Eventual consistency: client impacts - focus on interruptions

Author: Nicolas Liochon

This presents the client part of HBase reading with eventual consistency, and some implementation details, especially around interruptions.

Test scenario

Launch the PerformanceEvaluationTool with the appropriate number of replicas.
With the hbase shell ensure that the primary region is isolated on a region server without .meta. or .namespace.
Then stop the region server in a loop

```
while(true); do sleep 1; kill -SIGSTOP 10040; sleep 1; kill -SIGCONT 10040; done
```

This suspends the region server for one second, then make it available for one second, then loop. It's possible to increase the stop time, but you must stay under the zk timeout.

Expectations

Without eventual consistency; the test will hang every second.
Note that the latency presented will seem to be good, but it's because only a few calls will be stopped. The scenario will be
(not suspended): execute ~5000 calls in one second
(suspended): execute 1 call.
(not suspended): execute ~5000 calls in one second
(suspended): execute 1 call.
(not suspended): execute ~5000 calls in one second
(suspended): execute 1 call.

And so on. In the example above the average latency will look good, as 15000 calls will be great while 3 will be very bad, but hidden behing the mass of good ones.

With eventual consistency, depending on the timeout, you will have, for example for a 10 millisecond timeout:
(not suspended): execute ~5000 calls in one second
(suspended): execute 100 calls.
(not suspended): execute ~5000 calls in one second
(suspended): execute 100 call.
(not suspended): execute ~5000 calls in one second
(suspended): execute 100 call.

Lower the timeout and you will have more calls.

Implementation

eventual consistency is available for read only operations.
if eventual consistency is set, the operation is first sent to the main region server. If this region server does not answer before a configurable timeout (in micro seconds), the operation is sent to all other replica. The HBase API will return to its caller the first answer it gets. This answer may be stale, if it comes from a secondary replica, in this case the result will be marked as such.

Details

The timeout is considered globally: we can timeout because the server does not answer, because we got an exception and we entered a 'retry' loop with a default pause of 100 millisecond. It can also be that the client itself entered a garbage collection 'stop-the-world' that triggered. In all cases, we will send the call to the other replicas (after the stop-the-world in this last case: there is no magic).

This call (or these calls if there are multiple replicas) are sent on a thread pool. if the thread pool is full, the calls will be delayed.
The main call also goes through a thread pool. So the cause of the delay can be the thread pool itself. In any case, again, we don't try to interpret things: once the timeout is reached we send the calls to all replicas.

Once we have an answer from one of the replicas, we don't need the other replicas to answer. There are multiple cases:
- the calls was not yet started by the client: it's still in the thread pool
- the calls was being sent: the thread is working, but the query is not sent yet
- the call has been sent, but the server is not yet executing it
- the call has been sent and the server is executing it
- the call has been sent and executed.

There are multiple things we can do:
1) cancel the calls not yet sent
2) for the calls in progress, mark that we're not interested anymore by the answer, don't retry if they are currently failing, but don't stop them
3) stop the call on the client only - interrupt the thread that is waiting for an answer, but let the server go.
4) contact the server to ask it to cancel its operation.

1) Cancelling the calls not yet sent is obvious and trivial to write in Java.
2) Mark the calls in progress as abandoned is a little bit less trivial, but seems necessary.
3) Stopping the call on the client (interrupting), is trivial to write, but has very important implications. More on this later.
4) Cancelling the call on the server is a tradeoff: it means that you expect that

- cancelling is less expensive for the server than executing the action
- the server is available
- the client has enough resources to do the cancellation. For example, with 3 replicas, it means that instead of doing a single call when everything goes well,you will do:
- 1 call: the primary
- 2 calls: the secondary
- 2 calls: cancelling the 2 calls that has not made it.

That's 5 calls instead of 1, and instead of 3 if we don't cancel on the server. Sometimes it's better, sometimes it's not. If the call already in progress on the server, you sent the cancel for nothing, except if you're ready to interrupt the server as well (or to implement cooperative cancellation, but it might not cover the cases that actually happen in production).

Tests shows that the primary issue was the client: if the server does not answer, the client threads will get stuck. Without eventual consistency, it's not an issue: we're getting into the scenario described above: the client waits. With eventual consistency, the client does not wait anymore: it sends a call to the secondary, gets the answer, and does it next call -likely a call on the dead server-. So we can't have a thread stuck. We need to get it back at all cost, and this means interrupting it.

Interrupting

It's the standard way in Java. Hence it's trivial to do in Java (use Future#cancel(true) and you're done). The issue is that the underlying code must be resilient to interruption:
- don't shallow the interruption
- don't interpret it as an error that needs a retry.
- don't interpret it as an error that means that the server is in an illegal state that requires to abort it.

HBase code does rely on interruption today (the split worker can be interrupted), but we still have some area that are not as good as they could, and this is an understatement.

So a part of the work is to clean this part, on the client path at least. This can be in third parties as well.

Interruptions & exceptions

There are 4 exceptions to look at:
1) InterruptedException
2) InterruptedIOException extends IOException
3) ClosedByInterruptException extends … extends IOException
4) SocketTimeoutException extends InterruptedIOException

InterruptedException is simple. It's the historical exception in Java.

InterruptedIOException & ClosedByInterruptException are simple as well. The semantic has been changing a little around them, but we can consider they now mean the same thing as InterruptedException (the jdk source code itself can rethrow an InterruptedIOException when it catch an InterruptedException). But it means that any code that it catching IOException catches these exceptions, and must behave differently. Typically

```
for (int i=0; i< nbRetries; i++){
 try {
 catch (IOException ie){
   log.info("try " + i + " failed");
  }
}
```

will become:

```
for (int i=0; i< nbRetries; i++){
 try {
 catch (IOException ie){
   if (ie instance InterruptedIOException or ie instance of
ClosedByInterruptException) {
    break
   }
   log.info("try " + i + " failed");
  }
}
```

But here comes the fourth one: a SocketTimeoutException is an InterruptedIOException that must be retried. So the good code is actually:

```
for (int i=0; i< nbRetries; i++){
 try {
 catch (IOException ie){
   if (ie instance InterruptedIOException or ie instance of
ClosedByInterruptException and not ie instance SocketTimeoutException
) {
    break
  }
  log.info("try " + i + " failed");
 }
}
```

This pattern seems strange, but it's the one we find in various code. For example log4j does this.

Interruption and i/o

Interrupting an i/o is very system dependant. Java chose, after a few iterations, to limit the expectations it had, to work on all systems. It means that interrupting a thread that is writing close the stream. This is valid for synchronous i/o and asynchronous i/o (it closes the channel in this case).

One could think it's a non issue in HBase, with the following code;

```
prepare for write();
synchronized(outstream){
  if (Thread.interrupted()){
     throw new InterruptedIOException("interrupted before writing");
  }
   write(); // critical path: don't interrupt me here. Won't happen
often, so it's acceptable
}
waitForAnswer();
```

This does not work in practise, with 2 issues found:
- We are actually interrupted at the wrong point. This close the connection, and the other users of this multiplexed connections must retry.
- Sometimes (with a 1 / 10000 probability), the interruption does not work well, and we need to wait for the socket timeout (60 seconds by default). It could be a jdk issue. But anyway this shows the i/o are fragile.

So the solution finally retained was to delegate the write to a different thread. This thread is not interrupted, but the caller can be safely interrupted.

Alternatives

Just cancel the calls not done

We would go out of threads very quickly: a single client can send 5000 synchronous gets per second. A 500ms jitter would be enough to kill the client.

Heuristics to interrupt the thread 'only if necessary'

It's possible, I've tried different options:
1) interrupt only if the pool is taken at 50% or more
2) interrupt only if the call is older than 10 milliseconds

They both make sense, and work well for some scenarios. But it's fragile.

Cooperative cancellation

The JDK pushes to use Interrupt, but it does has some flaws. A solution would be to do our own mechanism. It would work, and I seriously considered it, but would be less standard. Using interrupt is standard, and users are requesting it (for example,HBASE-10337 and HBASE-10497 have been created this last month on this). And HBase already needs this on the server for splitting the logs (a critical feature). Lastly, the implementation would be either dirty (a thread local), or expensive (an extra parameter on all methods). As well, if the threads are actually writing, we can't be cooperative here: we need to stop them.