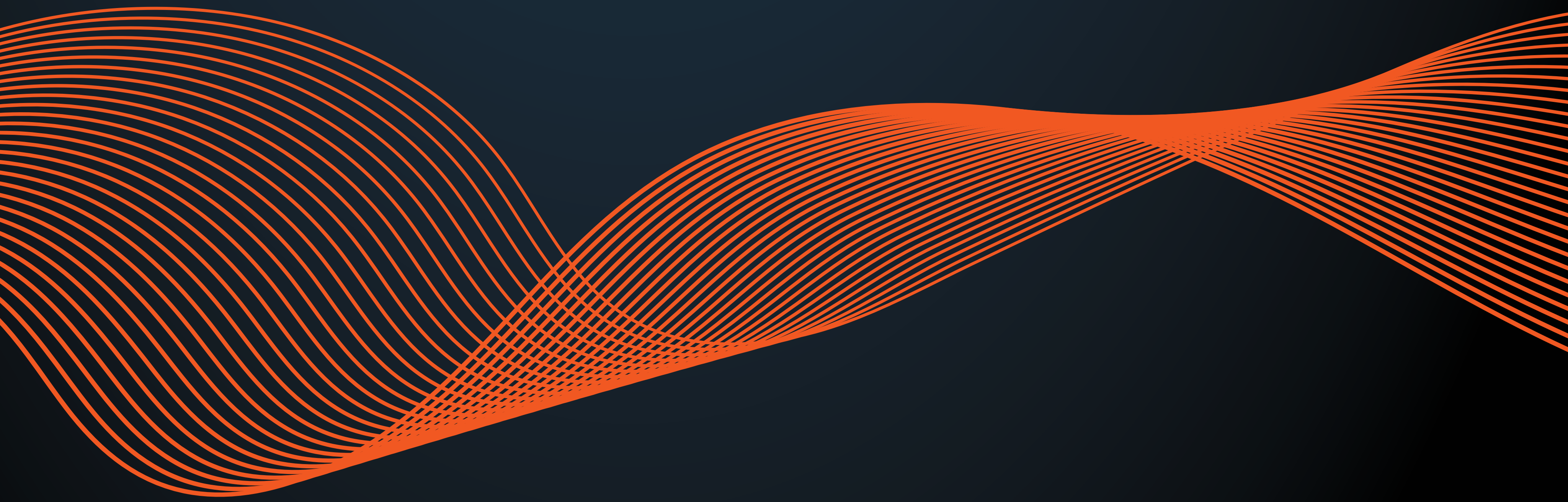
Corvil

FIX PERFORMANCE

Mechanisms, Measurements & Management



By Raymond Russell, CTO

20190628.v2

Contents

Background	3
Trading System Performance	4
Troubleshooting FIX Connectivity	6
Managing FIX Performance	9
Measuring Load Using FIX Timestamps	12
Measuring Latency Using FIX Timestamps	14
Operationalizing FIX Timestamps	21
Augmenting FIX Timestamps with Network Timestamps	23
Indicative Scenario	32
Recap and Conclusion	34

Background

FIX, or the Financial Information Exchange protocol, is the workhorse of communications between modern trading systems. FIX forms the backbone of communication between buy-side firms, sell-side brokers, market-makers, exchanges, ECNs, market data providers, and anyone participating in electronic trading. Even in areas where it has been superseded by other trading protocols, FIX often remains the de facto model for many of its replacements. For example, many equities exchanges offer binary order-entry interfaces as alternatives to FIX, but most of those binary protocols are modeled more or less directly on FIX. Examples include BOE's BATS Binary Order Entry and ICE's NYSE Pillar.

"Even in areas where it has been superseded by other trading protocols, FIX often remains the de facto model for many of its replacements."

The fact that FIX serves as a model for trading protocols is no surprise. With its long and distinguished history as an open standard for exchange of business-critical information, it has become the lingua franca of electronic trading systems. Although newer protocols have done away with some of the older incidental aspects of FIX, such as its ASCII-based variable-length encoding, they have preserved much of the vocabulary. For example, both BATS BOE and NYSE Pillar use FIX names for business message types and fields, reusing even the medial-capital (or camel-case) style of the FIX standard.

The FIX community itself fostered these developments. For example, FPL (FIX Protocol Ltd., now rebranded FIX Trading Community) started development of FAST in 2004, spurred by the ever increasing speeds of market data and processing overheads of the classic FIX format.

FAST, or FIX Adapted for Streaming, provided a completely new encoding of FIX messages, replacing the easily-inspected ASCII tag-value pairs with binary representations that are much more compact, and much more easily processed by modern CPUs. These benefits translate directly into dramatic reductions in bandwidth, processing power, and latency. However, the content carried by FAST – and its modern successor SBE (Simple Binary Encoding) – is exactly the same as carried by traditional FIX. It uses exactly the same tags and ascribes exactly the same meaning to the values of those tags. As a result, knowledge of FIX content serves as a solid foundation for understanding the content of newer trading protocols.

Trading System Performance

Electronic trading is, by virtue of being implemented by automated electronic systems, directly dependent on those systems. The success of an electronic trading business is inextricably linked to the reliability and performance of its systems. Trading is a highly performance-driven business, consuming vast quantities of fast-moving market data feeds and employing sophisticated decision-making algorithms.

Time is an especially sensitive factor in trading, primarily due to intense competitive pressures: when information, such as price updates or Fed announcements, is broadcast to market participants, those who react first reap most if not all of the financial advantage.

As a result, trading firms in pursuit of such advantages invest heavily in ensuring their internal systems can react and issue orders with as little delay or latency as possible. Their choices of brokers and venues on which to trade will be strongly influenced by the latency of those counterparties. As a result, brokers and exchanges in turn use latency as a competitive weapon in the battle for order-flow and liquidity. Even where low latency is not the primary determinant of trading success or failure, performance is still vitally important, underpinning efficient algorithmic execution and the timely computation of complex strategies.

FIX performance

The importance of performance, and in particular latency, to trading systems, and the widespread use of FIX for trading intersect in the area of central interest to us here, namely the performance of FIX implementations.

We noted above that FIX has been superseded in some areas with binary protocols, such as NASDAQ's OUCH. Some of the impetus for doing this is due to the somewhat complex design of FIX: it provides session control, application signaling, and business logic all in a single layer — functions which can be better served by being logically separated out. For example, OUCH defers session management to the SoupBin protocol, on top of which it is layered. Even the more recent revisions of the FIX standard have started to separate out session management into a distinct layer FIXT (FIX Transport). However the main reason that FAST and other binary-encoded alternatives to FIX were developed was for their performance characteristics, and in particular their ability to eliminate latency from the trading process.

Binary protocols generally do exhibit lower latency than FIX and, by virtue of making more efficient use of CPUs and bandwidth, can reduce the risk of software and network queues. As a result, one might be naively tempted to conclude that any trading performance problems experienced when using FIX can be remedied simply by switching to a binary protocol. A more subtle misconception would be to conclude that the performance management concepts and techniques we explore here are irrelevant for binary protocols. In fact, they are every bit as relevant and important; although the specific examples we use for illustration are tied to the details of FIX, the mechanisms and circumstances that are most detrimental to the operational health and performance of FIX connectivity can plague any electronic communication protocol. To use a crude analogy, although a Formula One race car can run rings around a family sedan, a flat tire will bring either kind of car to a screeching halt just as quickly.

Although the specific examples we use for illustration are tied to the details of FIX, the mechanisms and circumstances that are most detrimental to the operational health and performance of FIX connectivity can plague any electronic communication protocol.

Troubleshooting FIX Connectivity

In many respects, FIX is a relatively simple protocol to implement and manage. The simple ASCII encoding and flat tag-value structure of FIX messages means that it is easy to format them, to parse them, and to inspect them in their raw form.

Checking the FIX schema

The flexibility of the FIX format, along with its long pedigree, has resulted in multiple revisions and evolutions of the standard. In common use are FIX 4.2, FIX 4.4, and FIX 5.0 SP2 (Service Pack 2), as well as multiple extensions of these core standards via the use of user-defined tags, or indeed through any schema on which two counterparties might agree. Most FIX engines are built to accept schemas defining the set of tags to be used, usually in the form of an XML file defining the message types and the field tags supported. The schema can advertise the type of a field (STRING, INT, UTCTIMESTAMP, etc.), in what messages it appears, whether or not it is required, reflecting either the definition standardized by the community, or the local convention in use. Given these flexible but standardized ways of specifying the exact dialect of FIX to be used, most basic connectivity problems with FIX can be resolved simply by ensuring that both counterparties are using the same FIX schema.

"Most basic connectivity problems with FIX can be resolved simply by ensuring that both counterparties are using the same FIX schema."

Inspecting FIX messages

The most common form of FIX logging consists simply of the raw tag-value pairs concatenated to a file. For readability, some systems insert newlines in between individual messages, and some convert the standard FIX field separator SOH (ASCII value 0x01) into a pipe symbol | (ASCII value 0x7C). Such log files can provide a powerful way to troubleshoot FIX problems because they are fairly easy to inspect with a simple page or text editor, and simple enough to search using the editor's text search capabilities or even standard UNIX command-line tools such as grep. As a result, manual inspection of log files can provide a first port of call for retrospective troubleshooting of basic FIX connectivity problems.

"The most common form of FIX logging consists simply of the raw tag-value pairs concatenated to a file."

It is worth noting that this manual approach can rapidly become unwieldy or impossible in high-volume environments. Simple text-processing tools generally cannot cope with very large log files and basic regular-expression searching may not have enough discriminating power, or may be too slow, to execute to be practically useful. To wit, environments relying upon FIX or variants are likely by their business nature to be higher criticality and may not lend themselves to after-the-fact analysis and troubleshooting of degradation or disruption. In high-volume environments, a message store with suitable indexing of search criteria is likely to be required to enable effective troubleshooting — a traditional database is a natural choice for such a store, but note that it might struggle to ingest and index at the rates required.

“

In high-volume environments, a message store with suitable indexing of search criteria is likely to be required to enable effective troubleshooting — a traditional database is a natural choice for such a store, but note that it might struggle to ingest and index at the rates required.

”

Understanding FIX sequence numbers

The FIX standard does not specify what kind of transport is to be used to exchange messages. In practice, UDP multicast is often used for market data, while active trading sessions are usually connected over TCP. UDP provides no guarantees about the reliable delivery of packets and, even though TCP provides an automatic retransmission capability to protect against packet loss, it cannot provide any guarantees against disconnects (See Blog). As a result, FIX provides a reliable session model implemented using sequence numbers; in more recent versions of FIX, this is factored out into the FIXT (FIX Transport) layer.

Each outgoing message is given a unique in-order sequence number. These sequence numbers are used to detect, and optionally recover from, lost messages resulting from packet loss or session disconnects. Since a FIX session is bidirectional, there are two sets of sequence numbers to track, with each peer keeping track of the highest sequence number it has sent and the highest it has received. In the normal course of operations, a session may get disconnected, during which some messages that the sender has sent get discarded.

After a session has been disconnected and a new TCP connection is established, the initiator will send a logon message with its next sequence number. If this number matches the next sequence number the acceptor is expecting, then nothing from the initiator was lost during the disconnect and all is well. The acceptor then sends its own logon message with its next sequence number to the initiator, which performs the corresponding check. If either peer receives a sequence number which is higher than it was expecting, it knows that some messages were lost during the disconnect, and it will respond with a ResendRequest (MsgType=2). Its peer will either replay the lost messages or use a SequenceReset message (MsgType=4) to reset the sequence number to a higher number, which indicates that it cannot or will not replay the requested messages.

Interpreting the “sequence number too low” error

The above sequence number recovery is handled automatically by standards-compliant FIX engines and should rarely need attention unless there is a bug in the FIX implementation. However, another kind of error is worth understanding: when one side sends a logon message with a sequence number lower than previously received. This is an indication of serious confusion on the part of the sender about the shared session state.

In the worst case, this indicates that the sender has not persisted its state correctly (e.g., it crashed without fully flushing its log of sent messages to disk). It would be unaware of some of the messages it had previously sent. The reuse of old sequence numbers is an indication of a potentially critical business error and is not a condition from which a FIX engine can recover automatically. In these circumstances, the receiver must send a Logout message (MsgType=5) with a text message indicating the error, and manual recovery is required.

More often, this error arises in a much more innocuous fashion. FIX session peers typically agree on some schedule for restarting sequence numbers on both sides back to 1, usually before the first logon of the trading day. If this restart of expected sequence numbers is not properly coordinated, this too will result in the fatal “sequence number too low” error. In this case, the error message does not indicate a critical business problem and the situation can be remedied by a manual reset on both sides. Note that the SequenceReset (MsgType=4) message does not achieve this reset — it can only advance the sequence number, not restart it.

“

If this restart of expected sequence numbers is not properly coordinated, this too will result in the fatal “sequence number too low” error.

”

Managing FIX Performance

Establishing basic FIX connectivity is generally straightforward enough: once you have selected a solid FIX engine, agreeing the schema and sequence-number restarting schedule with your counterparty is generally enough to get you up and running. Managing the performance of your FIX implementation is, however, a much more challenging prospect. On one hand, the stakes are high as it directly supports the trading business, and the success and profitability of trades depends on reliable and timely execution of transactions over FIX. On the other hand, FIX itself provides little to no support for analysis or self-reporting of its performance, and practitioners are forced to use external systems and tools to remedy this.

“

Understanding the performance of any distributed system consists of first quantifying the two distinct, but related, dimensions of load and response. Load is how much work is demanded of the system, while response is how well the system handles that demand.

”

Understanding the performance of any distributed system consists of first quantifying the two distinct, but related, dimensions of load and response. Load is how much work is demanded of the system, while response is how well the system handles that demand. With accurate measurements available, you can then analyze the relationship between them for insights into managing system performance.

In the context of FIX-based trading systems, or trading protocols in general, the load is the rate of arrival of trade instructions, execution reports, and market data ticks, while the response is latency. Latency is the elapsed time between two causally related events, and is quantified by the difference between the timestamps of the two events; symbolically,

```
latency = time-of-effect - time-of-cause
```

Latencies of interest include overall response-times, such as the time from an order cancel request to the confirmation. Quantifying the component latencies of those response-times (such as processing times and network communication times) provides additional insight for managing and optimizing performance.

FIX timestamps

The essential foundation for a quantified understanding of the performance of FIX is timestamping. We must have accurate timestamps for each event to measure the rate at which events occur and the elapsed time between events. Unfortunately, it is in this respect that the FIX protocol is sorely lacking: the standards provide for very few timestamps for events of interest, and mandate only the poorest precision in, and no quantified accuracy of, those timestamps. There are only three commonly-provided timestamps that are relevant for performance purposes:

52	`SendingTime:`	time of message transmission
60	`TransactTime:`	when the advertised business transaction occurred
273	`MDEntryTime:`	time of Market Data Entry

Of these, SendingTime is the only one that is required. TransactTime is usually carried in messages reporting the results of a transaction, but MDEntryTime is not so consistent, often only appearing in full market data snapshots and not in incremental refreshes. Figure 1 shows where these timestamps are taken in some sample FIX transactions, with the grey columns representing counterparty systems and local applications that are communicating via FIX.

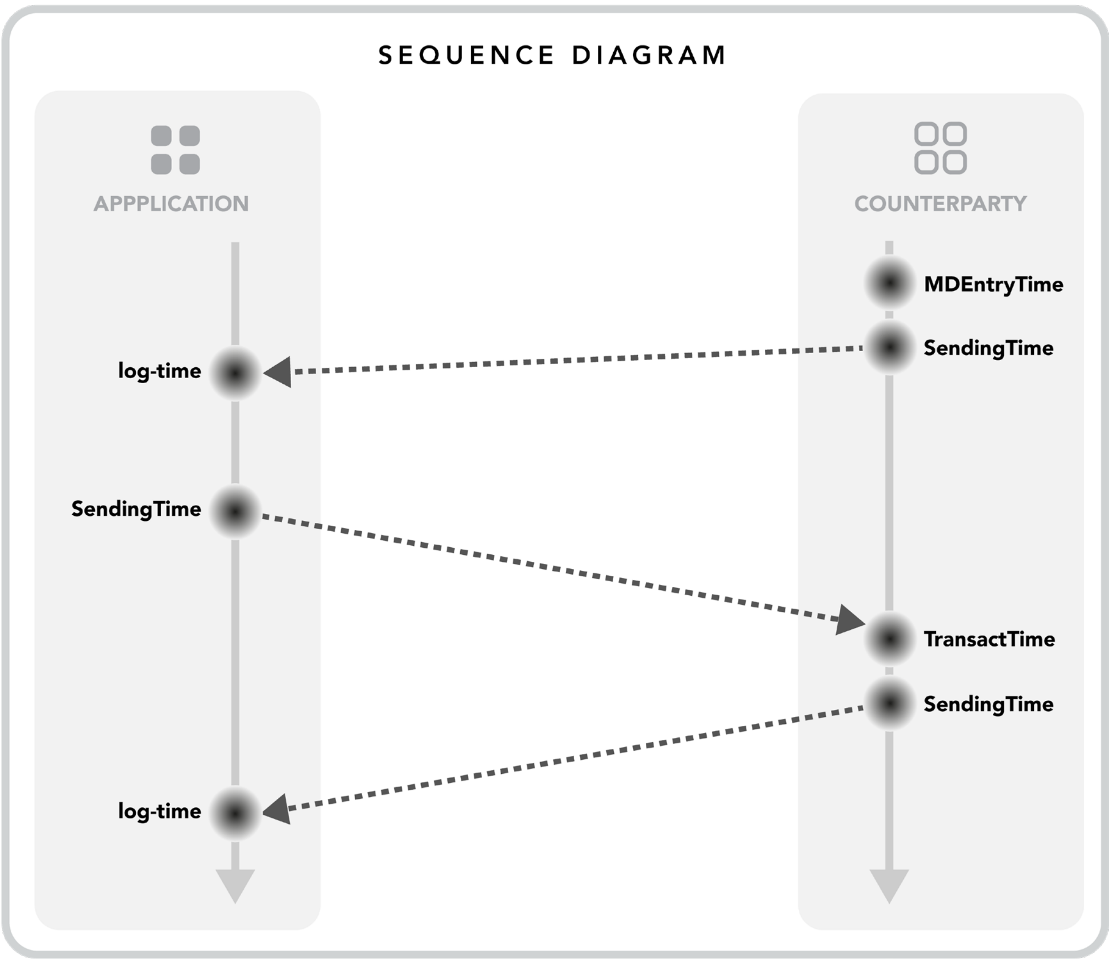


Figure 1. FIX timestamps

.

There are two further timestamps related to SendingTime, namely:

`122` `OrigSendingTime:` when a retransmitted message was originally sent

`370` `OnBehalfOfSendingTime:` when a message routed by a hub was sent to it

The former is useful when investigating for how long a session was disconnected, but the difference between `OrigSendingTime` and `SendingTime` is not really meaningful as a latency.

The latter is only rarely used and has been removed from FIX 5.0 SP2. There are several other time-related tags specified by the FIX standards, but most of them are used to specify the lifetimes of orders, the timing of trading sessions, or other events unrelated to performance. As a result, we effectively have just three FIX timestamps at our disposal.

"There are several other time-related tags specified by the FIX standards, but most of them are used to specify the lifetimes of orders, the timing of trading sessions, or other events unrelated to performance."

Measuring Load Using FIX Timestamps

The three FIX timestamps at our disposal can be used to identify how many events of each type occurred in specified time intervals. As a result, we can quantify the load on our FIX-based trading systems by measuring quantities such as the rate at which orders are being generated or filled, and the rate of market data feeds.

We mentioned before that FIX mandates only the poorest precision in, and no quantified accuracy of, those timestamps. Here we examine two scenarios where timestamping precision and accuracy in measuring load play a role.

Order-rate limits

Order rates have long been policed by some exchanges, which impose limits on how many orders per second members are permitted to send. A trading session that exceeds these limits may find its orders throttled, rejected, or even subjected to fines. Policing these limits, or ensuring one does not fall foul of them, requires careful measurement of order rates over the timescales specified — whether seconds or sub-second.

“We can quantify the load on our FIX-based trading systems by measuring quantities such as the rate at which orders are being generated or filled, and the rate of market data feeds.”

Market data microbursts

Market data is the fuel that powers most trading systems, providing the basis for trade-decision strategies and order-execution algorithms. Gaps in market data temporarily blind these engines, forcing them to wait for full snapshot refreshes. Stale price updates are even more pernicious, as an aggressive strategy will try to react to advantageous price opportunities, only to find that a competitor has beaten them to the punch. Similarly, stale pricing may cause an order-router may route orders to the wrong venue, in violation of regulatory best-execution obligations.

Market data is known not only for being voluminous, but also for being highly bursty. That is, not only is the total number of price updates per trading day constantly growing, but market activity is often clumped into short bursts of intense activity. Feed-handlers and other consumers of market data must keep up, not just with the average rate of market data, but with the peak bursts of ticks. If the processing capacity of a feed-handler is lower than these bursts of activity, it will fall behind the current market activity or even lose price updates entirely, resulting in missed trading opportunities.

The principles of queuing theory tell us that the appropriate timescale over which to measure these bursts is constrained by the amount of queuing in the feed-handler we are willing to tolerate. If we cannot afford for market data to be delayed by more than, say 1ms, then we must measure the rate to which market data bursts over a 1ms interval, and ensure the feed-handler has capacity to handle that rate. Measuring such short-timescale bandwidths, or microbursts, requires that the market data timestamp `MDEntryTime` have a commensurate precision and accuracy.

“

If we cannot afford for market data to be delayed by more than, say 1ms, then we must measure the rate to which market data bursts over a 1ms interval, and ensure the feed-handler has capacity to handle that rate.

”

Measuring Latency Using FIX Timestamps

Recall that latency is a key measure of the performance of a system and, in the trading context, of the competitiveness of the business processes it supports. Latency is quantified as the difference in timestamps of a pair of causally related events, and so FIX timestamps are a natural basis to consider for measuring FIX latencies.

Unfortunately, the three FIX timestamps available to us are generally unrelated to one another and are not directly usable to measure the latencies of most interest. However if our FIX logs also record the time at which a message was received (log-time), we can start to use the difference between that logged timestamp and one of our three timestamps as a measure of latency. For example, in Figure 2 (`log-time - SendingTime`) captures the FIX transport latency, which includes the network delay, as well as the latency through the network stack and the FIX stack on both the sender and the receiver.

One-way latency

This approach allows us to measure other one-way latencies. For example, if `MDEntryTime` is present in FIX market data messages, calculating (`log-time - MDEntryTime`) gives a measure of the market data latency (see Figure 2). This measurement would include the market data feed assembler latency and the FIX transport latency.

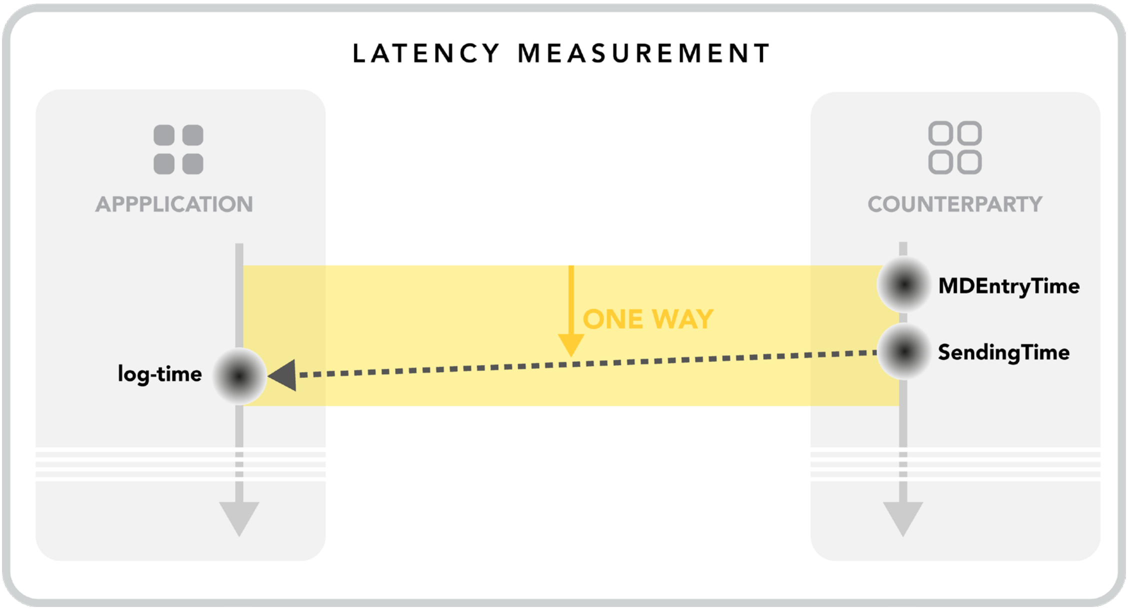


Figure 2. Measuring one-way market data latency

In cases where `TransactTime` is provided in a market data feed and `MDEntryTime` is not available, then `TransactTime` can serve to measure market data latency. In cases where neither is available, we can use the FIX transport latency as an approximation for the market data latency. As noted above, subtracting the `SendingTime` of a market data message from the time at which the message was received provides the FIX transport latency, but excludes the feed assembler latency.

Response-time latency

We can also extend this approach to response-time measurements. For example, we can measure the total time taken to cancel an order by subtracting the `SendingTime` of the OrderCancelRequest from the time at which the corresponding ExecutionReport was received – conceptually:

```
ExecutionReport.log-time - OrderCancelRequest.SendingTime
```

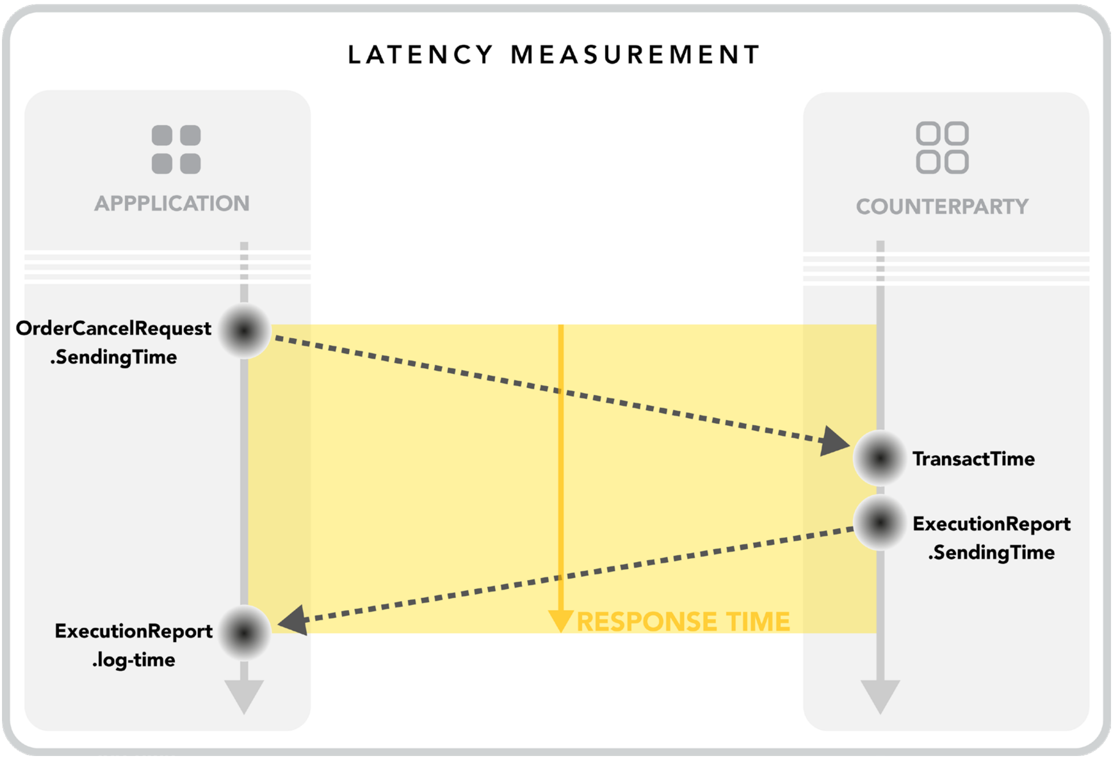


Figure 3. Measuring response-time latency for cancelling an order

However note that this calculation requires some non-trivial querying of the data: these two timestamps come from different messages, and the relationship between the messages is that they are carried on the opposite direction of the same FIX session and carry the same ClOrdID (tag 11). If we had the messages stored in a suitable database (even something as simple as SQLite), we might pull out the two timestamps with the following pair of SQL queries:

```
> SELECT SendingTime FROM messages WHERE MsgType = ‘F’
  AND SenderCompID = “Client” AND TargetCompID = “Server” AND ClOrdID = “o123”;
> SELECT log_time FROM messages WHERE MsgType = ‘8’ AND ExecType = ‘4’
  AND SenderCompID = “Server” AND TargetCompID = “Client” AND ClOrdID = “o123”;
```

For reference, as long as the timestamps are appropriately represented in the database, the latency could be pulled out directly by combining these queries as follows:

```
> SELECT rx - tx FROM (SELECT (
      SELECT SendingTime FROM messages WHERE MsgType = ‘F’
      AND SenderCompID = ‘Client’ AND TargetCompID = ‘Server’ AND ClOrdID = ‘id-123’
) AS tx, (
      SELECT log_time FROM messages WHERE MsgType = ‘8’ AND ExecType = ‘4’
      AND SenderCompID = ‘Server’ AND TargetCompID = ‘Client’ AND ClOrdID = “id-123”
) AS rx);
```

Of course, we don’t actually need to put the messages into a database to measure the latency but, in order to match up the execution report to the cancel instruction, we do need logic equivalent to these queries.

Request-response latencies

It is critical to know how quickly outstanding orders can be pulled or updated when reacting to fast-moving markets and synthesizing more sophisticated trading strategies, such as multi-leg orders. We highlight here some of the most important set of request-response pairs that should be measured, along with some comments on their business relevance:

1a. Order-to-ack:

From SendingTime of a NewOrderSingle to log-time of the corresponding ExecutionReport with ExecType=0 (New).

1b. MassQuote-to-ack:

From SendingTime of a MassQuote to log-time of the corresponding MassQuoteAcknowledgement.

Since these responses can be generated with minimal processing by the counterparty, these latencies assess the base FIX connectivity.

2a. Order-to-first-fill:

From SendingTime of a NewOrderSingle to log-time of the corresponding ExecutionReport with ExecType=F (Trade) and OrdStatus=1 (Partially filled) or =2 (Filled).

2b. Order-to-canceled:

From SendingTime of a NewOrderSingle to log-time of the corresponding ExecutionReport with ExecType=4 (Canceled).

These latencies are most meaningful for immediate-or-cancel (IOC) orders (TimeInForce=3) or fill-or-kill (TimeInForce=4), and also market-orders (OrdType=1), since the counterparty will take immediate action either way. It is possible to measure order-to-first-fill for limit day orders (TimeInForce=0) but this will not capture any response-time, instead simply reflecting market conditions (how long, if ever, it takes for top-of-book to meet the set limit).

3a. Cancel-to-confirmation:

From SendingTime of an OrderCancelRequest to log-time of the corresponding ExecutionReport with ExecType=4 (Canceled).

3b. Replace-to-confirmation:

From SendingTime of an OrderCancelRequest to log-time of the corresponding ExecutionReport with ExecType=5 (Replaced).

It is critical to know how quickly outstanding orders can be pulled or updated when reacting to fast-moving markets and synthesizing more sophisticated trading strategies, such as multi-leg orders.

4. QuoteRequest-to-Quote:

From SendingTime of a QuoteRequest to log-time of the corresponding Quote.

The time taken for the liquidity provider to respond to RFQs, as well as assessing FIX round-trip transport time, can be an important factor in the choice of which liquidity provider to use.

Two other measurements can be useful for monitoring the FIX transport time to and from the counterparty: cancel-to-pending and replace-to-pending. However, it is generally much more important for the business to understand how quickly orders can be fully canceled or replaced, and so those instructions are generally paired up with the confirmations to generate latencies, rather than the pendings (ExecType=6 or =E).

Caveats on FIX timestamps

There are a number of limitations on the value of FIX timestamps that it is important to understand. We discuss the top three here.

1. FIX timestamps are generally of unknown precision and accuracy

Precision:

FIX does support a flexible degree of precision; although the standard formats for the UTCTimestamp and UTCTimeOnly datatypes are either whole-second or millisecond precision, many modern implementations of FIX offer additional decimal places to represent microsecond- or nanosecond-precision timestamps. (A millisecond is one thousandth of a second, so millisecond precision is three decimal places; microsecond is six decimal places, and nanosecond is nine.) While the number of decimal places used certainly limits the precision of the timestamp, the converse is not true: a millisecond-precision timestamp can be presented in nanoseconds format simply by padding with six trailing zeros.

Accuracy:

More seriously, FIX mandates no accuracy for timestamps. Even if timestamps are presented with high precision, there is no guarantee that the clock from which the timestamp was generated is synchronized in any way to UTC.

As such, your mileage in using FIX timestamps may vary. This caveat must also be considered for logged timestamps, but they are generally easier to handle: logging formats are generally flexible enough that the timestamp precision is at least one microsecond and will almost always use the same system clock as the FIX engine. Whatever the accuracy of the local clock, logged timestamps can at least be accurately compared to local SendingTimes.

2. Problematic comparisons when FIX timestamps are taken from different clocks

The precision and accuracy problems are compounded when comparing timestamps taken on different systems. Taking the simplest example of market data latency, although both log-time and **`MDEntryTime`** are read from the same FIX log entry, they are taken from different clocks. An unusually high value of (**`log-time - MDEntryTime`**) could reflect a higher-than-usual delivery time for that tick, but it could also reflect data:

- A forward jump in the local system clock providing the log timestamp
- A backward jump in the clock used by the market data publisher FIX stack

Such jumps are a common occurrence when the system clock is managed by NTP (Network Time Protocol). Without some constraints on how the two clocks are synchronized, we cannot distinguish between these three possibilities.

3. **FIX timestamps are taken at relatively few points without location symmetry**

Perhaps the biggest challenge with making use of FIX timestamps is that there are relatively few points at which they are taken, and there is a lack of symmetry in their locations. For a market data tick, or other one- way transmissions, we can only rely on the **`SendingTime`** being present in all messages. As noted above, when **`MDEntryTime`** or **`TransactTime`** are available, these can also be used but they include more processing time and not just the FIX transport time.

For a request-response pair, we have more options, with four timestamps of interest: the local application **`SendingTime`**, the counterparty **`SendingTime`**, and the local **`log-time`**, and (if present) the counterparty **`TransactTime`**.

As we have seen, the local **`log-time`** can be matched up with suitable logic to the local application **`SendingTime`** to give the overall request-response latency. However, the **`TransactTime`** and counterparty **`SendingTime`** are less symmetrically arranged, and it can be hard to interpret them reliably in the absence of any understanding of the structure of the counterparty's system.

Counterparty SendingTime:

Since the counterparty **`SendingTime`** is the last application timestamp generated before an ExecutionReport is returned, (**`log-time - Counterparty SendingTime`**) is a useful measure of the latency of the FIX transport. However, there is no equivalent FIX timestamp that can be compared to our local application **`SendingTime`** to measure one-way latency to the counterparty. The counterparty could use their log-time for our application request to measure its transport time, but this is not shared in the matching ExecutionReport, so it is unavailable to us.

Counterparty TransactTime:

For a request-response pair, the first counterparty-side timestamp that could be available is **`TransactTime`**. For order instructions being sent to a central-limit order-book, this timestamp usually represents the matching-engine time at which the book is updated in response to the trader's instruction. Although it may represent the logical half-way mark of the total request-response cycle, there is no reason to expect that it will occur half way in time during the counterparty processing. As a result, there is little in general to be gleaned from the timing of `TransactTime` relative to the overall request-response timestamps (i.e. the local application's `SendingTime` and `log-time`).

However, it can be useful to keep track of (`log-time - Counterparty SendingTime`) along with (`Counterparty SendingTime - TransactTime`). If the overall request-response time is high, comparing these two latencies to their average behavior can be insightful:

- If (`Counterparty SendingTime - TransactTime`) is high, this indicates that something internal to the counterparty is driving the high response time.
- If on the other hand (`log-time - Counterparty SendingTime`) is high, then the problem is likely in the network or FIX stack.

Operationalizing FIX Timestamps

So far, we have discussed how timestamps provide the foundation for understanding FIX performance. We explored the timestamps that FIX provides and how to combine them to measure latency. These principles can be leveraged manually, providing the basis for troubleshooting individual transactions directly from FIX logs. Such a manual approach is inherently limited, and not just simply by virtue of the tiny number of transactions that can be dealt with by even a skilled expert.

These principles can be leveraged an automated way, with measurements made and recorded on a continuous basis. Operationalizing FIX timestamp analysis unlocks the full potential of these measurements. It is only with a comprehensive historical record that current measurements can be properly assessed, and meaningful baselines and thresholds established. More importantly, it is only with an accurate temporal record that the timeline of performance can be related to other infrastructure and business events. For example, surges in market data rates can be understood by relating them to events such as Fed announcements; the effectiveness of upgrades to a FIX engine can be assessed by tracking how much the latency drops at the time those upgrades are made.

“It is only with a comprehensive historical record that current measurements can be properly assessed, and meaningful baselines and thresholds established. More importantly, it is only with an accurate temporal record that the timeline of performance can be related to other infrastructure and business events.”

The collected data and analysis are useless unless they are accessible, and there are two primary modes of access to consider:

1. **Human access:**

   The best way to make data accessible to humans is visually and, in this context, the ideal vehicle for delivering this would be an interactive GUI equipped with widgets suited to presenting the different dimensions and aspects of the data:

   - Line and area-charts for time-series data
   - Big-numbers for compact single-value summaries
   - Pie-charts for at-a-glance breakdowns of composite values
   - Tables for sortable displays

Ideally, such a set of visualizations provides the ability to summarize meaningful ranges of time, such as the current trading day, and also to drill into specific short periods, such as around the market open. An example of a monitoring framework that supports a GUI with some of these capabilities is the open-source project Zabbix: https://www.zabbix.com/

2. **Machine access:**

   The best way to make data accessible to machines is through an open API, providing access to all the same data available through the GUI. In fact, the best way to build a GUI would be on top of such an API, and monitoring frameworks such as Zabbix use a flexible plug-in architecture to consume data from a variety of APIs.

Beyond access to data and analytics, the full operationalization of FIX performance management also demands the ability to schedule reports with GUI content (time-series and statistical summaries) and raise proactive alerts on performance problems — namely, spikes in load and/or latency.

Augmenting FIX Timestamps with Network Timestamps

On one hand, we have seen the potential for FIX timestamps to provide visibility and operational insight. On the other, we have seen how that potential is frustrated by the poor and uneven availability of those timestamps. The sometimes poor timestamp precision and accuracy only makes matters worse. This prompts us to ask if there is any way in which we can increase our reach with other timestamps of our FIX communications, and to improve the reliability and fidelity of those timestamps.

The answer lies in the network. There is a rich tradition of capturing timestamped network packets in service of performance monitoring and analysis, with supporting technologies to ensure the quality of the timestamps. Modern operating systems offer APIs to capture packets as they are sent to or received from network interfaces, with tcpdump being the classic example of software that uses these APIs to capture and filter network data of interest. For example, suppose we have a Linux machine hosting a FIX engine connecting to an exchange, liquidity provider, or other counterparty, on TCP port 4444. We can capture the packets carrying the FIX messages by running the following command in a shell on the machine hosting the engine:

```
$ sudo tcpdump -s 0 -w FIX.pcap -i eth0  tcp port 4444
```

(The interface name used here “eth0” may need to be changed to something appropriate to the machine’s networking setup.)

In the network itself, switches and routers support port-mirroring capabilities, where the network device can replicate some or all of the packets it forwards to a special mirror port (also known as a “SPAN” port). Many of these, notably products from Arista and Cisco, can also timestamp the mirrored packets in hardware. These ports typically feed “sniffer” or capture appliances, specialized servers dedicated to capturing and timestamping packets. Thus, whether through software APIs or network hardware, we have a range of options for augmenting FIX timestamps with network timestamps.

Hardware network timestamps

Hardware timestamping, whether done in the network fabric or by a dedicated appliance, offers important advantages over software timestamps:

- Hardware timestamps are captured at a very specific and identifiable point in the network, whether in the fabric of a switch, or the optical splitter in a tap. This eliminates any ambiguity as to where exactly along the transmission path the timestamp represents.

- A closely related advantage of using network packets as the basis of instrumentation is that they are a shared representation of the transaction being conducted. A software bug might cause a trading system to misrecord an order it has sent, but the packet it sends it in embodies the instruction as sent to the broker or exchange. As a result, a capture of that packet from the network provides the definitive record of that instruction.

- Hardware timestamping is either done in the fabric of the network switch, as part of the process of forwarding packets, or in a dedicated capture appliance out of band. As a result, it has absolutely zero impact on the forwarding of packets through the network, and neither reduces the bandwidth of the network nor adds any latency.

- Networks generally aggregate traffic to high degree in their core and so, by tapping or spanning core links, or cross-over points to third parties, we can capture and timestamp traffic between many systems at a single point. Instead of having to gather FIX logs from dozens of servers, we can capture equivalent (or richer) data just once.

- By virtue of being captured in hardware, network timestamps are very precise — 10ns or better with modern equipment. They are also generally highly accurate, as the hardware that captures them is designed to be synchronized with PTP (Precision Time Protocol) or 1PPS (one pulse per second), time distribution mechanisms that can deliver a time signal synchronized to within 100ns or better of UTC (Coordinated Universal Time).

- These mechanisms can also be used to discipline the operating system clocks of machines hosting trading systems — for example, PTP is designed to be distributed over standard Ethernet cabling. However designing, implementing, and operating a broad time distribution network is a demanding and costly challenge. A much simpler and more effective approach is to focus on distributing the time signal only to the devices that capture and timestamp packets.

Note that all these advantages accrue only to hardware-based network capture, and not to the host-based timestamping that tcpdump and similar software uses. The latter can be operated with minimal impact on their host machine, but its impact is non-zero. Furthermore, host timestamps need to be captured on every machine that FIX engines are running on, and suffer from all the precision and accuracy caveats as all software-based timestamps. Nevertheless, even host-based network timestamps captured with tcpdump or equivalent software are still unambiguous and authoritative — they are captured at a well-defined point in the operating system stack, and capture the transaction in-flight between counterparties.

Using network timestamps

The additional timestamps we can capture on the network are illustrated in an updated sequence diagram in Figure 4 (Page 25). These timestamps are taken on the network close to the application; for incoming messages, the network timestamp will be slightly earlier than the corresponding application timestamp, and slightly later for outgoing messages. These timestamps expand our horizons in two ways: they complement and improve on the FIX timestamps we already have, and they allow us to measure new latencies and shine a light into areas that were previously hidden from us.

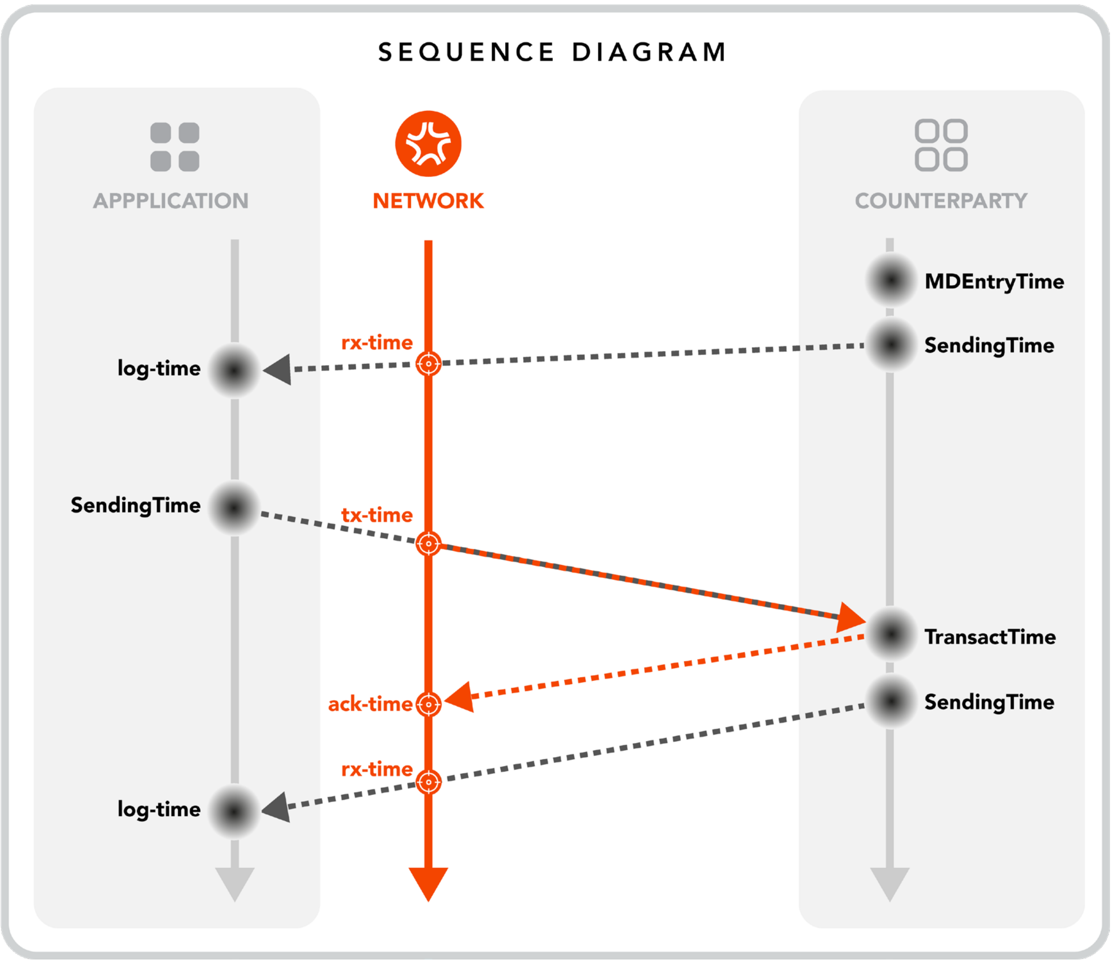


Figure 4. FIX and network timestamps.

Improves measurement of counterparty response time

The counter-party response time can be measured with the difference of two network timestamps, namely (`rx-time - tx-time`). In the case of order-to-ack (See 1a on page 16), this quantifies exactly how long it takes for a NewOrderSingle to reach the counterparty, processed, and for the ExecutionReport to return. It excludes the time spent in the application and local network stack and, by virtue of being based on network timestamps, is less ambiguous and more accurate.

Improves measurement of local application and network latency

Since we still have our FIX timestamps `SendingTime` and `log-time`, we can combine these with the network timestamps to measure how much of the total application round-trip time is due to our local stacks. In particular, we can break this down into the time taken to construct the order and publish it to the network (`tx-time - SendingTime`) and the time taken to receive the matching response from the network (`log-time - rx-time`).

Improves measurement of local application and network latency

The third network timestamp `ack-time` in Figure 4 is that of the TCP acknowledgment, which does not necessarily correspond to the transport of a message. TCP (Transmission Control Protocol) is a network protocol that implements its own sequencing and acknowledgment mechanism that is used to detect loss of network packets and arrange for retransmission where necessary. It is completely transparent to the application, and so the round-trip time quantified by (`ack-time - rx-time`) is a brand new measurement afforded to us by the network timestamps that has no application-level log counterpart. It quantifies the latency of the network connecting us to the counterparty, and comparing it to the network-derived response-time gives us a measure of the remote processing time: (`rx-time - ack-time`). .

Some caution needs to be taken when interpreting this latency. It is possible that the counterparty's TCP stack delays the ACK and piggy-backs it onto the packet carrying the execution report, in which case the `ack-time` and `rx-time` would be identical giving us an invalid measure of zero for the remote processing latency. There are techniques to allow delayed ACKs to be filtered out (U.S. patent number US8493875B2 and European patent number EP2234333), but these are beyond the scope of this document. Even when the NewOrderSingle elicits an immediate TCP ACK, we must be aware of the fact that the network transit time for the ACK may differ from that of the ExecutionReport. Nevertheless a systematic comparison of (`rx-time - ack-time`) with (`ack-time - tx-time`) gives a solid measure of the relative contributions of the network and counterparty processing to the overall transaction times. So, by gathering and analyzing network timestamps in their own right, and further combining them with FIX timestamps, we both unlock additional value from the latter and equip ourselves with far greater insight into the performance of the main building blocks of our trading infrastructure.

“By gathering and analyzing network timestamps in their own right, and further combining them with FIX timestamps, we both unlock additional value from the latter and equip ourselves with far greater insight into the performance of the main building blocks of our trading infrastructure.”

Network-level insight

Access to captures of network packets opens the door to a whole world of possibilities. Beyond our immediate need for accurate timestamps to replace or augment FIX timestamps, packet captures can enable insight into the performance of the network underlying our FIX transport. They can give us access to important events other than just the exchange of FIX messages, as we saw in the preceding section, where the time of receipt of a TCP ACK allows us to measure network round-trip time independently of the response-time of our counterparty. Examples of further insight available via an analysis of packet-captures include:

Reordered TCP packets:

Networks can reorder packets, but TCP delivers the transported data to the receiving application in-order. If TCP packets arrive out of order, they must be buffered until they can be put back in order, delaying any time-sensitive FIX messages carried in them. Furthermore, reordered packets are unusual, and nearly always diagnostic of deeper network issues such as packet loss or routing problems.

Retransmitted TCP packets:

TCP provides a reliable delivery model and will arrange for retransmission of packets that are not acknowledged in a timely fashion. Retransmission results from either packet loss or exceptionally high round-trip times, and causes the transport of FIX messages to stall and latency to spike up. The ability to detect and alert on TCP retransmission is valuable both for proactive diagnosis of FIX latency, and for monitoring of network health.

Packet loss:

Packet loss will trigger a TCP retransmission, which in turn will show up as packet reordering downstream of where the loss occurs. Note the two distinct ways in which the packet loss manifests itself:

1. Upstream, the lost packet is seen twice — once as the original transmission which was later dropped, and a second time when retransmitted.
2. Downstream, the retransmitted packet fills the gap that was left by the original loss.

This observation provides the basis for an important diagnostic: both reordered and retransmitted packets strongly indicate packet loss. Which of the two behaviors we see in our capture reveals whether the problem occurred upstream or downstream of our vantage point.

Microbursts:

We explained previously how short bursts in demand can temporarily exceed the processing capacity of a system, causing queuing and delay in input buffers. The same principles apply in packet networks: short bursts of network packets that exceed the bandwidth of the link they are routed over will be queued in the network device. These queues result in higher latency through the network and, if they fill the buffer holding them, packet loss.

We must consider the measurement location on the network, as it will influence our interpretation of the measured microburst profile (i.e., it's size, timeframe and shape). For the sake of illustration, consider market data traffic flowing from a 40Gb link through a 10Gb link and into another 40GB link (Figure 5). Assuming we cannot afford for traffic delays of more than 1ms, therefore we are measuring how high the traffic bursts over every 1ms interval.

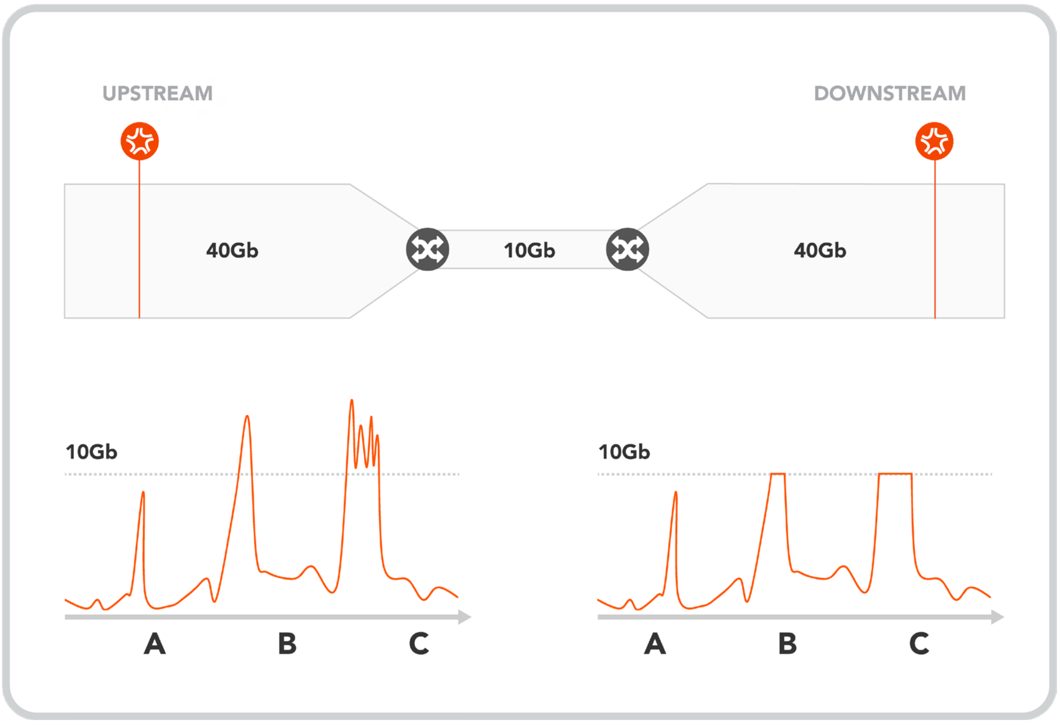


Figure 5. Microbursts measurements made at different points on the network

In the upstream profile, microbursts characterize the ability of the packet stream to cause congestion on the 10GB link, in our three examples:

A. The microburst bandwidth is below the link capacity, any queuing on the link will result in delays no longer than the preferred 1ms
B. The microburst bandwidth is above the link capacity, but as a single blip the network buffer may be able to handle the excess and deliver the market data without gaps, but with higher than preferred latency.
C. Sustained period of bursts above the link capacity almost always will overwhelm the 10Gb link causing market data gaps

In the downstream profile, it is easy to see how B and C have been shaped by their passage through the 10GB link. The "flat top" patterns are evidence that the 10Gb link was saturated, i.e., sending packets at its maximum rate. Extended busy-periods are indicative of heavy congestion with increased likelihood of dropped packets. However, it is less obvious from the profile pattern whether the 10G link was overwhelmed without the upstream profile for comparison or additional correlation with sequence gap analysis.

The topic of network traffic analytics is broad and deep, and not one we shall attempt to do any justice to here. However, we hope that these examples of network-level insight provide a flavor of what is possible with a suitable analysis of timestamped network packets.

Handling network packets

There is one critical task that we have glossed over in our discussion of network timestamps and how to combine them with FIX timestamps; that is the task of identifying which packets (timestamped with `tx-time`, `ack-time`, or `rx-time`) correspond to which messages (timestamped with `SendingTime`, `TransactTime`, and `log-time`).

In the very simplest case, each message will be sent in its own network packet, and a little searching through a pcap file will help locate the packet; for example, given the FIX.pcap we captured with tcpdump at the beginning of this section (Page 23),

```
$ tcpdump -nX -r FIX.pcap
```

will dump the timestamps of each packet, along with a somewhat readable rendering of any FIX content. If you try this, you will soon appreciate that, while this gives a quick-and-dirty way of finding the timestamp of a given message, it is quite impractical.

The core problem is that the network is quite a hostile environment from which to attempt to extract application visibility:

- One immediate problem is how FIX is encoded: while the traditional ASCII-based tag-value format of FIX is reasonably close to plain text, the field separator ASCII SOH is not printable and can throw off text-processing tools. FIXML (an XML schema for FIX) has all the readability of XML — which is to say none at all — while SBE (Simple Binary Encoding), like its predecessor FAST, cannot be interpreted at all without a software decoder equipped with the schema used for encoding.

- TCP segmentation and reassembly poses a broader difficulty — one that is not specific to FIX but is common to any attempt to extract application messages from data carried over TCP connections. TCP does not have any understanding of the boundary between successive application messages written to a socket, and simply treats the data to be transported as an undifferentiated stream of bytes. As such it will chop up that stream into packets as it sees fit, so that any given TCP packet may contain more than one FIX message, just the first bytes of a message, the remainder of a previous partially-transmitted message, or all of the above.

- Furthermore, the network may drop packets, forcing TCP to retransmit them; depending on whether the packet drop happened upstream or downstream of our vantage point on the network, we may see messages arriving multiple times (upstream) or out of order (downstream). A careful analysis of TCP sequence numbers is required to perform the clean reassembly of the byte stream that the receiving TCP stack must also do before delivering the complete and in-order data to the receiving application.

- The sheer scale of network data amplifies the previous problems. While a single TCP connection can only transmit data as fast as the applications driving the FIX session can process, a network tap or port-mirror can capture many thousands of connections pushing millions of FIX messages every second. In troubleshooting a specific problem, we may only be interested in a handful of messages on a single FIX session but, in order to identify that session and those specific messages, we are likely to need to:
    - » Track the state of every TCP connection
    - » Reassemble all their byte streams
    - » Decode all the constituent FIX messages

  Only then can we begin to search for the few messages we need.

Each of these problems is challenging in its own right and requires careful engineering to solve. Furthermore, each problem is orthogonal to the others, with no magic bullet that can tackle all simultaneously. Crafting a system capable of solving these compounded problems is a tempting challenge for talented engineers, and many trading firms have attempted this. Ultimately most have come to the conclusion that it does not make economic sense to do so. Instead they turn to a vendor such as Corvil, who has a proven track-record of delivering and supporting a complete professional-grade solution.

“Crafting a system capable of solving these compounded problems is a tempting challenge for talented engineers, and many trading firms have attempted this. Ultimately most have come to the conclusion that it does not make economic sense to do so.”

Indicative Scenario

We have discussed the principles of FIX performance management, covering the foundational importance of timestamps and how to weave them into insights into application and network performance. Let us now explore how they play out in a concrete scenario.

We use a two-part dataset drawn from a FIX environment as the basis for our illustration, consisting of a log-file and pcap (network packet-capture file) instrumenting a FIX trading application. The application issues aggressive IOC orders to a trading venue, and is generally quite successful in getting its orders filled, but occasionally it suffers from significant drops in fill-rate for no apparent business reason.

The scenario is partly synthetic, in that both the application's trading behavior and the market it trades against are artificial; however the infrastructure is real, with the application and the venue gateway running full FIX stacks, and connecting over a real network. More importantly, the phenomena that we observe in the FIX log and packet capture, and the root-cause that we uncover, are very real -- they can and do happen in real trading environments.

1. As a preliminary step, use the FIX-to-SQL.py python script to load the FIX log into an SQLite database.
2. Now explore the fill-status of the logged orders, counting how many were filled and how many were killed.
3. Select the fill-rate as a percentage, and plot over time; notice the significant drop at [time].
4. Explore that time-range in more detail, and note the series of orders in a row that were unsuccessful.
5. Copy the ClOrdID of the first of these orders to the clipboard.
6. Open the pcap file in Wireshark, and ensure the FIX plugin targets the TCP port used by the FIX server at the venue (port 4444).
7. Use the Wireshark filter box to search for the ClOrdID we copied earlier:

   a) `frame contains "11=o123"`
   b) `fix.ClOrdID == "o123"`

8. Note the TCP sequence number of the packet carrying the NewOrderSingle, and find how long it takes for the venue to ACK that data -- the difference of Xus between the two timestamps far exceeds the typical order-execution time.

9. A deeper look at the sequence numbers of subsequent packets shows out-of-sequence packet, with retransmission of packets out to the execution venue, and reordering of the return packets. As we saw in the network-level insight section (page 25), this clearly identifies that there is loss in the network leading to excessive FIX transport latencies.

10. It can be difficult to identify the cause of packet loss in the network, but microburst measurements are an important means of diagnosing the number one cause, namely link saturation. In this case, the subnet from which all packets from the venue are sourced is 10.0.0.x/24, and the microbursts generated by this subnet frequently flatline at 1Gb/s, the capacity of the cross-connect to the venue. In particular, the microburst measurements reveal significant link saturation exactly surrounding the times that TCP connections start to retransmit.

Putting all of this analysis together provides compelling evidence of the root-cause of the drop in fill-rates, and points to the steps that can be taken to remedy the problem.

Diagnosis:

Market data bursts in excess of the capacity of the network link to the exchange drive congestion and buffer overflow; the resulting packet loss stalls the TCP connections underlying the FIX connectivity, which causes the orders to miss the trading opportunity they were targeting, as manifested in the drop in fill-rates.

Remedy:

Upgrade link to accommodate market data bursts, or route order-flow over a separate dedicated circuit to avoid the congestion driven by market data. Note that a link upgrade is by far the superior option, as the congestion and packet loss also compromises the health and integrity of the market data feed.

Recap and Conclusion

The health and success of an electronic trading business depends on the performance of the systems that operate it. The performance of trading systems can be understood by analyzing the two primary dimensions of load and response -- load being how much work is demanded of the system, and response being how well it handles that demand -- and the relationship between them. These dimensions can be quantified with measurements of message- and bit-rates at short timescales, and of latency.

FIX provides some timestamps that can be used to provide these measurements, but there are some important caveats; briefly, FIX timestamps are of unknown precision and accuracy, and are not always available at ideal vantage points. Network timestamps, which are generally far superior in these respects, can be used both as a replacement for FIX timestamps and in conjunction with them for more accurate and refined visibility. Network timestamps also provide access to events that occur purely at the network layer and are invisible to applications, such as TCP stalls, packet loss, and network round-trip latencies.

Extracting application messages from network packet captures is a challenging task, requiring a careful handling of the TCP segmentation and reassembly process and the ability to deal with huge volumes of high-speed data. However the ultimate benefits are well-worth the effort, enabling a full forensic analysis from business-level symptoms, through the diagnosis of the underlying infrastructure problem, to an identified remedy.

Beyond FIX

Although we have focused on FIX, all of the principles we have explored here apply equally well to any other trading protocol. The specific details will differ, but these differences will mostly be limited to the naming of message types and fields and how that content is encoded for transmission across the network. Some protocols (notably Deutsche Boerse's Eurex ETI) offer better quality timestamps, and more of them, but these timestamps are complemented and augmented just as powerfully by network timestamps. Network visibility remains the gold-standard for gaining visibility and insight into the performance of the trading stack, and its impact on the business.

Corvil

Corvil is the leader in performance monitoring and analytics for electronic financial markets. The world's financial markets companies turn to Corvil analytics for the unique visibility and intelligence we provide to assure the speed, transparency, and compliance of their businesses globally. Corvil watches over and assures the outcome of electronic transactions with a value in excess of $1 trillion, every day.