreatePath); byte[] appStateData = appStateDataPB.getProto().toByteArray(); try { // currently throw all exceptions. May need to respond differently for HA // based on whether we have lost the right to write to FS writeFileWithRetries(nodeCreatePath, appStateData, true); } catch (Exception e) { LOG.info("Error storing info for app: " + appId, e); throw e; } } {code} Think thread B firstly comes into FileSystemRMStateStore.storeApplicationStateInternal method, then thread A will be blocked for a while because of synchronization. In ResourceManager there is only one RMStateStore instance. In my cluster it's FileSystemRMStateStore type. Debug the RMAppNewlySavingTransition.transition method, the thread stack shows it's called form AsyncDispatcher.dispatch method. This method code is as below. {code:title=AsyncDispatcher.java|borderStyle=solid} protected void dispatch(Event event) { //all events go thru this loop if (LOG.isDebugEnabled()) { LOG.debug("Dispatching the event " + event.getClass().getName() + "." + event.toString()); } Class<? extends Enum> type = event.getType().getDeclaringClass(); try{ EventHandler handler = eventDispatchers.get(type); if(handler != null) { handler.handle(event); } else { throw new Exception("No handler for registered for " + type); } } catch (Throwable t) { //TODO Maybe log the state of the queue LOG.fatal("Error in dispatcher thread", t); // If serviceStop is called, we should exit this thread gracefully. if (exitOnDispatchException && (ShutdownHookManager.get().isShutdownInProgress()) == false && stopped == false) { Thread shutDownThread = new Thread(createShutDownThread()); shutDownThread.setName("AsyncDispatcher ShutDown handler"); shutDownThread.start(); } } } {code} Above code shows AsyncDispatcher.dispatch method can process different type events. In fact this AsyncDispatcher instance is just ResourceManager.rmDispatcher created in ResourceManager.serviceInit method. You can find many eventTypes and handlers are registered in ResourceManager.rmDispatcher. In above scenario thread B blocks thread A, then many following events processing are blocked. In my testing cluster, there is only one queue and the client submits 1000 applications concurrently, the yarn cluster usage rate is 50. Many apps are pending. If I disable resourceManager recover functionality, the cluster usage can be 100. To solve this issue, I removed synchronized modifier on some methods defined in RMStateStore. Instead, in these methods I defined a dedicated lock object before calling dispatcher.getEventHandler().handle. In this way, the yarn cluster usage rate can be 100 and the whole cluster is good running. Please see my attached patch.

14. **summary:** Yarn recover functionality causes the cluster running slowly and the cluster usage rate is far below 100
description: In my hadoop cluster, the resourceManager recover functionality is enabled with FileSystemRMStateStore. I found this cause the yarn cluster running slowly and cluster usage rate is just 50 even there are many pending Apps. The scenario is below. In thread A, the RMAppImpl$RMAppNewlySavingTransition is calling storeNewApplication method defined in RMStateStore. This storeNewApplication method is synchronized. {code:title=RMAppImpl.java|borderStyle=solid} private static final class RMAppNewlySavingTransition extends RMAppTransition { @Override public void transition(RMAppImpl app, RMAppEvent event) { // If recovery is enabled then store the application information in a // non-blocking call so make sure that RM has stored the information // needed to restart the AM after RM restart without further client // communication LOG.info("Storing application with id " + app.applicationId); app.rmContext.getStateStore().storeNewApplication(app); } } {code} {code:title=RMStateStore.java|borderStyle=solid} public synchronized void storeNewApplication(RMApp app) { ApplicationSubmissionContext context = app .getApplicationSubmissionContext(); assert context instanceof ApplicationSubmissionContextPBImpl; ApplicationStateData appState = ApplicationStateData.newInstance(app.getSubmitTime(), app.getStartTime(), context, app.getUser()); dispatcher.getEventHandler().handle(new RMStateStoreAppEvent(appState)); } {code} In thread B, the FileSystemRMStateStore is calling storeApplicationStateInternal method. It's also synchronized. This storeApplicationStateInternal method saves an ApplicationStateData into HDFS and it normally costs

90~300 milliseconds in my hadoop cluster. {code:title=FileSystemRMStateStore.java|borderStyle=solid} public synchronized void storeApplicationStateInternal(ApplicationId appId, ApplicationStateData appStateDataPB) throws Exception { Path appDirPath = getAppDir(rmAppRoot, appId); mkdirsWithRetries(appDirPath); Path nodeCreatePath = getNodePath(appDirPath, appId.toString()); LOG.info("Storing info for app: " + appId + " at: " + nodeCreatePath); byte[] appStateData = appStateDataPB.getProto().toByteArray(); try { // currently throw all exceptions. May need to respond differently for HA // based on whether we have lost the right to write to FS writeFileWithRetries(nodeCreatePath, appStateData, true); } catch (Exception e) { LOG.info("Error storing info for app: " + appId, e); throw e; } } {code} Think thread B firstly comes into FileSystemRMStateStore.storeApplicationStateInternal method, then thread A will be blocked for a while because of synchronization. In ResourceManager there is only one RMStateStore instance. In my cluster it's FileSystemRMStateStore type. Debug the RMAppNewlySavingTransition.transition method, the thread stack shows it's called form AsyncDispatcher.dispatch method. This method code is as below. {code:title=AsyncDispatcher.java|borderStyle=solid} protected void dispatch(Event event) { //all events go thru this loop if (LOG.isDebugEnabled()) { LOG.debug("Dispatching the event " + event.getClass().getName() + "." + event.toString()); } Class<? extends Enum> type = event.getType().getDeclaringClass(); try{ EventHandler handler = eventDispatchers.get(type); if(handler != null) { handler.handle(event); } else { throw new Exception("No handler for registered for " + type); } } catch (Throwable t) { //TODO Maybe log the state of the queue LOG.fatal("Error in dispatcher thread", t); // If serviceStop is called, we should exit this thread gracefully. if (exitOnDispatchException && (ShutdownHookManager.get().isShutdownInProgress()) == false && stopped == false) { Thread shutDownThread = new Thread(createShutDownThread()); shutDownThread.setName("AsyncDispatcher ShutDown handler"); shutDownThread.start(); } } } {code} Above code shows AsyncDispatcher.dispatch method can process different type events. In fact this AsyncDispatcher instance is just ResourceManager.rmDispatcher created in ResourceManager.serviceInit method. You can find many eventTypes and handlers are registered in ResourceManager.rmDispatcher. In above scenario thread B blocks thread A, then many following events processing are blocked. In my testing cluster, there is only one queue and the client submits 1000 applications concurrently, the yarn cluster usage rate is 50. Many apps are pending. If I disable resourceManager recover functionality, the cluster usage can be 100. To solve this issue, I removed synchronized modifier on some methods defined in RMStateStore. Instead, in these methods I defined a dedicated lock object before calling dispatcher.getEventHandler().handle. In this way, the yarn cluster usage rate can be 100 and the whole cluster is good running. Please see my attached patch.

15. **summary:** Yarn recover functionality causes the cluster running slowly and the cluster usage rate is far below 100
description: In my hadoop cluster, the resourceManager recover functionality is enabled with FileSystemRMStateStore. I found this cause the yarn cluster running slowly and cluster usage rate is just 50 even there are many pending Apps. The scenario is below. In thread A, the RMAppImpl$RMAppNewlySavingTransition is calling storeNewApplication method defined in RMStateStore. This storeNewApplication method is synchronized. {code:title=RMAppImpl.java|borderStyle=solid} private static final class RMAppNewlySavingTransition extends RMAppTransition { @Override public void transition(RMAppImpl app, RMAppEvent event) { // If recovery is enabled then store the application information in a // non-blocking call so make sure that RM has stored the information // needed to restart the AM after RM restart without further client // communication LOG.info("Storing application with id " + app.applicationId); app.rmContext.getStateStore().storeNewApplication(app); } } {code} {code:title=RMStateStore.java|borderStyle=solid} public synchronized void storeNewApplication(RMApp app) { ApplicationSubmissionContext context = app .getApplicationSubmissionContext(); assert context instanceof ApplicationSubmissionContextPBImpl; ApplicationStateData appState = ApplicationStateData.newInstance(app.getSubmitTime(), app.getStartTime(), context, app.getUser()); dispatcher.getEventHandler().handle(new RMStateStoreAppEvent(appState)); } {code} In thread B, the FileSystemRMStateStore is calling storeApplicationStateInternal method. It's also synchronized. This storeApplicationStateInternal method saves an ApplicationStateData into HDFS and it normally costs 90~300 milliseconds in my hadoop cluster. {code:title=FileSystemRMStateStore.java|borderStyle=solid} public synchronized void storeApplicationStateInternal(ApplicationId appId, ApplicationStateData appStateDataPB) throws Exception { Path appDirPath = getAppDir(rmAppRoot, appId); mkdirsWithRetries(appDirPath); Path nodeCreatePath = getNodePath(appDirPath, appId.toString()); LOG.info("Storing info for app: " + appId + " at: " + nodeCreatePath); byte[] appStateData = appStateDataPB.getProto().toByteArray(); try { // currently throw all exceptions. May need to respond differently for HA // based on whether we have lost the right to write to FS writeFileWithRetries(nodeCreatePath, appStateData, true); } catch (Exception e) { LOG.info("Error storing

info for app: " + appId, e); throw e; } } {code} Think thread B firstly comes into FileSystemRMStateStore.storeApplicationStateInternal method, then thread A will be blocked for a while because of synchronization. In ResourceManager there is only one RMStateStore instance. In my cluster it's FileSystemRMStateStore type. Debug the RMAppNewlySavingTransition.transition method, the thread stack shows it's called form AsyncDispatcher.dispatch method. This method code is as below. {code:title=AsyncDispatcher.java|borderStyle=solid} protected void dispatch(Event event) { //all events go thru this loop if (LOG.isDebugEnabled()) { LOG.debug("Dispatching the event " + event.getClass().getName() + "." + event.toString()); } Class<? extends Enum> type = event.getType().getDeclaringClass(); try{ EventHandler handler = eventDispatchers.get(type); if(handler != null) { handler.handle(event); } else { throw new Exception("No handler for registered for " + type); } } catch (Throwable t) { //TODO Maybe log the state of the queue LOG.fatal("Error in dispatcher thread", t); // If serviceStop is called, we should exit this thread gracefully. if (exitOnDispatchException && (ShutdownHookManager.get().isShutdownInProgress()) == false && stopped == false) { Thread shutDownThread = new Thread(createShutDownThread()); shutDownThread.setName("AsyncDispatcher ShutDown handler"); shutDownThread.start(); } } } {code} Above code shows AsyncDispatcher.dispatch method can process different type events. In fact this AsyncDispatcher instance is just ResourceManager.rmDispatcher created in ResourceManager.serviceInit method. You can find many eventTypes and handlers are registered in ResourceManager.rmDispatcher. In above scenario thread B blocks thread A, then many following events processing are blocked. In my testing cluster, there is only one queue and the client submits 1000 applications concurrently, the yarn cluster usage rate is 50. Many apps are pending. If I disable resourceManager recover functionality, the cluster usage can be 100. To solve this issue, I removed synchronized modifier on some methods defined in RMStateStore. Instead, in these methods I defined a dedicated lock object before calling dispatcher.getEventHandler().handle. In this way, the yarn cluster usage rate can be 100 and the whole cluster is good running. Please see my attached patch.

16. **summary:** Yarn recover functionality causes the cluster running slowly and the cluster usage rate is far below 100
description: In my hadoop cluster, the resourceManager recover functionality is enabled with FileSystemRMStateStore. I found this cause the yarn cluster running slowly and cluster usage rate is just 50 even there are many pending Apps. The scenario is below. In thread A, the RMAppImpl$RMAppNewlySavingTransition is calling storeNewApplication method defined in RMStateStore. This storeNewApplication method is synchronized. {code:title=RMAppImpl.java|borderStyle=solid} private static final class RMAppNewlySavingTransition extends RMAppTransition { @Override public void transition(RMAppImpl app, RMAppEvent event) { // If recovery is enabled then store the application information in a // non-blocking call so make sure that RM has stored the information // needed to restart the AM after RM restart without further client // communication LOG.info("Storing application with id " + app.applicationId); app.rmContext.getStateStore().storeNewApplication(app); } } {code} {code:title=RMStateStore.java|borderStyle=solid} public synchronized void storeNewApplication(RMApp app) { ApplicationSubmissionContext context = app .getApplicationSubmissionContext(); assert context instanceof ApplicationSubmissionContextPBImpl; ApplicationStateData appState = ApplicationStateData.newInstance(app.getSubmitTime(), app.getStartTime(), context, app.getUser()); dispatcher.getEventHandler().handle(new RMStateStoreAppEvent(appState)); } {code} In thread B, the FileSystemRMStateStore is calling storeApplicationStateInternal method. It's also synchronized. This storeApplicationStateInternal method saves an ApplicationStateData into HDFS and it normally costs 90~300 milliseconds in my hadoop cluster. {code:title=FileSystemRMStateStore.java|borderStyle=solid} public synchronized void storeApplicationStateInternal(ApplicationId appId, ApplicationStateData appStateDataPB) throws Exception { Path appDirPath = getAppDir(rmAppRoot, appId); mkdirsWithRetries(appDirPath); Path nodeCreatePath = getNodePath(appDirPath, appId.toString()); LOG.info("Storing info for app: " + appId + " at: " + nodeCreatePath); byte[] appStateData = appStateDataPB.getProto().toByteArray(); try { // currently throw all exceptions. May need to respond differently for HA // based on whether we have lost the right to write to FS writeFileWithRetries(nodeCreatePath, appStateData, true); } catch (Exception e) { LOG.info("Error storing info for app: " + appId, e); throw e; } } {code} Think thread B firstly comes into FileSystemRMStateStore.storeApplicationStateInternal method, then thread A will be blocked for a while because of synchronization. In ResourceManager there is only one RMStateStore instance. In my cluster it's FileSystemRMStateStore type. Debug the RMAppNewlySavingTransition.transition method, the thread stack shows it's called form AsyncDispatcher.dispatch method. This method code is as below. {code:title=AsyncDispatcher.java|borderStyle=solid} protected void dispatch(Event event) { //all events go thru this loop if (LOG.isDebugEnabled()) { LOG.debug("Dispatching the event " + event.getClass().getName() + "." + event.toString()); } Class<? extends Enum> type =

event.getType().getDeclaringClass(); try{ EventHandler handler = eventDispatchers.get(type); if(handler != null) { handler.handle(event); } else { throw new Exception("No handler for registered for " + type); } } catch (Throwable t) { //TODO Maybe log the state of the queue LOG.fatal("Error in dispatcher thread", t); // If serviceStop is called, we should exit this thread gracefully. if (exitOnDispatchException && (ShutdownHookManager.get().isShutdownInProgress()) == false && stopped == false) { Thread shutDownThread = new Thread(createShutDownThread()); shutDownThread.setName("AsyncDispatcher ShutDown handler"); shutDownThread.start(); } } } {code} Above code shows AsyncDispatcher.dispatch method can process different type events. In fact this AsyncDispatcher instance is just ResourceManager.rmDispatcher created in ResourceManager.serviceInit method. You can find many eventTypes and handlers are registered in ResourceManager.rmDispatcher. In above scenario thread B blocks thread A, then many following events processing are blocked. In my testing cluster, there is only one queue and the client submits 1000 applications concurrently, the yarn cluster usage rate is 50. Many apps are pending. If I disable resourceManager recover functionality, the cluster usage can be 100. To solve this issue, I removed synchronized modifier on some methods defined in RMStateStore. Instead, in these methods I defined a dedicated lock object before calling dispatcher.getEventHandler().handle. In this way, the yarn cluster usage rate can be 100 and the whole cluster is good running. Please see my attached patch.

17. **summary:** Yarn recover functionality causes the cluster running slowly and the cluster usage rate is far below 100
description: In my hadoop cluster, the resourceManager recover functionality is enabled with FileSystemRMStateStore. I found this cause the yarn cluster running slowly and cluster usage rate is just 50 even there are many pending Apps. The scenario is below. In thread A, the RMAppImpl$RMAppNewlySavingTransition is calling storeNewApplication method defined in RMStateStore. This storeNewApplication method is synchronized. {code:title=RMAppImpl.java|borderStyle=solid} private static final class RMAppNewlySavingTransition extends RMAppTransition { @Override public void transition(RMAppImpl app, RMAppEvent event) { // If recovery is enabled then store the application information in a // non-blocking call so make sure that RM has stored the information // needed to restart the AM after RM restart without further client // communication LOG.info("Storing application with id " + app.applicationId); app.rmContext.getStateStore().storeNewApplication(app); } } {code} {code:title=RMStateStore.java|borderStyle=solid} public synchronized void storeNewApplication(RMApp app) { ApplicationSubmissionContext context = app .getApplicationSubmissionContext(); assert context instanceof ApplicationSubmissionContextPBImpl; ApplicationStateData appState = ApplicationStateData.newInstance(app.getSubmitTime(), app.getStartTime(), context, app.getUser()); dispatcher.getEventHandler().handle(new RMStateStoreAppEvent(appState)); } {code} In thread B, the FileSystemRMStateStore is calling storeApplicationStateInternal method. It's also synchronized. This storeApplicationStateInternal method saves an ApplicationStateData into HDFS and it normally costs 90~300 milliseconds in my hadoop cluster. {code:title=FileSystemRMStateStore.java|borderStyle=solid} public synchronized void storeApplicationStateInternal(ApplicationId appId, ApplicationStateData appStateDataPB) throws Exception { Path appDirPath = getAppDir(rmAppRoot, appId); mkdirsWithRetries(appDirPath); Path nodeCreatePath = getNodePath(appDirPath, appId.toString()); LOG.info("Storing info for app: " + appId + " at: " + nodeCreatePath); byte[] appStateData = appStateDataPB.getProto().toByteArray(); try { // currently throw all exceptions. May need to respond differently for HA // based on whether we have lost the right to write to FS writeFileWithRetries(nodeCreatePath, appStateData, true); } catch (Exception e) { LOG.info("Error storing info for app: " + appId, e); throw e; } } {code} Think thread B firstly comes into FileSystemRMStateStore.storeApplicationStateInternal method, then thread A will be blocked for a while because of synchronization. In ResourceManager there is only one RMStateStore instance. In my cluster it's FileSystemRMStateStore type. Debug the RMAppNewlySavingTransition.transition method, the thread stack shows it's called form AsyncDispatcher.dispatch method. This method code is as below. {code:title=AsyncDispatcher.java|borderStyle=solid} protected void dispatch(Event event) { //all events go thru this loop if (LOG.isDebugEnabled()) { LOG.debug("Dispatching the event " + event.getClass().getName() + "." + event.toString()); } Class<? extends Enum> type = event.getType().getDeclaringClass(); try{ EventHandler handler = eventDispatchers.get(type); if(handler != null) { handler.handle(event); } else { throw new Exception("No handler for registered for " + type); } } catch (Throwable t) { //TODO Maybe log the state of the queue LOG.fatal("Error in dispatcher thread", t); // If serviceStop is called, we should exit this thread gracefully. if (exitOnDispatchException && (ShutdownHookManager.get().isShutdownInProgress()) == false && stopped == false) { Thread shutDownThread = new Thread(createShutDownThread()); shutDownThread.setName("AsyncDispatcher ShutDown handler"); shutDownThread.start(); } } } {code} Above code shows AsyncDispatcher.dispatch method can process different type events. In fact this AsyncDispatcher instance is just

ResourceManager.rmDispatcher created in ResourceManager.serviceInit method. You can find many eventTypes and handlers are registered in ResourceManager.rmDispatcher. In above scenario thread B blocks thread A, then many following events processing are blocked. In my testing cluster, there is only one queue and the client submits 1000 applications concurrently, the yarn cluster usage rate is 50. Many apps are pending. If I disable resourceManager recover functionality, the cluster usage can be 100. To solve this issue, I removed synchronized modifier on some methods defined in RMStateStore. Instead, in these methods I defined a dedicated lock object before calling dispatcher.getEventHandler().handle. In this way, the yarn cluster usage rate can be 100 and the whole cluster is good running. Please see my attached patch.

18. **summary:** Yarn recover functionality causes the cluster running slowly and the cluster usage rate is far below 100
description: In my hadoop cluster, the resourceManager recover functionality is enabled with FileSystemRMStateStore. I found this cause the yarn cluster running slowly and cluster usage rate is just 50 even there are many pending Apps. The scenario is below. In thread A, the RMAppImpl$RMAppNewlySavingTransition is calling storeNewApplication method defined in RMStateStore. This storeNewApplication method is synchronized. {code:title=RMAppImpl.java|borderStyle=solid} private static final class RMAppNewlySavingTransition extends RMAppTransition { @Override public void transition(RMAppImpl app, RMAppEvent event) { // If recovery is enabled then store the application information in a // non-blocking call so make sure that RM has stored the information // needed to restart the AM after RM restart without further client // communication LOG.info("Storing application with id " + app.applicationId); app.rmContext.getStateStore().storeNewApplication(app); } } {code} {code:title=RMStateStore.java|borderStyle=solid} public synchronized void storeNewApplication(RMApp app) { ApplicationSubmissionContext context = app .getApplicationSubmissionContext(); assert context instanceof ApplicationSubmissionContextPBImpl; ApplicationStateData appState = ApplicationStateData.newInstance(app.getSubmitTime(), app.getStartTime(), context, app.getUser()); dispatcher.getEventHandler().handle(new RMStateStoreAppEvent(appState)); } {code} In thread B, the FileSystemRMStateStore is calling storeApplicationStateInternal method. It's also synchronized. This storeApplicationStateInternal method saves an ApplicationStateData into HDFS and it normally costs 90~300 milliseconds in my hadoop cluster. {code:title=FileSystemRMStateStore.java|borderStyle=solid} public synchronized void storeApplicationStateInternal(ApplicationId appId, ApplicationStateData appStateDataPB) throws Exception { Path appDirPath = getAppDir(rmAppRoot, appId); mkdirsWithRetries(appDirPath); Path nodeCreatePath = getNodePath(appDirPath, appId.toString()); LOG.info("Storing info for app: " + appId + " at: " + nodeCreatePath); byte[] appStateData = appStateDataPB.getProto().toByteArray(); try { // currently throw all exceptions. May need to respond differently for HA // based on whether we have lost the right to write to FS writeFileWithRetries(nodeCreatePath, appStateData, true); } catch (Exception e) { LOG.info("Error storing info for app: " + appId, e); throw e; } } {code} Think thread B firstly comes into FileSystemRMStateStore.storeApplicationStateInternal method, then thread A will be blocked for a while because of synchronization. In ResourceManager there is only one RMStateStore instance. In my cluster it's FileSystemRMStateStore type. Debug the RMAppNewlySavingTransition.transition method, the thread stack shows it's called form AsyncDispatcher.dispatch method. This method code is as below. {code:title=AsyncDispatcher.java|borderStyle=solid} protected void dispatch(Event event) { //all events go thru this loop if (LOG.isDebugEnabled()) { LOG.debug("Dispatching the event " + event.getClass().getName() + "." + event.toString()); } Class<? extends Enum> type = event.getType().getDeclaringClass(); try{ EventHandler handler = eventDispatchers.get(type); if(handler != null) { handler.handle(event); } else { throw new Exception("No handler for registered for " + type); } } catch (Throwable t) { //TODO Maybe log the state of the queue LOG.fatal("Error in dispatcher thread", t); // If serviceStop is called, we should exit this thread gracefully. if (exitOnDispatchException && (ShutdownHookManager.get().isShutdownInProgress()) == false && stopped == false) { Thread shutDownThread = new Thread(createShutDownThread()); shutDownThread.setName("AsyncDispatcher ShutDown handler"); shutDownThread.start(); } } } {code} Above code shows AsyncDispatcher.dispatch method can process different type events. In fact this AsyncDispatcher instance is just ResourceManager.rmDispatcher created in ResourceManager.serviceInit method. You can find many eventTypes and handlers are registered in ResourceManager.rmDispatcher. In above scenario thread B blocks thread A, then many following events processing are blocked. In my testing cluster, there is only one queue and the client submits 1000 applications concurrently, the yarn cluster usage rate is 50. Many apps are pending. If I disable resourceManager recover functionality, the cluster usage can be 100. To solve this issue, I removed synchronized modifier on some methods defined in RMStateStore. Instead, in these methods I defined a dedicated lock object before calling dispatcher.getEventHandler().handle. In this way, the yarn cluster usage rate can be 100 and the whole cluster is good running. Please see my attached patch.

19. **summary:** Yarn recover functionality causes the cluster running slowly and the cluster usage rate is far below 100
description: In my hadoop cluster, the resourceManager recover functionality is enabled with FileSystemRMStateStore. I found this cause the yarn cluster running slowly and cluster usage rate is just 50 even there are many pending Apps. The scenario is below. In thread A, the RMAppImpl$RMAppNewlySavingTransition is calling storeNewApplication method defined in RMStateStore. This storeNewApplication method is synchronized. {code:title=RMAppImpl.java|borderStyle=solid} private static final class RMAppNewlySavingTransition extends RMAppTransition { @Override public void transition(RMAppImpl app, RMAppEvent event) { // If recovery is enabled then store the application information in a // non-blocking call so make sure that RM has stored the information // needed to restart the AM after RM restart without further client // communication LOG.info("Storing application with id " + app.applicationId); app.rmContext.getStateStore().storeNewApplication(app); } } {code} {code:title=RMStateStore.java|borderStyle=solid} public synchronized void storeNewApplication(RMApp app) { ApplicationSubmissionContext context = app .getApplicationSubmissionContext(); assert context instanceof ApplicationSubmissionContextPBImpl; ApplicationStateData appState = ApplicationStateData.newInstance(app.getSubmitTime(), app.getStartTime(), context, app.getUser()); dispatcher.getEventHandler().handle(new RMStateStoreAppEvent(appState)); } {code} In thread B, the FileSystemRMStateStore is calling storeApplicationStateInternal method. It's also synchronized. This storeApplicationStateInternal method saves an ApplicationStateData into HDFS and it normally costs 90~300 milliseconds in my hadoop cluster. {code:title=FileSystemRMStateStore.java|borderStyle=solid} public synchronized void storeApplicationStateInternal(ApplicationId appId, ApplicationStateData appStateDataPB) throws Exception { Path appDirPath = getAppDir(rmAppRoot, appId); mkdirsWithRetries(appDirPath); Path nodeCreatePath = getNodePath(appDirPath, appId.toString()); LOG.info("Storing info for app: " + appId + " at: " + nodeCreatePath); byte[] appStateData = appStateDataPB.getProto().toByteArray(); try { // currently throw all exceptions. May need to respond differently for HA // based on whether we have lost the right to write to FS writeFileWithRetries(nodeCreatePath, appStateData, true); } catch (Exception e) { LOG.info("Error storing info for app: " + appId, e); throw e; } } {code} Think thread B firstly comes into FileSystemRMStateStore.storeApplicationStateInternal method, then thread A will be blocked for a while because of synchronization. In ResourceManager there is only one RMStateStore instance. In my cluster it's FileSystemRMStateStore type. Debug the RMAppNewlySavingTransition.transition method, the thread stack shows it's called form AsyncDispatcher.dispatch method. This method code is as below. {code:title=AsyncDispatcher.java|borderStyle=solid} protected void dispatch(Event event) { //all events go thru this loop if (LOG.isDebugEnabled()) { LOG.debug("Dispatching the event " + event.getClass().getName() + "." + event.toString()); } Class<? extends Enum> type = event.getType().getDeclaringClass(); try{ EventHandler handler = eventDispatchers.get(type); if(handler != null) { handler.handle(event); } else { throw new Exception("No handler for registered for " + type); } } catch (Throwable t) { //TODO Maybe log the state of the queue LOG.fatal("Error in dispatcher thread", t); // If serviceStop is called, we should exit this thread gracefully. if (exitOnDispatchException && (ShutdownHookManager.get().isShutdownInProgress()) == false && stopped == false) { Thread shutDownThread = new Thread(createShutDownThread()); shutDownThread.setName("AsyncDispatcher ShutDown handler"); shutDownThread.start(); } } } {code} Above code shows AsyncDispatcher.dispatch method can process different type events. In fact this AsyncDispatcher instance is just ResourceManager.rmDispatcher created in ResourceManager.serviceInit method. You can find many eventTypes and handlers are registered in ResourceManager.rmDispatcher. In above scenario thread B blocks thread A, then many following events processing are blocked. In my testing cluster, there is only one queue and the client submits 1000 applications concurrently, the yarn cluster usage rate is 50. Many apps are pending. If I disable resourceManager recover functionality, the cluster usage can be 100. To solve this issue, I removed synchronized modifier on some methods defined in RMStateStore. Instead, in these methods I defined a dedicated lock object before calling dispatcher.getEventHandler().handle. In this way, the yarn cluster usage rate can be 100 and the whole cluster is good running. Please see my attached patch.
label: code-design
20. **summary:** Yarn recover functionality causes the cluster running slowly and the cluster usage rate is far below 100
description: In my hadoop cluster, the resourceManager recover functionality is enabled with FileSystemRMStateStore. I found this cause the yarn cluster running slowly and cluster usage rate is just 50 even there are many pending Apps. The scenario is below. In thread A, the RMAppImpl$RMAppNewlySavingTransition is calling storeNewApplication method defined in RMStateStore. This storeNewApplication method is synchronized.

{code:title=RMAppImpl.java|borderStyle=solid} private static final class RMAppNewlySavingTransition extends RMAppTransition { @Override public void transition(RMAppImpl app, RMAppEvent event) { // If recovery is enabled then store the application information in a // non-blocking call so make sure that RM has stored the information // needed to restart the AM after RM restart without further client // communication LOG.info("Storing application with id " + app.applicationId); app.rmContext.getStateStore().storeNewApplication(app); } } {code} {code:title=RMStateStore.java|borderStyle=solid} public synchronized void storeNewApplication(RMApp app) { ApplicationSubmissionContext context = app .getApplicationSubmissionContext(); assert context instanceof ApplicationSubmissionContextPBImpl; ApplicationStateData appState = ApplicationStateData.newInstance(app.getSubmitTime(), app.getStartTime(), context, app.getUser()); dispatcher.getEventHandler().handle(new RMStateStoreAppEvent(appState)); } {code} In thread B, the FileSystemRMStateStore is calling storeApplicationStateInternal method. It's also synchronized. This storeApplicationStateInternal method saves an ApplicationStateData into HDFS and it normally costs 90~300 milliseconds in my hadoop cluster. {code:title=FileSystemRMStateStore.java|borderStyle=solid} public synchronized void storeApplicationStateInternal(ApplicationId appId, ApplicationStateData appStateDataPB) throws Exception { Path appDirPath = getAppDir(rmAppRoot, appId); mkdirsWithRetries(appDirPath); Path nodeCreatePath = getNodePath(appDirPath, appId.toString()); LOG.info("Storing info for app: " + appId + " at: " + nodeCreatePath); byte[] appStateData = appStateDataPB.getProto().toByteArray(); try { // currently throw all exceptions. May need to respond differently for HA // based on whether we have lost the right to write to FS writeFileWithRetries(nodeCreatePath, appStateData, true); } catch (Exception e) { LOG.info("Error storing info for app: " + appId, e); throw e; } } {code} Think thread B firstly comes into FileSystemRMStateStore.storeApplicationStateInternal method, then thread A will be blocked for a while because of synchronization. In ResourceManager there is only one RMStateStore instance. In my cluster it's FileSystemRMStateStore type. Debug the RMAppNewlySavingTransition.transition method, the thread stack shows it's called form AsyncDispatcher.dispatch method. This method code is as below. {code:title=AsyncDispatcher.java|borderStyle=solid} protected void dispatch(Event event) { //all events go thru this loop if (LOG.isDebugEnabled()) { LOG.debug("Dispatching the event " + event.getClass().getName() + "." + event.toString()); } Class<? extends Enum> type = event.getType().getDeclaringClass(); try{ EventHandler handler = eventDispatchers.get(type); if(handler != null) { handler.handle(event); } else { throw new Exception("No handler for registered for " + type); } } catch (Throwable t) { //TODO Maybe log the state of the queue LOG.fatal("Error in dispatcher thread", t); // If serviceStop is called, we should exit this thread gracefully. if (exitOnDispatchException && (ShutdownHookManager.get().isShutdownInProgress()) == false && stopped == false) { Thread shutDownThread = new Thread(createShutDownThread()); shutDownThread.setName("AsyncDispatcher ShutDown handler"); shutDownThread.start(); } } } {code} Above code shows AsyncDispatcher.dispatch method can process different type events. In fact this AsyncDispatcher instance is just ResourceManager.rmDispatcher created in ResourceManager.serviceInit method. You can find many eventTypes and handlers are registered in ResourceManager.rmDispatcher. In above scenario thread B blocks thread A, then many following events processing are blocked. In my testing cluster, there is only one queue and the client submits 1000 applications concurrently, the yarn cluster usage rate is 50. Many apps are pending. If I disable resourceManager recover functionality, the cluster usage can be 100. To solve this issue, I removed synchronized modifier on some methods defined in RMStateStore. Instead, in these methods I defined a dedicated lock object before calling dispatcher.getEventHandler().handle. In this way, the yarn cluster usage rate can be 100 and the whole cluster is good running. Please see my attached patch.

jira_issues_comments:

1. [~jianhe] would you kindly help to take a look on this issue?
2. **body:** The {{AsyncDispatcher.GenericEventHandler.handle()}} method is MT safe. The {{AsyncDispatcher.getEventHandler()}} is the unsafe call, and it's only unsafe because of the lazy initialization. Prior to YARN-1121, it was returning a new object every time, which was thread safe. I see two obvious options: revert the YARN-1121 optimization in the {{AsyncDispatcher.getEventHandler()}} method or do eager initialization into a final member variable. Either way, the calls become MT-safe, letting you just drop the synchronization.
   label: requirement
3. [~iceberg565], thanks for looking into this. analysis makes sense to me. I think we can just remove the synchronized keyword ? bq. the AsyncDispatcher.getEventHandler() is the unsafe call Suppose the call is unsafe, in the worst case when contention happens, separate new objects will return to each caller instead of one, which is equivalent to new object every time as before ?

4. **body:** [~jianhe], you are correct, but that approach just smells bad to me. It's behavior that someone will be confused by later. It would be better to do something intentional than something that accidentally works for a non-obvious reason.
   label: code-design
5. Thanks for all your comments. I prefer to do eager initialization handlerInstance in AsyncDispatcher, then remove synchronized modifier in RMStateStore. Pelese see my new patch.
6. patch looks good to me. thanks [~iceberg565], and [~templedf] for the review.
7. Yep. +1 (non-binding)
8. Actually, before I +1 that, [~iceberg565], did you run the full set of RM unit tests against the patch?
9. [~iceberg565], after you upload the patch, you can click the "Submit Patch" button, which will trigger jenkins to run the unit tests.
10. | (x) *{color:red}-1 overall{color}* | \\ \\ || Vote || Subsystem || Runtime || Comment || | {color:blue}0{color} | {color:blue} reexec {color} | {color:blue} 0m 0s {color} | {color:blue} Docker mode activated. {color} | | {color:green}+1{color} | {color:green} @author {color} | {color:green} 0m 0s {color} | {color:green} The patch does not contain any @author tags. {color} | | {color:red}-1{color} | {color:red} test4tests {color} | {color:red} 0m 0s {color} | {color:red} The patch doesn't appear to include any new or modified tests. Please justify why no new tests are needed for this patch. Also please list what manual steps were performed to verify this patch. {color} | | {color:green}+1{color} | {color:green} mvninstall {color} | {color:green} 9m 49s {color} | {color:green} trunk passed {color} | | {color:green}+1{color} | {color:green} compile {color} | {color:green} 3m 5s {color} | {color:green} trunk passed with JDK v1.8.0_66 {color} | | {color:green}+1{color} | {color:green} compile {color} | {color:green} 2m 49s {color} | {color:green} trunk passed with JDK v1.7.0_85 {color} | | {color:green}+1{color} | {color:green} checkstyle {color} | {color:green} 0m 35s {color} | {color:green} trunk passed {color} | | {color:green}+1{color} | {color:green} mvnsite {color} | {color:green} 1m 28s {color} | {color:green} trunk passed {color} | | {color:green}+1{color} | {color:green} mvneclipse {color} | {color:green} 0m 35s {color} | {color:green} trunk passed {color} | | {color:green}+1{color} | {color:green} findbugs {color} | {color:green} 3m 13s {color} | {color:green} trunk passed {color} | | {color:green}+1{color} | {color:green} javadoc {color} | {color:green} 1m 13s {color} | {color:green} trunk passed with JDK v1.8.0_66 {color} | | {color:green}+1{color} | {color:green} javadoc {color} | {color:green} 1m 17s {color} | {color:green} trunk passed with JDK v1.7.0_85 {color} | | {color:green}+1{color} | {color:green} mvninstall {color} | {color:green} 1m 23s {color} | {color:green} the patch passed {color} | | {color:green}+1{color} | {color:green} compile {color} | {color:green} 3m 4s {color} | {color:green} the patch passed with JDK v1.8.0_66 {color} | | {color:green}+1{color} | {color:green} javac {color} | {color:green} 3m 4s {color} | {color:green} the patch passed {color} | | {color:green}+1{color} | {color:green} compile {color} | {color:green} 2m 50s {color} | {color:green} the patch passed with JDK v1.7.0_85 {color} | | {color:green}+1{color} | {color:green} javac {color} | {color:green} 2m 50s {color} | {color:green} the patch passed {color} | | {color:green}+1{color} | {color:green} checkstyle {color} | {color:green} 0m 35s {color} | {color:green} the patch passed {color} | | {color:green}+1{color} | {color:green} mvnsite {color} | {color:green} 1m 27s {color} | {color:green} the patch passed {color} | | {color:green}+1{color} | {color:green} mvneclipse {color} | {color:green} 0m 35s {color} | {color:green} the patch passed {color} | | {color:red}-1{color} | {color:red} whitespace {color} | {color:red} 0m 0s {color} | {color:red} The patch has 1 line(s) that end in whitespace. Use git apply --whitespace=fix. {color} | | {color:green}+1{color} | {color:green} findbugs {color} | {color:green} 3m 33s {color} | {color:green} the patch passed {color} | | {color:green}+1{color} | {color:green} javadoc {color} | {color:green} 1m 14s {color} | {color:green} the patch passed with JDK v1.8.0_66 {color} | | {color:green}+1{color} | {color:green} javadoc {color} | {color:green} 1m 16s {color} | {color:green} the patch passed with JDK v1.7.0_85 {color} | | {color:green}+1{color} | {color:green} unit {color} | {color:green} 2m 35s {color} | {color:green} hadoop-yarn-common in the patch passed with JDK v1.8.0_66. {color} | | {color:red}-1{color} | {color:red} unit {color} | {color:red} 65m 28s {color} | {color:red} hadoop-yarn-server-resourcemanager in the patch failed with JDK v1.8.0_66. {color} | | {color:green}+1{color} | {color:green} unit {color} | {color:green} 2m 36s {color} | {color:green} hadoop-yarn-common in the patch passed with JDK v1.7.0_85. {color} | | {color:red}-1{color} | {color:red} unit {color} | {color:red} 65m 3s {color} | {color:red} hadoop-yarn-server-resourcemanager in the patch failed with JDK v1.7.0_85. {color} | | {color:green}+1{color} | {color:green} asflicense {color} | {color:green} 0m 28s {color} | {color:green} Patch does not generate ASF License warnings. {color} | | {color:black} {color} | {color:black} {color} | {color:black} 178m 3s {color} | {color:black} {color} | \\ \\ || Reason || Tests || | JDK v1.8.0_66 Failed junit tests | hadoop.yarn.server.resourcemanager.TestClientRMTokens | | | hadoop.yarn.server.resourcemanager.rmapp.TestRMAppTransitions | | | hadoop.yarn.server.resourcemanager.TestAMAuthorization | | JDK v1.7.0_85 Failed junit tests | hadoop.yarn.server.resourcemanager.TestClientRMTokens | | |

hadoop.yarn.server.resourcemanager.rmapp.TestRMAppTransitions | | | hadoop.yarn.server.resourcemanager.TestAMAuthorization | \\ \\ || Subsystem || Report/Notes || | Docker | Image:yetus/hadoop:0ca8df7 | | JIRA Patch URL | https://issues.apache.org/jira/secure/attachment/12774937/YARN-4398.3.patch | | JIRA Issue | YARN-4398 | | Optional Tests | asflicense compile javac javadoc mvninstall mvnsite unit findbugs checkstyle | | uname | Linux c0bfc366ed04 3.13.0-36-lowlatency #63-Ubuntu SMP PREEMPT Wed Sep 3 21:56:12 UTC 2014 x86_64 x86_64 x86_64 GNU/Linux | | Build tool | maven | | Personality | /testptch/hadoop/patchprocess/precommit/personality/provided.sh | | git revision | trunk / 830eb25 | | findbugs | v3.0.0 | | whitespace | https://builds.apache.org/job/PreCommit-YARN-Build/9829/artifact/patchprocess/whitespace-eol.txt | | unit | https://builds.apache.org/job/PreCommit-YARN-Build/9829/artifact/patchprocess/patch-unit-hadoop-yarn-project_hadoop-yarn_hadoop-yarn-server_hadoop-yarn-server-resourcemanager-jdk1.8.0_66.txt | | unit | https://builds.apache.org/job/PreCommit-YARN-Build/9829/artifact/patchprocess/patch-unit-hadoop-yarn-project_hadoop-yarn_hadoop-yarn-server_hadoop-yarn-server-resourcemanager-jdk1.7.0_85.txt | | unit test logs | https://builds.apache.org/job/PreCommit-YARN-Build/9829/artifact/patchprocess/patch-unit-hadoop-yarn-project_hadoop-yarn_hadoop-yarn-server_hadoop-yarn-server-resourcemanager-jdk1.8.0_66.txt https://builds.apache.org/job/PreCommit-YARN-Build/9829/artifact/patchprocess/patch-unit-hadoop-yarn-project_hadoop-yarn_hadoop-yarn-server_hadoop-yarn-server-resourcemanager-jdk1.7.0_85.txt | | JDK v1.7.0_85 Test Results | https://builds.apache.org/job/PreCommit-YARN-Build/9829/testReport/ | | modules | C: hadoop-yarn-project/hadoop-yarn/hadoop-yarn-common hadoop-yarn-project/hadoop-yarn/hadoop-yarn-server/hadoop-yarn-server-resourcemanager U: hadoop-yarn-project/hadoop-yarn | | Max memory used | 76MB | | Powered by | Apache Yetus http://yetus.apache.org | | Console output | https://builds.apache.org/job/PreCommit-YARN-Build/9829/console | This message was automatically generated.

11. **body:** I uploaded a new patch that removed useless whitespace. The current test cases can cover the modified code in this patch. This patch resolved performance issue. So no new unit test cases.
    label: code-design
12. | (x) *{color:red}-1 overall{color}* | \\ \\ || Vote || Subsystem || Runtime || Comment || | {color:blue}0{color} | {color:blue} reexec {color} | {color:blue} 0m 0s {color} | {color:blue} Docker mode activated. {color} | | {color:green}+1{color} | {color:green} @author {color} | {color:green} 0m 0s {color} | {color:green} The patch does not contain any @author tags. {color} | | {color:red}-1{color} | {color:red} test4tests {color} | {color:red} 0m 0s {color} | {color:red} The patch doesn't appear to include any new or modified tests. Please justify why no new tests are needed for this patch. Also please list what manual steps were performed to verify this patch. {color} | | {color:green}+1{color} | {color:green} mvninstall {color} | {color:green} 8m 9s {color} | {color:green} trunk passed {color} | | {color:green}+1{color} | {color:green} compile {color} | {color:green} 2m 10s {color} | {color:green} trunk passed with JDK v1.8.0_66 {color} | | {color:green}+1{color} | {color:green} compile {color} | {color:green} 2m 22s {color} | {color:green} trunk passed with JDK v1.7.0_85 {color} | | {color:green}+1{color} | {color:green} checkstyle {color} | {color:green} 0m 30s {color} | {color:green} trunk passed {color} | | {color:green}+1{color} | {color:green} mvnsite {color} | {color:green} 1m 20s {color} | {color:green} trunk passed {color} | | {color:green}+1{color} | {color:green} mvneclipse {color} | {color:green} 0m 29s {color} | {color:green} trunk passed {color} | | {color:green}+1{color} | {color:green} findbugs {color} | {color:green} 2m 46s {color} | {color:green} trunk passed {color} | | {color:green}+1{color} | {color:green} javadoc {color} | {color:green} 1m 16s {color} | {color:green} trunk passed with JDK v1.8.0_66 {color} | | {color:green}+1{color} | {color:green} javadoc {color} | {color:green} 1m 1s {color} | {color:green} trunk passed with JDK v1.7.0_85 {color} | | {color:green}+1{color} | {color:green} mvninstall {color} | {color:green} 1m 7s {color} | {color:green} the patch passed {color} | | {color:green}+1{color} | {color:green} compile {color} | {color:green} 1m 59s {color} | {color:green} the patch passed with JDK v1.8.0_66 {color} | | {color:green}+1{color} | {color:green} javac {color} | {color:green} 1m 59s {color} | {color:green} the patch passed {color} | | {color:green}+1{color} | {color:green} compile {color} | {color:green} 2m 14s {color} | {color:green} the patch passed with JDK v1.7.0_85 {color} | | {color:green}+1{color} | {color:green} javac {color} | {color:green} 2m 14s {color} | {color:green} the patch passed {color} | | {color:green}+1{color} | {color:green} checkstyle {color} | {color:green} 0m 28s {color} | {color:green} the patch passed {color} | | {color:green}+1{color} | {color:green} mvnsite {color} | {color:green} 1m 12s {color} | {color:green} the patch passed {color} | | {color:green}+1{color} | {color:green} mvneclipse {color} | {color:green} 0m 28s {color} | {color:green} the patch passed {color} | | {color:green}+1{color} | {color:green} whitespace {color} | {color:green} 0m 0s {color} | {color:green} Patch has no whitespace issues. {color} | | {color:green}+1{color} | {color:green} findbugs {color} | {color:green} 2m 55s {color} | {color:green} the patch passed {color} | | {color:green}+1{color} | {color:green} javadoc {color} | {color:green} 0m 56s

{color} | {color:green} the patch passed with JDK v1.8.0_66 {color} | | {color:green}+1{color} | {color:green} javadoc {color} | {color:green} 1m 6s {color} | {color:green} the patch passed with JDK v1.7.0_85 {color} | | {color:green}+1{color} | {color:green} unit {color} | {color:green} 2m 9s {color} | {color:green} hadoop-yarn-common in the patch passed with JDK v1.8.0_66. {color} | | {color:red}-1{color} | {color:red} unit {color} | {color:red} 64m 58s {color} | {color:red} hadoop-yarn-server-resourcemanager in the patch failed with JDK v1.8.0_66. {color} | | {color:green}+1{color} | {color:green} unit {color} | {color:green} 2m 13s {color} | {color:green} hadoop-yarn-common in the patch passed with JDK v1.7.0_85. {color} | | {color:red}-1{color} | {color:red} unit {color} | {color:red} 65m 29s {color} | {color:red} hadoop-yarn-server-resourcemanager in the patch failed with JDK v1.7.0_85. {color} | | {color:green}+1{color} | {color:green} asflicense {color} | {color:green} 0m 22s {color} | {color:green} Patch does not generate ASF License warnings. {color} | | {color:black}{color} | {color:black} {color} | {color:black} 169m 7s {color} | {color:black} {color} | \\ \\ || Reason || Tests || | JDK v1.8.0_66 Failed junit tests | hadoop.yarn.server.resourcemanager.TestClientRMTokens | | | hadoop.yarn.server.resourcemanager.TestAMAuthorization | | JDK v1.7.0_85 Failed junit tests | hadoop.yarn.server.resourcemanager.TestClientRMTokens | | | hadoop.yarn.server.resourcemanager.security.TestRMDelegationTokens | | | hadoop.yarn.server.resourcemanager.TestAMAuthorization | | | hadoop.yarn.server.resourcemanager.TestRM | \\ \\ || Subsystem || Report/Notes || | Docker | Image:yetus/hadoop:0ca8df7 | | JIRA Patch URL | https://issues.apache.org/jira/secure/attachment/12775241/YARN-4398.4.patch | | JIRA Issue | YARN-4398 | | Optional Tests | asflicense compile javac javadoc mvninstall mvnsite unit findbugs checkstyle | | uname | Linux bc3bf54a8c60 3.13.0-36-lowlatency #63-Ubuntu SMP PREEMPT Wed Sep 3 21:56:12 UTC 2014 x86_64 x86_64 x86_64 GNU/Linux | | Build tool | maven | | Personality | /testptch/hadoop/patchprocess/precommit/personality/provided.sh | | git revision | trunk / 53e3bf7 | | findbugs | v3.0.0 | | unit | https://builds.apache.org/job/PreCommit-YARN-Build/9834/artifact/patchprocess/patch-unit-hadoop-yarn-project_hadoop-yarn_hadoop-yarn-server_hadoop-yarn-server-resourcemanager-jdk1.8.0_66.txt | | unit | https://builds.apache.org/job/PreCommit-YARN-Build/9834/artifact/patchprocess/patch-unit-hadoop-yarn-project_hadoop-yarn_hadoop-yarn-server_hadoop-yarn-server-resourcemanager-jdk1.7.0_85.txt | | unit test logs | https://builds.apache.org/job/PreCommit-YARN-Build/9834/artifact/patchprocess/patch-unit-hadoop-yarn-project_hadoop-yarn_hadoop-yarn-server_hadoop-yarn-server-resourcemanager-jdk1.8.0_66.txt https://builds.apache.org/job/PreCommit-YARN-Build/9834/artifact/patchprocess/patch-unit-hadoop-yarn-project_hadoop-yarn_hadoop-yarn-server_hadoop-yarn-server-resourcemanager-jdk1.7.0_85.txt | | JDK v1.7.0_85 Test Results | https://builds.apache.org/job/PreCommit-YARN-Build/9834/testReport/ | | modules | C: hadoop-yarn-project/hadoop-yarn/hadoop-yarn-common hadoop-yarn-project/hadoop-yarn/hadoop-yarn-server/hadoop-yarn-server-resourcemanager U: hadoop-yarn-project/hadoop-yarn | | Max memory used | 76MB | | Powered by | Apache Yetus http://yetus.apache.org | | Console output | https://builds.apache.org/job/PreCommit-YARN-Build/9834/console | This message was automatically generated.

13. [~jianhe] and [~templedf], could you assign this jira to me and help to check the patch into trunk? Thanks.
14. [~iceberg565], added you to the contributor list. Assigned this to you. You can also now assign jira to yourself. Committing this.
15. Committed to trunk, branch-2, branch-2.7. [~iceberg565], congratulations on your first YARN patch !
16. FAILURE: Integrated in Hadoop-trunk-Commit #8910 (See [https://builds.apache.org/job/Hadoop-trunk-Commit/8910/]) YARN-4398. Remove unnecessary synchronization in RMStateStore. (jianhe: rev 6b9a5beb2b2f9589ef86670f2d763e8488ee5e90) * hadoop-yarn-project/hadoop-yarn/hadoop-yarn-common/src/main/java/org/apache/hadoop/yarn/event/AsyncDispatcher.java * hadoop-yarn-project/hadoop-yarn/hadoop-yarn-server/hadoop-yarn-server-resourcemanager/src/main/java/org/apache/hadoop/yarn/server/resourcemanager/recovery/RMStateStore.java * hadoop-yarn-project/CHANGES.txt
17. ABORTED: Integrated in Hadoop-Hdfs-trunk-Java8 #658 (See [https://builds.apache.org/job/Hadoop-Hdfs-trunk-Java8/658/]) YARN-4398. Remove unnecessary synchronization in RMStateStore. (jianhe: rev 6b9a5beb2b2f9589ef86670f2d763e8488ee5e90) * hadoop-yarn-project/hadoop-yarn/hadoop-yarn-server/hadoop-yarn-server-resourcemanager/src/main/java/org/apache/hadoop/yarn/server/resourcemanager/recovery/RMStateStore.java * hadoop-yarn-project/CHANGES.txt * hadoop-yarn-project/hadoop-yarn/hadoop-yarn-common/src/main/java/org/apache/hadoop/yarn/event/AsyncDispatcher.java
18. [~jianhe], thank you.
19. Committed to branch-2.8.
20. Closing the JIRA as part of 2.7.3 release.