Threading Issues And NetVC Migration

Threading in ATS

ATS was designed with the assumption that a single transaction is processed within a single thread. There are two cases that violate this single thread assumption. One is the case of a plugin that spawns threads to handle blocking operations. It is up to the plugin writer to make sure there is sufficient locking and rescheduling in place to avoid problems. The other case of multiple threads per transaction is due to server session reuse via a global pool.

Server session pools are great because we can amortize the potentially expensive connection start up over many transactions. If the server sessions are pooled per thread, then a particular server VC will always operate on the same thread (in the same net handler) as the matching client VC.

With a global session pool you get even better connection reuse. You do not need to keep as many connections open to the origin server so you consume fewer resources on ATS and on the origin servers. However, with a global session pool, you introduce cross thread issues with the processing of a single transaction. It is possible to have a client VC in the nethandler of thread 1 and the server VC in the nethandler of thread 2. The transaction processed by that VC pair will be executed partly on thread 1 and partly on thread 2. This means that locking becomes a much greater concern. Network IO from the server VC could occur at the same time as the client VC is operating on a client side IO.

For the most part this all works out, but there are latent issues. The latest was the issue captured in TS-3486/TS-3266. There was a race condition between reactivating a server session from the pool and a network event coming in at just the wrong time. It is an issue that has been there likely as long as global server session pool support has been there, but other things had changed to make that race condition occur frequently enough to debug.

Within Yahoo, we have encountered an environment that made another race condition far more active. We had seen crashes with the following stack trace occasionally, but recently we have found an environment where these crashes happen so frequently that running ATS with global session pools is not feasible. Filed as TS-3797.

```
#0  0x00000000004fac6e in Ptr<IOBufferBlock>::operator IOBufferBlock* (
    this=0x10) at ../lib/ts/Ptr.h:300
#1  0x00000000005109a2 in IOBufferReader::read_avail (this=0x0)
    at ../iocore/eventsystem/P_IOBuffer.h:606
#2  0x0000000000777b54 in write_to_net_io (nh=0x2acc365358a0,
    vc=0x2acd38024960, thread=0x2acc36532010) at UnixNetVConnection.cc:540
#3  0x000000000077747a in write_to_net (nh=0x2acc365358a0, vc=0x2acd38024960,
    thread=0x2acc36532010) at UnixNetVConnection.cc:407
#4  0x0000000000770378 in NetHandler::mainNetEvent (this=0x2acc365358a0,
    event=5, e=0x2244730) at UnixNet.cc:562
#5  0x0000000000510560 in Continuation::handleEvent (this=0x2acc365358a0,
    event=5, data=0x2244730) at ../iocore/eventsystem/I_Continuation.h:145
#6  0x0000000000796ffe in EThread::process_event (this=0x2acc36532010,
    e=0x2244730, calling_code=5) at UnixEThread.cc:128
#7  0x0000000000797508 in EThread::execute (this=0x2acc36532010)
    at UnixEThread.cc:252
#8  0x00000000007965a9 in spawn_thread_internal (a=0x2115540) at Thread.cc:85
#9  0x00002acc2edd49d1 in start_thread () from /lib64/libpthread.so.0
#10 0x000000032750e88fd in clone () from /lib64/libc.so.6
```

The immediate problem is that the IOBuffer is cleared, but looking at the path to the crash point, it is clear that the IOBuffer was not cleared at the start of the IO operation, so it must have been cleared by another thread.

We believe that our current issue is another race condition. One possible scenario is that an error occurs on the client VC, and the HttpSM shuts down both the server VC and the client VC (do_io_close). Since the server VC is in another thread, the do_io_close only clears the buffer and sets a flag. It will get rescheduled in the right thread before the socket is closed. But by clearing the buffer, any server-side network event that is processed will result in a crash if the buffer clear occurs during the input processing.

I have no doubt that we could adjust locks and fix this issue as we did for TS-3486. But I also have a strong feeling that in a couple months there will be another race condition that will emerge. So rather than applying another bandaid, we should restructure things a bit to stay closer to the original model of having a single transaction processed by a single thread.

VC Migration/Reconstruction to Reduce Cross Thread Coordination

We suggest adding the ability to “Migrate” a server VC from one thread to another. Moving the server VC data structure from one nethandler to another would be difficult. You’d probably need to hold the lock for both nethandlers at some point which would be likely to introduce performance problems.

However, the important items in the Server VC are the socket and the SSL object (in the SSL case). The actual server VC data structure isn’t that important. You could “gut” the existing server VC of its socket/SSL data and create a new server VC in the target thread’s nethandler. By keeping everything on the same thread we reduce the possibility of cross thread contention and we should experience better thread/memory locality. In addition, we propose doing this migration only during the HttpServerSession::acquire_session. This constraint limits the number of cases we must consider.

During HttpSessionManager::acquire_session, the thread holds the HttpSM lock and the ServerSessionPool lock. The server_vc that is in the pool has a read vio open against the ServerSessionPool object. When a write_to_net_io or a read_from_net operation happens on that server VC, the logic will attempt the gain the ServerSessionPool lock. So if our logic got into acquire_session, we are guaranteed that no IO operations will occur on that server session until the function has completed.

We need to perform the following steps to migrate serverVC to a new target thread.

1. Remove the serverSession from the HttpSessionCache.
2. Save aside the socket and ssl out of server session serverVC.
3. Call serverVC->do_io_zombie() which sets the close flag, marks the VC as a zombie and schedules the actual close to be handled on the original thread. Must leave the socket so the original thread can remove it from the event loop, but must mark the vc somehow so the socket is not closed.
    a. Augment close_UnixNetVConnection() to not close the socket if the VC zombie flag is set.
    b. NetHandler::mainNetEvent() checks the close flag. If it is set close_UnixNetVConnection() is called , which cleans up the original serverVC data structure, calls vc->ep->stop() to remove the file descriptor from the thread’s event, and removes the serverVC from the original nethandler lists.
    c. Augment read_from_net to check the close flag after it has acquired the vio mutex. It is possible that acquire_session is called from another thread and set the closed flag after the check in NetHandler::mainNetEvent() is made. If the flag is set, call close_UnixNetVConnection()
4. At this point it is safe for acquire_session to drop the ServerSessionPool lock. The HttpServerSession has been removed and the original serverVC has been marked to be closed and zombied. The acquire_session thread does not touch the zombie VC again.
5. Create a new VC using the saved socket and ssl object and the target thread. Pull logic out of connect_re and connectUp to do this. This will put the new VC in the appropriate lists for the target thread’s nethandler.

The race condition possibilities are limited because we do only limited IO processing for server sessions in the pool. They will never write data. If there is any data to be read it is assumed to be an EOS. If any other data shows up, the connection is closed as well.

Race condition analysis

1. A read event shows up on the serverVC after the acquire_session starts to reactivate the serverSession
    a. If the close flag is set when the mainNetEvent checks it, the original ServerVC will be freed via the call to close_UnixNetConnection
    b. If the close flag is not yet set, the read_from_net will be called and it will fail to get the vio->mutex (the ServerSessionPool mutex is held by acquire_session). The event will trigger on the next iteration and the close flag check in mainNetEvent should trigger.
2. The server socket has a read ready before the the original serverVC has been removed and after the new ServerVC has been inserted on the other nethandler. Both nethandlers should note that there is a read event on the socket file descriptor.
    a. The original ServerVC will have the close flag and the zombie flag set. The close flag check in mainNetEvent will cause the original serverVC to be freed.
    b. The new serverVC will not have the close flag set and the read_for_net should occur as normal.