RX/TX network timestamping in FIXAntenna C++

What is SO_TIMESTAMPING?

- Introduced in Linux 2.6.30, primarily aimed at enabling UDP timestamping capabilities, the feature has evolved significantly over time. By the Linux kernel 6.2, it gained the ability to trace specific parts of the TCP stream, thanks to the introduction of the final piece of the puzzle: the SOF_TIMESTAMPING_OPT_ID_TCP option. This option fixes the OPT_ID generation logic for TCP by accounting for packet reordering and retransmissions. Prior to the introduction of this option, the OPT_ID generation logic was not reliable for TCP streams.

- The feature was adopted by OpenOnload starting from version 8.2, which enabled SO_TIMESTAMPING users to leverage a unified timestamping API. Before this adoption, the only way to work with network timestamping through the OpenOnload wrapper was via a proprietary API.

- This functionality provides several nanosecond-precise timestamps for different parts of the TCP stream as they travel through the system. It delivers highly accurate data about the bytes transmitted over TCP, capturing their journey from the moment the application initiates the send to the point when the receiving counterparty confirms delivery.

What data does SO_TIMESTAMPING provide?

- For RX (Receive) timestamping:
  - The moment when the final byte of the data read from the socket was received by the NIC (Network Interface Card).
  - The moment when the final byte of the data read from the socket was received by the Linux kernel.

- For TX (Transmit) timestamping:
  - The moment when the final byte of the data sent to the socket was passed to the kernel's TCP packet scheduler.
  - The moment when the final byte of the data sent to the socket was handed off to the NIC driver from the kernel.
  - The moment when the final byte of the data sent to the socket was transmitted onto the wire.
  - The moment when the counterparty acknowledged receipt of the TCP byte stream up to the final byte of the data sent to the socket.

How to receive the RX timestamping data?

- For RX timestamps, the data receiving is simple. After configuring the timestamping socket options, the timestamps are available in a special structure contained in the data received from the socket.

```
const uint16_t BUFFER_SIZE = 1000;

msghdr msg{};
iovec iov{};
std::array<char, BUFFER_SIZE> control{};
cmsghdr* cmsg = nullptr;

iov.iov_base = buf; //Buffer to receive the data to
iov.iov_len = len;  //The buffer size
msg.msg_iov = &iov;
msg.msg_iovlen = 1;
msg.msg_control = control.data();
msg.msg_controllen = control.size();

const auto ret = ::recvmsg(socket_, &msg, flags);

for (cmsg = CMSG_FIRSTHDR(&msg); cmsg != nullptr; cmsg = CMSG_NXTHDR(&msg, cmsg))
{
    if (cmsg->cmsg_level == SOL_SOCKET)
    {
        if (cmsg->cmsg_type == SO_TIMESTAMPING_NEW)
        {
            auto tsdata = reinterpret_cast<scm_timestamping64*>(CMSG_DATA(cmsg));
        }
    }
}
```

How to receive the TX timestamping data?

- For TX timestamps, the data receiving is a complicated topic with a lot of nuances. The code is similar to the RX timestamp receiving, but the messages should be received from the `MSG_ERRQUEUE`(::recvmsg(socket_, &msg, MSG_ERRQUEUE);).
- Some of the most significant nuances:
  - Receiving is asynchronous. The message order is not and can not be guaranteed due to the fundamentals of the TCP protocol.
  - The data is not(and in some cases can not be) available instantly.
  - The data may not be available(and therefore, received) at all.
  - The OPT_ID identifier of the sent buffer can not be set manually and is generated by the kernel/NIC instead. The OPT_ID value is a position of the last byte of the buffer sent, in the whole TCP data stream. The application should calculate this value in order to match received timestamps to the original buffers.
  - The time required for the timestamps to arrive to the MSG_ERRQUEUE may vary from a few nanoseconds to a few dozens of seconds depending on the current state of the TCP stream and the network related conditions.
  - The messages in the MSG_ERRQUEUE are consuming the socket buffer budget.
  - The MSG_ERRQUEUE buffer is circular. If not consumed in time, the messages containing timestamps may be overridden by the new incoming data(and lost completely).

What else is there?

- Timestamping does not rely on TCP packets and does not adhere to TCP packet boundaries. While it may naturally coincide with packet boundaries at times, it cannot be processed reliably in this manner. The provided timestamping data is instead tied to the byte positions of the data being sent or received within the TCP byte stream for each specific send or receive call.

- If your application separates the networking layer from the processing layer, it can be challenging to match messages parsed from the TCP stream data (or sent to the network layer) with the exact byte positions within the receiving stream.

- There is no "reasonable" time interval for waiting on TX timestamps. One of the primary applications of network timestamping is network diagnostics, and if you discard data because it took too long to arrive, you lose the ability to diagnose the network. Worse, this loss occurs precisely at moments when such diagnostics are most critical.

Fundamental assumptions

- The following assumptions were made:
  - It is the user's responsibility to map the messages they send to the timestamps they receive. We provide the data and the means to facilitate the mapping; the user is responsible for completing the process.
  - It is the user's responsibility to decide how to respond if timestamping indicates potential issues. We provide the tools to detect such issues; the user must take appropriate actions based on their requirements.
  - It is the user's responsibility to manage delays related to receiving TX timestamps. While we ensure that timestamping data is made available as soon as it is obtained, we do not impose TX data-related delays on the FIX message exchange logic.
  - It is not critical for users to receive TX timestamps immediately upon availability, as long as we provide a mechanism to retrieve them when needed (or allow them to wait for the data at the time of retrieval).

Finally, API

- The user API:

```
class FIXMessage
{
    …
    inline Utils::Timestamping::TimepointHolder::Ptr getTimepointHolder() const;
    …
};

class TimepointHolder
    {
        …
        using Ptr = std::shared_ptr<TimepointHolder>;
        // Read the time point status.
        [[nodiscard]] TimepointStatus getTimepointStatus(Timepoint point);

        // Read the time point timestamp. Only call if the time point status is `TimepointStatus::READY`.
        // Will throw `ValueUnavailableException` if the time point status is `TimepointStatus::UNAVAILABLE`.
        // Will throw `ValueNotReadyException` if the time point status is `TimepointStatus::NOT_REACHED` or `TimepointStatus::WAITING`.
        timespec getTimepoint(Timepoint point);
        …
    };
```

- The timepoint holder instance is made available to the user immediately after a message is submitted to the engine for sending (or instantly for received messages).
- Each instance is unique to a specific send operation, allowing the user to retrieve it and associate it with any relevant information they deem important.

Applications?

- Network diagnostic.
- Latency-sensitive order execution(HFT)
- Post-trade latency analysis(identify delays in order routing and processing)
- Market data alignment(correlate FIX messages to the market data to restore the exact state of the order book at the moment of trading)
- Smart order routing(realtime order latency monitoring and routing changes depending on the wire to execution latency)
- Nanosecond-precise trade replay and debugging.
- Ability to tune wire-to-market latency to win the competition by estimating execution priority or slippage under market load.
- Precise monitoring and alerting. Realtime counterparty latency measurements to instantly indicate issues on the counterparty.

THANK YOU!