Kafka Error Handling and Logging

Background

Today we observe two common scenarios in our logging:

1. In some places, we creates too much not-so-useful logging content which pollute the logging files. For example, below is a snippet of the server log under normal operations (setting log4j level to INFO):

```
2014/08/24 00:00:12.145 INFO [Processor] [kafka-network-thread-10251-1] [kafka-server] [] Closing socket
connection to /172.17.198.21.
2014/08/24 00:00:12.152 INFO [Processor] [kafka-network-thread-10251-6] [kafka-server] [] Closing socket
connection to /172.24.25.165.
2014/08/24 00:00:12.191 INFO [Processor] [kafka-network-thread-10251-4] [kafka-server] [] Closing socket
connection to /172.17.198.24.
2014/08/24 00:00:12.209 INFO [Processor] [kafka-network-thread-10251-4] [kafka-server] [] Closing socket
connection to /172.17.232.135.
2014/08/24 00:00:12.218 INFO [Processor] [kafka-network-thread-10251-6] [kafka-server] [] Closing socket
connection to /172.17.198.41.
2014/08/24 00:00:12.235 INFO [Processor] [kafka-network-thread-10251-5] [kafka-server] [] Closing socket
connection to /172.17.198.30.
2014/08/24 00:00:12.326 INFO [Processor] [kafka-network-thread-10251-1] [kafka-server] [] Closing socket
connection to /172.17.198.27.
2014/08/24 00:00:12.362 INFO [Processor] [kafka-network-thread-10251-2] [kafka-server] [] Closing socket
connection to /172.17.198.23.
2014/08/24 00:00:12.455 INFO [Processor] [kafka-network-thread-10251-5] [kafka-server] [] Closing socket
connection to /172.17.198.94.
2014/08/24 00:00:12.488 INFO [Processor] [kafka-network-thread-10251-3] [kafka-server] [] Closing socket
connection to /172.17.198.20.
```

This is from the socket server, which logs closing a socket each time as INFO. However, since most clients use a temporary socket to refresh their metadata, this result in tons of closing socket INFO entries which are useless most of the time.

Another example is that when there is a network connection issue causing clients / follower-replicas not able to connect to servers / leader replicas, an ERROR / WARN log entry will be recorded. But since most of the clients will retry frequently upon failures, and usually log4j will be configured to at least include ERROR / WARN entries, this can cause the clients logs to be swamped with ERROR / WARN along with stack traces:

```
ERROR kafka.producer.SyncProducer  - Producer connection to localhost:1025 unsuccessful
java.net.ConnectException: Connection refused
    at sun.nio.ch.Net.connect0(Native Method)
    at sun.nio.ch.Net.connect(Net.java:465)
    at sun.nio.ch.Net.connect(Net.java:457)
    at sun.nio.ch.SocketChannelImpl.connect(SocketChannelImpl.java:639)
    at kafka.network.BlockingChannel.connect(BlockingChannel.scala:57)
    at kafka.producer.SyncProducer.connect(SyncProducer.scala:146)
    at kafka.producer.SyncProducer.getOrMakeConnection(SyncProducer.scala:161)
    at kafka.producer.SyncProducer.kafka$producer$SyncProducer$$doSend(SyncProducer.scala:68)
    at kafka.producer.SyncProducer.send(SyncProducer.scala:112)
    at kafka.client.ClientUtils$.fetchTopicMetadata(ClientUtils.scala:53)
    at kafka.producer.BrokerPartitionInfo.updateInfo(BrokerPartitionInfo.scala:82)
    at kafka.producer.async.DefaultEventHandler$$anonfun$handle$1.apply$mcV$sp(DefaultEventHandler.scala:69)
    at kafka.utils.Utils$.swallow(Utils.scala:186)
    at kafka.utils.Logging$class.swallowError(Logging.scala:105)
    at kafka.utils.Utils$.swallowError(Utils.scala:45)
    at kafka.producer.async.DefaultEventHandler.handle(DefaultEventHandler.scala:69)
    at kafka.producer.Producer.send(Producer.scala:74)
    at kafka.javaapi.producer.Producer.send(Producer.scala:32)
    ...
WARN kafka.client.ClientUtils$  - Fetching topic metadata with correlation id 0 for topics [Set(test-topic)]
from broker [id:0,host:localhost,port:1025] failed
java.net.ConnectException: Connection refused
    at sun.nio.ch.Net.connect0(Native Method)
    at sun.nio.ch.Net.connect(Net.java:465)
    at sun.nio.ch.Net.connect(Net.java:457)
    at sun.nio.ch.SocketChannelImpl.connect(SocketChannelImpl.java:639)
    at kafka.network.BlockingChannel.connect(BlockingChannel.scala:57)
    at kafka.producer.SyncProducer.connect(SyncProducer.scala:146)
    at kafka.producer.SyncProducer.getOrMakeConnection(SyncProducer.scala:161)
    at kafka.producer.SyncProducer.kafka$producer$SyncProducer$$doSend(SyncProducer.scala:68)
    at kafka.producer.SyncProducer.send(SyncProducer.scala:112)
    at kafka.client.ClientUtils$.fetchTopicMetadata(ClientUtils.scala:53)
    at kafka.producer.BrokerPartitionInfo.updateInfo(BrokerPartitionInfo.scala:82)
    at kafka.producer.async.DefaultEventHandler$$anonfun$handle$1.apply$mcV$sp(DefaultEventHandler.scala:69)
    at kafka.utils.Utils$.swallow(Utils.scala:186)
    at kafka.utils.Logging$class.swallowError(Logging.scala:105)
    at kafka.utils.Utils$.swallowError(Utils.scala:45)
    at kafka.producer.async.DefaultEventHandler.handle(DefaultEventHandler.scala:69)
    at kafka.producer.Producer.send(Producer.scala:74)
    at kafka.javaapi.producer.Producer.send(Producer.scala:32)
    ...
```

2. On the other hand, some important logging information is missing, which makes debugging / trouble shooting much more difficult than it should be, for example:

```
[2014-08-11 15:09:56,078] DEBUG [KafkaApi-1] Error while fetching metadata for [item_topic_0,0]. Possible
cause: null (kafka.server.KafkaApis)
```

Some related JIRAs have been filed for these issues, for example, KAFKA-1066 and KAFKA-1122.

In order to resolve this issue, it is necessary to discuss some logging and exception handling conventions along with the existing code styles described here . This wiki page is created to summarize some implicit conventions we have been following, and to inspire discussions about these conventions. The hope is that moving forward, we will have a better sense about exception handling and logging when writing / reviewing code.

Error Handling

1. When calling some API functions of some Java / Scala libraries or other Kafka modules, we need to make sure each of the possible throwing checked exceptions are either 1) caught and handled, or 2) bypassed and hence higher level callers needs to handle them.

Also it is recommended to add the throwable exceptions list for all public API functions with @exception.

2. When logging is needed in exception handling, we need to make a careful call about 1) which logging level should we use (a good guidance would be this , for example); and 2) whether to print exception name, exception message or exception trace.

Usually, if it is clear where the exception is thrown (for example, it is a Kafka exception and hence we are clear which inner functions may throw it), it is recommended to just print the exception name and message (i.e. exception.toString);

If the error is logged with WARN or below, usually we can just print the exception name; if the error is logged with ERROR or FATAL, we need to include the stack trace.

3. In addition, we usually include description of the current state of the system, possible causes of the thrown exceptions, etc in the logging entry.

Logging

When to Log

We usually need to add some logging entries that are expected to record under normal operations:

1. At the beginning / end of module startup(), and shutdown().

2. At the beginning / end of some important phases of background / special case logic such as LogManager.cleanupLogs(), Controller.onBrokerFailure(), etc

What to Log

In an log entry, we usually needs to include the following information:

1. Related local / global variable's values within the function / code block.

   For example, the fetch request string for logging "request handling failures", the current replicas LEO values when advancing the partition HW accordingly, etc.

2. For exception logging (WARN / ERROR), include the possible cause of the exception, and the handling logic that is going to execute (closing the module, killing the thread, etc).

And here are a few general rules we take for logging format:

1. Each logging entry is usually one-line for the ease of text parsing tools upon trouble shooting, unless it will cause a very long line that may span over multiple screens.

   For example, the topic-change listener firing log, that may contain hundreds of partitions information.

2. The logging indent need to be concise and self-distinctive (this is usually also related to client id naming).

3. Use canonical toString functions of the struct whenever possible.

   For example, instead of using "topic [%s, %d]".format(topic, partitionId), use "topic %s".format(TopicAndPartition(topic, partitionId)).

Logging Cleaning Actions

Based on some discussions around these points, we have proposed several actions so far to clean our logging for now, this following list may expand as we discuss along:

1. Compress request logs.

The request handling logs has been a pain for a long time for us. Since it records one entry for each of the request received and handled, and simply due to the large number of clients and some clients may not follow any backoff mechanism in sending metadata refresh requests, etc, its logging files can easily grow to hundreds of GBs in a day. The plan is to change the request logs to default to DEBUG level (summary) with TRACE level (requests detail) printed as binary format at the same time, along with compression / rolling of the binary logs. The DEBUG level information is sufficient under normal operations, and we only need to de-compress and transform the request details logs into text when trouble shooting is needed.

Details can be found in KAFKA-1590.

2. Clean-up unnecessary stack traces.

This one is kind of obvious, we need to go through the Kafka code base and check for all these prints of stack traces, if it is really necessary.

Details can be found in KAFKA-1591.

3. Clean-up unnecessary INFO level log4j entries.

Some of the INFO level logging entries should really be DEBUG or even TRACE, we need to reduce them so that the sever / client logs are clean and concise under normal operations.

Details can be found in KAFKA-1592.