Runtime Infrastructure - Feature #5002

implement fast in-memory based network transport for localhost connections

11/09/2020 08:47 AM - Eugenie Lyzenko

Status:	New	**Start date:**	
Priority:	Normal	**Due date:**	
Assignee:		**% Done:**	0%
Category:		**Estimated time:**	0.00 hour
Target version:			
billable:	No	**vendor_id:**	GCD
Description			

History

#1 - 11/09/2020 08:51 AM - Eugenie Lyzenko

- Tracker changed from Bug to Feature

#2 - 11/09/2020 09:06 AM - Eugenie Lyzenko

The idea behind is to replace currently used pretty slow SocketChannel for the communication cases certainly known as happening on the same physical machine.

In this case the faster approach can be used is to exchange data between server and client parts via reserved shared memory region. This can be some variation of the unnamed pipe. The areas can be involved in:

1. Swing client full code.
2. Web client Java part is currently works with both server side and web browser Javascript engine. In this case only Java client code that works on the same system as server part will be affected.

The advantages:

1. Minimal delay for data exchange.
2. Can be configured as LIFO or FIFO.
3. Not limited to be byte reading oriented, the data chunks can be char, int, double, ... arbitrary any size.

The disadvantages:

- It is complex and risky. More synchronization work is required to ensure data integrity.

#3 - 11/09/2020 09:33 AM - Greg Shah

- Start date deleted (11/09/2020)

Some options:

- Implement using real OS-level pipes or AF_UNIX sockets. This has the advantage of keeping the design the same, which is about serializing a byte stream. The localhost network layer is less costly than the full network stack because it avoids the Ethernet layer and it avoids some of the IP layer. But AF_UNIX sockets would be faster than the current socket approach because it avoids all the TCP layer processing. I don't know how big the advantage would be. Eric did find a small advantage in using AF_UNIX sockets but it was small because the overall network time for DB access is pretty small (see #4012). One disadvantage is that there are still multiple copies of the buffers that must happen. Typically, there would be a copy of the entire buffer with each shift into/out of ring 0 (the kernel-mode Intel CPU protection level as compared with ring 3/user mode). I think there is also a copy when the input data is sent across to the other process. This may mean at least 3 copies of the buffered data while being transferred by the pipe. These copies are expensive.
- Implement our own shared memory transport with some kind of pipe interface. This could be implemented to avoid the user mode/kernel mode transitions and to avoid the buffer copies. It is certainly much more work and more risky than using AF_SOCKETS. Crucially, this approach will still require all of the serialization overhead of creating the byte stream.
- We could implement a shared memory approach that is less of a transport and is more like shared storage of data. This could avoid the serialization but at a greater cost for reworking the data storage of any object that needs to be shared. This is really the ultimate approach for fast "synchronization". To the degree that our performance is gated by this spilt architecture (many thin clients + a server with context-local threads), this approach might resolve the contention. But the cost of the rework would be significant. With this in mind we should consider:
    - We do have a task #4912 to move the web clients into the server process (this cannot work for Swing). In my opinion, this is pretty "doable" but it is definitely some serious work.
    - We could consider something like Hazelcast which is an Apache 2.0 licensed in-memory data grid technology designed for exactly this use-case. We are also considering it for use in #4369. This has the real advantage of not re-inventing the wheel. But as we've found with most other low-level 3rd party frameworks, there is often a mismatch between our low-level needs and the design of the framework. My worry here is that it may solve the functional problem but at a cost of performance which would be useless in the end. Or if it performs well, then the cost may be complexity and fragility.

#4 - 11/10/2020 02:48 PM - Igor Skornyakov

I would start with logging the network I/O and implementing the playback test to figure the real cost of using the existing transport. This test can be used to test another options (at least those ones which use the same (de)serialization logic).
I've used Hazelast before (much earlier version than is avaialble now). It was pretty fast but required serialization. In fact I can hardly imagine more or less general purpose and flexible shared data framework which will not require it. However using a paradigm of shared data may in theory reduce the network traffic if the sgared data will be organized in small chunks.
Maybe it makes sense to consider using more lightweight encryption than one used in SSL, such as XOR with shared "codebook". If this codebook will be long enough and is changed frequently this can be secure enough. If we plan to secure non-SSL communication channel maybe it will be easier to use such channel for sharing the codebook?

#5 - 11/10/2020 03:14 PM - Greg Shah

These are useful thoughts.

In regard to the XOR + codebook idea, please note that we have multiple customers that have security requirements for SSL. The cryptography of SSL is well understood and it has been independently verified. Implementing our own approach brings a big set of problems in this regard. If customers want to use a private network or localhost implementation, then they can use the non-SSL approach which will be faster than our own crypto. If they need to traverse untrusted networks, then SSL is probably the only thing that they will trust.

#6 - 11/10/2020 03:33 PM - Igor Skornyakov

Greg Shah wrote:

> These are useful thoughts.
>
> In regard to the XOR + codebook idea, please note that we have multiple customers that have security requirements for SSL. The cryptography of SSL is well understood and it has been independently verified. Implementing our own approach brings a big set of problems in this regard. If customers want to use a private network or localhost implementation, then they can use the non-SSL approach which will be faster than our own crypto. If they need to traverse untrusted networks, then SSL is probably the only thing that they will trust.

I understand this. But I was talking only about communication between processes running on the same machine. I know that the question about secure communication is a very sensitive subject for customers. But this is applicable to other alternatives duscussed here if I understand correctly.

#7 - 11/10/2020 03:43 PM - Igor Skornyakov

BTW: As far as I understand, XOR with one-time codebook is a "perfect" cipher. The "only" problem is with sharing this codebook betweeen peers. I think I've seen this statement in the classical "Applied Cryptography" by Bruce Scheier. I have it on my bookshelf, but it will take some time to find an exact quote.

#8 - 11/11/2020 06:32 PM - Eugenie Lyzenko

Igor,

Can you clarify the following. The size of the BlockingSSL internal buffer is 16K. What was the criteria of selection this size? Why I can not increase the value to 32K - this cause the Web client to stop working, is it expected? Or may be means a kind of hidden issue in FWD code?

#9 - 11/12/2020 02:37 AM - Igor Skornyakov

Eugenie Lyzenko wrote:

> Igor,
>
> Can you clarify the following. The size of the BlockingSSL internal buffer is 16K. What was the criteria of selection this size? Why I can not increase the value to 32K - this cause the Web client to stop working, is it expected? Or may be means a kind of hidden issue in FWD code?

Eugenie,
As far as I remember, 16K is an SSL max value for a packet size. In any case changing the buffer size is a tricky business. There are several buffers and their sizes must match. I do not rememeber all the details at this moment, but I've spent signigicant amount of time to make things working. In particular it was not easy to modify the code to allow sending large messages, such as ones with fonts. If you wish to change something in the code, please be ready for a hard battle.