



DEVELOPERS GUIDE

HKEx Orion Market Data Platform Securities Market & Index Datafeed Products

Version 1.0
31 July 2012

DOCUMENT HISTORY

Distribution Version

Version	Date of Issue	Comments
V1.0	31 July 2012	First Distribution Issue

CONTENTS

1	INTRODUCTION	4
2	DATA STRUCTURE.....	5
2.1	Packet Header	5
2.2	Heartbeats.....	6
2.3	Message Header	6
2.4	Message Formats	6
3	ENDIAN	6
4	FIELD ATTRIBUTES	7
4.1	Null Values.....	7
4.2	Currency Values.....	7
5	MESSAGE PROCESSING	7
5.1	Start of Day.....	7
5.2	Normal Transaction	8
5.2.1	Receive Multicast	8
5.2.2	Line Arbitration	8
5.2.3	Process Data Message.....	10
5.2.4	Process Control Message (Heartbeats)	10
5.3	Recovery.....	11
5.3.1	Retransmission Service.....	11
5.3.1.1	Secondary Retransmission Server	12
5.3.1.2	RTS Logon	12
5.3.1.3	RTS Logon Response.....	12
5.3.1.4	RTS Heartbeats	12
5.3.1.5	RTS Request.....	13
5.3.1.6	RTS Response	13
5.3.1.7	RTS Message.....	14
5.3.1.8	RTS Limits	14
5.3.1.9	Processing of RTS retransmission data.....	14
5.3.2	Refresh Service	15
5.3.2.1	RFS Snapshot	16
5.3.2.2	Processing a Refresh	16
6	RACE CONDITIONS.....	19
7	AGGREGATE ORDER BOOK MANAGEMENT	19
8	FULL ORDER BOOK MANAGEMENT.....	19
9	EXCEPTION HANDLING	20
9.1	Late Connection / Startup Refresh	20
9.2	Intra-day Refresh.....	21
9.3	Client Application Restarts	21
9.4	Sequence Reset Message	21
9.5	OMD Restarts Before Market Open	22
9.6	OMD Component Failover.....	22
9.7	Site Failover	22
	APPENDIX A – Example of network diagram to OMD	23
	APPENDIX B – Pseudo code to connect and receive multicast channel	25
	APPENDIX C – Pseudo code of Line Arbitration	26
	APPENDIX D – Pseudo code for processing retransmission data.....	28
	APPENDIX E – Pseudo code for processing Refresh snapshot packet.....	29
	APPENDIX F – Pseudo code for processing Aggregate Order Book Message	30

1 INTRODUCTION

This document contains guidelines and suggestions for HKEx Orion Market Data Platform ("OMD") feed handler developers. All information included in this document is presented for reference only. Clients should design and implement their own OMD feed handler that are tailored to their business and technical requirements.

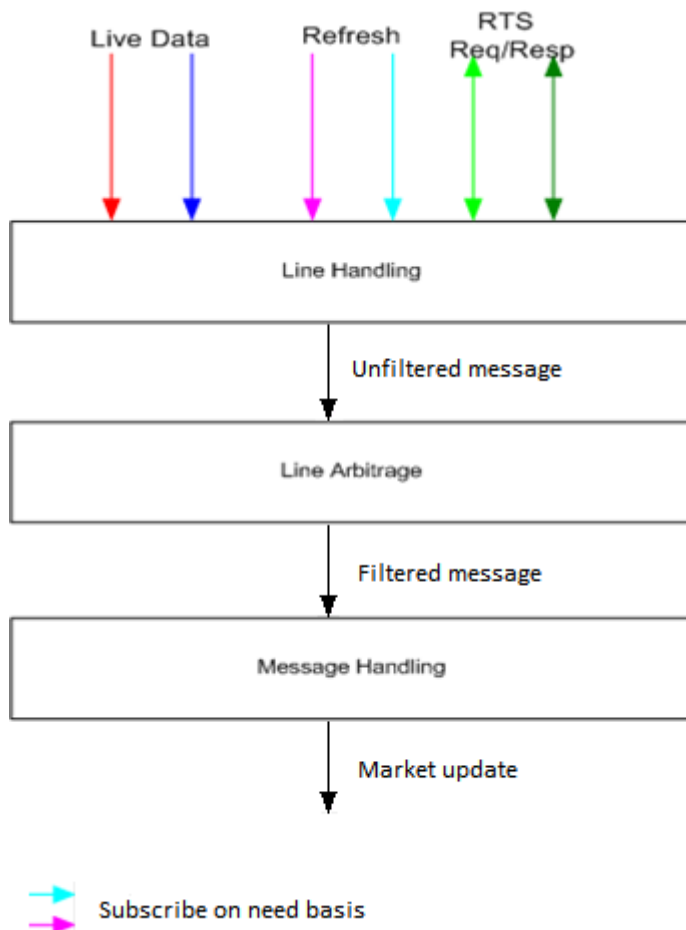
Its scope covers line arbitrage, packet and message processing, retransmission and refresh mechanisms, order book maintenance and exceptional handling procedures.

The purpose of this document is to answer any questions that developers may have after reading the OMD interface specification. It shows examples of usage and code snippets to help developers to understand the logic behind the market data disseminated from the OMD platform.

Table 1. Acronyms used in this document

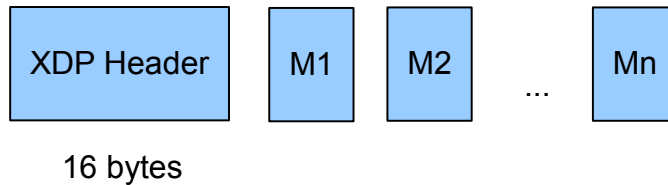
FH	Feed handler
HA	High Availability
MC	Multicast
RFS	Refresh Server
RTS	Retransmission Server
UDP	User Datagram Protocol
XDP	Exchange Data Publisher

Diagram 1. A Basic Client Application Layout



2 DATA STRUCTURE

Multicast packets are structured into a common packet header followed by zero or more messages. Messages within a packet are laid out sequentially, one after another without any spaces between them.



A packet will only ever contain complete messages. A single message will never be fragmented across packets.

2.1 Packet Header

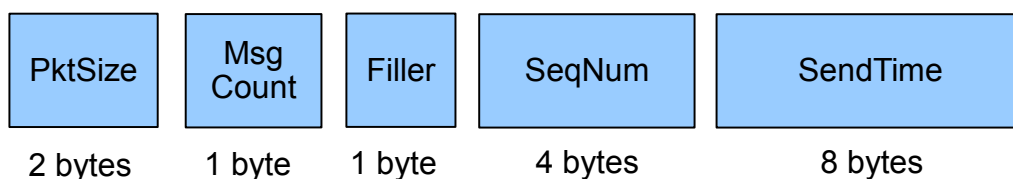
All packets disseminated from the OMD feed have a common packet header. This format is consistent across live, retransmission and refresh. XDP packet consists of 16-byte header followed by messages.

There are no delimiters between the packet header and messages or between messages themselves. One has to use the size of the header and in each individual message to determine the start of each message.

Table 2 below shows the packet header structure. The offsets in the table represent the number of bytes away from the beginning of the packet.

Table 2. Packet Header

Field	Offset	Length	Format	Description
PktSize	0	2	UInt16	Binary integer representing size of the packet (including this header)
MsgCount	2	1	UInt8	Binary integer representing number of messages included in the packet
Filler	3	1	String	
SeqNum	4	4	UInt32	Binary integer representing the sequence number of the first message in the packet
SendTime	8	8	UInt64	Binary integer representing the number of nanoseconds since January 1, 1970, 00:00:00 GMT, precision is provided to the nearest millisecond



2.2 Heartbeats

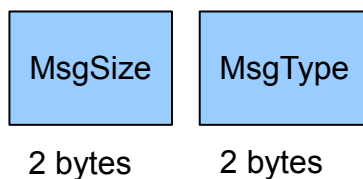
Heartbeats consist of a packet header with MsgCount set to 0 and do not increment the sequence number of the multicast channel. SeqNum in packet header is set to the sequence number of the previous message sent on the channel.

The Heartbeat message syntax is identical across OMD services.

2.3 Message Header

The format of each message within a packet will vary according to message type. However, regardless of the message type, all messages start with a two-byte message size followed by a two-byte message type.

MsgSize Binary integer representing the length of the message (including the header)
MsgType Binary integer representing the type of message. Please refer to the HKEx OMD Interface Specification for the full list of message type



2.4 Message Formats

Please refer to the HKEx OMD Interface Specification for details on the following message types:

- Control messages
- Retransmission
- Refresh
- Reference Data
- Status Data
- Order Book Data
- Trade and Price Data
- Value Added Data
- News
- Index Data

3 ENDIAN

All binary values are in Little Endian byte order, which means the first byte (lowest address) is the least significant one.

In C/C++, one solution is to create a structure containing all the fields from the packet header and cast the pointer to a packet, to a pointer to such a structure. For instance:

```

struct XdpPacketHeader
{
    unsigned short mPktSize;
    unsigned char mMsgCount;
    unsigned char mFiller;
    unsigned long mSeqNum;
    unsigned long long mSendTime;
}
  
```

```
};
```

Assume the packet is passed as a pointer to const unsigned char, which could look like this:

```
struct PacketHeader* hdr = static_cast<PacketHeader*>(packetPtr);
```

One packet can contain multiple messages. Clients should locate the beginning of each message based on the message length and process each message separately. The number of messages within a packet is indicated by MsgCount field in the packet header.

4 FIELD ATTRIBUTES

4.1 Null Values

From time to time certain fields cannot be populated and specific values are used to represent null. For example, it is currently used within Int64 fields of the Index Data (71) message.

The Int64 null representation is 0x8000000000000000 (Hex 2's complement) or -9223372036854775808 (Decimal).

4.2 Currency Values

See the ISO-4217 Currency Codes for a full list of possible data values. Currently, the system uses the following codes:

'HKD'	–	Hong Kong dollars
'USD'	–	US dollars
'CNY'	–	Chinese Renminbi
'EUR'	–	Euro
'JPY'	–	Japanese Yen
'GBP'	–	United Kingdom Sterling
'CAD'	–	Canadian Dollars
'SGD'	–	Singapore Dollars

HKEx may add or delete currency code(s), whenever applicable, in the future.

5 MESSAGE PROCESSING

Each multicast channel maintains its own session. A session is limited to one business day. During the day, message sequence number is strictly increasing and therefore unique within a channel.

5.1 Start of Day

OMD will normally be brought up around 1:30am. This start up time, however, is not rigid and HKEx has the right to adjust this time according to the different trading situations.

Clients start at OMD startup time

- Clients subscribe to real-time multicast channels (Each OMD data product is delivered via a group of real-time multicast channels)
- OMD sends Sequence Reset message (100). Please refer to section [9.4 Sequence Reset Message](#) for processing details.
- OMD sends Reference Data messages below

- ◆ Market Definition (10)
- ◆ Security Definition (11)
- ◆ Liquidity Provider (13)
- ◆ Currency Rate (14)
- Remark:
 - ◆ Clients may receive multiple Sequence Reset messages during the start of day. The general handling should be, reset the next expected sequence number and clear all cached data for all instruments. Please refer to section [9.5 OMD Restarts Before Market Open](#) for details.
 - ◆ After receiving the Sequence Reset message, clients should also check the sequence number of next incoming packet. If the sequence number is not equal to 1, it indicates that there is packet loss. Please refer to section [9.4 Sequence Reset Message](#) for details.

Clients start after OMD startup time and missed sequence reset message and reference data

- Please refer to section [5.3.2.2 Processing a Refresh](#) for exception handling of late connection

5.2 Normal Transaction

Normal message transmission is expected between when the market opens for trading and when the market is closed. Heartbeats are sent regularly (currently OMD sets to every 2 seconds) on each channel when there is no line activity.

UDP multicast network/transport protocol is used in OMD and data is sent to different broadcast streams (known as multicast channels).

UDP is not a reliable transport protocol. So packets may be lost or come out of order. The data on each channel comes from two redundant lines, A and B, to minimize the risk of losing a packet.

Clients receive and process OMD data

- Receive real-time multicast messages from Line A and Line B
 - Create two sockets using Multicast IP / Ports of Line A and Line B
 - Read data from multicast channel for Line A and Line B
- Line Arbitration using sequence number in packet header
 - Discard duplicate packets
 - Reorder packets
 - Gap Detection
- Process multicast messages
 - Process Data Message
 - Process Control Message (Heartbeat)

5.2.1 Receive Multicast

Clients join particular multicast group in order to receive the desired data. Data is categorized and available from dedicated multicast groups.

Clients connect and receive real-time multicast messages from Line A and Line B.

Please refer to [APPENDIX B – Pseudo code to connect and receive multicast channel](#) for example on connecting multicast channels.

5.2.2 Line Arbitration

The network/transport protocol used in OMD is UDP multicast. The data in OMD is divided into broadcast streams (known as channels). The data on each channel comes from two redundant lines,

A and B. UDP is not a reliable transport protocol like TCP but because of this it is much faster, although this means it is possible that packets may be lost or come out of order. Two lines with identical data minimize the risk of losing a packet; however the risk still exists.

Note

1. Clients should not prioritize line A over line B. They should listen to both line A and B at the same time. Line A is not guaranteed to be faster than B. They should both be treated with the same priority. The approach that assumes listening to line B only if there is a gap detected on line A is incorrect. In general, it is recommended to have an abstraction layer between the gap detection module and the source of packets. In other words, the gap detection module does not have to know where the packets are coming from, it just needs to monitor packet sequence numbers.
2. The packaging of messages between Line A and Line B may be different. In the example below, three packets are sent on each line, but message 'OrderUpdate3' appears in one packet from Line A but in the subsequent packet on Line B.

Diagram 2. Normal Message Delivery of Primary and Secondary Line (Line A and B)

Primary			Secondary		
Messages	MC	SN	SN	MC	Messages
OrderUpdate1 OrderUpdate2 OrderUpdate3	3	101	101	2	OrderUpdate1 OrderUpdate2
Trade1 OrderUpdate4	2	104	103	3	OrderUpdate3 Trade1 OrderUpdate4
Trade2 Statistics 1	2	106	106	2	Trade2 Statistics 1

Note

- MC: Message Count in a Packet

Clients receiving OMD feed are recommended to implement the following functionality in order to provide appropriate line handling:

1. Discarding duplicate messages
2. Reordering messages
3. Gap Detection

All of the above can be achieved by remembering the next expected sequence number. Please refer to the Gap Detection Diagram in the OMD Interface Specification for reference. Basically, a gap detection mechanism may work like this:

When clients receive a packet from Line A or Line B,

- Handle the first packet, process each message within the packet and advance the next expected sequence number (nextSeqNum) by 1
- When subsequent packet is received, compare the current seqNum in the packet header with the nextSeqNum
 - If seqNum > nextSeqNum, it is a gap and spool the message
 - If (seqNum + msgCount in packet) < nextSeqNum, it is a duplicate packet and skip
- When processing each message within the packet

- If (seqNum + message processed count in this packet) < nextSeqNum,
It is a duplicate message and skip
- If (seqNum + message processed count in this packet) = nextSeqNum,
Process it and advance the next expected sequence number (nextSeqNum) by 1

Please refer to [APPENDIX C – Pseudo code of Line Arbitration](#) for example on detecting gap or duplicate packet.

Possible approaches for handling message gap

Approach 1: Clients wait some time to fill the gap from the redundant line (or the packet may come from the same line, possibly out of order)

If a given amount of time has passed and there still is a gap, the clients should send a retransmission request. While awaiting for retransmission all packets coming from the live feed should be spooled. After processed the retransmitted packets, clients should process the spooled packets/messages.

Note

1. While waiting for the retransmission, another gap can occur. Clients should take this into account. One possible solution would be to keep track of how many gaps have been detected and for which gaps a retransmission request has already been sent.
2. Only a continuous series of packets/messages from the spool should be processed.
3. Any gaps should await to be filled either from the redundant line or the retransmission server.
4. Check if the gap in spool message can be filled at regular interval.
5. If the gap cannot be recovered for specified time, clients should recover from refresh server.

Please refer to [APPENDIX C – Pseudo code of Line Arbitration](#) for example on processing spooled messages.

Approach 2: Issue a retransmission request immediately after detecting a gap

If the missed packets/messages come on the redundant line before they come from the retransmission, clients will simply process them and discard the retransmitted ones.

This approach may waste a number of retransmission requests, but is faster.

5.2.3 Process Data Message

Message carrying information about a particular instrument has a Security Code field. This field is unique instrument identifiers. The Security name, ISIN code, etc. are only carried in the Security Definition (11) message, so clients must associate the Security Code with instrument's characteristics during the reference data processing. The Security Code, once allocated for an instrument, does not change.

5.2.4 Process Control Message (Heartbeats)

Heartbeats are disseminated at regular time intervals. Clients can use heartbeats to check if the feed is alive. If there is no heartbeat for longer than a configurable time, then it indicates that there is an outage at the exchange side.

Note that OMD sends heartbeats only when there is no market data being disseminated. When there is market data on the line, no heartbeat is available.

Heartbeats consist of a packet header with MsgCount set to 0 and do not increment the sequence number of the multicast channel. SeqNum in packet header is set to the sequence number of the previous message sent in the channel.

When receiving heartbeat packet, clients should ignore this packet in gap detection. Otherwise, clients may fail to detect the actual message gap.

Table 3. Gap Detection Example

Time	Packet sent from OMD	Packet received by Client	Remark
T1	101	101	
T2	102	102	
T3	103		Packet with seqNum 103 is lost
T4	103 (Heartbeat)	103 (Heartbeat)	If client receives heartbeat message but cannot find the corresponding packet with same sequence number, it should be a message gap and client should recover the lost message
T5	104	104	
T6	105	105	
T7	106	106	

5.3 Recovery

Since UDP multicast is not a reliable protocol, there is a risk of packet lost. Clients can recover lost messages using the retransmission server or the refresh, which depend on varies factors such as message gap size, recovery time/event and etc.

5.3.1 Retransmission Service

For small number of message gap, clients can recover lost messages using the retransmission server. The connection between the RTS and the client is reliable (TCP/IP). In order to receive lost messages, clients need to send a Retransmission Request. The RTS will respond with a Retransmission Response which can indicate that either the request has been accepted or rejected (the RetransStatus field). If accepted, the RetransStatus field will be 0, and if rejected, the values can be 1, 2, 100 or 101.

The retransmission server contains only a relatively small number of messages (50,000) from each broadcast stream. The RTS should not be thought of as a means of recovering intraday. It serves only as real time retransmission of a relatively small number of lost messages.

Clients can have only one connection with the RTS.

The sequence number range as well as the number of requests per day is limited to 1000 requests, and 10,000 messages per request.

Note

If clients need to issue a retransmission request for a gap bigger than the allowed limit, they need to split the requests into appropriate amount of smaller requests.

RTS Logon, Logon Response, RTS Request and RTS Response message will begin with packet header which is same format as real time. Clients should ignore the

sequence number in RTS packet header when sending or processing the RTS message.

5.3.1.1 Secondary Retransmission Server

There is a secondary RTS which should be used in case there are any problems encountered with the primary RTS. This is a part of the High Availability design and is meant to provide customers with a seamless service in case of the primary RTS failure.

5.3.1.2 RTS Logon

In order to receive retransmission, clients must establish a TCP/IP connection with the RTS and initiate a session by sending a Logon message within the logon timeout interval (5 seconds). If clients do not send a Logon message within the logon timeout interval, the server will close the connection.

Table 4. Logon Packet Header

PktSize	32
MsgCount	1
Filler	
SeqNum	Optional
SendTime	The number of nanoseconds since January 1, 1970, 00:00:00 GMT, precision is provided to the nearest millisecond.

Table 5. Logon Request Message

MsgSize	16
MsgType	101
Username	Username to logon in plain text

5.3.1.3 RTS Logon Response

The RTS immediately sends a LogonResponse message after it receives a Logon request. The SessionStatus field indicates if the Logon was successful. The possible values of this field are:

Table 6. Logon Session Statuses

Logon Session Status	Meaning
0	Session Active
5	Invalid Username
100	User already connected

The session, once established, can be reused for sending any subsequent retransmission requests. To maintain the session, a client must respond to heartbeats sent by the RTS within 5 seconds.

5.3.1.4 RTS Heartbeats

To determine the healthiness of the client connection on the TCP/IP channel, the RTS will regularly send heartbeats to the client. The heartbeat frequency is 30 seconds. The client must respond with a Heartbeat Response. The timeout of this heartbeat response is set at 5 seconds. If no response is received by the RTS within this timeframe, RTS will disconnect the session.

A Heartbeat Response is an exact copy of the incoming Heartbeat.

5.3.1.5 RTS Request

A retransmission request consists of a PacketHeader and a Retransmission Request (201).

Table 7. Retransmission Request Packet Header

PktSize	32
MsgCount	1
Filler	
SeqNum	Optional
SendTime	The number of <i>nanoseconds</i> since <i>January 1, 1970, 00:00:00 GMT</i> , precision is provided to the nearest millisecond.

Table 8. Retransmission Request Message

MsgSize	16
MsgType	201
ChannelID	Depending on the broadcast stream
Filler	
BeginSeqNum	Message sequence number of first message in range to be resent
EndSeqNum	Message sequence number of last message in range to be resent

Example of Retransmission Request

Assume client application received following packets from real time multicast channel 1

Channel	Packet Sequence number	Message	Message received	Message Gap (Y/N)
1	101	Msg1 Msg2 Msg3	Msg 1 (101) Msg 2 (102) Msg 3 (103)	N
1	104	Msg4 Msg5 Msg6	Msg 4 (104) Msg 5 (105) Msg 6 (106)	N
1	109	Msg7 Msg8	Msg 7 (109) Msg 8 (110)	Y (Missing messages with sequence number 107-108)

Client application should send following Retransmission Request Message to recover missing messages (Seq # 107-108)

MsgSize	16
MsgType	201
ChannelID	1
Filler	
BeginSeqNum	107
EndSeqNum	108

5.3.1.6 RTS Response

After sending a retransmission request, the RTS will respond with a retransmission response message. The most important field in the response message is the RetransStatus. Below are the possible values and what they indicate:

Table 9. Retransmission Response statuses

Retransmission Response Status	Meaning
0	Request accepted
1	Unknown/Unauthorised Channel ID
2	Messages not available
100	Exceeds maximum sequence range
101	Exceeds maximum requests in a day

Note

It is very important to stop sending retransmission requests for the current day after being rejected with reason 101. Client may contact HKEx OMD help desk for assistance.

5.3.1.7 RTS Message

Upon receiving retransmission response with Status '0', the RTS will start sending packets containing the requested messages. The sequence number of first requested message will be used as sequence number in packet header.

5.3.1.8 RTS Limits

Below is a table detailing the limits imposed on the Retransmission Service:

Table 10. Retransmission System Limits

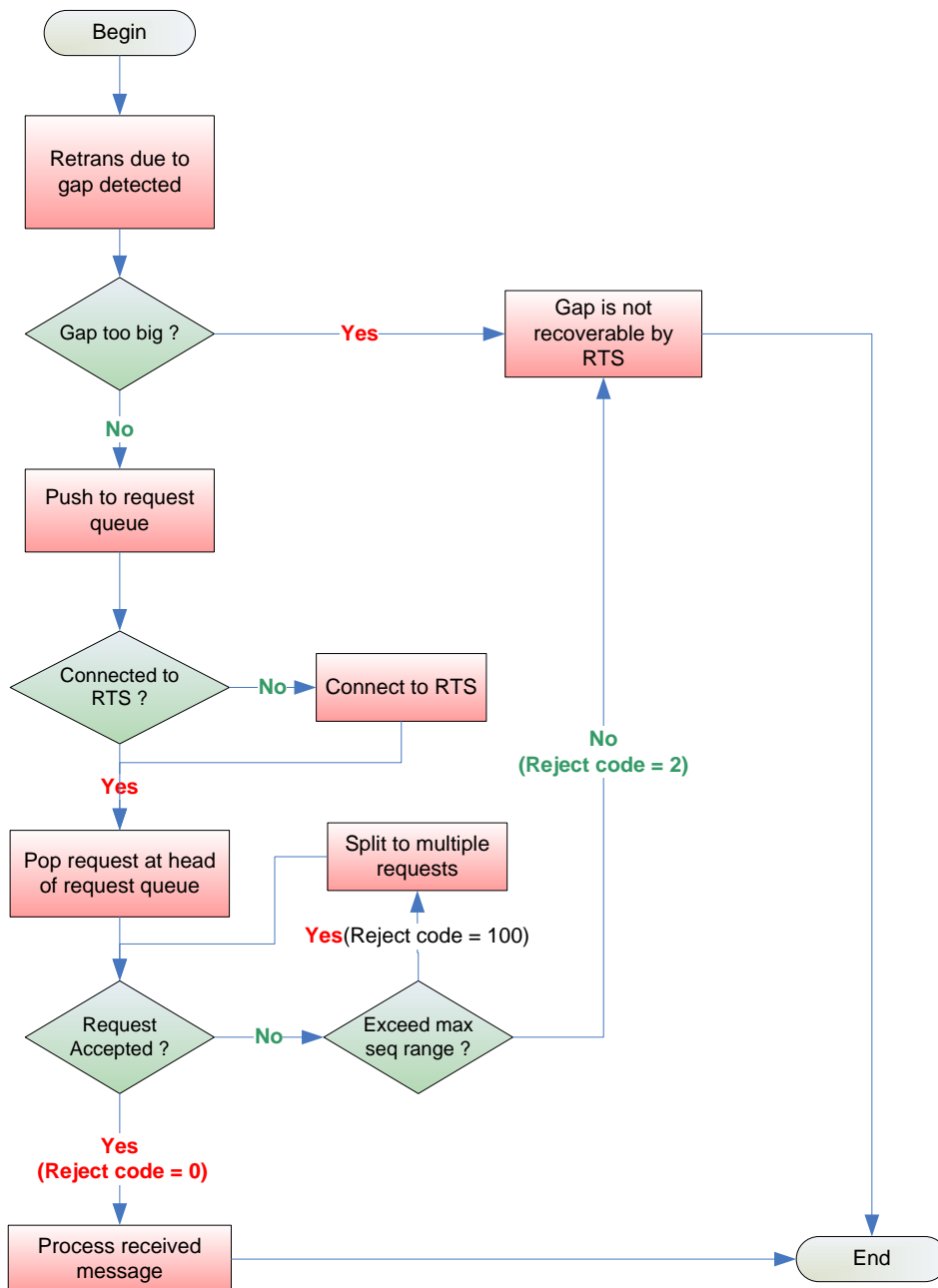
System Limit	Value
Last number of messages available per channel ID	50,000
Maximum sequence range that can be requested	10,000
Maximum number of requests per day	1,000
Logon timeout (seconds)	5
Heartbeat interval (seconds)	30

Note

If clients exceeded the daily limit of requesting OMD Retransmission services, you may contact OMD Help Desk for assistant.

5.3.1.9 Processing of RTS retransmission data

Figure 1. Workflow of Retransmission



(Assuming a client is authorised to that channel ID and has not reached the maximum request limit.)

Please refer to [APPENDIX D – Pseudo code for processing retransmission data](#) for example on handling data from OMD Retransmission server.

5.3.2 Refresh Service

The OMD feed provides a refresh facility, which allows clients to start intraday or recover from significant packet loss. The refresh is available per channel.

RFS periodically provides a full snapshot of the market. Not all the messages available from the live feed can be recovered from the refresh. However, all the message types, necessary for reconstructing an up-to-date image of the market, are available from the refresh.

The refresh packets are disseminated via dedicated multicast streams.

Similar to real time channel, the refresh data also come from two redundant lines, A and B. Clients can apply the Line Arbitration mechanism described in [5.2.2 Line Arbitration](#), except that there is no retransmission.

If clients connect to refresh multicast channel after OMD start up, they do not need to check any message gap before the first arrived packet. Clients can make reference to the sequence number of the first message and increment the next expected sequence number by 1 when processing incoming messages.

It is advisable that clients utilise the refresh service under the following situations:

1. Intraday start
2. Large message gap
3. Delay in the RTS retransmission
4. RTS retransmission failure

5.3.2.1 RFS Snapshot

Please refer to the OMD Interface Specification for the coverage of snapshot data.

5.3.2.2 Processing a Refresh

Processing the refresh while coping with the live feed may be a challenging piece of functionality in the feed handler. There are several things to think about in order to process the refresh properly. The 4 main areas, which problems may perhaps arise, are:

1. Connectivity
2. Synchronisation
3. Determine a full refresh snapshot
4. Sequencing of events

Connectivity

There are 2 data streams that need to be handled during the refresh:

1. Live feed multicast
2. Refresh feed multicast

Synchronization

1. Subscribe to the real time MC channel and cache received messages.
2. Subscribe to the corresponding refresh multicast channel. Once subscribed, if messages are received instantaneously, clients should discard all messages till the arrival of a Refresh Complete message.
3. Wait for the next wave of snapshot data. Process all messages until the next Refresh Complete message is received.
4. Store the LastSeqNum sequence number provided in the above message.
5. If Sequence Reset message is received, reset the next expected sequence number to a value of NewSeqNo (1) field in current Sequence Reset message
6. Unsubscribe from the refresh MC channel.
7. Discard the cached real time messages with sequence number less than or equal to LastSeqNum.
8. Process the remaining cached real-time messages and resume normal processing.

Determine a full refresh snapshot

When subscribing to OMD refresh multicast channel, clients should handle the following situations to recover a full image of the market:

1. The first message received is a Heartbeat

When there is no message transmission (channel idle) in a refresh multicast channel, OMD sends Heartbeat message at a regular time interval (currently it is set to 2 seconds). Clients wait for the full refresh snapshot starting with a message other than Heartbeat till the arrival of Refresh Complete message.

2. The first message received is a Refresh Complete message

Clients ignore the first Refresh Complete message and the subsequent Heartbeat message(s) in a refresh multicast channel. The next refresh snapshot starts with a message other than Heartbeat till the arrival of the second Refresh Complete message.

3. The first message received is neither Heartbeat nor Refresh Complete message

Clients cannot receive a full refresh snapshot and should **NOT** process any message at the moment. Simply skip message(s) until a Refresh Complete message is received. Usually, OMD sends refresh snapshot at a regular interval. Heartbeat is disseminated in between two rounds of refresh snapshot. Similar to point 1, clients will then obtain a full market snapshot in the next refresh interval

4. Receive refresh messages follow by Sequence Reset message

Receive "Sequence Reset Message" from refresh channel and clear the cached refresh messages from that refresh channel before. The next refresh snapshot starts with a message other than Heartbeat with sequence number start from 1 till the arrival of the Refresh Complete message.

Sequencing of events

It is important for clients to know which multicast channels hosts reference data for what other channels. Clients should not process any data (except for the reference data) until the full reference data is processed.

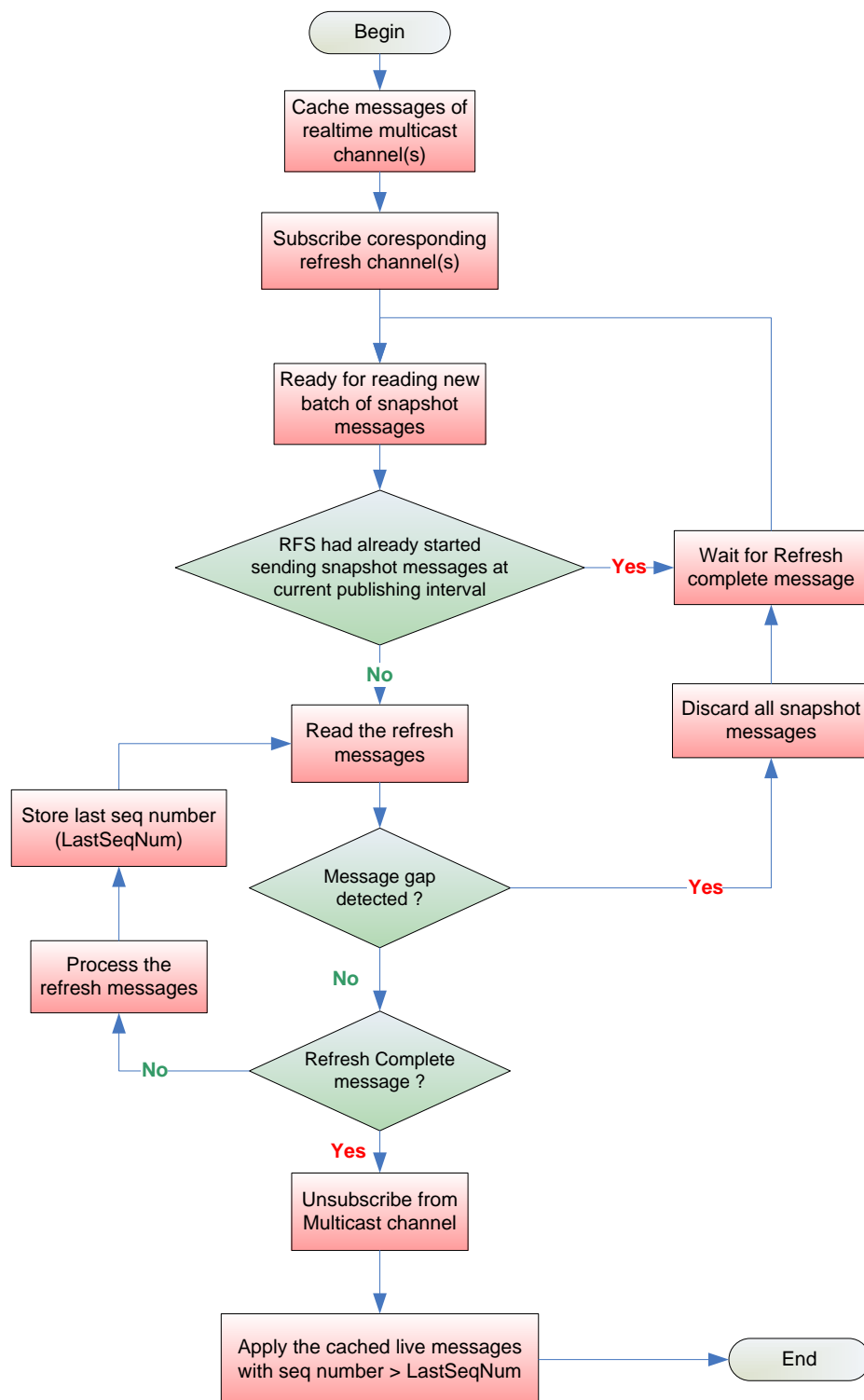
This implies the order of requesting refresh that clients should obey.

If the feed handler is started intraday, clients should **first go for the refresh of channels that serve reference data**. Only after the refresh of the reference data is received, clients should ask for the refresh of trades, order books, etc.

Note

There is no TCP retransmission for refresh. Clients must monitor for packet loss on the refresh channels and wait for the next snapshot if loss is detected.

Figure 2. Workflow of Refresh



Please refer to [APPENDIX E – Pseudo code for processing Refresh snapshot packet](#) for example on handling data from OMD RFS.

6 RACE CONDITIONS

The real-time order/trade data and reference data are disseminated via separate channels, so users need to be aware that there is a race condition.

For example, a Security Status (21) message may be sent marking a security as suspended. However, for a very short time after this message, the regular order and trade information for this security may continue to arrive.

Another example would be a Trading Session Status (20) message marking the trading session as halted, but real time data for the same market may continue to arrive for a short time afterwards.

7 AGGREGATE ORDER BOOK MANAGEMENT

Book updates are sent by OMD via Aggregate Order Book (53) messages. Each message may contain any combination of new, changed or deleted entries for a book or clear the whole book. The nature of an entry is defined by its UpdateAction.

Table 11. Actions on Aggregate Order Book Messages

Action	Description
New	Create/Insert a new price level
Delete	Remove a price level
Change	Update aggregate quantity at a price level
Clear	Clear the whole book

General Rules

- All entries within an Aggregate Order Book message must be applied sequentially.
- Clients must adjust the price level of entries below deleted or inserted entries. Note - Potential level adjustments must be carried out after each single entry in Aggregate Order Book message.
- If a new book entry causes the bottom entry of a book to be shifted out of the book,
 1. If the shifted out entry is within 10 Price level, OMD will send an explicit deletion entry (Explicit delete).
 2. If the shifted out entry is outside the 10 Price level, clients must delete the excess entry (Implicit delete).
 3. If the book shrinks again, the server resends the entries that have temporarily fallen out.
- If a match causes an order to be removed so that there are now less than 10 levels visible, then the server will also automatically send the additional level(s) that are now revealed.
- If a clear aggregate order book message is received, client should clear all entries of the order book.

Please refer to [Section 5 – Aggregate Order Book Management in the Interface Specification of HKEx Orion Market Data Platform Securities - Market & Index Datafeed Products](#) for different scenarios on how OMD sends Aggregate Order Book message.

You may also refer to [APPENDIX F – Pseudo code for processing Aggregate Order Book Message](#) for example on handling Order Book messages from OMD server.

8 FULL ORDER BOOK MANAGEMENT

Developers maintain order books from Order Update Messages. Every event in the full order book is reported by OMD with the following message types being disseminated:

Table 12. Order Update Message Types

Message Type	Name
30	Add Order
31	Modify Order
32	Delete Order
33	Add Odd Lot Order
34	Delete Odd Lot Order

General Rules

- Clients should be able to uniquely identify the order (OrderID is the unique identifier per security)
- Determine the price and size of an order

Order Types

Table 13. Order Types

Order Type	Description
Market (1)	Executed at current market price
Limit (2)	Contains an execution condition to buy or sell at a specified price or better

Note

The price field of Market Orders is always set to 0.

9 EXCEPTION HANDLING

Listed below are some common exception handling procedures that clients must be capable of when subscribing to OMD:

- Late connection
- Intra-day refresh
- Client application restarts
- Sequence Reset Message
- OMD restarts before market open
- OMD node failover
- Site failover

9.1 Late Connection / Startup Refresh

When client starts late, all reference data should be recovered before the current image for all instruments across all channels.

Please refer to section [5.3.2.2 Processing a Refresh](#) for recovery procedures.

Note

- Some channels may host reference data for other channels.
- Channels which depend on other channels for reference data cannot be processed before full reference data has been received
- Clients must define relationships between channels

9.2 Intra-day Refresh

For each real time multicast channel, there exists a corresponding refresh multicast channel on which snapshots of the market state are sent at regular intervals throughout the business day.

When clients experienced an unrecoverable packet loss on a certain channel during the day, a snapshot is only needed for that channel.

Sequencing of events

1. Caches real time messages in the multicast channel that previously experienced packet loss
2. Listens to the corresponding refresh multicast channel and waits for the next snapshot (*refer to [5.3.2.2 Processing a Refresh - Determine a full refresh snapshot](#)*)
3. Processes all refresh messages until the arrival of a Refresh Complete message
4. Store the LastSeqNum sequence number provided in the Refresh Complete message
5. Disconnects from the refresh multicast channel
6. Processes the cached real time messages with sequence number greater than the LastSeqNum. Otherwise, drop processing it.

Now the clients maintain the current market image.

9.3 Client Application Restarts

Similar to “Late Connection” as described earlier.

9.4 Sequence Reset Message

Sequence Reset Message from real time channels

Sequencing of events

1. Receive “Sequence Reset Message” from any real time multicast channel
2. Reset the next expected sequence number to a value of NewSeqNo (1) field in Sequence Reset message
3. Clear all cached data for all instruments.
4. Subscribe to the corresponding refresh channels of all subscribed real time multicast channels to receive the current state of the market. (*refer to [5.3.2.2 Processing a Refresh – for handling messages from Refresh channels](#)*)
5. Resume to process real time messages

Packet loss detection when processing Sequence Reset Message

After a Sequence Reset, the first UDP packet should have a sequence number 1. However, this packet is lost and clients start receiving packet with sequence number 2 and onwards. Clients can try to recover it from the redundant line. If the lost packet is unrecoverable, clients should start buffering the live feed and send a retransmission request immediately.

Once clients finished processing the retransmitted messages from RTS, clients can maintain the latest market image by handling the buffered data and then the live feed.

Sequence Reset Message from Refresh channels

Refer to [5.3.2.2 Processing a Refresh – for handling sequence reset message from Refresh channels](#)

9.5 OMD Restarts Before Market Open

In case of OMD performs start-of-day twice (errors encountered during first start-of-day). The second start-of-day should trigger Sequence Reset in all channels. Clients should discard all reference data received in the first start-of-day and process the second start-of-day.

9.6 OMD Component Failover

In case no live data can be received in one of the OMD lines (say line A), there is no impact to clients as OMD would continue publishing data via the alternate line (line B). Clients can reapply line arbitration to retrieve the latest market information as usual when OMD resumed the service from line A.

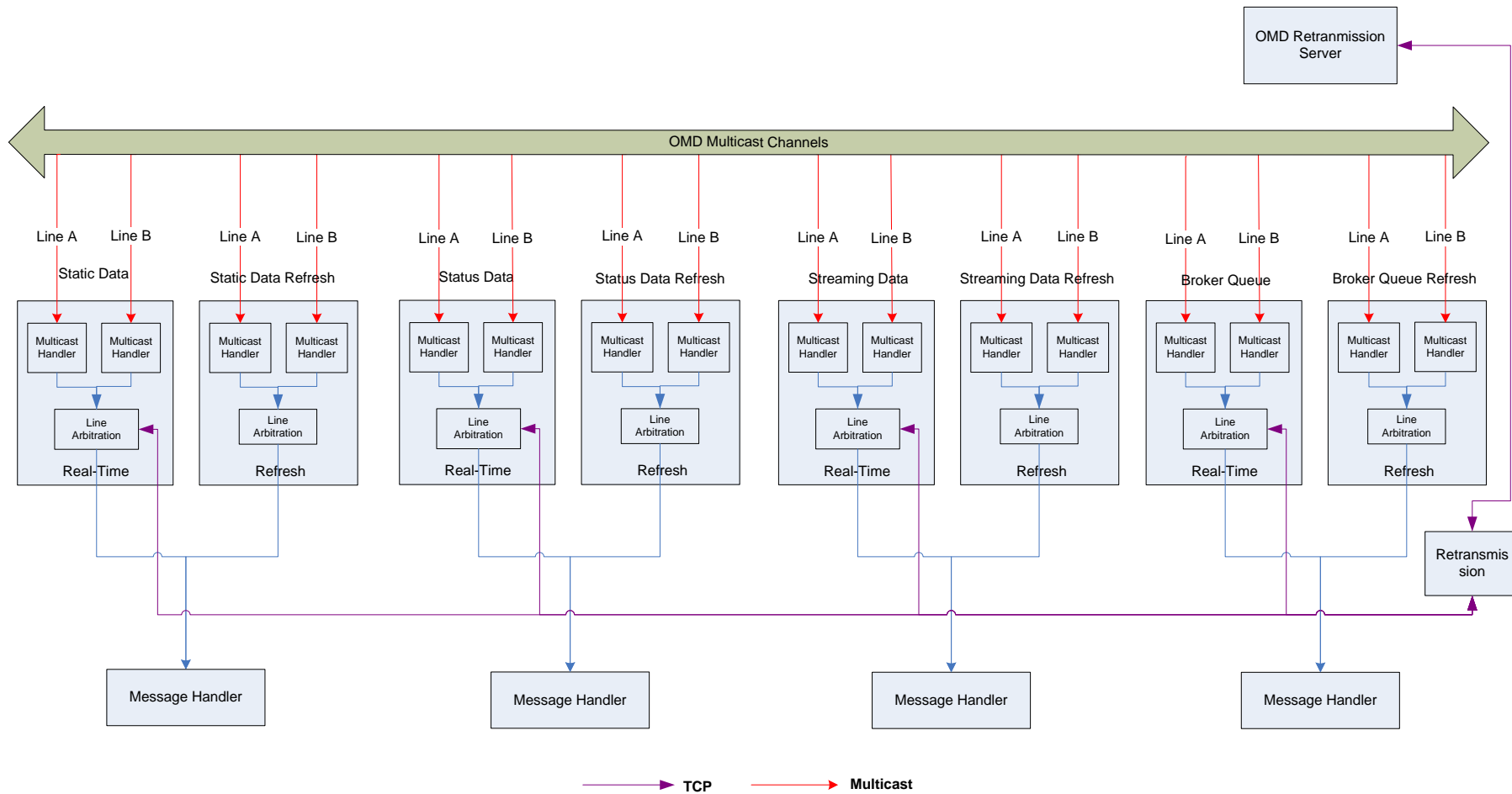
Trade Ticker may be resent for OMD Securities Standard service, Client should check the ticker ID when processing the Trade Ticker message to avoid duplication

9.7 Site Failover

In case of any problem in primary OMD servers, the OMD servers will be brought up at the backup site. The multicast addresses of live and refresh data will remain the same. The primary RTS servers will not be available. Clients should use the backup IP addresses to connect to backup RTS servers.

Once the OMD DR site is ready, OMD sends sequence reset message per multicast channel. Clients subscribe to the refresh channels in order to receive the current state of the market. Please refer to section [9.4 Sequence Reset Message](#) for details.

APPENDIX A – Example of network diagram to OMD



Module	Description	Network Type	Number of connection
Retransmission	Module that handle Retransmission in case there is any gap	TCP	1
Multicast Handler	Module that read data from multicast channels. <ul style="list-style-type: none">- Online message (Line A and Line B)- Refresh message (Line A and Line B) in case there is any missing packet that cannot be recovered from RTS	UDP	Number of MC x 4
Line Arbitration	Module that handle line arbitration.	N/A	
Message Handler	Module that process OMD messages	N/A	

Note

The above network diagram illustrates one of the possible network connections to OMD. Clients may have different designs when subscribing to different OMD multicast channels.

APPENDIX B – Pseudo code to connect and receive multicast channel

An example shows how to set up UDP socket and join multicast channel.

```
int sock_fd;
int flag = 1;
struct sockaddr_in sin;
struct ip_mreq imreq;

// Create a socket
sock_fd = socket(AF_INET, SOCK_DGRAM, 0);

// Set socket option
setsockopt(sock_fd, SOL_SOCKET, SO_REUSEADDR, &flag, sizeof(int));

// Set IP, Port
memset(&sin, 0, sizeof(sin));
sin.sin_family = AF_INET;
sin.sin_addr.s_addr = INADDR_ANY;
sin.sin_port = htons(port);

// Bind
bind(sock_fd, (struct sockaddr *) &sin, sizeof(struct sockaddr))

// Add to Multicast Group
imreq.imr_multiaddr.s_addr = inet_addr(mcAddress);
imreq.imr_interface.s_addr = inet_addr(interface);

setsockopt(sock_fd, IPPROTO_IP, IP_ADD_MEMBERSHIP, (const void *)&imreq, sizeof(struct ip_mreq));
```

An example shows how to read data from a multicast channel.

```
size_t len;
socklen_t size = sizeof(struct sockaddr);
struct sockaddr_in client_addr;
char mReadBuffer[2046];

memset(mReadBuffer, 0, sizeof(mReadBuffer));

// Read the data on the socket
len = recvfrom(fd, mReadBuffer, sizeof(mReadBuffer), 0, (struct sockaddr *) &client_addr, &size);
```

APPENDIX C – Pseudo code of Line Arbitration

An example shows how clients process a packet received from OMD. This function handles data received from Line A or Line B multicast channels.

```
void processPacket(Packet packetBuffer)
{
    if (packetBuffer.getSeqNum() > expectedSeqNum) {
        //Gap detected, recover lost messages

        //Spool Packet in memory and wait for short period
        //Gap may be filled from next few incoming packet,
        //either from same line or alternative line
        spoolMessages(packetBuffer);
    }
    else if (packetBuffer.getSeqNum() + packetBuffer.getMsgCount() < expectedSeqNum) {
        //Duplicate packet, ignore
    }
    else if (packetBuffer.containsSeqNum(expectedSeqNum)) {
        //Process the packet if it contains a message
        //with sequence number = expectedSeqNum
        int msgProcessCount = 0;

        for (int i=0; i < packetBuffer.getMsgCount(); i++) {
            if (packet.getSeqNum() + msgProcessCount == expectedSeqNum) {
                extractMessage(message, packetBuffer, msgProcessCount);
                processMessage(message);
                expectedSeqNum++;
            }
            else {
                //Duplicate message, ignore
            }
            msgProcessCount++;
        }
    }
}
```

An example shows a timer function processes the spooled messages at a regular time interval.

```
void checkMessageSpoolTimer()
{
    MessageSpool::iterator i = mMessageSpool.begin();

    // Iterate through the message spool
    while (i != mMessageSpool.end())
    {
        Message message = i->second;

        // If the current packet sequence number is larger than expected,
        // there's a gap

        if (message.getSeqNum() > mNextSeqNum)
        {
            //No retrans request sent for this message before
            if (! message.getRetransRequested()) {
                sendRetransRequest(mNextSeqNum,
                                   message.getSeqNum() - 1);

                return;
            }

            //time limit hasn't been reached, so it's still not an
            // unrecoverable gap. Return and wait..
            if (message.getTimeLimit() < poolTimeLimit) {
                return;
            } else {
                //The RetransRequest failed or took too long and the gap wasn't
                //filled by the other line - The messages have been permanently
                //missed. Recover the lost data from Refresh Server (RFS)
                recoverFromRefresh();
            }
        }
        if (message.containsSeqNum(mNextSeqNum))
        {
            // The packet contains the next expected sequence number, so
            //process it
            processPacket (message);
        }
    }
}
```

APPENDIX D – Pseudo code for processing retransmission data

An example shows how clients process incoming data from OMD Retransmission server. It handles Heartbeat, RTS Logon Response, RetransRequest Response and Retrans data.

```
void read()
{
    readBuffer(mRtsBuffer);

    while (true)
    {
        // Get packet information at mRtsBuffer
        PacketHeader* packet = (PacketHeader*) mRtsBuffer;

        // If the entire packet is in the buffer, process it
        if (isEntirePacket(mRtsBuffer))
        {
            // If Heartbeat (i.e. packet with 0 MsgCount)
            if (packet->mMsgCount == 0)
            {
                sendRtsHeartbeat(mRtsBufPos, packet->mPktSize);
            }
            else
            {
                // Determine the kind of message(s) in the packet
                uint16_t msgType = packet.getMsgType();

                switch (msgType)
                {
                    case LOGON_RESPONSE_TYPE:
                    {
                        LogonResponse *logonResponse
                            = (LogonResponse *) (mRtsBufPos + sizeof(PacketHeader));
                        processLogonResponse(logonResponse);
                        break;
                    }
                    case RETRANS_RESPONSE_TYPE:
                    {
                        RetransResponse* resp = (RetransResponse*) (mRtsBufPos + sizeof(PacketHeader));
                        processRetransResponse(resp);
                        break;
                    }
                    default:
                        processPacket(packet);
                }
            }
        }
        // Wait for the rest of the data to come from the socket
        else
        {
            break;
        }
    }
}
```

APPENDIX E – Pseudo code for processing Refresh snapshot packet

An example shows how clients process refresh snapshot data from OMD RFS server and merge with realtime messages

```
void processRefreshPacket(Packet packetBuffer) {
    static int expectedSeqNum = 0;
    //First Message is Heartbeat or Refresh Complete Message
    if(! isStartOfRefresh(packetBuff) {
        return;
    }

    if (packetBuffer.getSeqNum() > expectedSeqNum) {
        //Gap detected
        clearSpoolMessage();
        return;
    }
    else if (packetBuffer.getSeqNum() + packetBuffer.getMsgCount() < expectedSeqNum) {
        //Duplicate packet, ignore
        return;
    }
    else if (packetBuffer.containsSeqNum(expectedSeqNum)) {
        spoolMessages(PacketBuff, expectedSeqNum);
    }

    if (isRefreshComplete(packetBuffer)) {
        List refreshMessageList = getRefreshSpoolMessage();
        processMessages(refreshMessageList);

        //Get spooled realtime message with seq num >= expectedSeqNum;
        List realtimeMessageList = getRealtimeSpoolMessage(expectedSeqNum);
        processMessages(realtimeMessageList);
    }
}
```

APPENDIX F – Pseudo code for processing Aggregate Order Book Message

An example shows how clients process Aggregate Order Book Message and update the internal order book.

```
OrderBook mOrderBook;

void processAggregateOrderBook(AggregateOB aggregateOB) {

    switch(aggregateOB.getAction())

    case ADD:
        int tickLevel = getTickLevel(mOrderBook, aggregateOB.getPrice());

        insertOB(mOrderBook, tickLevel, aggregateOB);

        //If Price level > 10, delete those order from OBMMap
        deleteOBExceedMaxPriceLevel(mOrderBook);

    case Update:
        int tickLevel = getTickLevel(mOrderBook, aggregateOB);
        updateOB(mOrderBook, tickLevel, aggregateOB);

    case Delete:
        int tickLevel = getTickLevel(mOrderBook, aggregateOB);
        deleteOB(mOrderBook, tickLevel, aggregateOB);
        updateOBPriceLeve(mOrderBook)

    case Clear:
        clearOB(mOrderBook);

    }

    void insertOB(mOrderBook, tickLevel, newAggregateOB) {
        newAggregateOB.setTickLevel(tickLevel);
        mOrderBook.add(tickLevel-1, newAggregateOB);

        for (int i=tickLevel; i < mOrderBook.getSize(); i++) {
            AggregateOB aggregateOB = mOrderBook.get(i);
            aggregateOB.updateTickLevel(mOrderBook);
            aggregateOB.updatePriceLevel(mOrderBook);
        }
    }
}
```