

**git\_comments:**

1. \* Licensed to the Apache Software Foundation (ASF) under one or more \* contributor license agreements. See the NOTICE file distributed with \* this work for additional information regarding copyright ownership. \* The ASF licenses this file to You under the Apache License, Version 2.0 \* (the "License"); you may not use this file except in compliance with \* the License. You may obtain a copy of the License at \* \* <http://www.apache.org/licenses/LICENSE-2.0> \* \* Unless required by applicable law or agreed to in writing, software \* distributed under the License is distributed on an "AS IS" BASIS, \* WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied. \* See the License for the specific language governing permissions and \* limitations under the License.
2. Request3 will trigger the close of channel, because the number of max chunks being transferred is 2;
3. Prepare the stream.
4. Finish flushing the response for request0.
5. Used to keep track of the number of chunks being transferred and not finished yet.
6. Parse streamChunkId to be stream id and chunk id. This is used when fetch remote chunk as a stream.
7. \* \* Called when start sending a stream.
8. \* \* Called when a stream is successfully sent.
9. \* \* Called when a chunk is successfully sent.
10. \* \* Called when start sending a chunk.
11. \* \* Return the number of chunks being transferred and not finished yet in this StreamManager.
12. \* The max number of chunks being transferred and not finished yet.
13. \* \* The max number of chunks allowed to being transferred at the same time on shuffle service.

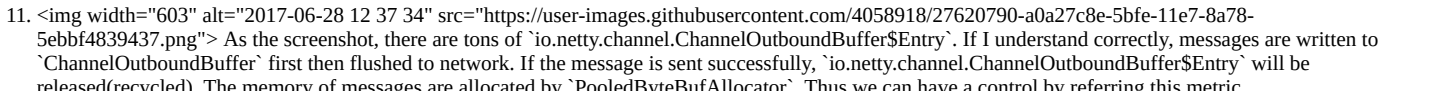
**git\_commits:**

1. **summary:** [SPARK-21175] Reject OpenBlocks when memory shortage on shuffle service.  
**message:** [SPARK-21175] Reject OpenBlocks when memory shortage on shuffle service. ## What changes were proposed in this pull request? A shuffle service can serves blocks from multiple apps/tasks. Thus the shuffle service can suffers high memory usage when lots of shuffle-reads happen at the same time. In my cluster, OOM always happens on shuffle service. Analyzing heap dump, memory cost by Netty(ChannelOutboundBufferEntry) can be up to 2~3G. It might make sense to reject "open blocks" request when memory usage is high on shuffle service.  
<https://github.com/apache/spark/commit/93dd0c518d040155b04e5ab258c5835aec7776fc> and  
<https://github.com/apache/spark/commit/85c6ce61930490e2247fb4b0e22dfecbb8b6a1ee> tried to alleviate the memory pressure on shuffle service but cannot solve the root cause. This pr proposes to control currency of shuffle read. ## How was this patch tested? Added unit test. Author: jinxing <jinxing6042@126.com>  
Closes #18388 from jinxing64/SPARK-21175.

**github\_issues:****github\_issues\_comments:****github\_pulls:**

1. **title:** [SPARK-21175] Reject OpenBlocks when memory shortage on shuffle service.  
**body:** ## What changes were proposed in this pull request? A shuffle service can serves blocks from multiple apps/tasks. Thus the shuffle service can suffers high memory usage when lots of shuffle-reads happen at the same time. In my cluster, OOM always happens on shuffle service. Analyzing heap dump, memory cost by Netty(ChannelOutboundBuffer@Entry) can be up to 2~3G. It might make sense to reject "open blocks" request when memory usage is high on shuffle service.  
<https://github.com/apache/spark/commit/93dd0c518d040155b04e5ab258c5835aec7776fc> and  
<https://github.com/apache/spark/commit/85c6ce61930490e2247fb4b0e22dfecbb8b6a1ee> tried to alleviate the memory pressure on shuffle service but cannot solve the root cause. This pr proposes to control currency of shuffle read. ## How was this patch tested? Added unit test.

**github\_pulls\_comments:**

1. \*\*[Test build #78439 has finished](https://amplab.cs.berkeley.edu/jenkins/job/SparkPullRequestBuilder/78439/testReport)\*\* for PR 18388 at commit [ed889b9] (<https://github.com/apache/spark/commit/ed889b96d938cc8d0dbd9f6b153a4f7c5b44c7d4>). \* This patch \*\*fails RAT tests\*\*. \* This patch merges cleanly. \* This patch adds the following public classes `_(experimental)_`: \* `public class PooledByteBufAllocatorWithMetrics extends PooledByteBufAllocator` \* `public class OpenBlocksFailed extends BlockTransferMessage`
2. \*\*[Test build #78440 has finished](https://amplab.cs.berkeley.edu/jenkins/job/SparkPullRequestBuilder/78440/testReport)\*\* for PR 18388 at commit [0a2bcee] (<https://github.com/apache/spark/commit/0a2bcee8821b36cb373e2f26438482b1f86e2b92>). \* This patch \*\*fails Spark unit tests\*\*. \* This patch merges cleanly. \* This patch adds the following public classes `_(experimental)_`: \* `public class PooledByteBufAllocatorWithMetrics extends PooledByteBufAllocator` \* `public class OpenBlocksFailed extends BlockTransferMessage`
3. \*\*[Test build #78455 has finished](https://amplab.cs.berkeley.edu/jenkins/job/SparkPullRequestBuilder/78455/testReport)\*\* for PR 18388 at commit [f4856c2] (<https://github.com/apache/spark/commit/f4856c20716d72c0cd26a468e36dbef7efbade41>). \* This patch passes all tests. \* This patch merges cleanly. \* This patch adds the following public classes `_(experimental)_`: \* `public class PooledByteBufAllocatorWithMetrics extends PooledByteBufAllocator` \* `public class OpenBlocksFailed extends BlockTransferMessage`
4. @cloud-fan @vanzin @tgraves How do you think about this idea ?
5. Haven't looked at the path in detail yet. High level questions/thoughts. So you say the memory usage is by the netty chunks, so my assumption is this is during the actual transfer? failing the open blocks isn't necessarily going to solve that. If a bunch of reducers all due open blocks at once, it won't reject any and when they all start to transfer it could still run out of memory. It could help in the normal case where some run openblocks while other transfers going on though. Have you been running this patch, what are results? So an alternative to this is limiting the number of blocks each reducer is fetching at once. Instead of calling open blocks with 500 at once, do them in chunks of say 20. We are working on a patch for that and should have it available in the next couple days. This again though doesn't guarantee it but it allows you to throttle down the # of blocks each reducer would get at once. MapReduce/Tez actually do this with a lot of success.
6. Thanks a lot for quick reply :) Yes, this patch doesn't guarantee avoiding the OOM on shuffle service when all reducers are opening the blocks at the same time. But we can alleviate this by adjusting ``spark.reducer.maxSizeInFlight``. ``ShuffleBlockFetcherIterator`` will break the blocks in several requests(see <https://github.com/apache/spark/blob/master/core/src/main/scala/org/apache/spark/storage/ShuffleBlockFetcherIterator.scala#L240>). e.g. 500 blocks can be split into 20 requests, which will be send one by one to shuffle service. If memory cost is high on shuffle service, following requests will be rejected. In normal case this is pretty useful. Also if the the ``OpenBlocks`` is rejected, reducer can sleep for a random duration, say 2s~5s. Thus help to avoid all reducer open blocks at the same time.
7. >So an alternative to this is limiting the number of blocks each reducer is fetching at once Is it relevant to ``spark.reducer.maxSizeInFlight`` ? Breaking ``OpenBlocks`` into more requests is helpful. But I really think we should have some defensive approach on shuffle service side.
8. I think having both sides would probably be good. limit the reducer connections and simultaneous block calls but have a fail safe on the shuffle server side where it can reject connections also makes sense. Can you please give more details what is using the memory? If its the netty blocks is it when its actually streaming the data back to reducer? I thought it was using direct buffers for that so it wouldn't show up on the heap. I'll have to look in more detail.
9. cc @jiangxb1987
10. Will review this tomorrow. Thanks!
11.  As the screenshot, there are tons of ``io.netty.channel.ChannelOutboundBuffer$Entry``. If I understand correctly, messages are written to ``ChannelOutboundBuffer`` first then flushed to network. If the message is sent successfully, ``io.netty.channel.ChannelOutboundBuffer$Entry`` will be released(recycled). The memory of messages are allocated by ``PooledByteBufAllocator``. Thus we can have a control by referring this metric.
12. @jiangxb1987 Thanks a lot for taking time review this pr. I will read your comments very carefully and refine it.
13. LGTM except some minor comments, thanks for working on it!
14. \*\*[Test build #78880 has finished](https://amplab.cs.berkeley.edu/jenkins/job/SparkPullRequestBuilder/78880/testReport)\*\* for PR 18388 at commit [c5a01aa] (<https://github.com/apache/spark/commit/c5a01aab913555157aaec66a23f6c5000e4cb243>). \* This patch \*\*fails Spark unit tests\*\*. \* This patch merges cleanly. \*

- This patch adds no public classes.
15. Jenkins, retest this please.
  16. **\*\*[Test build #78897 has finished](https://amplab.cs.berkeley.edu/jenkins/job/SparkPullRequestBuilder/78897/testReport)\*\*** for PR 18388 at commit [`c5a01aa`] (<https://github.com/apache/spark/commit/c5a01aab913555157aaec66a23f6c5000e4cb243>). \* This patch passes all tests. \* This patch merges cleanly. \* This patch adds no public classes.
  17. LGTM. Can we add descriptions of these new configs in `configuration.md`? thanks!
  18. **\*\*[Test build #78914 has finished](https://amplab.cs.berkeley.edu/jenkins/job/SparkPullRequestBuilder/78914/testReport)\*\*** for PR 18388 at commit [`1d34578`] (<https://github.com/apache/spark/commit/1d345784ca2fbaac463ebb5efaa93c27a5ed3342>). \* This patch passes all tests. \* This patch merges cleanly. \* This patch adds no public classes.
  19. does this patch require server side change for shuffle service?
  20. Yes, there is a change. Server side may return `OpenBlocksFailed` for the "open blocks" request, which means that old client is not compatible with new server. Is it acceptable?
  21. cc @zsxwing how strictly we require for shuffle service compatibility?
  22. Very gentle ping @zsxwing, would you mind help comment on this?
  23. making the external shuffle service incompatible is a huge issue for deployments. For the yarn side you would have to have the nodemanager run 2 versions (which as far as I know hasn't been tested). Unless we have a very good reason we shouldn't do it. I think that would be considered api change and would have to be in major version (3.x line)
  24. @tgravescs I think it's not that hurt. In current change, new client is compatible with the old and new shuffle service. In our clusters, we always upgrade the client first and then server side, which will not cause incompatible issue. The only risk here is that user upgrades the server but still using the old client. But I find no reason they do this. I think users usually tend to upgrade the client first and then deploy new servers gradually. In our cluster, there are nodemanagers failing everyday because of OOM of shuffle service. The root cause is that shuffle service is a hot point and there is no concurrency control.
  25. So that is an issue. If users are running spark 1.6 or spark 2.1 on the same cluster as the new one with this feature, you can't upgrade the shuffle service until no one runs those. We run multiple versions on a cluster at the same time and not everyone immediately upgrades. For instance when a new version comes out, like 2.2 its a bit unstable initially, so production jobs stay on older versions until the newer one stabilizes. You need to make it backwards compatible or come up with different approach. <https://github.com/apache/spark/pull/18487> is the pull request for limiting the reducer fetch at once. Still needs reviewed. It hasn't been run in production yet though but we haven't had issues with NM crashing since we changed the openblocks to be lazy so we wouldn't know immediately how much it helped. that approach definitely helps on the mapreduce/tez side. But depending on what is actually happening may or may not help. The other approach which is less nice is to just have to reject the connection (without returning the failure message) but the client side wouldn't necessarily know why so you would have to make sure it still retried. But I'm actually still wondering about the root cause here. I'm wondering what is actually using the memory. You said it was the netty chunks, Are you using SSL? I had thought that the netty calls we were using use transferTo which shouldn't pull the data into memory, that is of course unless you are using ssl which I don't think can use transferTo. Or are you seeing lots of chunks in memory from the same fetcher? ie you do openblocks of 500 blocks, is it opening all 500 file descriptors at once? I didn't think we did this but want to double check. If we were doing this we should stop by only open a few and when one finishes, open the next. Or is it just that you have 100's of connections from different fetchers and each one has 1 chunk in memory?
  26. @tgravescs As in the screenshot, we have tons of `ChunkOutboundBuffer$Entry`. Yes we are using `transferTo`. Netty will put the `Entry` (containing reference to the `MessageWithHeader`) into `ChannelOutboundBuffer`, then consumer will ship the data onto network. We are running OOM because of too many `ChunkOutboundBuffer$Entry`, as you can see 3GB almost.
  27. I removed the `OpenBlocksFailed` for compatibility. In current change, the server reject the "open blocks" request by closing the connection. Then `RetryingBlockFetcher` will retry.
  28. @jinjing64 what's the downside if we don't have the `OpenBlocksFailed`?
  29. **\*\*[Test build #79280 has finished](https://amplab.cs.berkeley.edu/jenkins/job/SparkPullRequestBuilder/79280/testReport)\*\*** for PR 18388 at commit [`91b05a2`] (<https://github.com/apache/spark/commit/91b05a226f26ba25612eb44b00c0df1ac91693b5>). \* This patch fails Spark unit tests. \* This patch merges cleanly. \* This patch adds no public classes.
  30. ok sorry I forgot you had the screenshot there. so as you mention in that post if we are just creating too many outboundbuffers before they can actual be sent over the network then we should try to add some flow control. did you check to see what the buffers were for? How many connections did you have any how many blocks was each fetching? a million is a lot either way. but I'm assuming its something like you had 500 connections each fetching 2000 blocks. If that is the case it seems like it would be good to add flow control here rather than just disconnecting based on memory. really having both would be good, this as a fall back, but the flow control part should allow everyone to start fetching without rejecting a bunch, especially if the network can't push it out that fast anyway. For instance only create a handful of those outgoing buffers and wait to get successfully sent messages back for the those before creating more. This might be a bit more complex
  31. Analyzing the heap dump, there are 200K+ connections and 3.5M blocks(`FileSegmentManagedBuffer`) being fetched. Yes, flow control is a good idea. But I still think it make much sense to control the concurrency. Reject some "open blocks" requests, thus we can have sufficient bandwidth for the existing connections and we can finish the reduce task as soon as possible. Simple flow control(slow down connections when pressure) can help avoid OOM, but it seems more reduce tasks will run longer.
  32. > there are 200K+ connections and 3.5M blocks(`FileSegmentManagedBuffer`) being fetched. Did you use a large `spark.shuffle.io.numConnectionsPerPeer`? If not, the number of connections seems too large since each `ShuffleClient` should have only one connection to one shuffle service. How large is your cluster and how many applications are running at the same time?
  33. @cloud-fan To be honest, it's a little bit tricky to reject "open blocks" by closing the connection. The following reconnection will surely have extra cost. In current change we are relying on retry mechanism of `RetryingBlockFetcher`. `spark.shuffle.io.maxRetries` and `spark.shuffle.io.retryWait` should also be tuned, with this change maybe their meanings become different, users should know this. This is the sacrifice for compatibility. It comes to me that can we add back `OpenBlocksFailed` and add a flag(default false)? If user wants to turned on, we can tell them they should upgrade the client.
  34. We didn't change `spark.shuffle.io.numConnectionsPerPeer`. Our biggest cluster has 6000 `NodeManager`'s. There are 64 executors at most running on a same host at the same time.
  35. 200k+ connections seems to be your problem then. Is this all a single application? You say 6000 nodes with 64 executors on each host, how many cores per executor? Or do you mean basically each host can run max 64 tasks in parallel. (6000\*64) = 384000 which would be your 200K. I'd be surprised if every reducer is hitting the all nodes at the same time. We are randomizing the blocks to fetch in hope they don't hit all the same one at once. have you tried using `spark.reducer.maxReqsInFlight`? Rejecting connection could also slow things down or even worse make them fail. You have a wait between retries and if you hit the max retries and fail tasks that is much worse then flow control. Reconnection does have a cost but either way you are going to wait some between retries, you don't actually want to retry to quickly or you will just have same issue. What problem are you seeing with the close? I agree that I think both are good to have but personally think the reject connections should be the last thing you want to do.
  36. @tgravescs Thanks a lot for reviewing this pr thus much. I think I'm making a stupid mistake. Can I ask a question, how to check the number of connections? I'm just counting the `org.jboss.netty.channel.socket.nio.NioAcceptedSocketChannel` in the heap dump. To be honest, 200k is also beyond my imagination.
  37. @jinjing64 Since [ExternalShuffleService] (<https://github.com/apache/spark/blob/a0fe32a219253f0abe9d67cf178c73daf5f6fcc1/core/src/main/scala/org/apache/spark/deploy/ExternalShuffleService.scala#L55>), will close idle connections, this looks like a connection leak. Could you dig more to see if you can find any connection leak issue?
  38. Thanks for reply. I will figure out what I can do for this :)
  39. yeah it would be interesting to see if those are all active. I guess the question still is how much memory are those using vs what the actual blocks are using. You said the memory was being used by the `ChannelOutBoundBuffer#Entry`. I think you would only have those on an actual transfer unless those are being leaked as well.
  40. @tgravescs Thanks a lot for your advice :) very helpful. I will try more on this.
  41. @jinjing64 I'm still a bit curious on a few of my previous questions to get exact usage here: Is this all a single application? You say 6000 nodes with 64 executors on each host, how many cores per executor? have you tried using `spark.reducer.maxReqsInFlight`?
  42. >Is this all a single application? No, it's data warehouse, there are thousands ETLs >You say 6000 nodes with 64 executors on each host, how many cores per executor? 1 core per executor. > have you tried using `spark.reducer.maxReqsInFlight` I think we didn't.
  43. Previously I was saying that I have 200k+ connections to one shuffle service. I'm sorry about this, the information is wrong. It turns out that our each `NodeManager` has two auxiliary shuffle services, one for Spark and one for "Hadoop MapReduce". Most of the connections are for "Hadoop MapReduce" shuffle service. Analyzing the heap dump, there are only 761 connection(`NioSocketChannel`)s to the Spark shuffle service. (How I found this? Spark shuffle service is using Netty4 for transferring blocks. I found tons of `org.jboss.netty.channel.socket.nio.NioAcceptedSocketChannel`, checking Netty code, I found they are only used in Netty3, that's used in our Hadoop.) So @zsxwing, there is no connection leak in my understanding. The situation is we have 10K map tasks ran on the same shuffle service and around 1K reduce tasks fetching the blocks. On java heap I found one `io.netty.channel.ChannelOutboundBuffer.Entry` (reference one

- block) will cost almost 1K bytes and we have 3.5M Entries. When OOM, we have `io.netty.channel.ChannelOutboundBuffer.Entry``s retaining 3GBytes. So the problem here is one connection is fetching too many blocks. I believe tuning `spark.reducer.maxReqsInFlight`` or `spark.reducer.maxBlocksInFlightPerAddress`` (#18487) can alleviate this issue. The question is how to set it appropriately. It seems hard because we need to make a balance between warehouse performance and stability. After all there are only 2~3 NodeManagers running OOM, we cannot set `spark.reducer.maxReqsInFlight`` too small to avoid performance degradation. I checked the connections of one shuffle services yesterday. 5K connections is very common during the night. It's easy to happen, say there are 5K reduces running at the same time. What if there are 5 applications and each has 5K reduces? That will be 25k connections. If each connection is fetching 100 blocks and each `Entry`` is 1KB. The memory cost is 2.5G. I think it's too much. So I'm still proposing concurrency control. Different from current change, can we control the number of blocks being transferred? If the number is above water mark, we can fail the new coming `OpenBlocks``.
44. shall we control it at reducer side or shuffle service side or both?
45. I think it could be more efficient to do the control on shuffle service side.
46. Ideally we do both. I think <https://github.com/apache/spark/pull/18487> already will help you on the reducer side. It allows you to limit the # of blocks its fetching in one call. So you should see `max 1k * spark.reducer.maxBlocksInFlightPerAddress` at any given time. say you set it to 20, `20*1k` is much better than 3.5M, even at the 5k you mention, thats only 100k. Note Mapreduce has a very similar concept and that is set to 20 and has been working well for many years. I think this should be easier to tune from the other configs as you are still fetching from the same # of hosts you were before, its just chunking it. We've done some testing with that and haven't seen any performance differences between `Int.max` and say 20. We are still doing more perf testing as well. We an also add flow control on the shuffle server side where we only create a configurable number of outbound buffers before waiting for them to be sent. once one is sent you create another. This might be a bit more work on the shuffle server but haven't looked in detail. I assume in your case the 1k reducers were spread across multiple different jobs? Is it always the same big jobs that cause issues? If it was the same job I would use `spark.reducer.maxReqsInFlight` temporarily. We use that for some of our larger jobs that were causing issues and didn't see any performance impact set to 100 but its going to be job specific.
47. +1 on both. We have to do it at server side, because there may be a lot of reducers fetching data from one shuffle service at the same time, which may OOM the server even each reducer has a limited requests size to this server. (stability) And we also need to do it at reducer side, in case several reducers exhaust the shuffle service and make all other reducers waiting. (performance)
48. @travesces Thanks a lot for advice. > the flow control part should allow everyone to start fetching without rejecting a bunch, especially if the network can't push it out that fast anyway. For instance only create a handful of those outgoing buffers and wait to get successfully sent messages back for the those before creating more. Regarding flow control, can I ask a question? In flow control, we won't reject `OpenBlocks``, but do we reject `ChunkFetchRequest``? If we don't reject `ChunkFetchRequest``, it means we will accept all `ChunkFetchRequest``s. We must store the `ChunkFetchRequest``s in some buffer when memory pressure from Netty is above water mark. But buffering the `ChunkFetchRequest``s can also take much memory. That will be another issue. The problem is the client never know when to slow down sending requests. Am I wrong?
49. I don't think we reject any requests at this point. So yes you could still run into an issue. Generally I think limiting the # of blocks you are fetching at once will solve this also, but its not guaranteed. By default we get one chunk per block, so again I think the most requests you would have buffered would be `spark.reducer.maxBlocksInFlightPerAddress * # reducer connected`. I'm still ok with adding in something like what you have here as a last resort type thing. I recommend just closing the connections rather than changing the api to keep backwards compatibility. If there is some issue with that though then perhaps we can config it off by default like you suggest but I think that is much harder for the broader community to see benefit from it. You mentioned a problem with just doing a close, what was the problem? The flow control part I mention above is really just if the outbound buffers are taking to much memory. I think doing all 3 of these would be good.
50. @travesces Thanks a lot for helping this pr ! I changed this pr, in current change: Shuffle server will track the number of chunks being transferred. Connection will be closed when the number is above water mark. I think it makes sense -- shuffle server should close the connection rather than OOM when memory pressure is high. For the flow control part, it is not included in current change. Because I don't have a proper way to balance between buffering requests and caching `ChannelOutboundBuffer$Entry``. >You mentioned a problem with just doing a close, what was the problem? I just think user should know there's another condition that can trigger closing connection. Also reconnection can have some cost.
51. **\*\*[Test build #79640 has finished](<https://amplab.cs.berkeley.edu/jenkins/job/SparkPullRequestBuilder/79640/testReport>)\*\*** for PR 18388 at commit [`d7c71a4``] (<https://github.com/apache/spark/commit/d7c71a4e35597375dfd1e47c4bde222a09a6a395>). \* This patch passes all tests. \* This patch merges cleanly. \* This patch adds no public classes.
52. **\*\*[Test build #79642 has finished](<https://amplab.cs.berkeley.edu/jenkins/job/SparkPullRequestBuilder/79642/testReport>)\*\*** for PR 18388 at commit [`3cc29a7``] (<https://github.com/apache/spark/commit/3cc29a7ab4e9418ab86dd91821e3a1dfb31c20d7>). \* This patch passes all tests. \* This patch merges cleanly. \* This patch adds no public classes.
53. **\*\*[Test build #79754 has finished](<https://amplab.cs.berkeley.edu/jenkins/job/SparkPullRequestBuilder/79754/testReport>)\*\*** for PR 18388 at commit [`70aaf9a``] (<https://github.com/apache/spark/commit/70aaf9a56c91ecc997c0094c0f1c4a3f72fc362>). \* This patch passes all tests. \* This patch merges cleanly. \* This patch adds no public classes.
54. **\*\*[Test build #79785 has finished](<https://amplab.cs.berkeley.edu/jenkins/job/SparkPullRequestBuilder/79785/testReport>)\*\*** for PR 18388 at commit [`7dd2cec``] (<https://github.com/apache/spark/commit/7dd2cec311189feb555f3cfd8bb27b29676efc18b>). \* This patch passes all tests. \* This patch merges cleanly. \* This patch adds no public classes.
55. a question: can we track the number of sending chunks in `TransportRequestHandler`` instead of `StreamManager``? It looks weird to me that only `OneForOneStreamManager`` does the tracking while we put this concept in the `StreamManager`` interface, and makes me doubt that if `StreamManager`` is the right abstraction level for tracking the number of sending chunks.
56. **\*\*[Test build #79800 has finished](<https://amplab.cs.berkeley.edu/jenkins/job/SparkPullRequestBuilder/79800/testReport>)\*\*** for PR 18388 at commit [`ef89321``] (<https://github.com/apache/spark/commit/ef893215e1076ee4758d19240aa2d0342b1d544d>). \* This patch passes all tests. \* This patch merges cleanly. \* This patch adds no public classes.
57. @cloud-fan I understand your concern. A `TransportRequestHandler`` is for a channel/connection. We want to track the sending chunks of all connections. So I guess we must have a manager for all the connections. Currently, all chunks are served from `OneForOneStreamManager``, so I put the logic there.
58. **\*\*[Test build #79840 has finished](<https://amplab.cs.berkeley.edu/jenkins/job/SparkPullRequestBuilder/79840/testReport>)\*\*** for PR 18388 at commit [`98123ee``] (<https://github.com/apache/spark/commit/98123ee6e4bbe685f75db6cd55a1d7e9c87ee9d2>). \* This patch passes all tests. \* This patch merges cleanly. \* This patch adds no public classes.
59. **\*\*[Test build #79855 has finished](<https://amplab.cs.berkeley.edu/jenkins/job/SparkPullRequestBuilder/79855/testReport>)\*\*** for PR 18388 at commit [`4bfeabb``] (<https://github.com/apache/spark/commit/4bfeabb8755b71f161f086ef68f95f522b848f23>). \* This patch \*\*fails from timeout after a configured wait of `\`250m``\*\*. \* This patch merges cleanly. \* This patch adds no public classes.
60. **\*\*[Test build #79856 has finished](<https://amplab.cs.berkeley.edu/jenkins/job/SparkPullRequestBuilder/79856/testReport>)\*\*** for PR 18388 at commit [`5f622c3``] (<https://github.com/apache/spark/commit/5f622c3da3b65b8d183e329ac641caa1c9aed9bb>). \* This patch \*\*fails from timeout after a configured wait of `\`250m``\*\*. \* This patch merges cleanly. \* This patch adds no public classes.
61. **\*\*[Test build #79857 has finished](<https://amplab.cs.berkeley.edu/jenkins/job/SparkPullRequestBuilder/79857/testReport>)\*\*** for PR 18388 at commit [`4de417f``] (<https://github.com/apache/spark/commit/4de417f946430dd6d963768583d5fa1f22fe4622>). \* This patch \*\*fails from timeout after a configured wait of `\`250m``\*\*. \* This patch merges cleanly. \* This patch adds no public classes.
62. retest this please
63. @jinxing64 Sorry, I forgot to mention one request. Could you add a unit test? Right now it's disabled so the new codes are not tested. It will help avoid some obvious mistakes, such as the missing ``return`` issue :)
64. **\*\*[Test build #79858 has finished](<https://amplab.cs.berkeley.edu/jenkins/job/SparkPullRequestBuilder/79858/testReport>)\*\*** for PR 18388 at commit [`4de417f``] (<https://github.com/apache/spark/commit/4de417f946430dd6d963768583d5fa1f22fe4622>). \* This patch \*\*fails due to an unknown error code, -9\*\*\*. \* This patch merges cleanly. \* This patch adds no public classes.
65. **\*\*[Test build #79879 has finished](<https://amplab.cs.berkeley.edu/jenkins/job/SparkPullRequestBuilder/79879/testReport>)\*\*** for PR 18388 at commit [`8ee60f8``] (<https://github.com/apache/spark/commit/8ee60f8c95b9afb61bed8cd34f21f2c755ecd79>). \* This patch \*\*fails due to an unknown error code, -9\*\*\*. \* This patch merges cleanly. \* This patch adds no public classes.
66. Jenkins, retest this please.
67. **\*\*[Test build #79886 has finished](<https://amplab.cs.berkeley.edu/jenkins/job/SparkPullRequestBuilder/79886/testReport>)\*\*** for PR 18388 at commit [`8ee60f8``] (<https://github.com/apache/spark/commit/8ee60f8c95b9afb61bed8cd34f21f2c755ecd79>). \* This patch passes all tests. \* This patch merges cleanly. \* This patch adds no public classes.
68. **\*\*[Test build #79923 has finished](<https://amplab.cs.berkeley.edu/jenkins/job/SparkPullRequestBuilder/79923/testReport>)\*\*** for PR 18388 at commit [`3a018b1``] (<https://github.com/apache/spark/commit/3a018b14ce4c5bbb894df8a85ce3ec0586f276b>). \* This patch passes all tests. \* This patch merges cleanly. \* This patch adds no public classes.

69. thanks, merging to master!
70. Thanks for merging !
71. sorry I didn't get a chance to review this. Started but kept getting distracted by other higher priority things. I think we should expand the description of the config to say what happens when the limit is hit. Since its not using real flow control a user might set this thinking nothing bad will happen, but its dropping connections so could cause failures if the retries don't work. I'll file a separate jira for that. Also what was the issue with implementing the actual flow control part? Was it just adding a queueing type mechanism? We should file a separate jira so we can add that later.
72. <https://issues.apache.org/jira/browse/SPARK-21530>
73. @tgraves Thanks for help. > I think we should expand the description of the config to say what happens when the limit is hit. Since its not using real flow control a user might set this thinking nothing bad will happen, but its dropping connections so could cause failures if the retries don't work. Could you give the link for the JIRA ? I'm happy to work on a follow-up PR if possible. For the flow control part, I'm just worrying the queue will be too large and causing memory issue.
74. if it's ok to break shuffle service backward compatibility (by default this config is off), I think we should introduce a new response type to tell the client that, the shuffle service is still up but just in memory shortage, please do not give up and keep trying. Currently we just close the connection, so the client has no idea what's going on and may mistakenly report FetchFailure and fail the stage/job.
75. its not ok to break the shuffle service backward compatibility though. Especially not in a minor release. We may choose to do it in like a 3.0 but even then it makes upgrading very hard to users.
76. OK then let's go with the flow control direction. > For the flow control part, I'm just worrying the queue will be too large and causing memory issue. We can make an external queue, i.e. if it's too large, spill to disk. Another concern is, with flow control, shuffle service may hang a request for a long time, and cause the client to timeout and fail. It's better than just closing the connection, but there is still a chance that the client mistakenly reports FetchFailure.
77. the idea of the queue is not to queue entire requests, its just to flow control the # chunks being sent at once. for example you only create 5 outgoing chunks at a time per connection, once one of those has been sent you add another one. This limits the amount of memory being used by those outgoing chunks. This should not affect closing the connection, at least not change it from the current behavior.
78. oh i see, it's orthogonal to the current approach. Makes sense.

#### github\_pulls\_reviews:

1. Should we move these configs to `org.apache.spark.internal.config`? In java, you can use them by: `` import org.apache.spark.internal.config.package\$; ... package\$.MODULE\$.XXX() ``
2. ditto
3. ditto
4. ditto
5. Do we still need this trace?
6. What's the purpose of this change?
7. Sure, I will refine.
8. I'm not sure, just for debug :)
9. `MesosExternalShuffleService` can use it.
10. oh, make sense.
11. Yes, I do think it's good to put the config into `org.apache.spark.internal.config`. But I found it hard. Since `org.apache.spark.internal.config` in `core` module. I didn't find a good way to import it from module `spark-network-yarn` or `spark-network-shuffle`. Did I miss something?
12. oh, `core` module relies on `spark-network-common`, so we don't have to change here.
13. nit: it's better to use `Iterator` pattern here, as the input list may not be an indexed list and `list.get(i)` becomes `O(n)`.
14. do we have a config for shuffle service JVM heap size? maybe we can use that.
15. shall we merge this and the above log into one log entry?
16. nit: `this.reason == o.reason`?
17. can you add some comments to explain the test?
18. Yes, I should refine.
19. I'm hesitant. Because Netty could use off-heap or on-heap (depends on `spark.shuffle.io.preferDirectBufs`) for allocating memory.
20. This is inconsistent with the executor memory configuration. For executor memory, we have `spark.executor.memory` for heap size, and `spark.memory.offHeap.size` for off-heap size, and these 2 together is the total memory consumption for each executor process. Now we have a single config for shuffle service total memory consumption, this seems better, shall we fix the executor memory config?
21. Sorry, you mean: 1. the change( `spark.network.netty.memCostWaterMark` ) in this pr is ok 2. we merge `spark.executor.memory` (heap size) and `spark.memory.offHeap.size` (off-heap size, used by Tungsten) to be one in executor memory config. Do I understand correctly?
22. Currently, I do think `spark.memory.offHeap.size` is quite confusing.
23. maybe we can create a JIRA and send it to dev-list to gather feedbacks. Do you mind to do this if you have time?
24. Sure, really would love to :)
25. cc @rxin @JoshRosen @zsxwing any suggestion for the config name?
26. @cloud-fan I made a JIRA(<https://issues.apache.org/jira/browse/SPARK-21270>) about merging the memory configs. Please take a look when you have time and give some comments.
27. Instead of adding this class, can we upgrade to Netty 4.0.45.Final which allows us to use metrics APIs? (See <https://github.com/netty/netty/pull/6460>)
28. This is only for shuffle service. Right? The config should start with `spark.shuffle`. I'm okay with one config since Netty either uses heap memory or off-heap memory.
29. according to the usage, seems `volatile long chunksBeingTransferred` is enough?
30. shall we accept a `long streamId` here?
31. nit: remove this blank line
32. Really? we do `chunksBeingTransferred++` and `chunksBeingTransferred--`? I think it's not safe.
33. Yes, a `long streamId` will be cast to `Long`, right?
34. Ah, sorry..
35. I mean `chunkSent(long streamId)`, the caller side is responsible for converting stream id to long.
36. you are right
37. In which case the `streamManager` may not be `OneForOneStreamManager`?
38. can we have a reasonable default value?
39. When executor fetch file from driver, it will send `StreamRequest`. `NettyStreamManager` on driver will serve the stream.
40. It depends on memory config of NodeManager. In our cluster, we will set it to be 3G(memory overhead for Netty)/1k(size of ChannelOutboundBuffer\$Entry)=3000000. I think it's ok to leave it MAX\_VALUE (by default it's disabled).
41. `NettyStreamManager.chunkSent` is no-op, seems we don't need this `if`?
42. I feel there are too many "by default disabled" optimizations, which can't benefit most of the spark users. I'm ok to leave it disabled for now, but we should definitely revisit it and think about reasonable default values. also cc @tgraves @zsxwing BTW, please document this config in `configuration.md`
43. The `streamId` sent to `NettyStreamManager` is not a "{streamId}\_{chunkIndex}", so fail to parse?
44. oh i see
45. `spark.network.shuffle` -> `spark.shuffle`
46. I think it makes more sense to have 2 methods: `` chunkSent(long streamId); streamSent(String streamId); ``
47. Yes, it would be much better
48. This default value totally depends on the JVM heap size. Seems hard to pick up a reasonable value. If it's too small, then if a user uses a large heap size, when they upgrade, their shuffle service may start to fail. If it's too large, it's just the same as MAX\_VALUE.
49. missing `return`.
50. missing `return`
51. nit: `The max number of chunks allowed to being transferred at the same time on shuffle service.`
52. Please also move this to `Shuffle Behavior` section.
53. @jinxiang64 `chunksBeingTransferred` is modified in the same thread. Not a big deal though.
54. nit: extra empty line

- 55. nit: please document the meaning of the return value for this public method.
- 56. nit: extra empty line
- 57. Also please decrease `chunksBeingTransferred` for when sending ChunkFetchFailure and StreamFailure if `chunksBeingTransferred` is increased.
- 58. To make the error handling simple, you can increase chunksBeingTransferred just before writing the chunk to the channel, and decrease it in the future returned by write.

jira\_issues:

jira\_issues\_comments: