XMIT Specification

alpha15 - 2014-09-05

XMIT defines lightweight logical sessions that are used to carry application messages. XMIT defines a set of basic patterns for communication modes and reliability guarantees. XMIT is designed to meet the requirements of automated trading in the financial markets.

Contents

Appendices

1 Alpha Version Document Notice

NOTE: The XMIT specification is not stable yet.

2 Overview

XMIT defines lightweight logical sessions. A session is used to carry application messages. XMIT is responsible for using the selected communications mechanism to deliver messages orderly from one application to one or more other applications in the context of a session.

XMIT is named after the abbreviation of *transmit* and is pronounced X-MIT.

XMIT is designed to meet the requirements of automated trading in the financial markets. These include the use of open protocols, simple design, freedom from being tied to a particular messaging library implementation, and eliminating unnecesssary overhead that affects communications latency.

2.1 Supports Common Workflows

XMIT is designed to support basic communications patterns, primarily targeting automated trading workflows.

XMIT defines two basic asynchronous communications patterns - point to point and one to many. Applications can synthesize more involved synchronous or asynchronous patterns from these two basic patterns.

XMIT supports multiple delivery contracts.

- *Recoverable* - traditional reliable messaging in the form of a bidirectional message queue.
- *Idempotent Operations* - where messaging is defined in terms of operations to be carried out. XMIT ensures that each operation is applied at most once. XMIT does not queue. Instead, it lets the application decide to retry an operation that is about to be delayed. The return flow, comprising the server messages that the operations result in, is recoverable.
- *Application Defined* - XMIT uses an unsequenced transmission in both directions, leaving control to the application.

XMIT can be used to communicate the same messages to multiple receivers. When multicasting messages using UDP, data groups or topics at the application level are sometimes more fine grained than the grouping that is available at the network level. XMIT enables flexible configuration of application topics onto network groups. XMIT does so by decoupling topic definition at the application level from multicast addressing at the networking level, allowing multiple application topics to be multiplexed over a single network address.

2.2 Session Lifetime

Logical sessions are defined online, eliminating the complexity involved in recycling static session definitions. The identifier of a session is universally collision free, which eliminates the need to manage sessions manually. It is sufficient to manage how sessions are allowed to be authenticated and how an XMIT session is related to the corresponding application level session. There are hooks for applications to control the creation and authentication of sessions.

For stacks that do not require a separate encoding of the OSI Session Layer implementation, like Blink, XMIT can be tunneled over the presentation layer without any overhead. For other stacks, XMIT specifies requirements on a concrete OSI Session Layer encoding.

The session creation and establishment procedures can be authenticated.

2.3 Application Independency

An application message does not need to have a session header and does not need to implement or subclass a session layer entity. This means that any message can be an application message and that any message can be passed over XMIT non-intrusively.

2.4 Minimized Overhead

No control data is specified per application message. XMIT functions by injecting control messages in the stream of application messages. XMIT adds a minimal overhead to a conversation. It is mostly silent during the transmission of an application message flow.

2.5 Efficient Use of the Network

XMIT supports [UDP] and [TCP], as well as other communication mechanisms. UDP is unreliable. TCP is reliable but not always connected, which means that any queuing has to be done in parallel with the one done inside a TCP implementation.

UDP is lightweight and reduces the complexity in a high performance environment. UDP also provides a higher transparency as it does not have algorithms that require opaque queueing inside.

> **NOTE**: XMIT supports passing a message over a datagram transport only if the message fits in a single datagram.

TCP offers congestion and flow control, which make it a better option for communications between data centers, unless bandwidth is guaranteed.

XMIT also works with other messaging primitives, like ones based on inter-process communications [IPC].

2.6 Protocol Layering

In terms of the OSI model [OSI], XMIT is an implementation of the session layer.

While carrying out session layer functions (OSI Session Layer), XMIT is working at its simplest when relying on a unified encoding (OSI Presentation Layer), and is tunneled to appear at the same level as application messages (OSI Application Layer).

This specification is made in terms of Blink [BLINK], but it is straightforward to define support for other encodings. There is a Blink schema that specifies the XMIT messages in [SCHEMA].

2.7 History

XMIT draws upon the design of the Soup, Mold, and UFO session protocols defined by Island and later NASDAQ. XMIT provides a combined feature set in a single general purpose protocol and adds features such as the negotiation, in-band creation, of new sessions.

XMIT was designed and prototyped at Pantor Engineering in 2013. At Pantor, it is used to network distributed systems and as the session layer in internal and external high performance trading and market data interfaces. XMIT is also used for custom-built integration interfaces.

2.8 References

SCHEMA	http://blinkprotocol.org/s/xmit.blink
BLINK	http://blinkprotocol.org/spec/BlinkSpec-beta3.pdf
RFC4122	http://tools.ietf.org/html/rfc4122
TCP	Transmission Control Protocol
UDP	User Datagram Protocol
IPC	Inter-Process Communication
IP MC	IP Multicast
OSI	ISO standard ISO/IEC 7498-1:1994
IDEMP	Idempotence means that an operation that is applied multiple times does not change the outcome, the result, after the first time.
GITHUB	https://github.com/pantor-engineering/xmit

2.9 Blink Relationship

In this specification, XMIT messages are defined in terms of the Blink schema. Blink meets XMIT requirements on message framing and identification natively, without the use of a session layer message header. XMIT is not part of the Blink protocol or vice versa. The role of XMIT in a protocol stack is outlined in Section 12 (page 12) .

2.10 Implementation Independency

XMIT is straight forward to implement and is not tied to a reference implementation or platform. Pantor Engineering AB provides open source implementation of XMIT that is available at [GITHUB].

3 Session

A new session can be created at any time, using an XMIT in band negotiation protocol. The session identifier is suggested by the client and negotiated with the server. The use of the session is separate from the act of creating the session.

The negotiation of a session *involves the application* in validating the credentials of the negotiating client.

A session can be established only after it has been negotiated. The session can be terminated and re-established without any additional negotiation. The server application has the chance to perform authentication during session establishment, in addition to any authentication carried out during the negotiation of the session.

3.1 Identification

```
Uuid = @blink:type="UUID" fixed (16)
```

A session has no predefined duration. A session is abandoned and replaced with a new session where some other session level protocols would instead have a mechanism for reusing the existing session definition. The key to the XMIT model is its use of an easy-to-allocate surrogate session identifier that has no semantic meaning. In other words, it uses an identifier that is not defined and used by humans.

3.1.1 Unique, Distributed Allocation

A session is defined by its universally unique identifier (UUID). A session identifier is a 128 bit entity that is supposed to never be reused anywhere and to be collision free. The UUID is allocated by the client before it attempts to negotiate the session. The allocation scheme is to generate a random UUID according to UUID Version 4 [RFC4122].

3.1.2 Rationale

The benefit of using an UUID instead of a sequence value is that it is effortless to allocate in a distributed system. It is also simple and efficient to hash and therefore easy to lookup at the endpoints. The downside is a larger size overhead. The identifier however does not appear in the stream once a session has been established and bound to an underlying transport that identifies the session. When sessions are being multiplexed onto the same transport session, the identifier appears whenever the sender switches into a new session, at minimum once per datagram when using UDP.

> **NOTE**: XMIT has an optional optimization of the identifier size that can be used when multiplexing sessions.

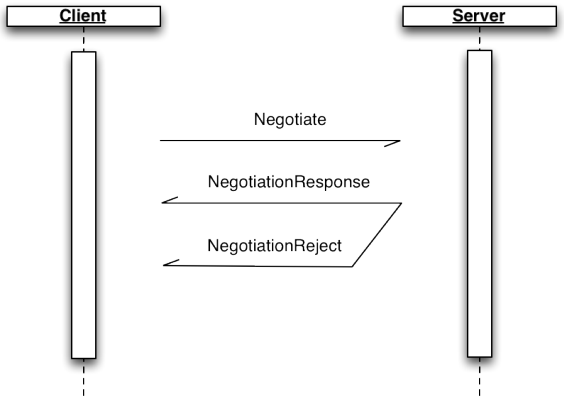
3.1.3 Lifetime

The lifetime of the session is until both flow directions have ended (finalized). For instance, in a financial trading session that has a daily schedule, a session would typically be allocated once per day. If there is an unrecoverable problem with the session, it would however be abandoned and replaced with a newly negotiated session.

3.2 Creation / Negotiation

A client creates a session identifier and initiates the negotiation of that session by sending *Negotiate*. XMIT leaves authentication credentials to be defined by the application via a generic *object* typed field.

Figure 3-1 *Session Negotiation*



3.2.1 Negotiate

The client selects a *Timestamp* that it will not reuse in another *Negotiate* message, and allocates a random UUID for the *SessionId*. It defines the type of flow it will be using in *ClientFlow*. The client optionally selects *Credentials* as defined by the rules of engagement that are in place. Standard credentials models are defined in Appendix A (page 13) .

```
FlowType =
          Recoverable | Unsequenced | Idempotent
```

```
Negotiate/0x10000 ->
          nanotime Timestamp,
          Uuid SessionId,
          FlowType ClientFlow,
          object Credentials?
```

3.2.2 Negotiation Response

The server responds with a *NegotiationResponse* or a *NegotiationReject*. The session is referenced via its *SessionId*. The specific negotiation request is referenced via *RequestTimestamp* that matches the *Timestamp* field in the *Negotiate* request.

If the server accepts the session, it will declare the flow type it will be using in *ServerFlow*.

The server will accept the session again if the negotiation request is repeated.

```
NegotiationResponse/0x10001 ->
          nanotime RequestTimestamp, # ref Negotiate
          Uuid SessionId,
          FlowType ServerFlow
```

3.2.3 Negotiation Reject

If the server rejects the session, it will specify a *Reason* in free text form.

```
NegotiationRejectCode = Credentials | Unspecified |
 FlowTypeNotSupported
```

```
NegotiationReject/0x10002 ->
          nanotime RequestTimestamp, # ref Negotiate
          Uuid SessionId,
          NegotiationRejectCode Code,
          string Reason?
```

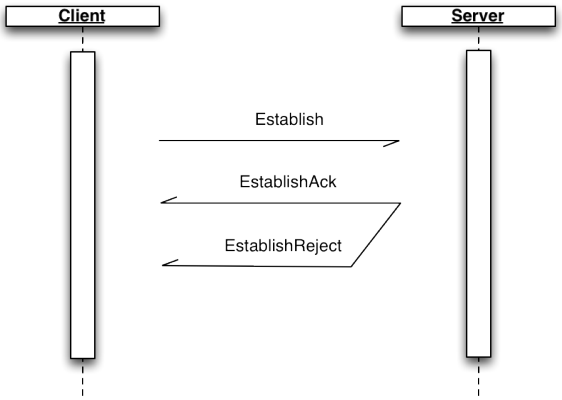
3.3 Establishment and Re-Establishment

A client attempts to establish a session, which is the same as saying that it binds a logical session to an underlying transport, by sending an *Establish* message. The same message exchange is used the first time the session is established and any later time it is re-established.

The server responds with an *EstablishmentAck* or *EstablishmentReject* message.

As described in Section 8 (page 8), multiple logical XMIT sessions can be multiplexed over a transport by repeating the exchange of establishment messages but using a different session identifier.

Figure 3-2 *Session Establishment*



3.3.1 Establish

The client selects a *Timestamp* that it will not reuse in another *Establish* message, and selects the *SessionId* for the logical session it wants to establish. It optionally selects *Credentials* as defined by the rules of engagement that are in place. For simplistic credentials implementations, it does not provide any value to specify *Credentials* in the establishment phase in addition to the negotiation phase. The field is left as a hook for more involved credentials mechanisms. In *KeepaliveInterval*, the client informs the server the longest it will remain silent for before sending a keep alive message.

If the client is producing a recoverable flow, it will declare the sequence number of the next message it will produce in *NextSeqNo*. This sequence number is unrelated to what, if any,

messages have previously been sent to the server. It allows the server to immediately initiate message replay if there are messages that it has not received. For an idempotent flow, the field is not set as messages are not recoverable by the server.

```
Establish/0x10010 ->
          nanotime Timestamp,
          Uuid SessionId,
          DeltaMillisecs KeepaliveInterval,
          u64 NextSeqNo?,
          object Credentials?
```

```
DeltaMillisecs = u32
```

3.3.2 Establishment Ack

The server accepts the session by responding with an *EstablishmentAck* message that references the session *SessionId*. *RequestTimestamp* references the *Timestamp* of the *Establish* message being responded to. In *KeepaliveInterval*, the server informs the client the longest it will remain silent for before sending a keep alive message.

If the server is producing a recoverable flow, it will declare the sequence number of the next message it will produce in *NextSeqNo*. It is used for the same purpose as the same field in the establish message.

If the client has made multiple attempts to establish the session, *RequestTimestamp* allows the client to determine which attempt the server is responding to.

Unless the session identifier and the request timestamp in the acknowledgment match a request, the acknowledgment *must* be ignored and an internal alert may be generated.

```
EstablishmentAck/0x10011 ->
          Uuid SessionId, # Robustness, is redundant
          nanotime RequestTimestamp, # ref Establish
          DeltaMillisecs KeepaliveInterval,
          u64 NextSeqNo?
```

3.3.3 Establishment Reject

If the server rejects the attempt to establish the session, it will specify a *Reason* in free text form.

```
EstablishmentRejectCode = Unnegotiated
 | AlreadyEstablished | SessionBlocked |
KeepaliveInterval | Credentials | Unspecified
```

```
EstablishmentReject/0x10012 ->
          Uuid SessionId, # Robustness, is redundant
          nanotime RequestTimestamp, # ref Establish
          EstablishRejectCode Code,
          string Reason?
```

3.4 Termination

When a session has been terminated, it is no longer established, but can be re-established again.

To terminate a session is to break the binding that was created during session establishment, the binding from the XMIT session to the underlying transport.

An established session becomes terminated, stops being established, for one of the following reasons:

- If one of the peers sends or receives a *Terminate* message;
- or, if the transport level session is disconnected;
- or, as a result of the keep alive mechanism defined in Section 9.2 (page 9) is expiring its timer.

If the termination is not conveyed to the other peer, both peers will eventually consider the session to be terminated for one of the reasons listed. A server will not accept the session to be re-established if the session is still in the established state.

Session termination does not imply logical flow finalization, as defined in Section 3.5 (page 5) .

Session termination should ideally complete on both sides without having to wait for a timer to expire, with the exception that some communications or application problems can only be detected using a timer. A peer *must* therefore respond to *Terminate* with a *Terminate* response, unless one such message has already been sent after the session was established.

3.4.1 Terminate

```
TerminationCode = Finished | UnspecifiedError |
 ReRequestOutOfBounds | ReRequestInProgress
```

```
Terminate/0x10015 ->
          Uuid SessionId, # Robustness, is redundant
          TerminationCode Code,
          string Reason?
```

3.5 Finalization

Any flow is finalized orderly to ensure both sides know what the other side knows. This procedure is independent of, but is one of the causes of, the termination of the transport binding using *Terminate*.

The producer of a flow signals that the flow has logically reached its end by sending *FinishedSending*.

A producer of a finalized flow where the transport has not yet been terminated uses *FinishedSending* as the keepalive message.

In some scenarios, a peer cannot determine when the session has ended safely unless there is an procedure for orderly finalization. The receiving side therefore responds to *FinishedSending* with a *FinishedReceiving* message after having received all messages. When a peer has received and sent a *FinishedReceiving*, it *must* terminate with a *Terminate* message. If there is message loss during the termination protocol exchange, the session keep alive mechanism will tear down the session. The session will be in an unfinalized state and it must be re-established before a new attempt can be made to terminate it orderly.

3.5.1 Finished Sending

```
FinishedSending/0x10025 ->
          Uuid SessionId, # Robustness, is redundant
          u64 LastSeqNo?
```

3.5.2 Finished Receiving

```
FinishedReceiving/0x10026 ->
          Uuid SessionId # Robustness, is redundant
```

4 Message Flows

XMIT defines a direction to carry a flow of messages. The type of flow is independent of the type of the other direction in a point to point session. There are three flow types, *recoverable*, *idempotent*, and *unsequenced*.

The client defines the flow type it is going to use in the *Negotiate* message and the server defines its flow type in the *NegotiationResponse* messsage.

When an idempotent flow is configured, XMIT ensures that when a client requests the server to carry out an operation, the operation will neither be forgotten nor applied multiple times. This mode is used by clients that need to control whether to retry or cancel an operation that failed to reach the server on a previous attempt.

If an *idempotent* flow is negotiated by the client, the server will always use a *recoverable* flow towards the client.

One-to-many sessions can be recoverable or unsequenced. The behavior is declared using the *Topic* message. An example application of the unsequenced flow is to provide information snapshots without allowing historic values to be retrieved.

5 Recoverable Flows

The purpose of a recoverable flow is to allow the streaming of a *recoverable*, serially ordered, sequence of messages. The message sequence is fully defined at the XMIT level, in the session layer, using sequence numbers. Sequence numbers appear in the stream only when required for the endpoints to agree on the sequencing. The sequence number is passed in an *EstablishmentAck*, *Sequence*, *Context*, *PackedContext*, or a *Retransmission* message.

A recoverable flow starts at the sequence number one.

A recoverable flow is idempotent [IDEMP] because it has exactly-once semantics.

At-least-once semantics is *provisioned* by the retransmission of messages based on the detection of an out of order flow or the expiration of a timer set when the previous message arrived. There is a *guarantee* that a message is delivered including a verification to the sender for a session that *terminates at the flow level*.

At-most-once semantics is *guaranteed* by the sequencing of messages as this allows the receiver-side XMIT logic to order the incoming flow and filter duplicated messages.

Any message passed after *Sequence* is implicitly numbered, where the first message after Sequence has the sequence number NextSeqNo. Session level messages are not part of the recoverable flow.

5.1 Sequence

```
Sequence/0x01 ->
        u64 NextSeqNo
```

5.2 Sequencing

In a non-multiplexed flow that runs over a reliable transport, the sequence number of an application message is calculated implicitly from the one specified explictly during session establishment.

In a non-multiplexed flow that runs over an unreliable transport, the sequence *must* be established at the start of each datagram using a *Sequence* message.

5.3 Message Mode

The *Sequence*, *Context*, *PackedContext*, *EstblishmentAck*, and *Retransmission* messages are sequence forming. They define the implicit sequence numbering for subsequent messages.

Any non-XMIT message, as defined in Section 13 (page 12), has the next implicit sequence number and is dispatched to the application layer. An XMIT message does not have a sequence number.

6 Idempotent Flows

When a client uses the idempotent flow type, each application message becomes a request to the server to perfom an operation. XMIT ensures that an operation has the property that it will neither be forgotten nor applied multiple times. XMIT provides the option for the client application to prefer canceling an undelivered, potentially stale, operation to retrying it.

To guarantee idempotence, a unique identifier has to be allocated to each operation to be carried out. XMIT uses a sequence number for the identifier. The response flow must identify which operations that have been carried out. When the client negotiates a session with the idempotent flow type, the server must respond that its return flow is recoverable.

6.1 Definition of An Operation

Each individual application message is an idempotent operation and is identified using a sequence number that is unique within the context of the session. The sequence number is implicit and is defined using a *Sequence* message. The first message after Sequence has the sequence number *NextSeqNo*. The same lifetime rules apply for the implicit sequence number in the idempotent flow, as for the implicit sequence number in the recoverable flow.

The current version of XMIT does not provide a mechanism for composing an operation that logically comprises multiple application messages. To achieve atomicity over multiple application messages, the application layer has to package them into a single application message. The client application can also achieve atomicity over multiple operations by either using a reliable transport or using a datagram transport and sending operations that belong together in a single datagram. XMIT requires but does not control that the server application carries out all operations delivered to it.

6.2 Client Requirements

The client-side XMIT implementation is responsible for reattempting an operation that is not acknowledged by the server (at least once semantics). It is also reponsible for sending keep alive messages. The requirements are described in more detail in Section 9.4 (page 9) .

6.3 Server Requirements

The server-side XMIT implementation is responsible for discarding any operation that has a sequence number lower or equal to the operation that was carried out last (at most once semantics). The combination of at-most-once and at-least-once semantics provide exactly-once semantics, making any operation tagged with a unique identifier to be idempotent.

The server accepts an operation with a higher sequence number, even if there is a sequence gap. The server must notify the client that operations in the sequence gap will be discarded, by sending a *NotApplied* message. In other words, the server will apply a more recent operation that arrives before a previous operation that is delayed. When the previous operation finally arrives, it will not be applied.

If the server receives a *Sequence* message with a higher *NextSeqNo* than expected, the server has information about potential message loss. If the client sent the *Sequence* message as a heartbeat, no application message follows. The server must delay discarding messages in the potential gap until an actual application message arrives that makes the gap definitive. This means that the *Sequence* message does not function as a heartbeat message for recovery purposes. The reason for this behavior is explained in Section 9.4 (page 9) .

6.4 Algorithm Properties

This algorithm is designed to be simple and highly efficient. It provides clear semantics to the client in how it can issue operations concurrently and how to proceed in a failure scenario.

A consequence of the algorithm is that a lost operation will not hold up other processing if the client decides to issue operations concurrently. It is the responsibility of the client to hold back issuing any operation that has a dependency on a previous operation, or to transport the operations in a way that they are guaranteed to either arrive in order, or to not arrive at all. It can do so by either using a reliable transport, or by sending them in the same datagram using an unreliable transport. Consequences of the algorithm in terms of concurrency are described in more detail in Section 6.7 (page 8) .

6.5 Applied

The server confirms that a range of operations has been applied by returning an *Applied* message.

If the server receives an operation that it has already applied, there is no need to regenerate an applied message due to a lost signal scenario. The return flow is recoverable, message by message.

NOTE: The server-side application layer may identify to the client-side application layer which operations that have been applied, making the *Applied* message redundant. XMIT requires *Applied* messages to be sent explicitly, unless the parties use an application layer that overrides this requirement. The application layer specification should state explicitly in which scenarios an application message takes over the role of the *Applied* message, and when not. XMIT does not include a mechanism for specifying in band that the use of *Applied* has been selectively replaced with the use of a corresponding application message, as such a mechanism would introduce significant complexity while not necessarily being generic.

```
Applied/0x10201 ->
        u64 FromSeqNo,
        u32 Count
```

6.6 Not Applied

The server specifies that a range of operations has not been applied by returning a *NotApplied* message.

```
NotApplied/0x10202 ->
        u64 FromSeqNo,
        u32 Count
```

6.7 Concurrency

The client can issue multiple operations without waiting for previous operations to complete. This means that network transfer latency does not limit the client for as long as all messages arrive in order. If an operation arrives out of order and is discarded by the server, the server will send a NotApplied notification without a delay, allowing the client to reissue the operation quickly using a new sequence number.

The client ensures that all operations are applied in order by sending operations in controlled batches.

The client can ensure that an operation is applied in order while issuing multiple operations concurrently, despite the simplistic semantics guaranteed by the server. The base premise is that the client has to hold back sending new operations until previous operations have been applied. This limitation is lifted if a reliable transport is used. Also with a reliable transport, the client needs to able to reattempt operations orderly, as the transport may be disconnected and later reconnected.

6.7.1 Unreliable Transport

The limitation is worked around if an unreliable transport is used, by relying on datagram boundaries and the property that a datagram is either delivered completely or not at all. As all or none operations in a datagram will be delivered at the server, the client only needs to consider holding back subsequent datagrams until all operations in the previous datagram have been acknowledged. In other words, the client remains in full control while being able to have multiple operations in flight to the server, yielding a concurrency equal to the number of operations that fit into one datagram.

6.7.2 Latency Hiding

Total ordering with concurrency can be achieved without being limited by network or server latency, at the cost of increased complexity.

As only one datagram can be in flight, a long distance network will limit the effective concurrency available to a client that is waiting for acknowledgments before being able to issue the next round of operations.

A straight forward solution to avoid being limited by delays in this scenario is to use multiple independent XMIT sessions. XMIT is defined in a way that it is cheap to use multiple sessions, even allowing ones to be created dynamically.

6.8 Server-Side State And Efficiency

By requiring operation identifiers to be allocated in a strictly monotonic sequence, the server side can be optimized.

The server does not need to store the full set of identifiers for previously applied operations. It does not even need to store the set of identifiers that were not applied (note: it would have, if the idempotent flow was to support the use of an unsequenced, unrecoverable flow out of the server).

The flow outbound from the server is required to use the recoverable XMIT flow mode. This allows the server to rely on its outbound flow to be recoverable and forget state for any historic operations.

The server generates a NotApplied message for each expected operation that it did not see by the time it applies the next operation. The client will see the result of all operations including lost ones by processing the recoverable server message flow.

If and when the server finally receives the lost or out of order operation, the operation will be ignored and no new response will be generated for it.

7 Unsequenced Flows

The purpose of an unsequenced flow is to leave the control of messaging to the application.

8 Multiplexing Recoverable Flows

Sessions in a recoverable flow direction can be multiplexed over a connection. When multiplexing is active, the *Context* message is used instead of *Sequence*, to include information about what session that is being recoverable. *PackedContext* has the same semantics as *Context* and can optionally be used to reduce message size overhead. The *Retransmission* message also functions as a context message. Only these three messages sets which session that is active. Other messages that specify a session identifier do not imply a change of which multiplexed session that is active.

The sequence must be established at the start of each datagram or every time the flow switches to the next logical session.

8.1 Context

```
Context/0x10050 ->
        Uuid SessionId,
        u64 NextSeqNo?
```

PackedContext can be used when the next sequence number is inside a 32 bit range. The prefix is selected by using the first eight bytes of the identifier. The sender must ensure that the prefix uniquely identifies the session. The size of *PackedContext* is half the size of *Context*.

```
PackedContext/0x10051 ->
        binary (8) SessionIdPrefix,
        u32 NextSeqNo?
```

9 Error Detection

9.1 Silence Means Error

Both sides must send messages regularly to enable the other side to determine that the application is intentionally silent, so that complete silence clearly indicates that there is either a loss of communications or that the other side malfunctions.

9.2 Unsequenced Heartbeat

```
UnsequencedHeartbeat/0x10020
```

Each peer declares how often it is going to send keep alive messages during the establishment phase. The server can decide to reject the establishment attempt based on the interval selected by the client. It is up to each peer to determine the relationship between the announced keep alive interval and the length of the silence it accepts before terminating the session.

9.3 Recoverable Flows

In a recoverable flow, the *Sequence* message functions as a keep alive message in addition to its sequencing role. In case the transport is multiplexed, its *Context* sibling is used instead.

9.4 Idempotent Flows

In the scenarios below, it is stated that the client may reattempt an operation. In each of these cases, the client-side application has the opportunity to instead instruct the XMIT implementation to not reattempt the operation. In that case, the server will later go forward with the next future operation and acknowledge the discarding of one or more operations using a *NotApplied* message.

In addition to using a timer for sending messages to keep the connection alive, the client-side XMIT implementation is responsible for regularly reattempting unacknowledged operations.

An operation is determined by the client to remain in transit or remain in progress until it receives either an *Applied* or a *NotApplied* notification. If a message has been lost, and no new operations are being generated, there will be silence. The use of *NotApplied* messages means that some scenarios will be resolved without waiting for a timer to expire. For the remaining scenarios, a timer based retry mechanism is required. As stated before, the *Sequence* message may indicate to the server that there is a potential problem. If the server would decide to act upon that information and discard any potentially lost operations, the client would have the option to retry these operations using a new sequence number. To avoid having multiple ways of achieving similar semantics, this option is closed by this specification, and the simplest mechanism remains. That mechanism is to use timer based *Sequence* messages for indicating that the client is active, and timer based reattempts of operations.

The *Sequence* message is used as a keep alive message and must be sent by the client unless the connection would otherwise be silent, in accordance with the negotiated keep alive interval. The *UnsequencedHeartbeat* message is not allowed to be used. It could

serve the same role, but the *Sequence* message has the benefit of being more explicit and therefore more useful for any logging purposes.

9.5 Unsequenced Flows

For unsequenced flows, the session level does not use the keep alive mechanism to trigger recovery, but the application level is free to rely on sesion level messages as a trigger. The use of keep alive is therefore required also for unsequenced flows. In an unsequenced flow, an *UnsequencedHeartbeat* functions as a keep alive message. In the multiplexing case, such a heartbeat message implies a keep alive for all established sessions with an unsequenced flow and not only the active session.

10 Resynchronization

NOTE: Resynchronization is replay based in this version of XMIT. Any snapshot style resynchronization is left to the application to define. A future version of XMIT may include messages that allow application snapshot synchronization to be matched to sequence numbers in the XMIT flow.

A peer consuming a flow can request recoverable messages to be retransmitted by sending *RetransmitRequest*. The peer producing the flow responds with a *Retransmission* message before retransmitting messages, or with a *Terminate* message if the request violates the protocol.

Retransmitted messages are sequenced using the *Retransmission* message instead of using one of the *Sequence* or *Context* messages. In a datagram oriented transport, *Retransmission* is passed in every single retransmission datagram. In addition to reseting the implicit sequence number, the *Retransmission* message signals that the request has been or is being processed and also informs the consumer whether it needs to issue a follow up *RetransmitRequest* message.

10.1 Retransmit Request

```
RetransmitRequest/0x10021 ->
        Uuid SessionId,
        nanotime Timestamp,
        u64 FromSeqNo,
        u32 Count
```

10.2 Retransmission

```
Retransmission/0x10022 ->
        Uuid SessionId,
        u64 NextSeqNo,
        nanotime RequestTimestamp,
        u32 Count
```

10.3 Request Constraints

XMIT conveys sufficient information for a flow consumer to know what messages are available to be re-requested. If a re-request despite this specifies a range that is fully or partially out of bounds, the server will terminate the session with the *ReRequestOutOfBounds* code. As the session is re-established, the consumer will get a fresh reminder of what messages that are available and will get a new chance to avoid the error logic that made it send an uninformed *RetransmitRequest* in the first place. The server is free to take appropriate action, like generating an internal alert or blocking the session, if the problem reoccurs after the session is re-established.

10.4 Resynchronization Pacing

In a datagram transport without builtin flow control, like UDP, resending many messages in a short time may result in packet loss. *RetransmitRequest* and *Retransmission* messages must therefore be exchanged in a way that the correspondence is rate paced. Unless the underlying transport provides guaranteed delivery, the

sender *must* limit how many messages are retransmitted per request to what can be sent in a single datagram.

To successfully resynchronize with the sender, the receiver needs to take part in the rate pacing algorithm. It does so by issuing a new *RetransmitRequest* to get the next set of messages, as a reaction to a *Retransmission* response that does not close the message gap. A receiver *must not* send a new RetransmitRequest until the previous request has completed or until a timer has expired, indicating packet loss.

The sender will terminate the session with the *ReRequestInProgress* code if it sees a premature retransmit request. The reason for not allowing multiple retransmit requests to be in progress is that it would potentially disrupt the rate pacing algorithm.

Using a UDP datagram transport, the sender *must* bound each retransmission round to what can be sent in a single datagram. In a high latency network, this requirement severely limits the ability to take advantage of available bandwidth. To support high speed recovery in a high delay and variable bandwidth environment, it is recommended that the recovery is carried out over a transport with built in congestion management, like TCP. Alternatively, the sender implementation may include a congestion management policy that adapts the response size dynamically and accepts multiple requests to be in progress.

10.5 Concurrent Transmission

10.5.1 Multiplexed Recovery

A producer is free to multiplex the realtime message flow for one session with retransmission for another session. If multiple session flows are being retransmitted concurrently and these sessions are related at the application level, it is up to the consumer to control how the retransmission should be interleaved. The consumer may want to interleave its input in a logical input order to avoid having to excessively buffer the input for some session that is getting ahead of another. The consumer can do that by controlling the retransmission in as fine grained way as it desires. There is some complexity as only the application layer can determine the logical relationship between the multiplexed and translate that into sequence numbers for its XMIT session layer implementation. The alternative is not to control

10.5.2 Realtime Interleaving

A producer may also send the realtime flow concurrently with messages that have been re-requested on the same session. The receiver may elect not to queue the realtime data if the gap of messages to recover is large. It can later request these voluntarily dropped messages to be resent. There is a potential race condition in the resynchronization procedure as recovery nears completion and the receiver becomes ready to only process the realtime flow. The receiver *must* therefore start to queue incoming realtime data for processing, at the latest when it has had to issue a retransmit request for data that it dropped voluntarily earlier in the current recovery phase. The receiver *must* also queue incoming realtime data after the initial recovery phase has completed, in conjunction with session (re)establishment.

11 Messaging Patterns

XMIT provides two communications models primitively, asynchronous point to point and asynchronous one to many communications. Other common patterns, synchronous or asynchronous, can be synthesized from these two base patterns, at the application level.

11.1 Point To Point

A point to point session has a client and a server. The client initiates the establishment of a session against the server. After the session has been established, the relationship between the parties are peers, in the sense that there is no inherent client and server relationship beyond the session establishment.

The point to point session is full duplex, i.e., messages can be sent in both directions independently.

11.2 One To Many

An XMIT session can be producer defined instead of being negotiated by a client. The producer declares the session using a *Topic* message.

The *Topic* message binds a topic or logical channel to a session and also declares that session.

In the *Classification* field, the producer specifies or classifies the content of the flow. A topic can for instance be fully detailed market data updates for one financial instrument. The field has the type *object* and is defined by the application. In the financial instrument example, the object would identify the instrument and other aspects that distinguishes this topic, such as the detail level of the market data updates it conveys.

11.2.1 Topic

```
Topic/0x10003 ->
          Uuid SessionId,
          FlowType Flow,
          object Classification
```

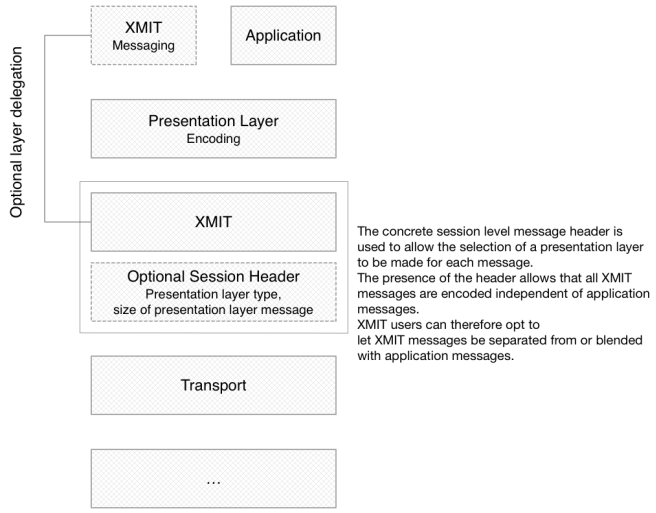
Recovery of a producer flow is initiated by establishing the session using the topic's SessionId against a point to point recovery service and then using the ordinary retransmission request mechanism. The producer can opt to perform the retransmission over the multicast flow or in a directed reply.

As a receiver may not have seen the initial *Topic* message for a session, it has to be resent with regularity or sent unsolicited in a response to *RetransmissionRequest* for the session.

12 Protocol Layering

Blink based protocol stacks are self sufficient in determining message framing and type identifiers. However, a session layer message framing can be used also with Blink.

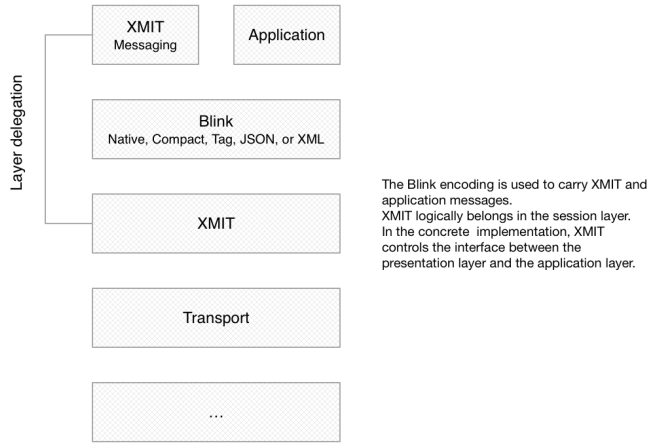
Figure 12-1 *Protocol Stack*



XMIT messaging can be delegated to use the same presentation layer that the application layer uses. In this mode, the XMIT implementation on each side is responsible for dispatching messages to the relevant layer. This mode is ideal in stacks using a unified encoding across layers that perform integrated layer processing. It allows the combined stream of session and application messages to be transformed into an alternative encoding without any loss in fidelity. This is is the preferred mode in a Blink based stack.

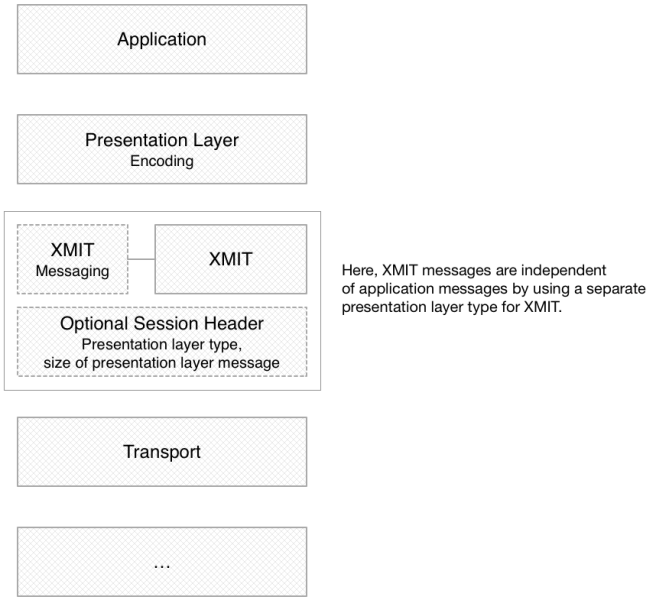
NOTE: With Blink, the delegation of messaging to the presentation layer is not performed using traditional tunneling. As XMIT is optimized for performance, it instead relies on letting session layer and application layer messages share the namespace of message type identifiers in the presentation layer. In Blink, this is done by reserving identifiers for session messages and letting XMIT perform message dispatching.

Figure 12-2 *Blink Stack*



The use of an optional framing protocol encapsulates the presentation layer, allowing the encoding for session messages and application messages to be separated.

Figure 12-3 *Switched Stack*



The framing protocol comprises a session layer header on each message. XMIT does not define a concrete header. The requirements on the header are defined in Section 13.2 (page 12) .

13 Reserved Message Identifiers

Unless a framing protocol is used to separate messages, XMIT shares the presentation layer with application messages. The XMIT messages therefore need to allocate type identifiers in the same namespace. The allocation is specific to each session layer.

13.1 Blink Message Identifiers

When XMIT is used with Blink, the message type id 1 and id 2 and the types in the range 10000_{hex} to 10210_{hex}, inclusively, are defined to be XMIT messages.

13.2 Session Layer Header

The optional XMIT presentation header is a concrete OSI layer 5 entity that wraps the presentation layer. Encodings, i.e., presentation layer implementations, are assigned a protocol number. Each message is prefixed with a presentation header that specifies the size of the message, including the presentation header. The protocol type determines which encoding that is used, allowing the encoding to be set individually for each message.

NOTE: This document does not specify a concrete header or a registry for protocol types.

Figure 13-1 *XMIT Session Layer Header*

Figure 13-2 *Framing Blink Native*

Figure 13-3 *Framing Blink Compact*

13.3 OSI Session Layer

Use of framing enables different encodings to be used in parallel. This means that XMIT control messages can be separated from application messsages, effectively turning XMIT into a traditional OSI Session Layer.

Figure 13-4 *Single Encoding Example*

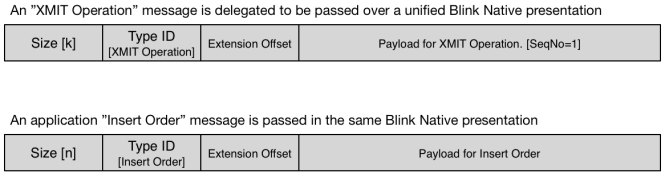
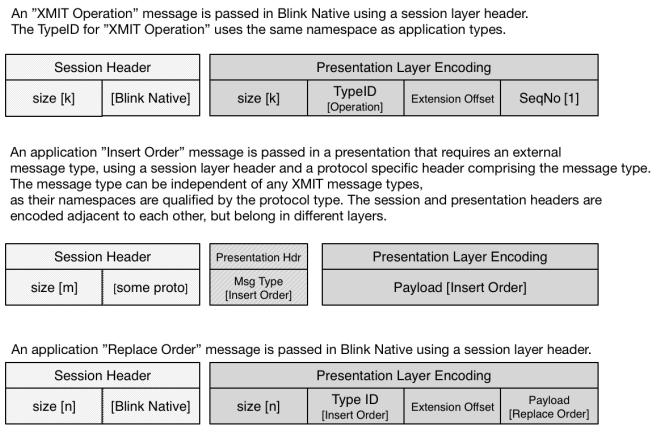


Figure 13-5 *Multi Encoding Example*



A Standardized Credentials Configurations

A.1 Minimal Identification

A minimal, polite use of the Credentials object is to identify the client creating the session using a string. Secure authentication is implicitly left to a separate layer, like using a secure VPN. The content of the string is implementation specific.

StringIdentification should not be used as a composite to pass a tuple of sub identifiers. Instead, the tuple should be modeled fully at the schema level, i.e., in a separate message.

It is not necessary to pass the identifer in the establishment phase, as it has already been passed during the negotiation phase.

```
StringIdentification/0x10170 ->
          string Identity
```

B Release Notes

The release notes start at version *alpha15*.

B.1 alpha15

Zhu Li provided feedback and several of the suggestions introduced in this release.

Don Mendelson came up with the improved naming of the *Recoverable* flow type.

- The flow type *Sequenced* has been renamed to *Recoverable*.
- Distinguish between flow termination and session transport binding termination better by calling the former flow finalization.
- Added a *NextSeqNo* to *Established* for better symmetry with the response message and making client message recovery more responsive, otherwise being delayed until the next new message is produced by the client.
- Define *FinishedSending* to be used for all flow types, making *LastSeqNo* optional for the unsequenced flow.
- Define *FinishedSending* to be the keepalive message for a finalized flow.