THE COPPELIA GUIDE (HA)

This Document is available to HA clients ONLY

Version	Audience	Comments
BETA	EXTERNAL	This is a Beta Release.
	HA Clients	

Last Updated May 10, 2000

Javelin Copyright Statement

Copyright

© 1996-2000 Javelin Technologies, Inc.

All rights reserved. No part of this document covered by the copyright hereon may be reproduced or copied by any means or in any form without the written consent of Javelin Technologies, Inc. Any software furnished under a license may be used or copied only in accordance with the terms set forth in the license agreement or contract.

Javelin Technologies, Inc. reserves the right to amend, modify, or revise all or part of this document without notice and shall not be responsible for any loss, cost, or damage, including, but not limited to, consequential damage, caused by reliance of the information contain herein.

Javelin Technologies, Inc. reserves the right to make changes in the product design without reservation and without notification to its users.

All other products and company names herein may be trademarks of their respective owners.

This document and the information it contains are proprietary and confidential to JAVELIN TECHNOLOGIES, INC. and are intended solely for authorized recipients. Unauthorized distributions is strictly prohibited by law. If you are in receipt of a copy of this document without the permission of JAVELIN TECHNOLOGIES, INC., notify Javelin at 212-422-6000 to arrange for its return or destruction.

The Coppelia Guide is a compendium of documents that integrate to cover the various aspects of Coppelia. The Coppelia Guide is part of a support effort that includes test scripts, a FAQ and a forthcoming troubleshooting guide on Javelin's website. The primary focus of these is to assist the client and add value to Javelin's product suite.

The present document is a ***Beta release***. There are a few sections that are blank and others that are incomplete. This work is under creation and revision.

Any comments or suggestions from users and readers would measureably enhance the quality of this document.

We use the terms *master* and *slave* purely as technical terms to define dependent functions and in no manner it reflects any insensitivity to a chiefly historical or social reference.

Thank you for your collaboration and support.

Javelin Technologies, Inc.

Contents

1.0 Introduction to FIX

FIX (for **F**inancial **I**nformation e**X**change) is a message protocol used to transmit and receive information related to financial transactions, such as orders, executions, cancels, and other pre-trading, trading, and post-trading related business messages. Starting in 1992, Fidelity Management and Research Co and Salomon Brothers, Inc. worked on a project linking the two companies' trading systems for the purpose of automated trading and error reduction. This initial effort evolved over the years past, and later in the process included more and more firms participating in the steering and technical committees of the FIX organization.

1.1 Background

1.1.1 Organization

In April 1999, FIX Protocol Limited was established. This is a private company limited by guarantee and formed in the United Kingdom. FIX Protocol Limited acts as the global umbrella for all FIX Protocol activities.

1.1.1.1 Global Steering Committee

Structurally, FIX is run by a Global Steering Committee that contains the co-chairs of regional steering committees in the U.S., Europe and Japan.

1.1.1.2 Global Technical Committee

On the technical side, a Global Technical Committee reports to the Global Steering Committee. The Global Technical Committee is responsible for maintaining the FIX specification and the FIX web site located at http://www.fixprotocol.org. The committee is composed of buy and sell side representatives from member firms whose responsibility is to ensure that the protocol supports the needs and requirements of the industry by leveraging their firm's specific implementation experiences. A buy-side co-chairman and sell-side co-chairman head the FIX Technical Committee.

1.1.1.3 Working Groups

Membership in FIX Committees is generally reserved for members of buy and sell side firms. However, specific technical and business working groups, which address unique needs and new work items are open to all interested parties. Technical working groups report their recommendations to the Global Technical Committee and business working groups report to the steering committees. Javelin Technologies, Inc. is represented in several such working groups.

1.2 The FIX Protocol

The FIX Protocol is a stream of ASCII characters, which is sent between two counter-parties. As of the time of writing, FIX is a point-to-point communication mechanism. While one FIX-enabled system can handle multiple connections, and one FIX session can handle information pertaining to more than one recipient or firm, publish-and-subscribe or broadcast-style dissemination is not implemented.

1.2.1 Layers

1.2.1.1 Network Protocol

FIX is designed to be independent of the network protocol used to transport the FIX message itself. TCP/IP is the option most widely used.

1.2.1.2 Session Layer

The Session Layer handles administrative messages like Logon and Logoff and ensures message delivery. FIX is based on an optimistic model; therefore, the term 'to ensure message delivery' should not be mistaken for 'guaranteed message delivery.' FIX incorporates an almost automatic recovery mechanism at the session layer, based on ordered sequencing in both directions of the FIX connection.

1.2.1.3 Application Layer

The Application Layer deals with the actual business-related content in the message flow. Among the messages considered "Application Messages" are Orders, Execution Reports, and Allocations, to name a few. All application messages are delivered via the session layer.

1.3 FIX Versions

1.3.1 Versions

Today, we are looking at five different versions of FIX:

2.7 (released in July, 1994 – DEFUNCT and generally not supported
3.0 (released in August, 1995)
4.0 (released in January, 1996)
4.1 (released in April, 1998)
4.2 (released in March, 2000)

For 1999, it was decided that the FIX Committee would not release any new versions. This decision allowed implementers to focus on Y2K issues and also catch up to the latest version of the protocol.

1.3.2 Differences

There are a number of differences between these FIX versions that are worth mentioning upfront. The items mentioned here do not pertain to the rather minor adjustments, such as adding a new field to a message, or clarifying a value allowed for a certain tag, but rather architectural and fundamental differences between the FIX versions.

1.3.2.1 From 3.0 to 4.0

Between versions 3.0 and 4.0 the session layer has been significantly altered.

In version 4.0 and above, every FIX message, whether administrative or application message, increments the message sequence number. In versions 3.0 and below, only application messages would increment the sequence number. All administrative messages would contain a sequence number that denoted the next expected application message's sequence number. Therefore, sequence numbers could exist more than once within the same FIX session. In the case of a resend request, only messages with a 'real' sequence number would be resent. As a business logic enhancement, some people would modify their software so that a reject message increments the sequence number as well, for the purpose of being able to request a resend of that type of (administrative) message.
In version 4.0, the concept of GapFill was introduced. This mechanism comes into effect when a resend request is received by one party, and the range of sequence numbers requested contains one ore more administrative messages, such as heartbeats. Instead of actually resending these messages that have no business related meaning, the sequence reset message was modified to contain an identifier, marking it as a GapFill. This message then means that a gap between sequence numbers is being bridged, for example 'this is sequence number 10, expect sequence number 20 as my next message, since there is no relevant information to be resent between 10 and 20'.
In version 4.0 and above, each party is required to send a logon, whether that party actually initiates the FIX connection by sending the logon message first, or reacts to an incoming logon message.
The meaning of the (administrative) reject message has been clarified. In version 3.0 and below, a reject message was sometimes used to reject an order directly from the executing side's application. However, the nature of the reject message being administrative, emphasis was placed on the fact that a reject should be used only to reject message for administrative reasons, such as insufficient fields, values within fields out of bounds, incorrect body-length or checksum, or other related errors. To reject an order, for example, one would exclusively use an execution report message, and set the order status tag within the execution report message to "rejected".

In version 4.0, the tag OrigSendingTime (122) was added to the message header. This time denotes the time the message has been sent originally from the originating system in the case of a resend of this message.

From version 4.0 on, the time and date fields within FIX message have been combined and normalized to be in the format YYYYMMDD-HH:MM:SS. Formerly, the

SendingDate tag had the format YYMMDD, and the SendingTime tag was separate from the SendingDate tag. The SendingDate tag no longer exists in version 4.0 and above.

There have been additional changes to several messages, and additional message types were introduced as well. Lastly, changes have been incorporated to allow broader international use of the FIX protocol. The official specifications for all FIX versions are freely available at http://www.fixprotocol.org, and are also part of this document.

1.3.2.2 From 4.0 to 4.1

FIX version 4.1 introduced the following changes in the session layer:

Addition of the ResetSeqNumFlag into the logon message. This flag enables (rudimentary) 24 by 7 operation of a FIX engine by allowing the reset of sequence numbers 'on the fly' by means of a logon message.
Clarification of the fact that NewSeqNum means 'next transmitted / to be expected' sequence number.

Additional changes introduced with version 4.1 include extensive modifications to the allocation message, and the addition of option-related fields into order and execution report messages. There have been a number of minor clarifications and redefinitions that were published with version 4.1 of the FIX protocol, including a re-definition of the business flow or order flow model.

1.3.2.3 From 4.1 to 4.2

A few minor changes to the session level have been made for release 4.2 of the FIX specifications. The changes include:

Remove MsgSeqNum upper bounds
Remove restriction of message length
Use "-1" as the value to denote "infinity" for resend requests
Include millisecond precision for all time stamps
Discovery and / or negotiation of the counter party's support of messages by means of optional fields in the logon message
Include new optional fields XmlData and XmlDataLen to enable 'embedded FIXML'
Numerous data type and data type definition changes
Introduction of LastSeqNumProcessed tag to identify delays at the other side of a FIX connection

There are quite a few changes that have been made on the application layer, and in the documentation of the protocol itself. A good indication is the following table:

	3.0	4.0	4.1	4.2
Released	09/1995	01/1996	04/1998	04/2000
# of Admin Messages	7	7	7	7
# of Business Messages	17	20	21	39
# Fields	112	138	208	396
# Appendices in document	4	4	7	16
# Pages in document	57	69	106	265

Specifically, some of the changes in the application layer include:

Enhanced Good-Til (non-day order) model
Introduction of a Business Reject Message
Pre-allocation of orders now possible
Introduced procedures to detect and handle stale orders
Execution reports support CMS-style multi-fills
Introduction of Mass Quoting mechanism
Introduction of Security Definition, Security Status, and Trading Session Status messages
Introduction of Trading Session ID model to distinguish between pre- and after-hours trading sessions
Introduction of Market Data messages to publish ECN and exchange book information
Multicast extensions to support market data feed transmissions in a semi-FIX way without extensive session layer
Extensive Program and List Trading enhancements
Introduction of FIXML

1.4 FIX in Detail

1.4.1 Message Format

The general format of a FIX message as per specifications is a standard header followed by the message body fields and terminated with a standard trailer. See also the following section showing an example.

Each message is constructed of a stream of `<tag>=<value>` fields.

Except where explicitly stated otherwise, fields within a message can be defined in any sequence, in other words, the relative position of a field within a record is inconsequential. Any exceptions are defined otherwise:

Four header / trailer fields (BeginString, BodyLength, MsgType, and CheckSum)
fields within repeating data groups, and
general message format of standard header followed by body followed by standard trailer.

It is permissible for fields to be repeated. In the case where a field allows multiple values, these repeating fields are logically added together to form the data for that field. It is also possible for a field to be contained in both the clear text portion and the encrypted data sections of the same message. This is useful for validation and verification. For example, sending the *SenderCompID* in the encrypted data section can be used as a rudimentary validation technique. In the cases where the clear text data differs from the encrypted data, the encrypted data should be considered more reliable; however, a security warning should be generated.

All fields, including those of data type *data*, i.e. SecureData, RawData, SignatureData, etc.) in a FIX message are terminated by a delimiter character. The non-printing, ASCII "SOH" (#001), is used for field termination. Records are delimited by the "SOH" character following the CheckSum field. All records begin with the "8=FIX.x.y" string and terminate with "10=nnn<SOH>".

There shall be no embedded delimiter characters within fields except for data type *data*. If delimiters are necessary, they must be different from the ASCII SOH character used here.

1.4.2 Anatomy of a FIX Message

All FIX messages begin with a standard header and end with a standard trailer. The header has the following structure, for example:

Tag	*Field Name*	*Contents*	*Comments*
8	BeginString	FIX.4.2	Protocol version.
9	BodyLength	99999	Length in byte of the message body.
35	MsgType		Message Type: 0 = Heartbeat 1 = Test Request 2 = Resend Request 3 = Reject 4 = Sequence Reset 5 = Logout 6 = Indication of Interest 7 = Advertisement 8 = Execution Report 9 = Order Cancel Reject A = Logon B = News C = Email D = Order – Single E = Order – List F = Order Cancel Request G = Order Cancel/Replace Request H = Order Status Request J = Allocation K = List Cancel Request L = List Execute M = List Status Request N = List Status P = Allocation ACK Q = Don't Know Trade (DK) R = Quote Request S = Quote T = Settlement Instructions V = Market Data Request W = Market Data-Snapshot/Full Refresh X = Market Data-Incremental Refresh Y = Market Data Request Reject Z = Quote Cancel a = Quote Status Request b = Quote Acknowledgement c = Security Definition Request d = Security Definition e = Security Status Request f = Security Status g = Trading Session Status Request h = Trading Session Status i = Mass Quote j = Business Message Reject k = Bid Request l = Bid Response m = List Strike Price
49	SenderCompID		Firm ID.
50	SenderSubID		User ID.
56	TargetCompID		Firm ID.

57	TargetSubID		User ID.
34	MsgSeqNum	999999	Next expected sequence number.
43	PossDupFlag		Optional: Indicates possible retransmission of this sequence number: Y = Possible duplicate. N = Original transmission.
97	PossResend		Optional: Indicates that message may contain information that has been sent under another sequence number: Y = Possible resend. N = Original transmission.
52	SendingTime	YYYYMMDD-HH:MM:SS	GMT

The header is followed by the Application Data part of the FIX message, the actual contents. Last is the standard checksum trailer:

10	CheckSum	0..255	Byte Sum

The following sample logon message serves as an example of how the message length and checksum fields are calculated. Field delimiters (ASCII SOH) are depicted as vertical bars '|'.

Body length (9=0059) is 59 characters long.

```
8=FIX.4.2|9=0059| 35=A|49=BULL|56=JAVT|34=1|52=19990815-20:03:21|98=0|108=30| 10=xxx|
```

Checksum (10=xxx) is modulo 256 of the simple byte sum of these characters.

1.4.3 A Sample FIX Message

The following is a sample FIX message generated by Coppelia, and is further explained below to clarify the way a FIX message is built:

8=FIX.4.0|9=0174|35=8|56=SLGM|49=SBI|34=7|50=HARRY|57=GEORGE|52=19990816:15:28:30|37=Order#85|11=1|17=498|20=0|39=2|55=INTC|54=1|38=1000|40=2|44=44.0|59=1|32=1000|31=43.875|14=1000|6=43.875|10=208|

"Tag=Value" Pair	TagName	Meaning
8=FIX4.0	BeginString	The FIX version is 4.0
9=0174	BodyLength	The message body is 174 bytes long.
35=8	MsgType	This is an execution report (8 means execution report).
56=SLGM	TargetCompID	Where the message is headed, here it's Seligman (SLGM) for example purposes.
49=SBI	SenderCompID	Where the message is from, here it is SBI for Salomon Brothers, Inc.
34=7	MsgSeqNum	This message's sequence number.
50=HARRY	SenderSubID	The trader or desk this message is from, here it's Harry at SBI.
57=GEORGE	TargetSubID	The trader or desk the message is directed to, here it's George at SLGM.
52=19990816-15:28:30	SendingTime	Always in GMT.
37=Order #85	OrderID	The ID given to the order that resulted in this Execution Report by SBI.
11=1	ClOrdID	Client Order ID. This is the ID Seligman's system gave this order for their own purposes.
17=498	ExecID	The ID of this execution report message.
20=0	ExecTransType	This is a new (0) transaction.
39=2	OrdStatus	The order referenced in this execution report has been filled (status 2).
55=INTC	Symbol	Intel
54=1	Side	This was a BUY order.
38=1000	OrderQty	Number of shares, here 1000.
40=2	OrderType	This was a limit order.
44=44.0	Price	The limit was 44.00.
59=1	TimeInForce	This was a GTC (Good-'til-Cancel) order.
32=1000	LastShares	The number of shares filled by means of this particular execution report.
31=43.875	LastPx	The price at which the 1000 shares have been filled, here 43 7/8.
14=1000	CumQty	The total number of shares filled on this ORDER.
6=43.875	AvgPx	The average price for all fills on this order (since there was only this one in this order example, it's the same as last price).
10=208	CheckSum	The modulo 256 of the byte-sum of this message.

The first grayed part is the header, the second gray part represents the trailer. The section in-between is the actual application data section.

1.5 Session Layer

A FIX session is defined as a bi-directional stream of ordered messages between two parties within a continuous sequence number series. A single FIX session can exist across multiple physical connections. Parties can connect and disconnect multiple times while maintaining a single FIX session. Connecting parties must bi-laterally agree as to when sessions are to be started or stopped based upon individual system and time zone requirements. It is recommended that a new FIX session be established once within each 24 hour period.

It is possible to maintain 24 hour connectivity and establish a new set of sequence numbers by sending a Logon message with the ResetSeqNumFlag set. This feature was made available in FIX version 4.1.

The FIX session protocol is based on an optimistic model. Normal delivery of data is assumed (i.e. no communication level acknowledgment of individual messages) with errors in delivery identified by message sequence number gaps. This section provides details on the implementation of the FIX session layer and dealing with message sequence gaps.

1.5.1 FIX Session

A FIX session is comprised of three parts: Logon, Message Exchange, and Logout.

Logon

Establishing a FIX connection involves three distinct operations:

Creation of a telecommunications level link
Authentication and acceptance of the initiating party by the accepting party
Message synchronization (initialization)

The sequence of events is as follows:

The session initiator establishes a telecommunication link with the session acceptor. The initiator sends a *Logon* message. The acceptor will authenticate the identity of the initiator by examining the *Logon* message. The *Logon* message will contain the data necessary to support the previously agreed upon authentication method. If the initiator is successfully authenticated, the acceptor responds with a *Logon* message (FIX versions 4.0 and above). If authentication fails, the session acceptor should shut down the connection. The session initiator may begin to send messages immediately following the *Logon* message, however, the acceptor may not be ready to receive them. The initiator must wait for the confirming *Logon* message from the acceptor before declaring the session fully established.

After the initiator has been authenticated, the acceptor will respond immediately with a confirming *Logon* message.

Depending on the encryption method being used for that session, this *Logon* message may or may not contain the same session encryption key. The initiator side will use the *Logon* message being returned from the acceptor as confirmation that a FIX session has been established. If the session acceptor has chosen to change the session encryption key, the session initiator must send a third *Logon* back to the other side in order to acknowledge the key change request. This also allows the session acceptor to know when the session initiator has started to encrypt using the new session key. Both parties are responsible for infinite loop detection and prevention during this phase of the session.

After authentication, the initiator and acceptor must synchronize their messages through interrogation of the *MsgSeqNum* field before sending any queued or new messages. A comparison of the *MsgSeqNum* in the *Logon* message to the internally monitored next expected sequence number will indicate any message gaps. Likewise, the initiator can detect gaps by comparing the acknowledgment *Logon* message *MsgSeqNum* to the next expected value. The section on message recovery later in this document deals with message gap handling.

Note on 24x7 Connectivity

When using the ResetSeqNumFlag to maintain 24 hour connectivity and establish a new set of sequence numbers, the process should be as follows:

Both sides should agree on a reset time and the party that will be the initiator of the process. Note that the initiator of the ResetSeqNum process may be different than the initiator of the Logon process. One side will initiate the process by sending a TestRequest and wait for a Heartbeat in response to ensure of no sequence number gaps. Once the Heartbeat has been received, the initiator should send a Logon with ResetSeqNumFlag set to Y and with MsgSeqNum of 1. The acceptor should respond with a Logon with ResetSeqNumFlag set to Y and with MsgSeqNum of 1. At this point new messages from either side will continue with MsgSeqNum of 2. It should be noted that once the initiator sends the Logon with the ResetSeqNumFlag set, the acceptor must obey this request and the message with the last sequence number transmitted “yesterday” will no longer be available, which means that Resend Request messages asking for “old” sequence numbers will be rejected.

Message Exchange

After completion of the initialization process, normal message exchange begins. 'Application Messages'.

Logout

Normal termination of the message exchange session will be completed via the exchange of *Logout* messages. Termination by other means should be considered an abnormal condition and dealt with as an error. Session termination without receiving a Logout should treat the counter party as logged out.

It is recommended that a TestRequest be issued before sending the Logout message to force a Heartbeat from the other side. This helps to ensure that there are no sequence number gaps.

Note: Logging out does not affect the state of any orders. All active orders will continue to be eligible for execution after logout, unless specifically agreed upon otherwise with the counter-party.

1.5.2 Message Recovery

During initialization, or in the middle of a FIX session, message gaps may occur which are detected via the tracking of incoming sequence numbers. Coppelia tracks both the incoming and outgoing message sequence numbers automatically. The following section provides details on how messages are recovered.

Each message is assigned a unique (by connection) sequence number which is incremented after each message. Likewise, every message received has a unique sequence number and the incoming sequence counter is incremented after each message.

When the incoming sequence number does not match the expected number, corrective processing is required.

If the incoming message has a sequence number less than expected and the PossDupFlag is not set, this indicates a serious error. Coppelia will reject this message, and terminate the connection immediately. Manual intervention is necessary in this case.

If the incoming sequence number is greater than expected, this indicates that messages were missed and retransmission of the messages is requested via the *Resend Request*. This is automatically handled by Coppelia.

Upon receipt of a *Resend Request,* the resender can respond in one of three ways:

Retransmit the requested messages (in order) with the original sequence numbers and *PossDupFlag* set to "Y". This will be the case if application messages were in the range of messages requested to be resent.
Issue a *SeqReset-GapFill with PossDupFlag set to "Y"* message to replace the retransmission of administrative and application messages (this is a feature first introduced in FIX4.0). Administrative messages are generally not resent (except *Rejects*), but rather "gap-filled" over in this manner. See the section below describing administrative messages.

Issue a *SeqReset-Reset with PossDupFlag set to "Y"* to force sequence number synchronization.

The sequence number of the *SeqReset-GapFill* message is the next expected outbound sequence number. The *NewSeqNum* field of the GapFill message contains the sequence number of the highest administrative message in this group plus 1.

For example, during a Resend operation there are 7 sequential administrative messages waiting to be resent. They start with sequence number 9 and end with sequence number 15. Instead of transmitting 7 Gap Fill messages (which is perfectly legal, but not network friendly), a *SeqReset-GapFill* message may be sent. **The sequence number of the Gap Fill message is set to 9 because the remote side is expecting that as the next sequence number.** The *NewSeqNum* field of the GapFill message contains the number 16, because that will be the sequence number of the next message to be transmitted. All this is automatically taken care of by Coppelia, which sends ONE *SeqReset-GapFill* message only.

1.5.3 Administrative Messages

Administrative messages are what the session layer of FIX is comprised of. The following message types are considered part of the session layer:

Message Type Designation	Message
A	Logon
0	Heartbeat
1	Test Request
2	Resend Request
3	Reject
4	Sequence Reset / Gap Fill
5	Logout

1.6 Application Layer

The FIX application layer deals with the actual business related data content of a FIX session.

1.6.1 Application messages

The following messages are considered part of the application layer as of FIX 4.2.

Message Type Designation	Message
6	Indication of Interest
7	Advertisement
8	Execution Report
9	Order Cancel Reject
B	News
C	Email
D	Order – Single
E	Order – List
F	Order Cancel Request
G	Order Cancel/Replace Request
H	Order Status Request
J	Allocation
K	List Cancel Request
L	List Execute
M	List Status Request
N	List Status
P	Allocation ACK
Q	Don't Know Trade (DK)
R	Quote Request
S	Quote
T	Settlement Instructions
V	Market Data Request
W	Market Data – Snapshot / Full Refresh
X	Market Data – Incremental Refresh
Y	Market Data Request Reject
Z	Quote Cancel
a	Quote Status Request
b	Quote Acknowledgement
c	Security Definition Request
d	Security Definition
e	Security Status Request
f	Security Status
g	Trading Session Status Request
h	Trading Session Status
i	Mass Quote
j	Business Message Reject
k	Bid Request

l	Bid Response
m	List Strike Price

1.6.2 Routing of Messages

The fields SenderCompID and TargetCompID in the header of a FIX message are used to route messages from sender to receiver companies. In a third-party setup, where a FIX engine functions as a router, additional fields, such as OnBehalfOf...IDs and DeliverTo...IDs can be used to identify the original sender, and the 'end user' of the message sent. The values of these fields are agreed upon between parties before establishing a connection. In many cases, the market maker acronym of a company is used, if one exists. The sub routing fields, SenderSubID and TargetSubID are used to identify senders and receivers on a lower level, such as DESK or TRADERNAME, for example.

In the following two tables, you are the firm of the name BROKER, the client to your FIX connection is called FIRM in this example. The tables give a brief example of how the header fields are filled in order to route messages properly.

Messages *from clients* could be filled as:

Tag	*Field Name*	*Contents*	*Comments*
49	SenderCompID	FIRM	From company "FIRM"
50	SenderSubID	USER	From a user at "FIRM"
56	TargetCompID	BROKER	To firm BROKER
57	TargetSubID	DESK	To destination DESK within BROKER

while messages *to clients* could be filled as:

Tag	*Field Name*	*Contents*	*Comments*
49	SenderCompID	BROKER	From firm BROKER
50	SenderSubID	DESK	From DESK within BROKER
56	TargetCompID	FIRM	To company "FIRM"
57	TargetSubID	USER	To user "USER" at firm "FIRM"

1.6.3 Identification of Messages

The following fields are an example of how to identify and correlate messages.

Tag#	Field Name	Description
34	MsgSeqNum	This value is only useful at the session level to identify messages that have been sent. This value does not increment when the connection between you and a client is abnormally disconnected.
97	PossResend	The application sets this tag to 'Y' for messages that are sent / received 'again', but under a different sequence number. For example, that would happen when an Order Status is requested, and the Execution Reports are being sent to the requesting party.
37	OrderID	Each order accepted by a broker, ECN, or you for that matter should be assigned a daily-unique number. The value in this field then accompanies all Execution Report messages against this order.
11	ClientOrderId	Each order received via FIX must be tagged with this identifier value. Most applications accept up to 32 characters. This field will accompany all Execution Report messages received against this order. The difference between tag 11 and tag 37 is that the value in tag 37 is assigned by the receiver of the order, while the value in tag 11 is assigned by the sender of the order.
17	ExecID	Each distinct trade execution performed by a broker, exchange, or ECN is assigned a daily-unique number. Note that FIX requires an ExecID for all (including new and canceled) orders. Since some applications do not consider a new or cancel transaction an execution, a value of zero can be used in these cases. This becomes very important to know when checking for duplicates, since the value of zero can occur more than once during a session or day.

This table is not necessarily complete, but shows a point to start from when identifying messages.

1.6.4 Order and Execution Exchange

Ordinarily, each order placed will result in a series of Execution Report messages detailing its status from its initial acceptance (New, Rejected, or Canceled) to its eventual completion (Partial Fill, Filled, Canceled, etc).

Please note that the FIX related term "Execution Report" does not necessarily mean it contains an execution or fill. An Execution Report is a very versatile message, which can be used to acknowledge or reject an order, acknowledge a cancel request, and submit full or partial fills, for example.

Most applications uniquely identify each trade execution. Please also see the previous section on message identification. These trade execution identifiers are sent back in the ExecID (integer) field in Execution Report messages. Since an order may be incrementally filled, an order may be associated with many trade executions. Execution IDs are unique but not necessarily sequential per order.

FIX also requires an ExecID field for new order confirmations. In some cases, applications will set the ExecID tag in this message to zero, which is not a mistake per se, but can make the detection of duplicates a little harder.

1.6.5 Order State Change Matrixes

The following section shows and explains the FIX order state change matrixes with notes as they can be found in the specification document of FIX 4.2.

Notes:

The following state change diagrams are presented from the broker's point of view. X refers to the original order, Y refers to the cancel / replacing order. If you are not running Coppelia in a sell-side mode, i.e., you are not a broker receiving orders but rather a buy-side money manager (for example), and you are sending orders, you need to substitute 'receive' with 'send' in the following tables.

Any fills which occur and need to be communicated to the customer of that broker while an order is "pending" and waiting to achieve a new state (i.e., via a Order Cancel Replace Request) must contain the "original" (current order prior to state change request) order parameters (i.e., ClOrdID, OrderQty, LeavesQty, Price, etc). An order cannot be considered replaced until it has been explicitly accepted and confirmed to have reached the replaced status (i.e. OrdStatus = "Replaced"). Care should be taken as the replaced order could still have reports forthcoming which will update the CumQty and AvgPx of both the original and replacement order, however, the effect on the replacement (ClOrdID, new quantity or limit price, etc.) will not be seen until a report on the replacement has been generated.

When a "Fill Or Kill" (FOK) order cannot be filled or an "Immediate Or Cancel" (IOC) order cannot be immediately hit, the proper response to "kill" the order is an Execution Report with ExecType= "Cancelled". Note that this is the equivalent of an "UNSOLICITED UR OUT" CMS message.

The equivalent of a "NOTHING DONE" CMS message should be sent as a "status report", i.e., an Execution Report message with ExecTransType = "Status" and ExecType and OrdStatus equal to that of the previous ExecutionRpt message for this order (usually "New" or "Replaced" when "nothing has been done").

The table below shows which state transitions have been illustrated by the matrices in this section of the document. The row represents the current value of OrdStatus and the column represents the next value as reported back to the buy-side via an execution report or order cancel reject message. Next to each OrdStatus value is its precedence – this is used when the order exists in a number of states simultaneously to determine the value that should be reported back. Note that absence of a scenario should not necessarily be interpreted as meaning that the state transition is not allowed.

OrdStatus (precedence value)	New (2)	Partially Filled (4)	Filled (8)	Done For Day (10)	Pending Cancel (12)	Pending Replace (11)	Replaced (3)	Canceled (5)	Rejected (2)	Stopped (7)
Pending New (2)	*								*	
New (2)	*	*	*	*	*	*	*		*	*
Partially Filled (4)		*	*	*	*	*		*		
Filled (8)		*	*			*				
Done for Day (10)		*								
Pending Cancel (12)	*	*	*		*			*		
Pending Replace (11)	*	*	*			*	*	*		
Replaced (3)		*								
Canceled (5)										
Rejected (2)										
Stopped (7)		*								

How to read the Order State Change Matrices:

- The 'Execution Report' message is referred to simply as 'Execution'
- The 'Order Cancel/Replace Request' and 'Order Cancel Request' messages are referred to as 'Replace Request' and 'Cancel Request' respectively
- The shaded rows represent messages sent from buy-side to the sell-side
- In general where two lines of a matrix share the same time, this means either
    - that there are two possible paths (e.g. a request is accepted or rejected) – in this case the first row of the two possible paths is the reject case which is italicized. The non-italicized row is the path that is continued by the remainder of the matrix
    - that two messages are being sent at the same time but in different directions such that the messages cross on the connection (e.g. a cancel request is sent at the same time as the sell-side is sending an execution) – in this case both lines have bold text
- For scenarios involving cancel requests or cancel/replace requests ' X' refers to the original order, 'Y' refers to the cancel/replacing order. A similar convention is used for corrections or cancels to executions

1.6.5.1 Filled order

Time	**Message Received** (ClOrdID, OrigClOrdID)	**Message Sent** (ClOrdID, OrigClOrdID)	**Exec Type**	**OrdStatus**	**Exec Trans Type**	**Order Qty**	**Cum Qty**	**Leaves Qty**	**Last Shares**	**Comment**
1	New Order(X)					10000				
2		*Execution(X)*	*Rejected*	*Rejected*	*New*	*10000*	*0*	*0*	*0*	*If order is rejected by sales*
2		Execution(X)	New	New	New	10000	0	10000	0	
3		*Execution(X)*	*Rejected*	*Rejected*	*New*	*10000*	*0*	*0*	*0*	*If order is rejected by trader/exchange*
3		Execution(X)	Partial Fill	Partially Filled	New	10000	2000	8000	2000	Execution of 2000
4		Execution(X)	Partial Fill	Partially Filled	New	10000	3000	7000	1000	Execution of 1000
5		Execution(X)	Fill	Filled	New	10000	10000	0	7000	Execution of 7000

1.6.5.2 Part-filled day order, done for day

Time	**Message Received** (ClOrdID, OrigClOrdID)	**Message Sent** (ClOrdID, OrigClOrdID)	**Exec Type**	**OrdStatus**	**Exec Trans Type**	**Order Qty**	**Cum Qty**	**Leaves Qty**	**Last Shares**	**Comment**
1	New Order(X)					10000				
2		*Execution(X)*	*Rejected*	*Rejected*	*New*	*10000*	*0*	*0*	*0*	*If order is rejected*
2		Execution(X)	New	New	New	10000	0	10000	0	
3		Execution(X)	Partial Fill	Partially Filled	New	10000	2000	8000	2000	Execution of 2000
4		Execution(X)	Partial Fill	Partially Filled	New	10000	3000	7000	1000	Execution of 1000
5		Execution(X)	Done for Day	Done for Day	New	10000	3000	0	0	Assuming day order. See other examples which cover GT orders

1.6.5.3 Cancel request issued for a zero-filled order

Time	**Message Received** (ClOrdID, OrigClOrdID)	**Message Sent** (ClOrdID, OrigClOrdID)	**Exec Type**	**OrdStatus**	**Exec Trans Type**	**Order Qty**	**Cum Qty**	**Leaves Qty**	**Last Shares**	**Comment**
1	New Order(X)					10000				
2		*Execution(X)*	*Rejected*	*Rejected*	*New*	*10000*	*0*	*0*	*0*	*If order is rejected*
2		Execution(X)	New	New	New	10000	0	10000	0	
3	Cancel Request(Y,X)					10000				
4		*Cancel Reject (Y,X)*		*New*		*10000*				*If rejected by salesperson*
4		Execution (Y,X)	Pending Cancel	Pending Cancel	New	10000	0	10000	0	
5		*Cancel Reject (Y,X)*		*New*		*10000*				*If rejected by trader/exchange*
5		Execution (Y,X)	Canceled	Canceled	New	10000	0	0	0	

1.6.5.4 Cancel request issued for a part-filled order; executions occur while cancel request is active

Time	**Message Received** (ClOrdID, OrigClOrdID)	**Message Sent** (ClOrdID, OrigClOrdID)	**Exec Type**	**OrdStatus**	**Exec Trans Type**	**Order Qty**	**Cum Qty**	**Leaves Qty**	**Last Shares**	**Comment**
1	New Order(X)					10000				
2		*Execution(X)*	*Rejected*	*Rejected*	*New*	*10000*	*0*	*0*	*0*	*If order is rejected*
2		Execution(X)	New	New	New	10000	0	10000	0	
3		Execution(X)	Partial Fill	Partially Filled	New	10000	2000	8000	2000	Execution for 2000
4	**Cancel Request(Y,X)**					**10000**				
4		**Execution(X)**	**Partial Fill**	**Partially Filled**	**New**	**10000**	**5000**	**5000**	**3000**	**Execution for 3000. This execution passes the cancel request on the connection**
5		*Cancel Reject (Y,X)*		*Partially Filled*		*10000*				*If request is rejected*
5		Execution (Y,X)	Pending Cancel	Pending Cancel	New	10000	5000	5000	0	'Pending cancel' order status takes precedence over 'partially filled' order status
6		Execution(X)	Partial Fill	Pending Cancel	New	10000	6000	4000	1000	Execution for 1000 whilst order is pending cancel – 'pending cancel' order status takes precedence over 'partially filled' order status
7		*Cancel Reject (Y,X)*		*Partially Filled*		*10000*				*If request is rejected*
7		Execution (Y,X)	Canceled	Canceled	New	10000	6000	0	0	'Canceled' order status takes precedence over 'partially filled' order status

1.6.5.5 Cancel request issued for an order that becomes filled before cancel request can be accepted

Time	**Message Received** (ClOrdID, OrigClOrdID)	**Message Sent** (ClOrdID, OrigClOrdID)	**Exec Type**	**OrdStatus**	**Exec Trans Type**	**Order Qty**	**Cum Qty**	**Leaves Qty**	**Last Shares**	**Comment**
1	New Order(X)					10000				
2		*Execution(X)*	*Rejected*	*Rejected*	*New*	*10000*	*0*	*0*	*0*	*If order is rejected*
2		Execution(X)	New	New	New	10000	0	10000	0	
3		Execution(X)	Partial Fill	Partially Filled	New	10000	2000	8000	2000	Execution for 2000
4	**Cancel Request(Y,X)**					**10000**				
4		**Execution(X)**	**Partial Fill**	**Partially Filled**	**New**	**10000**	**5000**	**5000**	**3000**	**Execution for 3000. This execution passes the cancel request on the connection**
5		*Cancel Reject (Y,X)*		*Partially Filled*		*10000*				*If request is rejected*
5		Execution (Y,X)	Pending Cancel	Pending Cancel	New	10000	5000	5000	0	'Pending cancel' order status takes precedence over 'partially filled' order status
6		Execution(X)	Fill	Pending Cancel	New	10000	10000	0	5000	Execution for 5000 whilst order is pending cancel. 'Pending cancel' order status takes precedence over 'filled' order status
7		Cancel Reject (Y,X)		Filled		10000				Cancel request rejected – CxlRejectReason = 0 (too late to cancel)

1.6.5.6 Zero-filled order, cancel/replace request issued to increase order qty

Time	**Message Received** (ClOrdID, OrigClOrdID)	**Message Sent** (ClOrdID, OrigClOrdID)	**Exec Type**	**OrdStatus**	**Exec Trans Type**	**Order Qty**	**Cum Qty**	**Leaves Qty**	**Last Shares**	**Comment**
1	New Order(X)					10000				
2		*Execution(X)*	*Rejected*	*Rejected*	*New*	*10000*	*0*	*0*	*0*	*If order is rejected by broker*
2		Execution(X)	New	New	New	10000	0	10000	0	
3	Replace Request(Y,X)					11000				Request to increase order qty to 11000
4		*Cancel Reject (Y,X)*		*New*		*10000*				*If request is rejected by salesperson*
4		Execution (Y,X)	Pending Replace	Pending Replace	New	10000	0	10000	0	
5		*Cancel Reject (Y,X)*		*New*		*10000*				*If rejected by trader/exchange*
5		Execution (Y,X)	Replace	Replaced	New	11000	0	11000	0	'Replaced' order status takes precedence over 'new' order status
6		Execution (Y)	Partial Fill	Partially Filled	New	11000	1000	10000	1000	Execution for 1000
7		Execution (Y)	Partial Fill	Partially Filled	New	11000	3000	8000	2000	Execution for 2000

1.6.5.7 Part-filled order, followed by cancel/replace request to increase order qty, execution occurs while order is pending replace

Time	**Message Received** (ClOrdID, OrigClOrdID)	**Message Sent** (ClOrdID, OrigClOrdID)	**Exec Type**	**OrdStatus**	**Exec Trans Type**	**Order Qty**	**Cum Qty**	**Leaves Qty**	**Last Shares**	**Comment**
1	New Order(X)					10000				
2		*Execution(X)*	*Rejected*	*Rejected*	*New*	*10000*	*0*	*0*	*0*	*If order is rejected by broker*
2		Execution(X)	New	New	New	10000	0	10000	0	
3		Execution(X)	Partial Fill	Partially Filled	New	10000	1000	9000	1000	Execution for 1000
4	Replace Request(Y,X)					12000				Request increase in order quantity to 12000
5		*Cancel Reject (Y,X)*		*Partially Filled*		*10000*				*If request is rejected*
5		Execution (Y,X)	Pending Replace	Pending Replace	New	10000	1000	9000	0	'Pending replace' order status takes precedence over 'partially filled' order status
6		Execution(X)	Partial Fill	Pending Replace	New	10000	1100	8900	100	Execution for 100 before cancel/replace request is responded to
7		*Cancel Reject (Y,X)*		*Partially Filled*		*10000*				*If request is rejected*
7		Execution (Y,X)	Replace	Partially Filled	New	12000	1100	10900	0	'Partially filled'' order status takes precedence over 'replaced' order status
8		Execution(Y)	Fill	Filled	New	12000	12000	0	10900	Execution for 10900

1.6.5.8 Filled order, followed by cancel/replace request to increase order quantity

Time	**Message Received** (ClOrdID, OrigClOrdID)	**Message Sent** (ClOrdID, OrigClOrdID)	**Exec Type**	**OrdStatus**	**Exec Trans Type**	**Order Qty**	**Cum Qty**	**Leaves Qty**	**Last Shares**	**Comment**
1	New Order(X)					10000				
2		*Execution(X)*	*Rejected*	*Rejected*	*New*	*10000*	*0*	*0*	*0*	*If order is rejected by broker*
2		Execution(X)	New	New	New	10000	0	10000	0	
3		Execution(X)	Fill	Filled	New	10000	10000	0	10000	Execution for 10000
4	Replace Request(Y,X)					12000				Request increase in order quantity to 12000
5		*Cancel Reject (Y,X)*		*Filled*		*10000*				*If request is rejected*
5		Execution (Y,X)	Pending Replace	Pending Replace	New	10000	10000	0	0	'Pending replace' order status takes precedence over 'partially filled' order status
6		*Cancel Reject (Y,X)*		*Filled*		*10000*				*If request is rejected*
6		Execution (Y,X)	Replace	Partially Filled	New	12000	10000	2000	0	'Partially filled' order status takes precedence over 'replaced' order status.
7		Execution(Y)	Fill	Filled	New	12000	12000	0	2000	Execution for 2000

1.6.5.9 Cancel/replace request (not for quantity change) is rejected as a fill has occurred

Time	**Message Received** (ClOrdID, OrigClOrdID)	**Message Sent** (ClOrdID, OrigClOrdID)	**Exec Type**	**OrdStatus**	**Exec Trans Type**	**Order Qty**	**Cum Qty**	**Leaves Qty**	**Last Shares**	**Comment**
1	New Order(X)					10000				
2		*Execution(X)*	*Rejected*	*Rejected*	*New*	*10000*	*0*	*0*	*0*	*If order is rejected by broker*
2		Execution(X)	New	New	New	10000	0	10000	0	
3		Execution(X)	Partial Fill	Partially Filled	New	10000	1000	9000	1000	Execution for 1000
4	**Replace Request(Y,X)**					**10000**				**Assume in this scenario that client does not wish to increase qty (e.g. client wants to amend limit price)**
4		**Execution (X)**	**Fill**	**Filled**	**New**	**10000**	**10000**	**0**	**9000**	**Execution for 9000 – the replace request message and this execution report pass each other on the connection**
5		Cancel Reject (Y,X)		Filled		10000				CxlRejectReason = 0 (too late to cancel)

1.6.5.10 Cancel/replace request sent while execution is being reported – the requested order qty exceeds the cum qty. Order is replaced then filled

Time	**Message Received** (ClOrdID, OrigClOrdID)	**Message Sent** (ClOrdID, OrigClOrdID)	**Exec Type**	**OrdStatus**	**Exec Trans Type**	**Order Qty**	**Cum Qty**	**Leaves Qty**	**Last Shares**	**Comment**
1	New Order(X)					10000				
2		*Execution(X)*	*Rejected*	*Rejected*	*New*	*10000*	*0*	*0*	*0*	*If order is rejected*
2		Execution(X)	New	New	New	10000	0	10000	0	
3		Execution(X)	Partial Fill	Partially Filled	New	10000	1000	9000	1000	Execution for 1000
4	**Replace Request(Y,X)**					**8000**				**Request a decrease order quantity to 8000 (leaving 7000 open)**
4		**Execution(X)**	**Partial Fill**	**Partially Filled**	**New**	**10000**	**1500**	**8500**	**500**	**Execution for 500 sent. Replace request and this execution report pass each other on the connection**
5		*Cancel Reject (Y,X)*		*Partially Filled*		*10000*				*If request is rejected by salesperson*
5		Execution (Y,X)	Pending Replace	Pending Replace	New	10000	1500	8500	0	'Pending replace' order status takes precedence over 'partially filled' order status
6		Execution(X)	Partial Fill	Pending Replace	New	10000	1600	8400	100	Execution for 100 occurs before cancel/replace request is accepted
7		*Cancel Reject (Y,X)*		*Partially Filled*		*10000*				*If request is rejected by trader/exchange*
7		Execution (Y,X)	Replace	Partially Filled	New	8000	1600	6400	0	'Partially filled' order status takes precedence over 'replaced' order status. Replace is accepted as requested order qty exceeds cum qty
8		Execution (Y)	Fill	Filled	New	8000	8000	0	6400	Execution for 6400.

1.6.5.11 Cancel/replace request sent while execution is being reported – the requested order qty equals the cum qty – order qty is amended to cum qty

Time	**Message Received** (ClOrdID, OrigClOrdID)	**Message Sent** (ClOrdID, OrigClOrdID)	**Exec Type**	**OrdStatus**	**Exec Trans Type**	**Order Qty**	**Cum Qty**	**Leaves Qty**	**Last Shares**	**Comment**
1	New Order(X)					10000				
2		*Execution(X)*	*Rejected*	*Rejected*	*New*	*10000*	*0*	*0*	*0*	*If order is rejected by broker*
2		Execution(X)	New	New	New	10000	0	10000	0	
3	**Replace Request(Y,X)**					**7000**				**Client wishes to amend order qty to 7000 shares**
3		**Execution(X)**	**Partial Fill**	**Partially Filled**	**New**	**10000**	**7000**	**3000**	**7000**	**Execution for 7000 - the replace message and this execution report pass each other on the connection**
4		Execution (Y,X)	Replace	Filled	New	7000	7000	0	0	The replace request is interpreted as requiring the balance of the order to be canceled – the 'filled' order status takes precedence over 'canceled' or 'replaced'

1.6.5.12 Cancel/replace request sent while execution is being reported – the requested order qty is below cum qty – order qty is amended to cum qty

Time	**Message Received** (ClOrdID, OrigClOrdID)	**Message Sent** (ClOrdID, OrigClOrdID)	**Exec Type**	**OrdStatus**	**Exec Trans Type**	**Order Qty**	**Cum Qty**	**Leaves Qty**	**Last Shares**	**Comment**
1	New Order(X)					10000				
2		*Execution(X)*	*Rejected*	*Rejected*	*New*	*10000*	*0*	*0*	*0*	*If order is rejected by broker*
2		Execution(X)	New	New	New	10000	0	10000	0	
3	**Replace Request(Y,X)**					**7000**				**Client wishes to amend order qty to 7000 shares**
3		**Execution(X)**	**Partial Fill**	**Partially Filled**	**New**	**10000**	**8000**	**2000**	**8000**	**Execution for 8000 - the replace message and this execution report pass each other on the connection**
4		Execution (Y,X)	Replace	Filled	New	8000	8000	0	0	The replace request is interpreted as requiring the balance of the order to be canceled – the 'filled' order status takes precedence over 'canceled' or 'replaced'

1.6.5.13 One cancel/replace request is issued which is accepted – another one is issued which is also accepted

Time	**Message Received** (ClOrdID, OrigClOrdID)	**Message Sent** (ClOrdID, OrigClOrdID)	**Exec Type**	**OrdStatus**	**Exec Trans Type**	**Order Qty**	**Cum Qty**	**Leaves Qty**	**Last Shares**	**Comment**
1	New Order(X)					10000				
2		*Execution(X)*	*Rejected*	*Rejected*	*New*	*10000*	*0*	*0*	*0*	*If order is rejected by broker*
2		Execution(X)	New	New	New	10000	0	10000	0	
3		Execution(X)	Partial Fill	Partially Filled	New	10000	1000	9000	1000	Execution for 1000
4	Replace Request(Y,X)					8000				Request decrease in order quantity to 8000, leaving 7000 open
5		Execution (Y,X)	Pending Replace	Pending Replace	New	10000	1000	9000	0	'Pending replace' order status takes precedence over 'partially filled' order status
6		Execution(X)	Partial Fill	Pending Replace	New	10000	1500	8500	500	Execution for 500
7		Execution (Y,X)	Replace	Partially Filled	New	8000	1500	6500	0	'Partially filled' order status takes precedence over 'replaced' order status
8		Execution (Y)	Partial Fill	Partially Filled	New	8000	3500	4500	2000	Execution for 2000
9	Replace Request(Z,Y)					6000				Request decrease in order quantity to 6000, leaving 2500 open
10		Execution (Z,Y)	Pending Replace	Pending Replace	New	8000	3500	4500	0	
11		Execution (Z,Y)	Replace	Partially Filled	New	6000	3500	2500	0	'Partially filled' order status takes precedence over 'replaced' order status
12		Execution(Z)	Fill	Filled	New	6000	6000	0	2500	Execution for 2500

1.6.5.14 One cancel/replace request is issued which is rejected before order becomes pending replace – then another one is issued which is accepted

Time	**Message Received** (ClOrdID, OrigClOrdID)	**Message Sent** (ClOrdID, OrigClOrdID)	**Exec Type**	**OrdStatus**	**Exec Trans Type**	**Order Qty**	**Cum Qty**	**Leaves Qty**	**Last Shares**	**Comment**
1	New Order(X)					10000				
2		*Execution(X)*	*Rejected*	*Rejected*	*New*	*10000*	*0*	*0*	*0*	*If order is rejected by broker*
2		Execution(X)	New	New	New	10000	0	10000	0	
3		Execution(X)	Partial Fill	Partially Filled	New	10000	1000	9000	1000	Execution for 1000
4	Replace Request(Y,X)					8000				Request decrease in order quantity to 8000, leaving 7000 open
5		Cancel Reject (Y,X)		Partially Filled		10000				Request is rejected
6		Execution(X)	Partial Fill	Partially Filled	New	10000	1500	8500	500	Execution for 500
7		Execution(X)	Partial Fill	Partially Filled	New	10000	3500	6500	2000	Execution for 2000
8	Replace Request(Z,X)					6000				Request decrease in order quantity to 6000, leaving 2500 open. Note that OrigClOrdID = X
9		Execution (Z,X)	Pending Replace	Pending Replace	New	10000	3500	6500	0	Note that OrigClOrdID = X
10		Execution (Z,X)	Replace	Partially Filled	New	6000	3500	2500	0	Note that OrigClOrdID = X
11		Execution(Z)	Partial Fill	Partially Filled	New	6000	5000	1000	1500	Execution for 1500

1.6.5.15 One cancel/replace request is issued which is rejected after it is in pending replace – then another one is issued which is accepted

Time	**Message Received** (ClOrdID, OrigClOrdID)	**Message Sent** (ClOrdID, OrigClOrdID)	**Exec Type**	**OrdStatus**	**Exec Trans Type**	**Order Qty**	**Cum Qty**	**Leaves Qty**	**Last Shares**	**Comment**
1	New Order(X)					10000				
2		*Execution(X)*	*Rejected*	*Rejected*	*New*	*10000*	*0*	*0*	*0*	*If order is rejected by broker*
2		Execution(X)	New	New	New	10000	0	10000	0	
3		Execution(X)	Partial Fill	Partially Filled	New	10000	1000	9000	1000	Execution for 1000
4	Replace Request(Y,X)					8000				Request decrease in order quantity to 8000, leaving 7000 open
5		Execution (Y,X)	Pending Replace	Pending Replace		10000	1000	9000	0	
6		Execution(X)	Partial Fill	Pending Replace	New	10000	1500	8500	500	Execution for 500. 'Pending replace' order status takes precedence over 'partially filled' order status
7		Cancel Reject (Y,X)		Partially Filled		10000				Request is rejected (e.g. by trader/exchange)
8		Execution(X)	Partial Fill	Partially Filled	New	10000	3500	6500	2000	Execution for 2000
9	Replace Request(Z,X)					6000				Request decrease in order quantity to 6000, leaving 2500 open. Note that OrigClOrdID = X
10		Execution (Z,X)	Pending Replace	Pending Replace	New	10000	3500	6500	0	
11		Execution (Z,X)	Replace	Partially Filled	New	6000	3500	2500	0	
12		Execution(Z)	Partial Fill	Partially Filled	New	6000	5000	1000	1500	Execution for 1500

1.6.5.16 One cancel/replace request is issued followed immediately by another – broker processes sequentially

Time	**Message Received** (ClOrdID, OrigClOrdID)	**Message Sent** (ClOrdID, OrigClOrdID)	**Exec Type**	**OrdStatus**	**Exec Trans Type**	**Order Qty**	**Cum Qty**	**Leaves Qty**	**Last Shares**	**Comment**
1	New Order(X)					10000				
2		Execution(X)	New	New	New	10000	0	10000	0	
3		Execution(X)	Partial Fill	Partially Filled	New	10000	1000	9000	1000	Execution for 1000
4	Replace Request(Y,X)					8000				Request decrease in order quantity to 8000, leaving 7000 open
5	Replace Request(Z,Y)					7000				Request decrease in order quantity to 7000, leaving 6000 open
6		Execution (Y,X)	Pending Replace	Pending Replace	New	10000	1000	9000	0	Broker processes Replace (Y,X) first
7		Execution (Y,X)	Replace	Partially Filled	New	8000	1000	7000	0	Broker processes Replace (Y,X) first
8		Execution (Z,Y)	Pending Replace	Pending Replace	New	8000	1000	7000	0	Broker then processes Replace (Z,Y)
9		Execution (Z,Y)	Replace	Partially Filled	New	7000	1000	6000	0	Broker then processes Replace (Z,Y)
10		Execution(Z)	Fill	Filled	New	7000	7000	0	6000	Execution for 6000

1.6.5.17 One cancel/replace request is issued followed immediately by another – broker rejects the second as order is pending replace

Time	**Message Received** (ClOrdID, OrigClOrdID)	**Message Sent** (ClOrdID, OrigClOrdID)	**Exec Type**	**OrdStatus**	**Exec Trans Type**	**Order Qty**	**Cum Qty**	**Leaves Qty**	**Last Shares**	**Comment**
1	New Order(X)					10000				
2		Execution(X)	New	New	New	10000	0	10000	0	
3		Execution(X)	Partial Fill	Partially Filled	New	10000	1000	9000	1000	Execution for 1000
4	Replace Request(Y,X)					8000				Request decrease in order quantity to 8000, leaving 7000 open
5	Replace Request(Z,Y)					7000				Request decrease in order quantity to 7000, leaving 6000 open
6		Execution (Y,X)	Pending Replace	Pending Replace	New	10000	1000	9000	0	
7		Cancel Reject (Z,Y)		Pending Replace		10000				Rejected because broker does not support processing of order cancel replace request whilst order is pending cancel. CxlRejReason = 'Order already in pending cancel or pending replace status'
8		Execution (Y,X)	Replace	Partially Filled	New	8000	1000	7000	0	'Partially filled' order status takes precedence over 'replaced' order status
9		Execution (Y)	Partial Fill	Partially Filled	New	8000	3000	5000	2000	Execution for 2000

This matrix illustrates the case where the broker does not support multiple outstanding order cancel or order cancel/replace requests.

1.6.5.18 Telephoned order

Time	**Message Received** (ClOrdID, OrigClOrdID)	**Message Sent** (ClOrdID, OrigClOrdID)	**Exec Type**	**OrdStatus**	**Exec Trans Type**	**Order Qty**	**Cum Qty**	**Leaves Qty**	**Last Shares**	**Comment**
1										Order for 10000 shares phoned to broker
2		Execution	New	New	New	10000	0	0	0	Confirm that the broker has accepted the order – note that broker does not need to capture a ClOrdID
3		Execution	Partial Fill	Partially Filled	New	10000	2000	8000	2000	Execution of 2000
4		Execution	Partial Fill	Partially Filled	New	10000	3000	7000	1000	Execution of 1000
5		Execution	Fill	Filled	New	10000	10000	0	7000	Execution of 7000

1.6.5.19 Unsolicited cancel of a part-filled order

Time	**Message Received** (ClOrdID, OrigClOrdID)	**Message Sent** (ClOrdID, OrigClOrdID)	**Exec Type**	**OrdStatus**	**Exec Trans Type**	**Order Qty**	**Cum Qty**	**Leaves Qty**	**Last Shares**	**Comment**
1	New Order(X)					10000				
2		*Execution(X)*	*Rejected*	*Rejected*	*New*	*10000*	*0*	*0*	*0*	*If order is rejected by broker*
2		Execution(X)	New	New	New	10000	0	10000	0	
3		Execution(X)	Partial Fill	Partially Filled	New	10000	1000	9000	1000	Execution for 1000
4										Broker verbally agrees to cancel order
5		Execution(X)	Canceled	Canceled	New	10000	1000	0	0	Broker signifies that order has been canceled - ExecRestatementReason = Verbal change

This scenario might occur if the buy-side has not implemented order cancel requests or alternatively there is an electronic communication problem at the point that the buy-side wishes to send a cancel request.

1.6.5.20 Unsolicited replacement of a part-filled order

Time	**Message Received** (ClOrdID, OrigClOrdID)	**Message Sent** (ClOrdID, OrigClOrdID)	**Exec Type**	**OrdStatus**	**Exec Trans Type**	**Order Qty**	**Cum Qty**	**Leaves Qty**	**Last Shares**	**Comment**
1	New Order(X)					10000				
2		*Execution(X)*	*Rejected*	*Rejected*	*New*	*10000*	*0*	*0*	*0*	*If order is rejected by broker*
2		Execution(X)	New	New	New	10000	0	10000	0	
3										Broker verbally agrees to increase order quantity to 11000
4		Execution(X)	Restated	New	New	11000	0	0	0	Broker signifies that order has been replaced ExecRestatementReason = Verbal
5		Execution(X)	Partial Fill	Partially Filled	New	11000	1000	10000	1000	Execution for 1000
6										Broker verbally agrees to increase order quantity to 12000
7		Execution(X)	Restated	Partially Filled	New	12000	1000	11000	0	Broker signifies that order has been replaced ExecRestatementReason = Verbal change

This scenario might occur if the buy-side has not implemented order cancel/replace requests or alternatively there is an electronic communication problem at the point that the buy-side wishes to send a cancel replace request.

1.6.5.21 Unsolicited reduction of order quantity by sell side (e.g. for US ECNs to communicate Nasdaq SelectNet declines)

Time	**Message Received** (ClOrdID, OrigClOrdID)	**Message Sent** (ClOrdID, OrigClOrdID)	**Exec Type**	**OrdStatus**	**Exec Trans Type**	**Order Qty**	**Cum Qty**	**Leaves Qty**	**Last Shares**	**Comment**
1	New Order(X)					10000				
2		*Execution(X)*	*Rejected*	*Rejected*	*New*	*10000*	*0*	*0*	*0*	*If order is rejected by broker*
2		Execution(X)	New	New	New	10000	0	10000	0	
3		Execution(X)	Restated	New	New	9000	0	9000	0	ExecRestatementReason="Partial Decline of OrderQty"
4		Execution(X)	Fill	Filled	New	9000	9000	0	9000	

1.6.5.22 Order rejected due to duplicate ClOrdID

Time	**Message Received** (ClOrdID, OrigClOrdID)	**Message Sent** (ClOrdID, OrigClOrdID)	**Exec Type**	**OrdStatus**	**Exec Trans Type**	**Order Qty**	**Cum Qty**	**Leaves Qty**	**Last Shares**	**Comment**
1	New Order(X)					10000				
2		Execution(X)	New	New	New	10000	0	10000	0	
3		Execution(X)	Partial Fill	Partially Filled	New	10000	1000	9000	1000	Execution for 1000
4	New Order(X)					10000				Order submitted with the same order id
5		Execution(X)	Rejected	Partially Filled	New	10000	1000	9000	0	OrdRejReason = duplicate order

1.6.5.23 Order rejected because the order has already been verbally submitted

Time	**Message Received** (ClOrdID, OrigClOrdID)	**Message Sent** (ClOrdID, OrigClOrdID)	**Exec Type**	**OrdStatus**	**Exec Trans Type**	**Order Qty**	**Cum Qty**	**Leaves Qty**	**Last Shares**	**Comment**
1	New Order(X)					10000				Order for 10000 sent electronically
2										Order passed verbally as there is communication problem and order does not arrive. The verbally passed order starts getting executed
3		Execution(X)	Rejected	Rejected	New	10000	0	0	0	Order finally arrives and is detected as a duplicate of a verbal order and is therefore rejected. OrdRejReason = duplicate of a verbal order

Note that the sell-side may employ a number of mechanisms to detect that the electronic order is potentially a duplicate of a verbally passed order, e.g.:
Checking the PossDup flag on the order message header
Checking the incoming order details against other orders from the same client (e.g. side, quantity)
Looking at the transact time on the order as a guide to 'staleness'

1.6.5.24 Order status request rejected for unknown order

Time	**Message Received** (ClOrdID, OrigClOrdID)	**Message Sent** (ClOrdID, OrigClOrdID)	**Exec Type**	**OrdStatus**	**Exec Trans Type**	**Order Qty**	**Cum Qty**	**Leaves Qty**	**Last Shares**	**Comment**
1	New Order(X)					10000				
2		Execution(X)	New	New	New	10000	0	10000	0	
3		Execution(X)	Partial Fill	Partially Filled	New	10000	1000	9000	1000	Execution for 1000
4	Status Request (Y)									
5		Execution(Y)	Rejected	Rejected	Status	0	0	0		OrdRejReason = unknown order LastShares not required when ExecTransType=Status

1.6.5.25 Transmitting a CMS-style "Nothing Done" in response to a status request

Time	**Message Received** (ClOrdID, OrigClOrdID)	**Message Sent** (ClOrdID, OrigClOrdID)	**Exec Type**	**OrdStatus**	**Exec Trans Type**	**Order Qty**	**Cum Qty**	**Leaves Qty**	**Last Shares**	**Comment**
1	New Order(X)					10000				
2		*Execution(X)*	*Rejected*	*Rejected*	*New*	*10000*	*0*	*0*	*0*	*If order is rejected by broker*
2		Execution(X)	New	New	New	10000	0	10000	0	
3	Status Request(X)									
4		Execution(X)	New	New	Status	10000	0	10000	0	Text="Nothing Done"

1.6.5.26 Order sent, immediately followed by a status request. Subsequent status requests sent during life of order

Time	**Message Received** (ClOrdID, OrigClOrdID)	**Message Sent** (ClOrdID, OrigClOrdID)	**Exec Type**	**OrdStatus**	**Exec Trans Type**	**Order Qty**	**Cum Qty**	**Leaves Qty**	**Last Shares**	**Comment**
1	New Order(X)					10000				
2	Status Request (X)									
3		Execution(X)	Pending New	Pending New	Status	10000	0	10000		Sent in response to status request. LastShares not required when ExecTransType=status
4		*Execution(X)*	*Rejected*	*Rejected*	*New*	*10000*	*0*	*0*	*0*	*If order is rejected*
4		Execution(X)	New	New	New	10000	0	10000	0	
5	Status Request (X)									
6		Execution(X)	New	New	Status	10000	0	10000		Sent in response to status request
7		Execution(X)	Partial Fill	Partially Filled	New	10000	2000	8000	2000	Execution for 2000
8	Status Request (X)									
9		Execution(X)	Partial Fill	Partially Filled	Status	10000	2000	8000		Sent in response to status request
10		Execution(X)	Fill	Filled	New	10000	10000	0	8000	Execution for 8000
11	Status Request (X)									
12		Execution(X)	Fill	Filled	Status	10000	10000	0		Sent in response to status request
13	Replace Request(Y,X)					12000				Request to increase order qty
14		Execution (Y,X)	Pending Replace	Pending Replace	New	10000	10000	0	0	
15		Execution (Y,X)	Replace	Partially Filled	New	12000	10000	2000	0	
16	Status Request (X)									
17		Execution (Y,X)	Partial Fill	Partially Filled	Status	12000	10000	2000		Sent in response to status request. Note reference to X to allow tie back of execution report to status request
18	Status Request (Y)									
19		Execution(Y)	Partial Fill	Partially Filled	Status	12000	10000	2000		Sent in response to status request

1.6.5.27 GTC order partially filled, restated (renewed) and partially filled the following day

Time	**Message Received** (ClOrdID, OrigClOrdID)	**Message Sent** (ClOrdID, OrigClOrdID)	**Exec Type**	**Ord Status**	**Exec Trans Type**	**Order Qty**	**Cum Qty**	**Leaves Qty**	**Last Shares**	**Day Order Qty**	**Day Cum Qty**	**Comment**
Day 1,1	New Order(X)					10000						
Day 1,2		Execution(X)	New	New	New	10000	0	10000	0			
Day 1,3		Execution(X)	Partial Fill	Partially Filled	New	10000	2000	8000	2000			Execution for 2000
Day 1,4		Execution(X)	Done for Day	Done for Day	New	10000	2000	8000	0			Optional at end of trading day
Day 2,1		Execution(X)	Restated	Partially Filled	New	10000	2000	8000	0	8000	0	ExecRestatementReason = GTC renewal/restatement (no change) – optionally sent the following morning
Day 2,2		Execution(X)	Partial Fill	Partially Filled	New	10000	3000	7000	1000	8000	1000	Execution for 1000

1.6.5.28 GTC order with partial fill, a 2:1 stock split then a partial fill and fill the following day

Time	**Message Received** (ClOrdID, OrigClOrdID)	**Message Sent** (ClOrdID, OrigClOrdID)	**Exec Type**	**Ord Status**	**Exec Trans Type**	**Order Qty**	**Cum Qty**	**Leaves Qty**	**Last Shares**	**Day Order Qty**	**Day Cum Qty**	**Comment**
Day 1,1	New Order(X)					10000						
Day 1,2		Execution(X)	New	New	New	10000	0	10000	0			
Day 1,3		Execution(X)	Partial Fill	Partially Filled	New	10000	2000	8000	2000			Execution for 2000 @ 50
Day 1,4		Execution(X)	Done for Day	Done for Day	New	10000	2000	8000	0			Optional at end of trading day
Day 2,1		Execution(X)	Restated	Partially Filled	New	20000	4000	16000	0	16000	0	Sent the following morning after the split ExecRestatementReason = GTC corporate action. AvgPx=25, DayAvgPx=0
Day 2,2		Execution(X)	Partial Fill	Partially Filled	New	20000	9000	11000	5000	16000	5000	Execution for 5000
Day 2,3		Execution(X)	Fill	Filled	New	20000	20000	0	11000	16000	16000	Execution for 11000

1.6.5.29 GTC order partially filled, restated(renewed) and canceled the following day

Time	**Message Received** (ClOrdID, OrigClOrdID)	**Message Sent** (ClOrdID, OrigClOrdID)	**Exec Type**	**Ord Status**	**Exec Trans Type**	**Order Qty**	**Cum Qty**	**Leaves Qty**	**Last Shares**	**Day Order Qty**	**Day Cum Qty**	**Comment**
Day 1,1	New Order(X)					10000						
Day 1,2		Execution(X)	New	New	New	10000	0	10000	0			
Day 1,3		Execution(X)	Partial Fill	Partially Filled	New	10000	2000	8000	2000			Execution for 2000
Day 1,4		Execution(X)	Done for Day	Done for Day	New	10000	2000	8000	0			Optional at end of trading day
Day 2,1		Execution(X)	Restated	Partially Filled	New	10000	2000	8000	0	8000	0	ExecRestatementReason = GTC renewal/restatement (no change) – optionally sent the following morning
Day 2,2	Cancel Request (Y,X)					10000						
Day 2,3		*Cancel Reject (Y,X)*		*Partially Filled*		*10000*						*If rejected by salesperson*
Day 2,3		Execution (Y,X)	Pending Cancel	Pending Cancel		10000	2000	8000	0	8000	0	
Day 2,4		Cancel Reject (Y,X)		Partially Filled		10000						If rejected by trader/exchange
Day 2,4		Execution (Y,X)	Canceled	Canceled		10000	2000	0	0	8000	0	

1.6.5.30 GTC order partially filled, restated(renewed) followed by replace request to increase quantity

Time	**Message Received** (ClOrdID, OrigClOrdID)	**Message Sent** (ClOrdID, OrigClOrdID)	**Exec Type**	**Ord Status**	**Exec Trans Type**	**Order Qty**	**Cum Qty**	**Leaves Qty**	**Last Shares**	**Day Order Qty**	**Day Cum Qty**	**Comment**
Day 1,1	New Order(X)					10000						
Day 1,2		Execution(X)	New	New	New	10000	0	10000	0			
Day 1,3		Execution(X)	Partial Fill	Partially Filled	New	10000	2000	8000	2000			Execution for 2000
Day 1,4		Execution(X)	Done for Day	Done for Day	New	10000	2000	8000	0			Optional at end of trading day
Day 2,1		Execution(X)	Restated	Partially Filled	New	10000	2000	8000	0	8000	0	ExecRestatementReason = GTC renewal/restatement (no change) – optionally sent the following morning
Day 2,2	Replace Request(Y,X)					15000						Increasing qty
Day 2,3		*Cancel Reject (Y,X)*		*Partially Filled*		*10000*						*If rejected by salesperson*
Day 2,3		Execution (Y,X)	Pending Replace	Pending Replace		10000	2000	8000	0	8000	0	
Day 2,4		Execution (X)	Partial Fill	Pending Replace		10000	3000	7000	1000	8000	1000	Execution for 1000
Day 2,5		Cancel Reject (Y,X)		Partially Filled		10000						If rejected by trader/exchange
Day 2,5		Execution (Y,X)	Replace	Partially Filled		15000	3000	12000	0	13000	1000	

1.6.5.31 Poss resend order

Time	**Message Received** (ClOrdID, OrigClOrdID)	**Message Sent** (ClOrdID, OrigClOrdID)	**Exec Type**	**OrdStatus**	**Exec Trans Type**	**Order Qty**	**Cum Qty**	**Leaves Qty**	**Last Shares**	**Comment**
1	New Order(X)					10000				
2		Execution(X)	New	New	New	10000	0	10000	0	
3	New Order(X)					10000				PossResend=Y
4		Execution(X)	New	New	Status	10000	0	10000		Because order X has already been received, confirm back the current state of the order. Last shares not required when ExecTransType = Status
5	New Order(Y)					15000				PossResend=Y
6		Execution(Y)	New	New	New	10000	0	10000	0	Because order Y has not been received before, confirm back as a new order.

1.6.5.32 Fill or Kill order cannot be filled

Time	**Message Received** (ClOrdID, OrigClOrdID)	**Message Sent** (ClOrdID, OrigClOrdID)	**Exec Type**	**OrdStatus**	**Exec Trans Type**	**Order Qty**	**Cum Qty**	**Leaves Qty**	**Last Shares**	**Comment**
1	New Order(X)					10000				Order is FOK
2		*Execution(X)*	*Rejected*	*Rejected*	*New*	*10000*	*0*	*0*	*0*	*If order is rejected by broker*
2		Execution(X)	New	New	New	10000	0	10000	0	
3		Execution(X)	Canceled	Canceled	New	10000	0	0	0	If order cannot be immediately filled

1.6.5.33 Immediate or Cancel order that cannot be immediately hit

Time	**Message Received** (ClOrdID, OrigClOrdID)	**Message Sent** (ClOrdID, OrigClOrdID)	**Exec Type**	**OrdStatus**	**Exec Trans Type**	**Order Qty**	**Cum Qty**	**Leaves Qty**	**Last Shares**	**Comment**
1	New Order(X)					10000				Order is IOC
2		*Execution(X)*	*Rejected*	*Rejected*	*New*	*10000*	*0*	*0*	*0*	*If order is rejected by broker*
2		Execution(X)	New	New	New	10000	0	10000	0	
3		Execution(X)	Partial Fill	Partially Filled	New	10000	1000	9000	1000	Execution for 1000
4		Execution(X)	Canceled	Canceled	New	10000	1000	0	0	If order cannot be immediately hit

1.6.5.34 Filled order, followed by correction and cancellation of executions

Time	**Message Received** (ClOrdID, OrigClOrdID)	**Message Sent** (ClOrdID, OrigClOrdID)	**Exec Type**	**OrdStatus**	**Exec Trans Type**	**Order Qty**	**Cum Qty**	**Leaves Qty**	**AvgPx**	**Last Shares**	**Last Px**	**ExecID (ExecRefID)**	**Comment**
1	New Order(X)					10000							
2		*Execution(X)*	*Rejected*	*Rejected*	*New*	*10000*	*0*	*0*		*0*		*A*	*If order is rejected by broker*
2		Execution(X)	New	New	New	10000	0	10000	0	0		B	
3		Execution(X)	Partial Fill	Partially Filled	New	10000	1000	9000	100	1000	100	C	Execution for 1000 @ 100
4		Execution(X)	Fill	Filled	New	10000	10000	0	109	9000	110	D	Execution for 9000 @ 110
5		Execution(X)	Partial Fill	Partially Filled	Cancel	10000	9000	1000	110	0	0	E (C)	Cancel execution for 1000
6		Execution(X)	Partial Fill	Partially Filled	Correct	10000	9000	1000	100	9000	100	F (D)	Correct price on execution for 9000 to 100
7		Execution(X)	Fill	Filled	New	10000	10000	0	102	1000	120	G	Execution for 1000 @ 120
8		Execution(X)	Fill	Filled	Correct	10000	10000	0	120	9000	120	H(F)	Correct price on execution for 9000 to 120
9	Replace Request (Y,X)					12000							Request to increase order qty
10		Execution (Y,X)	Pending Replace	Pending Replace	New	10000	10000	0	120	0	0	I	
11		Execution (Y,X)	Replace	Partially Filled	New	12000	10000	2000	120	0	0	J	
12		Execution(Y)	Partial Fill	Partially Filled	Correct	12000	10500	1500	120	9500	120	K(H)	Correct execution of 9000 @ 120 to 9500 @ 120

1.6.5.35 A canceled order followed by a busted execution and a new execution

Time	**Message Received** (ClOrdID, OrigClOrdID)	**Message Sent** (ClOrdID, OrigClOrdID)	**Exec Type**	**Ord Status**	**Exec Trans Type**	**Order Qty**	**Cum Qty**	**Leaves Qty**	**Last Shares**	**ExecID (ExecRefID**	**Comment**
1	New Order(X)					10000					
2		Execution(X)	New	New	New	10000	0	10000	0	A	
3		Execution(X)	Partial Fill	Partially Filled	New	10000	5000	5000	5000	B	LastPx=50
4	Cancel Request(Y,X)					10000					
5		Execution (Y,X)	Pending Cancel	Pending Cancel	New	10000	5000	5000	0	C	
6		Execution (Y,X)	Canceled	Canceled	New	10000	5000	0	0	D	
7		Execution(Y)	Partial Fill	Canceled	Cancel	10000	0	0	0	E(B)	Cancel of the execution. 'Canceled' order status takes precedence over 'New'
8		Execution(Y)	Partial Fill	Canceled	New	10000	4000	0	4000	F	Fill for 4000. LastPx=51

1.6.5.36 GTC order partially filled, restated (renewed) and partially filled the following day, with corrections of quantity on both executions

Time	**Message Received** (ClOrdID, OrigClOrdID)	**Message Sent** (ClOrdID, OrigClOrdID)	**Exec Type**	**Ord Status**	**Exec Trans Type**	**Order Qty**	**Cum Qty**	**Leaves Qty**	**Last Shares**	**Day Order Qty**	**Day Cum Qty**	**ExecID (ExecRefID)**	**Comment**
Day 1,1	New Order(X)					10000							
Day 1,2		Execution(X)	New	New	New	10000	0	10000	0			A	
Day 1,3		Execution(X)	Partial Fill	Partially Filled	New	10000	2000	8000	2000			B	Execution for 2000
Day 1,4		Execution(X)	Done for Day	Done for Day	New	10000	2000	8000	0			C	Optional at end of trading day
Day 2,1		Execution(X)	Restated	Partially Filled	New	10000	2000	8000	0	8000	0	D	ExecRestatementReason = GTC renewal/restatement (no change) – optionally sent the following morning
Day 2,2		Execution(X)	Partial Fill	Partially Filled	New	10000	3000	7000	1000	8000	1000	E	Execution for 1000
Day 2,3		Execution(X)	Partial Fill	Partially Filled	Correct	10000	2500	7500	1500	8500	1000	F (B)	Correct quantity on previous day's execution from 2000 to 1500
Day 2,4		Execution(X)	Partial Fill	Partially Filled	Correct	10000	2000	8000	500	8500	500	G (E)	Correct quantity on today's execution from 1000 to 500

1.6.5.37 Transmitting a guarantee of execution prior to execution

Time	**Message Received** (ClOrdID, OrigClOrdID)	**Message Sent** (ClOrdID, OrigClOrdID)	**Exec Type**	**OrdStatus**	**Exec Trans Type**	**Order Qty**	**Cum Qty**	**Leaves Qty**	**Last Shares**	**Comment**
1	New Order(X)					10000				
2		*Execution(X)*	*Rejected*	*Rejected*	*New*	*10000*	*0*	*0*	*0*	*If order is rejected by broker*
2		Execution(X)	New	New	New	10000	0	10000	0	
3		Execution(X)	Stopped	Stopped	New	10000	0	10000	1000	Text="You are guaranteed to buy 1000 at 50.10"; LastPx=50.10. This is similar to the concept of a 'protected' trade
4		Execution(X)	Partial Fill	Partially Filled	New	10000	1000	9000	1000	LastPx=50 * executed price is better than guaranteed

2.0 Coppelia

2.0.1 Introduction

Javelin's Coppelia server is a highly reliable FIX messaging solution that easily integrates into almost any existing system environment. Coppelia FIX server technology is available with many modules to provide the optimal solution for clients' specific business needs. These modules and how to configure Coppelia accordingly are described later in this document.

Coppelia is a multi-threaded financial transaction engine written 100 percent in Java. Coppelia is designed to fully comply with the FIX standards across all versions with a versatile architecture that allows Javelin developers to add new protocols to leverage the existing infrastructures of persistence, high availability and numerous interfaces. The engine is designed using a blocking queue mechanism that allows messages to flow from different subsystems in an extremely flexible, robust and efficient manner. Coppelia is a multipurpose adapter which can bridge CORBA, RMI, Observer Observable, Com/DCOM, ActiveX, Talarian and TIBCO Rendez-vous to a financial protocol like FIX, ACT and CMS. Persistence has always been an integral part of the original design. It captures all connection and state information to ensure increased data integrity. One of the major strengths of Coppelia is its platform and interface flexibility that allows us to maintain a single source code version for all of our customers. The result is a development effort that is focused on a single product and not spread across multiple versions resulting in a better-tested and maintained engine.

Coppelia is bundled with a support toolkit that includes a Broker Simulator, FIXometer Network Monitor, FIXionary and a compendium of Test Scripts.

2.0.2 Features and Benefits

FIX Version Support

Coppelia supports all the various FIX versions, 2.7, 3.0, 4.0, and 4.1. Coppelia allows users to communicate simultaneously in all FIX versions and across the entire FIX message suite (except for list-related messages at the point of this writing) in these versions. Future enhancements and additions to the protocol will be incorporated as they are released. Users can communicate to trading counter parties regardless of which version of FIX the other party is running.

Java and Portability

Coppelia is written 100% in Java and is dependent on the proper functioning of the Java Virtual Machine. Using the Java Virtual Machine, Javelin adapts Coppelia to run on Windows NT and many forms of UNIX, including Solaris, AIX, and HP/UX.

Network Options

Currently, Coppelia only supports TCP/IP. Currently, there are no plans to support any other network protocol.

Coppelia API Options

The Coppelia server can be integrated with clients' systems through CORBA, ActiveX, IBM MQSeries, TIB/Rendezvous, Talarian Smart Sockets, Observer / Observable, and Java RMI.

Single-process / Multi-threaded

Rather than running a separate session for each trading counter-party, Coppelia utilizes a single process, multi-threaded design that allows for scalability as well as reduced operational overhead.

Database

All versions of Coppelia come with a database from Object Design for data persistence and replication capabilities. No database administration is required, as only same-day messages are stored. Coppelia's design flexibility allows integration with other databases via JDBC per client's environment.

2.0.3 Coppelia Modules

Single Connect - "One FIX Connection"

- 3,000 - 4,000 trades
- 10K - 15K messages

Standard (ST) - "Multiple Connection, Limited Throughput"

Enhanced Performance (EP) - "Multiple Connection, High Throughput"

- 1MM+ messages (Allows for maximum speed and throughput)
- High performance database with replication

High Availability (Beta)

- Provides a robust high availability fault tolerant environment utilizing a primary/backup configuration ensuring no down time
- Constantly replicates orders into database or in memory

Coppelia Web - "Cherubino"

- Web delivery option facilitates counterparty communication
- FIX is integrated "inside" the solution, offering rapid deployment

FIX to CMS - "Lolita"

- Provides CMS connectivity and FIX to CMS conversions

FIX Box (still in development)

- Javelin's FIX hardware solution offering cost-effective, quick and easy integration

ACT Reporter

- Provides interface to NASD NWII server to satisfy the 90-second rule

2.1 Connections

2.1.1 Network Connectivity

Every FIX session requires at least one open TCP/IP session. It is imperative that network connectivity be constantly monitored to make sure all connections can be made without problems. That is especially important for firewall and router configurations.

In order to ensure proper network connectivity, a ping to the address in question will not suffice. Instead, a telnet into the counter party's IP and port will tell you whether the FIX engine would be able to connect to the process on the other side.

2.1.1.1 Test network connectivity:

```
telnet [hostname / host_IP] [port_number]
```

If network connectivity exists, and an application is listening on the port number specified, you will be offered an escape sequence, otherwise, there will be a note saying "Trying..." only.

2.2 Session Layer

The session level logic, such as resends, detection of gaps in the series of sequence numbers, etc., is automatic.

Please note that there are two sets of sequence numbers for each party to a FIX connection, incoming and outgoing sequence numbers. Each of these is being watched and kept track of.

2.2.1 Sequence Number Differences

As mentioned before, there are two sets of sequence numbers to every FIX session, an incoming sequence number, and an outgoing one. The incoming sequence number count pertains to messages received, and the outgoing count to the ones sent. Obviously, the other party to the FIX connection will keep the same tally, vice versa.

In FIX version 4.0 and higher, every message in the protocol increments the message sequence number, including all administrative messages. Therefore, every message should have it's unique sequence number, and if it does not, it should have the PossDup flag set (i.e., it should contain "43=Y". This does not mean that the message is indeed a duplicate, but it shows that the sending application thinks it is from its point of view. The receiving application is responsible to check for duplicates all the time, regardless of whether this flag is set or not.

There might be situations (network problems, abnormal disconnects) where, in order to re-establish a FIX connection, sequence numbers have to be reset to a certain value manually. Please note that resetting sequence numbers here means ***resetting to the next***

expected outgoing or incoming sequence number (see also below). It is recommended you be disconnected when you perform this operation.

2.2.1.1 Reset the incoming sequence number for a connection:

From Coppelia's command prompt, type

`msg_seq_num_in [ID] [Seq_Num]`

This will reset the counter in Coppelia to expect [Seq_Num] as the next incoming message sequence number.

2.2.1.2 Reset the outgoing sequence number for a connection:

From Coppelia's command prompt, type

`msg_seq_num_out [ID] [Seq_Num]`

This will reset the counter in Coppelia to send [Seq_Num] as the next message sequence number to the counter party.

2.2.2 Events and Reactions

Regular Disconnect (Logout)

This scenario might happen when the counter party's day is over, their application is not able to process any more orders, etc. In this case, the counter party intentionally terminates the connection by means of a logout message. If that happens during regular business hours (i.e., you do not expect it), wait a minute or two to give the other party time to reconnect. If Coppelia initiates the connection, it will try to re-logon automatically within a time interval specified in the configuration file.

Once you have waited a reasonable amount of time for a reconnect from the counter party, and there is no re-connection, it is recommended to call the counter party and ask for a reason, and a time frame when a reconnection can be expected. If Coppelia attempts to auto-connect to the other party, also wait a few times, and if there's no success, ask the counter party for a reason and time frame.

Note that a logout and re-logon is no catastrophe, but can have many application related reasons. FIX's provisions to recover from out-of-sync scenarios will automatically 'kick in' on re-logon.

2.2.2.1 Manually connect a specific session:

From Coppelia's command prompt, type

`connect [ID]`

This will connect the party specified as ID.

2.2.2.2 Manually connect ALL sessions:

From Coppelia's command prompt, type

`connect ALL`

Coppelia will attempt to (re-)establish all connections.

2.2.2.3 Manually disconnect a specific session:

From Coppelia's command prompt, type

`disconnect [ID]`

This will disconnect the party specified as ID.

2.2.2.4 Manually disconnect ALL sessions:

From Coppelia's command prompt, type

`disconnect ALL`

Coppelia will disconnect (logout) connections.

2.2.3 Abnormal Disconnect

In the case of an "abnormal" disconnect (no logout message has been received), you should test your network connection as described above. Contact the counter party to verify failure or crash at their end. Find out a time frame as to when normal communications are expected to resume. When re-establishing the connection, follow the procedures outlined under normal disconnection.

2.2.4 Encryption – RESTRICTED ACCESS

This section is not yet publicly available. Please call Javelin Technologies' support to obtain information about encryption and related topics.

2.3 Configuration

Coppelia relies on a configuration file in order to function. This file has to exist at startup time.

The configuration file should not need to be changed other than for purposes of adding or modifying an existing connection.

You should make sure that all TargetCompID, SenderCompID, (DeliverTo...IDs and OnBehalfOf...IDs if any) are agreed upon, and that the other party is notified of changes. This also applies to IP addresses, ports, FIX versions, and SubIDs.

Lastly, it is recommended that you suggest to your counter parties to be notified in advance of any changes to their FIX engine, IP addresses, ports, protocol version, business logic, etc. If there is enough advance notice, you will have a chance to test these changes before going into production. You should ALWAYS test when connecting to a new FIX engine altogether.

2.3.1 Configuration File Format

2.3.1.1 Blocks (References)

Important note: The following section refers to the new configuration file format that will be released shortly.

The configuration file format is line based i.e. the file is read in one line at a time. All characters after the "#" symbol are ignored no matter where on the line they exist. Blocks are defined as [BLOCK_NAME] there can be no white space inside the brackets and non-white space characters before the brackets are considered to be attribute names. All attributes following a block name are associated with that block until another block name appears. The same block name can be used again but it refers to the same block. Block names will always be converted to uppercase when used as a reference or a block name.

The "MAIN" block is where all global attributes are stored. To allow multiple servers to read from the same configuration file the concept of a server was introduced. SysConfig can load a config file in two ways:

Single Reference: This simply takes the name of the file or URL and loads the config file. It's called a single reference since only one server can access it at any given time.

Multi Reference: This takes the file/URL parameter and the Server Block. It then reads in the config file and any attributes defined in the server block are overlaid into the "MAIN" block. By doing this the same config file can be referenced by multiple servers.

2.3.1.2 Attributes

As mentioned above attributes are associated with blocks, any attributes that occur in a config file before the first block is ignored and will produce an error message. All attribute names are converted to lowercase. Attributes can have spaces but it's not recommended. All attributes must have an equals symbol on the line otherwise an error is thrown.

Attributes can reference blocks e.g. my_args = [MAIN_BLOCK] these attributes then internally map to the hash ref of that block

If an attribute name appears more than once in a block the values are then mapped to a vector. All values of attributes are automatically striped of leading and trailing white spaces.

2.3.2 Examples of current configuration files

Section TBA.

2.3.3 Coppelia dat. File Configuration

Each Coppelia engine is configured by using a configuration file, or otherwise known as a ".dat file". This file, which always ends in the ".dat" extension, contains all the information about a particular Coppelia engine, including what kind it is (a client or a server), what type (Buy or Sell), what interface it will use, the target engines it will connect to, and many many other details.

Configuration of a Coppelia .dat file is essential to proper Coppelia operation. Misconfigured .dat files can cause errors of many kinds, including connection failures to Coppelia failing to start. Most Coppelia problems that are investigated by the Javelin Support department turn out to be problems in dat file configuration.

There are some sample dat files in the Coppelia package. In the Coppelia/buy directory there is a buy.dat file (for the buy side) and in the Coppelia/sell directory a sell.dat (for the sell side) These sample dat files will be examined in further detail.

Commented out lines (lines that will be ignored by the Coppelia server) are lines with the '#' in the first position. All text after the '#' will be ignored, and these lines are used as reference.

Parameters in **bold type** are mandatory required configuration options! If these fields are not configured, the Coppelia will not be able to run.

2.3.3.1 Standard Coppelia Configuration Options

Note that the following list is comprised of elements that are independent of platform, interface and database. This list changes from time to time, and is not yet complete.

Parameter	*Description*	*Possible Values*
ADMIN_STRING	Instructs Coppelia to place this string into the FIX TargetSubID field in each administrative message (Heartbeat, TestRequest, Resend, etc.)	Any string, usually ADMIN
AUTOCONNECT	Automatically tries to connect any IDs not connected at all times, whether a connection has gone down, or a Coppelia has just been brought up.	Number of seconds to wait in between tries, default is 15
CHECK_REMOTE	Instructs Coppelia to check if the target is still connected before sending. If this is set to ON and the remote is not connected, Coppelia will return a REMOTEDOWN code and –important- the message that would have been send will not be in the outbound queue – to resend the message the user will have to recreate it from scratch and send it a second time.	*ON* *OFF – default*
CONNECT	**Whether the server should initiate connections or should accept (listen) for connections. A Coppelia engine that is set as a CLIENT will not accept any conntions. The only way for a Coppelia to accept connections is for it to be set as a SERVER A SERVER connection also allows for making outbound connections.**	**SERVER – listens for calls, and initiates connections** **CLIENT – initiates calls only**
CONSOLE	To run Coppelia properly in the background, the parameter CONSOLE must be turned OFF.	ON – default OFF
CORBA_DEBUG	This will provide debugging of the CORBA libraries on a socket level to stderr.	ON OFF – default
DESCRIPTION	This is kept internally in the server. This can be any description for the Coppelia Engine.	Any string

DISCONNECT_SCRIPT	A script, executable, or batch file that is run when a connection becomes disconnected.	Full path and name of the script, including a trailing (back)slash. i.e. D:\coppelia\connect.bat
DNS	Instructs Coppelia to use DNS for connecting to Coppelia via CORBA. Connections do not have to be specified as IP addresses. For example, this: 192.168.129.25 can be specified differently by using JAVELIN1	*ON* *OFF* – default
FILEPATH	The path where you want Coppelia to write all log files and database files to.	Full (absolute), including a trailing (back)slash.
GUI	With the Coppelia software comes a very simple GUI that can be used for demonstrations of the capabilities of the Coppelia engine. Using the GUI is good for development and debugging. However, for running the server in production, the GUI should not be used, as it will have a negative impact on performance and is more difficult to control. The functionality of the GUI is limited as well, as it only is able to send simple orders and execution reports. Note that there is available a GUI-network monitor, the FIXometer, which allows you to monitor the process remotely and adds quite a bit of functionality.	ON OFF
ID	**This parameter is discussed in more extensive detail in the section directly following this one.**	
IIOP_IP	**The IP address of this server. This parameter is required for proper Coppelia functionality. It is required regardless of the interface type – even if a user is using an interface that is not CORBA, the IIOP_IP parameter must be set with the correct IP address for Coppelia to run properly.**	**In IP format, such as 192.0.0.5**
IIOP_PORT	**Which port number that will be**	**Any unique numeric port**

	used to communicate to the server via the CORBA interface.	**number. All port numbers must be unique.**
INTERFACE	The type of binding to your own middleware. For types other than CORBA, see the appropriate section in the document for that interface.	CORBA (default) RV (TIB/Rendezvous) DCOM (Microsoft OCX/COM) MQSERIES (IBM MQSeries) OBSERVER (Native or Observer / Observable interface) AMBROSIA (IXNet pub/sub) RMI (RMI)
LOCAL_PORT	**Port number for this server that is used for TCP/IP connections. Note – for Coppelias configured as Servers, this is the port number that remote clients will connect to, so remote clients will need to know this port.** **For Coppelias that are configured as Clients, this port number does not need to be known by Servers that will be connected to, but this still must be set.**	**Any unique port number.**
LOG_DAYS	The number of days after which Coppelia will automatically delete the generated .log files and .rej files.	Number of days Default is 14 days.
LOG_HEARTBEAT	Whether to log heartbeats to the screen and to the log file or not. No logging of heartbeats will save on log file clutter, while logging them may help for debugging purposes.	ON OFF – default
MESSAGE_ON_LOGON	Enter any text here that you want to include into your logon message's text field.	Any string(s)
NO_SERVER_CHECK	When running multiple Coppelia engines on a single machine, this parameter prevents the Coppelia from getting confused between multiple targets. Failure to connect a Coppelia engine that is on the same machine is usually caused when this parameter is	ON OFF – default

	not set to ON.	
NO_IP_CHECK	Instructs Coppelia to NOT care about the incoming connections' source IP, regardless of what is in the .dat file. This is per server, not per connection. What this means is that if a connection from a remote ID comes in on a different IP address than what is specified in the .dat file, Coppelia will allow the connection. Use this parameter carefully, as it will allow any engine with the proper Firm ID to connect to you.	ON OFF – default
NO_PERSISTANCE	This prevents certain message types from being persisted (saved) to the Coppelia database. This is used when expecting large messgae traffic, and not all messages need to be saved.	Example: 8, 6 *will not save Execution Report and Indication of Interest messages in the Coppelia database*
REJECT_MSGTYPES	Coppelia will automatically send a Reject message if a message is received in this list. Administration messages are ignored.	Example: 8, 6 *will reject Execution Report and Indication of Interest messages*
TITLE	Coppelia GUI ONLY - This goes on the title bar of the server; it can be used for informational purposes.	Any string
TRADER_IDS	Coppelia GUI ONLY - There should be an entry for each *remote id* for the ID parameter. This is only for using the built in GUI. It is ignored in the non-GUI mode.	*Remote id; trader name 1, trader name 2, trader name 3, ...*
TYPE	**Whether the server is configured for the buy side (sending orders, receiving execution reports, receiving indications) or the sell side (receiving orders, sending indications, sending execution reports)**	**BUY** **SELL**

2.3.3.2 Remote Connection configuration

Each remote connection for a Coppelia engine must be specified with its own unique ID line, which contains all information needed to connect to a remote FIX engine. This is an example of an ID line.

The format is

```
ID; TargetCompID; SenderCompID; SenderSubID; IP address;
Port number; Target Identification and Description; Contact
Information; Heartbeat Interval; Encryption Type; FIX
Version and Starting sequence number

ID; SBI; SLGM; GEORGE; 192.168.129.25;9876; Salomon
Brothers Inc; Tech Support (212) 555-1212;30;0;401
```

The different parameters are divided by semicolons - ; All fields are requied to be set! The meaning of each parameter is explained in the following table

Parameter	*Description*	*Possible values*
ID	Each ID line must beings with the "ID" character	ID
SBI	This is the TargetCompID, or the Firm ID of the remote connection that will be communicated with	Any string
SLGM	This is the SenderCompID, or the Local Firm ID for the Coppelia engine. This is how this Coppelia engine will identify itself to remote FIX engines.	Any string
GEORGE	This is the SenderSubID, a parameter that will provide more information about the Sender, which is used in identification to remote FIX engines.	Any string
192.168.129.25	The IP address of the remote FIX server. Or, if the DNS parameter is set to ON, any DNS name	Any IP address, or, in the case of DNS, any string.
9876	The port number of the remote connection – what port number is used for that targets connections. **Note** – for a Coppelia engine that is set as Server and will not connect to anyone, the value of this field is irrelevant. However, it must be set with a value for the Coppelia engine to start up properly.	The appropriate port number for the remote FIX engine.

Salomon Brothers Inc	This string provides more identification information about the remote connection – the name of the company, exchange, ECN, etc. It can contain any value and is included for reference	Any string
Tech Support (212) 555-1212	Contact information for the Remote connection. Again, it can contain any value and is included for reference	Any string
30	The heartbeat interval – seconds to wait before sending a heartbeat to the remote FIX engine.	Any numeric value greater than zero
0	The encryption used on this connection. 0 is used if connection is unencrypted	0 – No encryption 5 – PGP/DES/MD5 encryption are the only values for this field
401	Fix version and starting sequence in this format – (FIX version * 100) + Starting sequence number. i.e. Fix 4.0 and starting seq # 1 (4.0 * 100) + 1 = 401	Must be properly formatted numeric value

2.4 System Administration

This section of the document attempts to help the operations department at our clients' sites to test, run and trouble-shoot Coppelia and its FIX-based connections to their clients. In the course of doing so, basic knowledge in FIX is necessary. This document uses the term "client" to mean the application and FIX engine at your clients' site. The Coppelia FIX engine is the "server" to that "client."

2.4.1 Installation

2.4.1.1 Downloading Coppelia

Using your web browser (i.e. Netscape Navigator), enter in the "Location" box the following:

http://www.javtech.com/downloads

Note #1: The above URL is password protected. You will not be able to download the software without a username and password. Please contact Javelin Technologies, Inc. if you do not have a username and password for the site.

Note #2: The software on the website is a limited version of Coppelia. You must contact Javelin Technologies, Inc. for other versions.

Pick the version of software you would like to download from the list, i.e., coppelia_winnt.zip for Windows NT installations, or Coppelia_UNIX.tar.gz for UNIX-based operating systems. Save in a directory where you want Coppelia to be installed. Downloading will commence.

2.4.1.2 Uncompressing the file

For UNIX:

In a terminal window, uncompress "coppelia_solaris.tar.Z" by running the command:

```
uncompress coppelia_UNIX.tar.gz
```

Extract the tar file by running the tar command:

```
tar xvf coppelia_UNIX.tar
```

Extraction will take place and you will end up with a directory called "Coppelia". Within the directory, there are three subdirectories (buy, sell and classes) and one HOWTO.TXT file.

Make sure that system-specific run-time for Java (JRE) is installed and operating properly on your system. Verify that your path variables are correct to ensure proper functionality of Coppelia before you proceed.

For Windows NT:

Unzip the file using either WinZip or any other decompressing tool for Windows. Once you finish unzipping, You will see a folder called "Coppelia". Inside the directory, there will be five subfolders (buy, sell, classes, bin and lib).

Windows NT users already have run-time for Java included with the zip package.

Please call Javelin Technologies' support if you need more information about the currently recommended Java versions.

2.4.1.3 Configuring Coppelia

Please make sure that the Run-Time for Java (JRE) is installed and operating properly in your Unix system. Verify that your paths are correct to ensure proper functionality of Coppelia before you proceed. Windows NT users already have Run-Time for Java included with the software package just downloaded.

Configuring the **Buy.dat** file

*The downloaded package requires NO further modifications for proper operation.

Within the "buy" directory, there is a file called "buy.dat". There are the parameters needed to set the "Buy Side" for communicating with the sell side.

If you want for testing to modify that file, open it with the "Vi", Word or any other editor. Do your changes and modification and save it before exit

An example of the structure of "buy.dat" file is shown below:

```
    TYPE                    BUY
    LOCAL_PORT              7200
    IIOP_PORT               7100
    CONNECT                 CLIENT
    DESCRIPTION             Buy Side configuration
    LOG_HEARTBEAT           ON
    TITLE                   Coppelia v4.1, BUY SIDE
    GUI                     ON
    IIOP_IP                 127.0.0.1

# ID; TargetCompID; SenderCompID; SenderSubID; network
address; description; contact; heart beat; encryption type;
version number+start seq num
ID; SBI; SLGM; GEORGE; 127.0.0.1;7000; Salomon Brothers
Inc; Tech Support (212) 555-1212;30;0;401

TRADER_IDS; SBI; JOE, HARRY, SAM, MAGGIE, LISA, HOMER,
MARGE
```

Explanation of the "Buy.dat " file parameters:

- **LOCAL_PORT** represents the TCP/IP port that is available to the network. This is where the
- Computer "listens" for FIX messages. (This must be agree with the remote party)
- **IIOP_PORT** represents the port that will be used to communicate to the server via the CORBA
- Interface.
- **CONNECT** specifies whether it will be a Client or a Server. For the demo, the buy side is always client. The difference between client and server is that the client can make and to accept a connection .The server only can accept a connection.
- **DESCRIPTION** Comments .Can be any string. This is kept internally in the server.
- **LOG_HEARTBEAT** is the setting to screen the heartbeats in the log.
- **TITLE** specifies which engine is running, whether buy side or sell side.
- **GUI** is the graphical interface for the demo. You could set it ON to run or OFF to not.
- **TRADETABLE** setting is adjusted here
- **IIOP_IP** Represents the IP address of this server
- **ID** Contains connections information delimited by a " ; " char. All information for the connection
- must be appear on this line
- **TRADER_ID** line at the bottom of the file indicated the company and traders the buy side is
- communicating to.

The settings **ID** featured are: (needed in order for the buy side to communicate with the sell side.)

- **TargetCompID** is the ID of the company that the buy side is connecting to. The demo setting is SBI or Salomon Brothers, Inc.
- **SenderCompID** is the ID of the company of the buy side. Here it is set to SLGM.
- **SenderSubID** is the name of the trader who is sending the message.
- **Network Address** is the IP Address of the server you are going to connect and the port number where he "listen" for connection.
- **Description and Contact** are text fields describing the name of the company the buy side is connecting to and the technician available in case of communication problems.
- **Heartbeat** is the setting for the time interval between heartbeats.
- **Encryption type** you set the encryption type.

- **Version number + start sequence number** is the setting for which FIX version is used and what the beginning sequence number is set. Here, it is set for version 4.0 with a start sequence number of 1.

Configuring the **sell.dat** file

*The downloaded package requires NO further modifications for proper operation.

Within the "sell" directory, there is a file called "sell.dat".

There are the parameters needed to set the "Sell Side" for communicating with the Buy side. As described above, if you want for testing to modify that file, open it with the "Vi", Word or any other editor. Do your changes and modification and save them before exiting the editor.

An example of the structure of "sell.dat" file is shown below:

```
TYPE                    SELL
LOCAL_PORT              7000
IIOP_PORT               7150
CONNECT                 SERVER
DESCRIPTION             Sell Side configuration
LOG_HEARTBEAT           ON
TITLE                   Coppelia v4.1, SELL SIDE
GUI                     ON
IIOP_IP                  127.0.0.1


# ID; TargetCompID; SenderCompID; SenderSubID; network
address; description; contact; heart beat; encryption type;
version number+start seq num
ID; SLGM; SBI; GEORGE; 127.0.0.1; 7200; Salomon Brothers
Inc; Tech Support (212) 555-1212; 30; 0; 401

TRADER_IDS; SBI; JOE, HARRY, SAM
```

Explanation of the "Sell.dat " file parameters

Similar to the buy.dat file, see above "Explanation of the "Buy.dat " file parameters"

The settings are slightly different than buy.dat file. The differences are:

- **TargetCompID** and **SenderCompID** which now is opposite than Buy.dat file
- **LOCAl_PORT** Because the sell side " listen" for communication to the port number 7000

- **IIOP_PORT** now the port for the communication with CORBA interface is 7150
- **ID** To the **Network address** field, the port number where the buy side "listen" is 7200

2.4.1.4 Starting Coppelia

For UNIX:

- Within the "buy" directory, execute the buy side by typing "go_buy" in the command line. This will activate Coppelia w/ its GUI (provided that it was set on in the buy.dat).
- Do the same for the sell side, but this time type "go_sell" in the "sell" directory.
- You are now ready to initiate a mock trade.

For Windows NT:

- Within the "buy" folder, execute the buy side by double-clicking on the icon entitled, "go_buy.bat". This will activate Coppelia w/ its GUI (provided that it was set on in the buy.dat).
- Do the same for the sell side, but this time type "go_sell" in the "sell" directory.
- You are now ready to initiate a mock trade.

The Coppelia engine should be started from a script, (see the package for NT examples). There are three ways to start a Coppelia server:

```
Coppelia [config file]
Coppelia [config file] [Server Name]
Coppelia –remote [URL] [Server Name]
```

See Section 2.3.1 "Configuration Syntax" for more information.

Examples

Command Line	Description
Coppelia buy.ini	Load configuration information from buy.ini
Coppelia sell.ini SERVER1	Load configuration from sell.ini and set the server name.
Coppelia –remote http://config/sell.ini SERVER10	Load remote config file and set the server to SERVER10

The following is a UNIX script example the classpath has been intentional ignored. This example sets two environment variables ORBIX_PATCH and RMI_SEC.

```
export ORBIX_PATCH="-Djava.compiler=NONE \
-Dorg.omg.CORBA.ORBClass=IE.Iona.OrbixWeb.CORBA.ORB \
-
Dorg.omg.CORBA.ORBSingletonClass=IE.Iona.OrbixWeb.CORBA.sin
gletonORB"

export RMI_SEC="\
 -
Djava.security.policy=/export/home/ben/dev/Coppelia41eHA/cl
asses/policy.txt \
 -
Djava.rmi.server.codebase=file:/export/home/ben/dev/Coppeli
a41eHA/classes/coppelia.jar"

java -mx512M $ORBIX_PATCH $RMI_SEC -classpath $CPATH
Coppelia sell.ini
```

The ORBIX_PATCH is set to allow Orbix to run in Java 1.2 this is to ensure there is no name conflicts with the CORBA now supplied in Java 1.2. If you are running in Java 1.1.* you need not include this atttribute. See Section 3.4.1 CORBA for more information.

The RMI_SEC attribute defines the security policy and the code base for the RMI Registry, it is required to ensure the server has the correct permissions to connect to the RMI registry. The attribute is not required if you don't intend to use RMI or High Availability. See Section 3.4.2 RMI for more information.

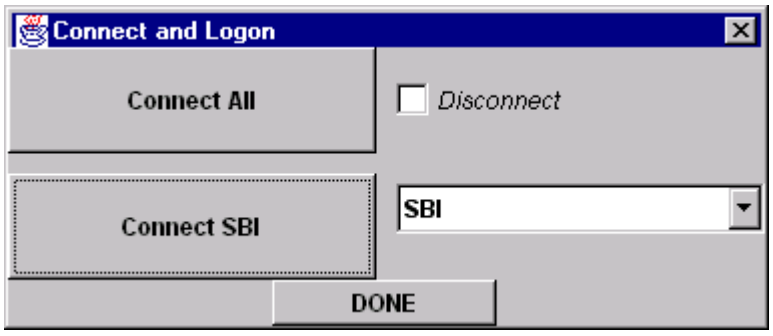
2.4.1.5 Running a mock trade

When both the buy side and sell side are running, you will see a GUI displayed for both sides:

BUY SIDE

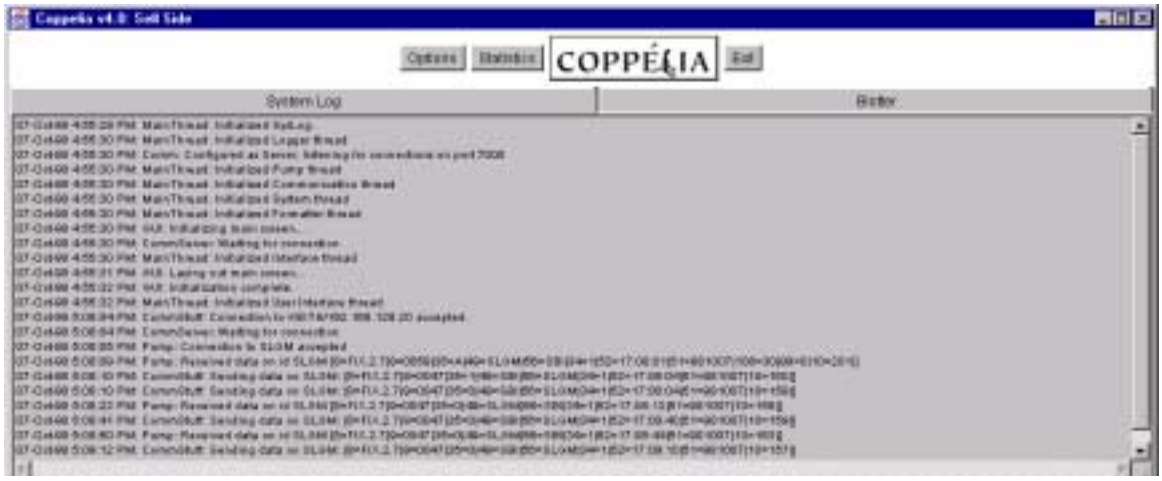
SELL SIDE

To start a session, click on the "Connect" button located at the buy side. A dialog box will appear afterwards:



For demo, we will pretend the Buy side is Selgmund Funds (SLGM) connecting to Salomon Brothers (SBI). Click on the "Connect SBI" button. Give it two to three seconds for connection to establish and then click on "Done".

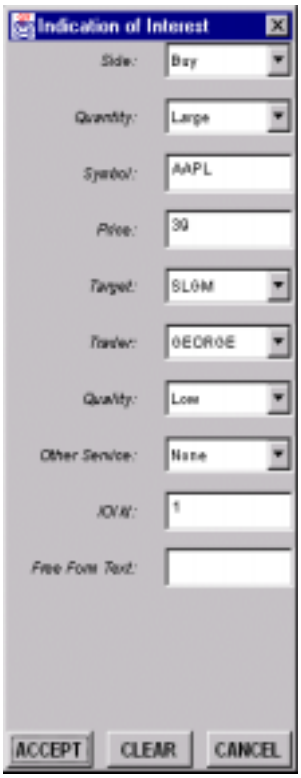
You will notice that in the System Log on both sides, a connection was established:



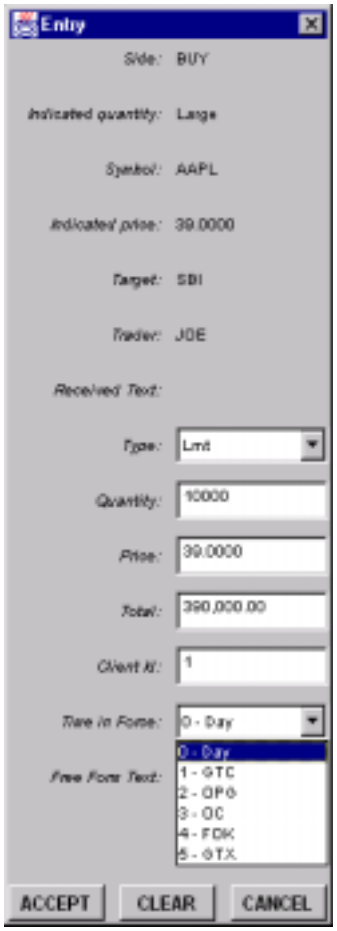
Click on the "Blotter" Tab in order to start trading. The GUI's will look like this:

(Insert BuySide Blotter Picture here)

To send an Indication of Interest (IOI), you have to start in the sell side. Within the "Action" Column, click on the "New" Button. A new dialog box will appear. The "Side" Field indicates whether you want to "buy" or "sell". "Quantity" specifies the size of the trade. "Symbol" would be the stock symbol of the company. "Price" would be the price where the IOI wants to be met. "Target" is the company where this IOI is intended. "Trader" would be the trader of the target. "Quality would mean the quality of the trade. "Other services" indicate whether the trade will require "Autex" "Bridge" or "Both". For the demo, you can keep "Target", "Trader", "Quality" and "Other Services" to their default settings (as shown in the example to your right). "IOI id" should be a unique alphanumeric input. It is like the "Ticket #"of the IOI, that's why it has to be unique. For demo purposes, it is set to "1". "Free form Text" is for sending text messages across the wire to the target. You can skip that field. Once you have all your fields filled up, click on the "Accept" button to complete the IOI.



On the buy side, you will notice that the IOI was accepted and noted in the blotter:

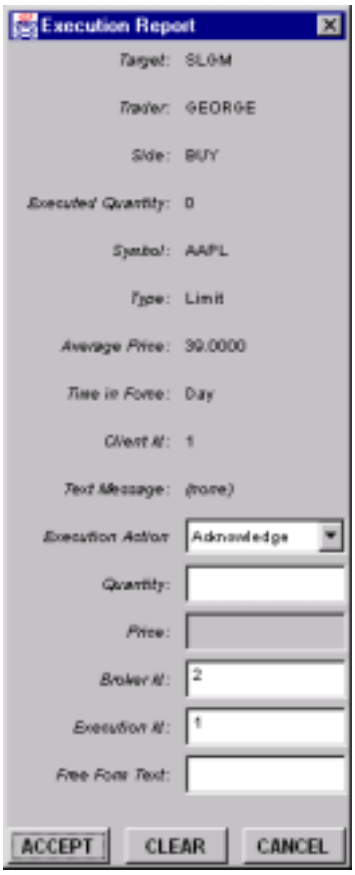


Click on the "New IOI" within the "Action column. Another dialog box will appear (left figure). You will notice that the "Side", "Indicated Quantity", "Symbol", "Indicated price", "Target" and "Trader" have all been filled out. These parameters have been set from the IOI initiated before. "Type" would indicate what kind of trade will occur (market or limit). Quantity would be the size of the trade. "Price" is the price of the stock. "Total" is the total price of the trade (Quantity x Price). Client ID is similar to IOI id and it must be unique. In this example, we'll keep it a "1". "Time in Force" shows what time will the trade take place. When you have finished completing the fields, you can click on the "Accept" button.

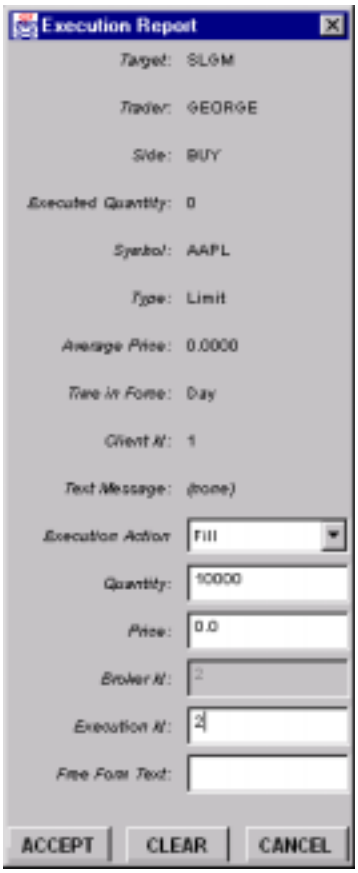
Note: Due to the limitations of this demo, NYSE/AMEX stocks (3-letter symbols) can only have a maximum quantity of 5000. NASDAQ stocks (4 or more letter symbols) can have up to 1000000.

Go back to the sell side. You will see that the order is ready for execution:

Click on the "Execute" button. This time, the Execution Report Dialog box appears. In the "Execution Action" field, choose "Acknowledge" to tell the buy side that you have received and will work on the Execution. You will not be able to enter a "Quantity" nor a "Price" because this is only an acknowledgement, not an actual trade. The "Broker ID" field has to be unique. It is like the broker ticket number. In this example, we will use "2". "Execution ID" should also be unique. Here, we will use "1". Click on "Accept" to continue.



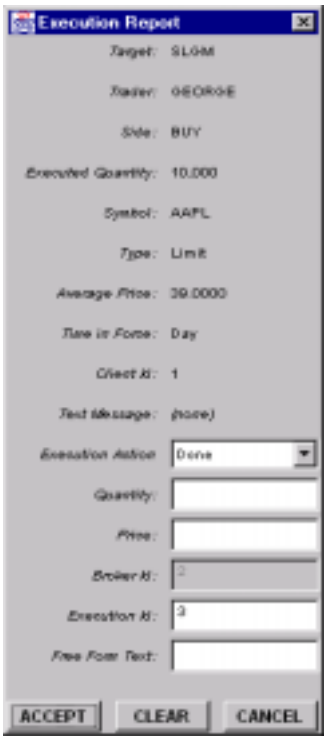
Click on "Execute" within the blotter of the sell side. This time, we will complete the order. Once the Execution Report comes up, choose "Fill" in the Execution Action field. Enter the quantity of shares you want. Set the price of the stock. In this example, we are using 10000 and 39 respectively. In the Execution id, which has to be unique once more, set it to 2. Click on "Accept" to continue.



Notice that the buy side blotter shows the trade has been complete and filled. The Action column indicates that the trade is now "Done".

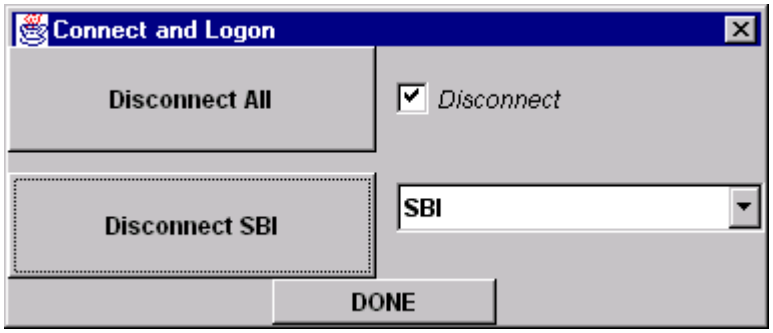
One the sell side, you will see that the blotter has logged the details of the trade. To finalize the trade, click on the "Execute" button to bring up the Execution Report for one last time.

Set the "Execution Action" to "Done." Change the "Execution ID" to another unique value. Here, we set it to "3". The rest of the fields will be inaccessible. Click on "Accept" to complete the process.



Shown below is the change in the "Action column. Execute has changed to "Done"

To end the session, go to the buy side, click on the "Connect" button. Check on the "Disconnect" box and click on the "Disconnect SBI" button and then "Done".



Click on the "Exit" button on both sides to exit the whole GUI. You have finished your mock trade.

2.4.1.6 Remote Configuration Loading

The ability to remotely load the Coppelia configuration file has two main advantages. They are:

- It allows a single configuration file to be distributed to all Coppelia Servers and ensures that they are kept in sync.
- The configuration file can be kept behind a firewall to protect sensitive configuration information.

The configuration file can be distributed to the servers via URL.

2.4.1.7 HINTS

(This section TBA)

2.4.2 Database

The FIX Protocol implementation dictates that messages be persisted. Javelin Technologies has chosen to persist the information in a database. Coppelia is bundled with a database from eXcelon (formerly Object Design). However, Coppelia EP and Coppelia HA can use other databases besides the database that is bundled with Coppelia.

At startup, Coppelia will look for its files to be in the path provided to it in the startup script, and also check whether there are any database files already exist. If they do, the existing database will be read in. If there are no database files found, Coppelia will create new database tables.

2.4.2.1 eXcelon Databases

Coppelia supports two databases from Excelon Corp (http://www.exceloncorp.com/), Pse Pro and Objectstore.

2.4.2.1.1 Pse Pro

Pse Pro is a lightweight database that is included with Coppelia.

2.4.2.2 JDBC Databases

Coppelia's support of other databases is made possible through the use of JDBC drivers. Many SQL databases have JDBC drivers. Databases with only ODBC drivers can be used through an ODBC-JDBC bridge driver. The Java Development Kit (JDK) includes an ODBC-JDBC bridge driver. There are also third party ODBC-JDBC bridge driver. Javelin Technologies recommends that you use databases with JDBC drivers if at all possible.

2.4.2.2.1 IBM DB2

(Section TBA)

2.4.2.2.2 Oracle

2.4.2.2.2.1 Supported Version(s)

Currently, Coppelia works with Oracle database servers versions 8.x, including 8i. No other versions or releases have been sufficiently tested. The connection to the Oracle database is made possible by using a JDBC interface supplied by Oracle. Coppelia supports both the `oci`-based and `thin` drivers.

2.4.2.2.2.2 Pre-Requisites

You must have Oracle installed and running on a local or remote machine.

1. The database has to be already created in Oracle; that means the database only, not the actual tables.
2. The database must be accessible from Coppelia via a username and password known to you.
3. The user and password must have permissions to create, read and write tables.

2.4.2.2.2.3 Configuration File Entries

In the appropriate .dat file, you need to enter the following parameters for Coppelia to be able to work with the database server.

2.4.2.2.2.3.1 OCI JDBC Driver

```
PERSISTENT_DB       SQL
SQL_URL             jdbc:oracle:oci8:@[DATABASE_SERVICE]
SQL_DRIVER          oracle.jdbc.driver.OracleDriver
SQL_USER            [USER]
SQL_PASSWORD        [PASSWORD]
```

where:

SQL_URL describes the location and 'contact' information for the database. "jdbc:oracle:oci8:@" is constant, followed by the service name of your database.

The service name does not make any differentiation between local or remote databases, since that definition is implied in the service name itself.

"LOCAL" here refers to a local database. If your database were on a different machine on your network, say on enterprise_server, and named CDB, you define a service name for this database, and refer to this service name in the .dat file, say for example "jdbc:oracle:oci8:@REMT". REMT as service name contains your server name and database name implicitly.

SQL_DRIVER is the name of the driver to use, leave that as is.

[SQL_USER] and **[SQL_PASSWORD]** are set to whatever the password and user are for the particular database you intend to use.

2.4.2.2.2.3.2 THIN Driver

If you are using the THIN driver, replace "oci8" with "thin" in the configuration file entries mentioned in section 3.1 above.

In addition, you need to give the path of the database explicitly, not by means of a service name as described above. For example, you would give Coppelia the following strings for the SQL_URL and SQL_DRIVER parameters:

```
SQL_URL         jdbc:oracle:thin:@[HOST]:[PORT]:[DATABASE]
SQL_DRIVER      oracle.jdbc.driver.OracleDriver
```

In this example, HOST is the DNS name of the machine your database server runs on, while PORT is the port on that machine to connect to. If you do not have these parameters, ask your database administrator for help.

2.4.2.2.2.4 Running Coppelia with Oracle

When running Coppelia with an Orcale database server, your startup script must point to the Oracle-supplied java archive as outlined below in order for the JDBC driver to work.

Use the appropriate the java archive, based upon which driver you are using and also which version of the java virtual machine (JVM) you intend to use to run Coppelia itself.

If you intend to run Coppelia with Oracle using a Java 1.2.x implementation, you need to have a file named classes12.zip. This file is not supplied with all Oracle installation packages. If you cannot find this file in your installation, you can download it from Oracle's website at

http://technet.oracle.com.

2.4.2.2.2.4.1 OCI Drivers for Java 1.1.x

```
SET CLASSPATH=d:\orant\JDBC\LIB\classes111.zip;
```

The above `SET CLASSPATH` statement implies that you have the classes111.zip file in drive `D:\` on a WindowsNT machine. For UNIX machines, you path statement may differ.

This statement is only part of your startup script.

Note: Some OCI drivers might need additional setup, e.g. setting the path. Please refer to the Oracle JDBC driver documentation for more details on this procedure.

2.4.2.2.2.4.2 OCI Drivers for Java 1.2.x

```
SET CLASSPATH=d:\orant\JDBC\LIB\classes12.zip;
```

The above `SET CLASSPATH` statement implies that you have the classes12.zip file in drive `D:\` on a WindowsNT machine. For UNIX machines, you path statement may differ.

This statement is only part of your startup script.

Note: Some OCI drivers might need additional setup, e.g. setting the path. Please refer to the Oracle JDBC driver documentation for more details on this procedure.

2.4.2.2.2.4.3 THIN Drivers for Java 1.1.x

```
SET CLASSPATH=d:\orant\JDBC\LIB\classes111.zip;
```

The above `SET CLASSPATH` statement implies that you have the classes111.zip file in drive `D:\` on a WindowsNT machine. For UNIX machines, you path statement may differ.

This statement is only part of your startup script.

Note: Some OCI drivers might need additional setup, e.g. setting the path. Please refer to the Oracle JDBC driver documentation for more details on this procedure.

2.4.2.2.2.4.4 THIN Drivers for Java 1.2.x

```
SET CLASSPATH=d:\orant\JDBC\LIB\classes12.zip;
```

The above `SET CLASSPATH` statement implies that you have the classes12.zip file in drive `D:\` on a WindowsNT machine. For UNIX machines, you path statement may differ.

This statement is only part of your startup script.

Note: Some OCI drivers might need additional setup, e.g. setting the path. Please refer to the Oracle JDBC driver documentation for more details on this procedure.

2.4.2.2.2.5 Example Full Startup Scripts

Please note that the file `pro.zip` has to appear in the classpath when starting Coppelia, even though Oracle instead of the default PSE Pro database by eXcelon is used. This is a known issue with Coppelia, and will be addressed in an upcoming release.

2.4.2.2.2.5.1 Java 1.1.x

```
set PATH=d:/jdk1.1.8/bin;%PATH%
set CLASSPATH=.;../classes;../classes/pro.jar;
../classes/coppelia.jar;../classes/mct3_0.zip;
../classes/rogue.zip;../classes/OrbixWeb31c.jar;
../classes/tools.jar;../lib;../classes/rvjpro.jar;
../classes/classes111.zip;d:/jdk1.1.8/lib/classes.zip;
../classes/jconnect.jar
java -mx256m -classpath %CLASSPATH% Coppelia buy.dat
```

This script assumes that Oracle's `classes111.zip` resides in the same directory as Coppelia's class files, i.e., `../classes`. The parameter `"-mx256M"` is not necessarily the value for your machine.

Note that the line breaks in this example appear for printing purposes only; the file shown above actually has three lines.

2.4.2.2.2.5.2 Java 1.2.x

```
set PATH=d:/jdk1.2.2/bin;%PATH%
set CLASSPATH=.;../classes;../classes/pro.jar;
../classes/coppelia.jar;../classes/mct3_0.zip;
../classes/rogue.zip;../classes/OrbixWeb31c.jar;
../classes/tools.jar;../lib;../classes/rvjpro.jar;
../classes/classes12.zip;d:/jdk1.2.2/jre/lib/rt.jar;
../classes/jconnect.jar
java -mx256m -classpath %CLASSPATH%
-Dorg.omg.CORBA.ORBClass=IE.Iona.OrbixWeb.CORBA.ORB
-Dorg.omg.CORBA.ORBSingletonClass=IE.Iona.OrbixWeb.CORBA.
singletonORB Coppelia buy.dat
```

This script assumes that Oracle's `classes12.zip` resides in the same directory as Coppelia's class files, i.e., `../classes`. The parameter `"-mx256M"` is not necessarily the value for your machine.

The parameters starting with "-Dorg.omg..." are necessary for Coppelia to be able to use the integrated OrbixWeb CORBA code (used intally by Coppelia regardless of external interface option). If these parameters are not set, Coppelia would try to use the ORB integrated into Java 2, and incompatibilities would occur.

Note that the line breaks in this example appear for printing purposes only; the file shown above actually has three lines

2.4.2.2.2.6 Database Structure

Coppelia will connect to the database specified, and create tables following the listed ddls (if these tables do not already exist). It is recommended to let Coppelia create the tables itself.

If Coppelia is brought down, and no end-of-day has been run to truncate records and tables, Coppelia will connect to the database, and read the existing tables. This will result in Coppelia using the same sequence numbers as they have been left when Coppelia was brought down.

```
create table outbound
(
id                  varchar(255),
msg                 text,
)
```

```
create table inbound
(
id                  varchar(255),
msg                 text,
)
```

```
create table stats
(
firm_id                 varchar(255),
msg_seq_num_out         integer,
msg_seq_num_in          integer,
last_interface_num      integer,
messages_in             integer,
messages_out            integer,
orders                  integer,
executions              integer,
execution_acks          integer,
rejects                 integer,
allocations             integer,
iois                    integer,
heartbeats              integer,
cancels                 integer,
corrects                integer,
logins                  integer,
bytes_in                integer,
bytes_out               integer
)
```

2.4.2.2.2.7 Troubleshooting

Following are a few errors we have encountered, and the appropriate procedure to fix the problem.

Error #1:

When starting up Coppelia, you see the following lines displayed in your console window:

```
.
.
Using JDBC/SQL as persistent database...
java.lang.UnsatisfiedLinkError: no ocijdbc8 in java.library.path
.
.
```

This usually means that your `PATH` variable does not include the Oracle libraries. Make sure you include the `/bin` directory in your PATH setting.

Error #2:

When starting up or running Coppelia, you see the following line displayed in your console window:

```
.
.
[Date] JdbcPersistentDB: Warning: Database access error – ORA-
00904: invalid column name
.
.
```

This error message means that you are using a version of Coppelia that is not compatible with the current structure of the database. You need to cleanup all tables, and let Coppelia create new tables in order to avoid this error message.

2.4.2.2.3 Sybase

2.4.2.2.3.1 Supported Versions

Currently, Coppelia works with version 6.x of the Sybase SQL Anywhere Server, as well as Sybase SQL Server 11.0.x. Support is through the JDBC driver.

2.4.2.2.3.2 Tested Driver(s)

Javelin has uniquely tested the JDBC driver included in Sybase's database server installation package.

2.4.2.2.3.3 Pre-Requisites

1) You must have Sybase installed and running on a local or remote machine.

2) There has to be a database (its name is unimportant—name it whatever you want) created in Sybase. Note that there has to be one database (not necessarily one database *server*) per Coppelia Server. That is, if you run a buy side and a sell side Coppelia, you need two databases, one for each Coppelia server. Use Sybase Central for an example on how to do this, and make sure all services are up (you can use Sybase Central to check and correct if necessary). Note also that database instances in Sybase for WinNT register themselves as services on this machine.

3) The database you create must be accessible from Coppelia via a user and password, which you define in Enterprise Manager. It's also a good idea to test the connectivity to the database with Sybase's supplied Database Wizard tool.

2.4.2.2.3.4 .dat File Entries

In the .dat file, you need to enter the following for Coppelia to work with the database server:

```
PERSISTENT_DB          SQL
SQL_URL                Jdbc:sybase:Tds:localhost:2638?Serv
                       iceName=[SERVICE NAME}
SQL_DRIVER             com. bsyase.jdbc.SybDriver
SQL_USER               USER
SQL_PASSWORD           PASSWORD
```

where:

SQL_URL describes the location and 'contact' information for the database. "jdbc:sybase:Tds:localhost:2638?ServiceName=" is constant (the port number, of course, is a variable), followed by the service name of your database. The service name does not make any differentiation between local or remote databases, since that definition resides in the service name itself. "LOCAL" here refers to a local database. If your database were on a different machine on your network (say on 'enterprise_server'), and it was named "CDB", you would define a service name for this database, and refer to the service name in the .dat file. For example:

"`jdbc:sybase:Tds:localhost:2638?ServiceName=test`".

Using 'test' as the service name implicitly contains your server name and database name.

SQL_DRIVER specifies the name of the driver to use. You might want to check the Java examples provided with your server product for the name of this driver when not using Adaptive Server Anywhere 6.x, or SQL Server 11.0.x. Of course, you can always contact Javelin Technologies when you encounter problems or have a question.

SQL_USER and SQL_PASSWORD are set to whatever the user and password are for the particular database you intend to use.

2.4.2.2.3.5 Startup Script

Your startup script must point to the Sybase-supplied archive as outlined below in order for the JDBC driver to work:

```
SET CLASSPATH=[path to the driver library]\jdbcdrv.zip;
```

In addition, you have to have your other path and classpath environment set in this script.

2.4.2.2.3.6 Database Schema for Sybase ONLY

The following tables were changed to improve performance of Coppelia when using a Sybase database as the persistence storage. The new columns are shortMsg and msgSize. If the FIX message is less than or equal to 255 characters, the FIX message will be stored in shortMsg and the length will be stored in msgSize. If the FIX message is longer than 255 characters, the message will continue to be stored in msg and the length will be stored in msgSize. Any exisiting INBOUND and OUTBOUND tables must be dropped before using Coppelia version 41e07.

```
create table outbound
(
id                  varchar(255),
msg                 text,
shortMsg            varchar(255),
msgSize             integer
)

create table inbound
(
id                  varchar(255),
msg                 text,
shortMsg            varchar(255),
msgSize             integer
)

create table stats
(
firm_id             varchar(255),
msg_seq_num_out          integer,
msg_seq_num_in      integer,
last_interface_num  integer,
messages_in              integer,
messages_out        integer,
orders              integer,
executions               integer,
execution_acks      integer,
rejects             integer,
allocations              integer,
iois                integer,
heartbeats               integer,
cancels             integer,
corrects            integer,
logins              integer,
bytes_in            integer,
bytes_out           integer
)
```

2.4.2.2.3 MS SQL Server

(Section TBA)

2.4.3 Command Line Interface

2.4.3.1 Help

The **?** command displays the available commands on the command line.

?

2.4.3.2 Statistics

The **stats** command displays the current status for each the connection.

The information contains where the connection is up or down. It also contains the current inbound and outbound FIX sequence numbers.

stats

2.4.3.3 Connect/Disconnect

The **connect** command runs logons a FIX connection to a counterparty.

```
connect [ID] - Logon to a specific connection.
connect ALL  - Logon to all connections.
```

The dis**connect** command runs logs off a FIX connection to a counterparty.

```
disconnect [ID] - Logoff a specific connection.
disconnect ALL  - Logoff all connections.
```

2.4.3.4 End of day

The **eod** command runs end of day.

```
eod [ID] - Runs End of day for a specific connection.
eod ALL  - Runs End of day on all of the connections.
```

2.4.3.5 Sequence Reset

`msg_seq_num_in [ID] [Seq_Num]`

The msg_seq_num_in command changes the inbound FIX sequence number for a specific connection.

`msg_seq_num_out [ID] [Seq_Num]`

The msg_seq_num_out command changes the outbound FIX sequence number for a specific connection.

`seq_reset [ID] [Seq_Num]`

The seq_reset command resets both the inbound and outbound FIX sequence number for a specific connection.

2.4.3.6 Version

`version`

The version command displays the version of Coppelia.

2.4.3.7 Reconfigure

`reconfigure`

The reconfigure command reloads the .dat file into Coppelia.

Current limitations of `reconfigure` command:

- Cannot delete ID lines online
- Cannot reconfigure upper part of .dat file
- (bug) doesn't create new RV subjects when adding an ID line...

2.4.3.8 Garbage Collection Time

`set_gc_time TIME(in seconds)`

2.4.3.9 Check Memory

`check_memory`

2.4.3.10 Autoconnect

`autoconnect NUMBER`

2.4.3.11 Exit

`exit`

2.4.4 Blotter/GUI

The Blotter/GUI is not officially part of the Coppelia server. Please do not use the GUI as part of your production environment.

2.4.4.1 Usage

(Section TBA)

2.4.4.2 Interface

(Section TBA)

2.4.4.3 Limitations

(Section TBA)

2.4.5 Day to Day Maintenance

2.4.5.1 Adding new clients and new connections

Client Contact Information Sheet

It is recommended that you keep the following information on hand at all times to be able to quickly trouble-shoot problems with connections. Make sure you update this information from time to time as necessary.

Client Name	
Client's ID (TargetCompID)	
Client's IP address	
Client's port (if you connect to Client)	
Your own SenderCompID for Client	
Business Hours for Client	
End-Of-Day window for Client	
FIX version for Client	
Heart Beat Interval for Client	
Contact Name(s) for Client	
Contact Phone / Pager / Email	
Contact at your site to notify of problems	

2.4.5.2 End-of-Day

It is important to be in agreement with your counter party as to when the end of a business day is reached. Most parties will have set times when it comes to database maintenance, daily turn-around procedures, etc. Every single one will have to be dealt with individually in order to avoid sequence numbers being out of sync, or even old data being transmitted by accident.

2.4.5.2.1 Run End-Of-Day (EOD) for a specific connection

From Coppelia's command prompt, type:

```
eod [ID]
```

2.4.5.2.2 Run End-Of-Day (EOD) for a all connections

From Coppelia's command prompt, type

```
eod ALL
```

If EOD succeeded, the console will notify you accordingly.

2.4.6 Upgrading Coppelia

(Section TBA)

2.4.7 Troubleshooting

2.4.7.1 Tools

2.4.7.1.1 Location of Files, OS Version, Java

In the event of a problem, Javelin Technologies, Inc., or your own development team might ask you for certain files, such as the startup script used, the log file (in the format CYYYYMMDD.log, where C stands for Coppelia), or the configuration file (in the format file_name.dat). Be prepared by knowing where these files reside, and how to email them if needed.

In addition, it is good to know what operating system you run, and what version of Coppelia and of Java you are implementing at the time.

2.4.7.1.1.1 To find out the version of Coppelia:

From Coppelia's command prompt, type

`version`

This will print the current version of Coppelia to the command window.

2.4.7.1.2 Viewing Log Files in Unix

In order to view log files, first find the location of them. Two helpful scripts you might want to incorporate for viewing and / or tailing FIX log files:

`fixtail`

This script will tail a log file, similar to "tail –f", converting the ASCII SOH delimiter into the pipe ("|") character. Create an executable shell script on your machine:

```
tail –f $1 |tr '\001' '|'
```

The syntax is **`fixtail logfile`**.

fixmore

This little script works like the UNIX command "more", and will simultaneously replace the (unprintable) ASCII SOH delimiter character in the FIX message and log files with the pipe ("|") character:

```
more $1 |tr '\001' '|' |more
```

The syntax is **fixmore logfile**.

2.4.7.2 Startup

(Section TBA)

2.4.7.3 Interface Connection

(Section TBA)

2.4.7.4 FIX Connection

2.4.7.4.1 Machine Crash at Your Site

If you crash, be courteous, and contact all counter parties. Provide them with an estimate as to when they can expect you to be up and running again. Involve your network group so that in case of router, firewall, or Telco problems action can be taken quickly.

2.4.7.4.2 Machine Turn-Around at Counter party's Site

It might be that other parties you connect to change machines (and therefore, IP addresses) when a primary machine crashes or becomes otherwise non-functional. Be prepared to change the IP address you are to connect to (this is done in the configuration file), unless you use aliasing in Coppelia.

2.4.7.5 Database

(Section TBA)

2.4.7.6 End of Day

(Section TBA)

2.5 Programmer's Guide

The Coppelia FIX engine handles the FIX session layer for the client. It manages logon protocol, sends out the required heartbeats, and persists messages automatically. Any other logic resides within the domain of the Application layer. It is the task of the *interface* to provide this functionality. The Interface is a connection or bridging between two systems through which information is exchanged.

The interface is a 'driver' that controls the application logic of Coppelia. It can direct Coppelia to establish or break FIX connections or send FIX messages. It is notified of events that happen within Coppelia such as incoming messages, and disconnects. Coppelia will put these events on a queue. It is the responsibility of the interface to drain this queue.

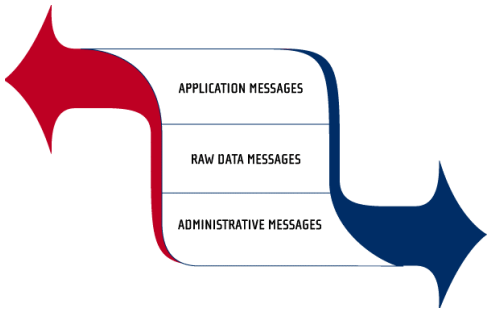
Although Coppelia is written in Java, there are many interface options:

- CORBA
- RMI
- Observer/Observable
- ActiveX
- TIB/Rendezvous
- MQ Series
- Talarian SmartSockets

These options derive from the more popular distributed computing technologies. Most of the interfaces into Coppelia differ greatly one from another reflecting the vastly different approaches each of the above technologies has to distributed computing.

2.5.0.1 Sending messages

Coppelia's architecture can send FIX messages of different types:



Application Messages: Are all allowed application messages as specified by the FIX protocol.

Raw Data Messages: RDMs send messages exclusively in raw data format (string) and not in the form of message objects. This format gives the user the freedom to package pr-existing FIX message strings and send them via Coppelia without having to create them from scratch.

Administrative Messages: As specified by the FIX protocol, these messages pertain to the FIX session rather than the data content transmitted.

Message types:

Application Messages	Raw Data Message	Administrative Messages
Advertisements	**String message in raw**	**Heartbeat**
Indications of Interest	**data format can contain**	**Logon Message**
News	**any FIX message**	**Test Request**
E-mail		**Resend Request**
Quote Request		**Reject**
Quote		**Resend Request**
New Order		**Reject**
Execution Report		**Sequence Reset**
Don't Know Trade		**Logout**
Order Cancel/Replace Trade		
Order Cancel Request		
Order Cancel Reject		
Order Status Request		
Allocation		
Allocations ACK		
New Order List		
List Status		
List Execute		
List Cancel Request		
List Status Request		

2.5.0.2 Receiving Messages

Coppelia is designed to send and receive ALL existing FIX messages. Further, Coppelia allows USER flexibility to specify his/her own custom message types. This mechanism is detailed in the section: "User Defined Messages."

2.5.0.3 Coppelia Data Types and Message Format

.

2.5.0.3.1 Description of Message Table Columns

Field Name	Coppelia Data Type	Required	FIX Tag #	Description and Comments

The actual FIX field name according to the FIX specs.

Field Name	Coppelia Data Type	Required	FIX Tag #	Description and Comments

The data type for a particular field in Coppelia: Below is a mapping of Java, C++, VB, and CORBA to Coppelia FIX 4.0:

Coppelia	int sequence_int	long sequence_long	float sequence_float	double sequence_double	string sequence_string
Java, C++,VB	int	long	float	double	string
CORBA	long	long long	long double	long double	string
	32 bits	64 bits	32 bits	64bits	unlimited

Field Name	Coppelia Data Type	Required	FIX Tag #	Description and Comments

Y (yes)	YD (yes dependent)
N (no)	ND (no dependent)

YD (Yes – dependent) indicates the field is normally required, but in certain cases the field does not need to be set (elaborated in the Description and Comments section). For example: in the Execution Report object, there are fields LastPx and LastShares, which are normally required, but in the case of a Status Execution Report, these fields need not be set.

ND (No – dependent) indicates that normally the field is not required, but in certain cases is required (elaborated in the Description and Comments section). For example, in the Order object, there is a field Price, which is only required when the order is of type Limit.

Field Name	Coppelia Data Type	Required	FIX Tag #	Description and Comments

(TBA)

Field Name	Coppelia Data Type	Required	FIX Tag #	Description and Comments

Further information about the field. Possible values, description of the field or values, and dependencies.

Note: Coppelia automatically generates values for many Header Objects and for all Trailer Objects. Fields that are automatically handled by Coppelia are shown in red <<<and are distinguished by the accompanying symbol XXX in the message table cell.>>>>

2.5.0.4 Coppelia Header object

Each FIX message, regardless of type, requires a header. The header of the message contains important information such as sequence number, target company ID, sender company ID, and sending time. The header fields are always the first fields in a FIX message.

Each message object in Coppelia has its own header object. Creation of a header in a Coppelia FIX message is very simple. If there is an Order object called `ord1`, to add the header information to that object, the programmer would use (in Java using standard CORBA)

```
ord1.header = new CHeader();
```

The header (and trailer) objects in Coppelia are unique in that all the required fields will be set by Coppelia, and require no user interaction. In all other user created objects, it is up to the user to make sure that the required fields are filled in. To fill in a non-required field in the header, a user would use code such as:

```
ord1.header.DeliverToCompID = "Javelin";
```

after the header itself has been initialized

2.5.0.4.1 Header Object – Table of Fields

Field Name	Coppelia Data Type	*Req'd*	*FIX Tag #*	*Description and Comments*
BeginString	**string**	**Y**	**8**	**Coppelia generated value. always "FIX.4.0"** **Identifies beginning of new message and protocol version. ALWAYS FIRST FIELD IN MESSAGE. Always unencrypted**
BodyLength	**long**	**Y**	**9**	**Coppelia generated value** **Message length, in bytes, forward to the CheckSum field. ALWAYS SECOND FIELD IN MESSAGE. (Always unencrypted)** **Valid values:** **0 – 9999**
MsgType	**string**	**Y**	**35**	**Message Type: Coppelia generated value** **Defines message type. ALWAYS THIRD FIELD IN MESSAGE. (Always unencrypted)** **Note: A "U" as the first character in the MsgType field indicates that the message format is privately defined between the sender and receiver.** **Valid values:** **0 = Heartbeat** **1 = Test Request** **2 = Resend Request** **3 = Reject** **4 = Sequence Reset** **5 = Logout** **6 = Indication of Interest** **7 = Advertisement** **8 = Execution Report** **9 = Order Cancel Reject** **A = Logon** **B = News** **C = Email** **D = Order - Single** **E = Order - List** **F = Order Cancel Request** **G = Order Cancel/Replace Request** **H = Order Status Request** **J = Allocation** **K = List Cancel Request** **L = List Execute** **M = List Status Request** **N = List Status** **P = Allocation ACK** **Q = Don't Know Trade (DK)** **R = Quote Request** **S = Quote**

Legend

Red Text	Automatically Generated by Coppelia
Bold Text	Required FIX field
	Conditionally Required
	Repeating
	Optional

SenderCompID	**string**	**Y**	**49**	**Coppelia generated value** **When calling a send function (i.e. sendOrder) this field is specified as an argument.** **Identifies the sender of the message. Always unencrypted**
TargetCompID	**string**	**Y**	**56**	**Coppelia generated value** **When calling a send function (i.e. sendOrder) this field is specified as an argument.** **Identifies the intended target of the message. Always unencrypted**
OnBehalfOfCompID	string	N	115	Assigned value used to identify firm originating message if the message was delivered by a third party i.e. the third party firm identifier would be delivered in the SenderCompID field and the firm originating the message in this field. Can be embedded within encrypted data section.
DeliverToCompID	string	N	128	Assigned value used to identify the firm targeted to receive the message if the message is delivered by a third party i.e. the third party firm identifier would be delivered in the TargetCompID field and the ultimate receiver firm ID in this field. Trading partner company ID used when sending messages via a third party Can be embedded within encrypted data section.
SecureDataLen	long	ND	90	Coppelia generated value – only when using Coppelia with PGP Length of encrypted message Required to identify length of encrypted section of message. (*Always unencrypted*)
SecureData	string	ND	91	Coppelia generated value – only when using Coppelia with PGP Required when message body is encrypted. Always immediately follows SecureDataLen field.
MsgSeqNum	**long**	**Y**	**34**	Unique integer message sequence number. Valid values: 0 - 999999 ***(Can be embedded within encrypted data section.)***
SenderSubID	string	N	50	Assigned value used to identify specific message originator (desk, trader, etc.) Useful for keeping uniqueness of messages that are received from a single SenderCompID *(Can be embedded within encrypted data section.)*

TargetSubID	string	N	57	Assigned value used to identify specific individual or unit intended to receive message. Useful for keeping uniqueness of messages that are sent to a single SenderCompID "ADMIN" reserved for administrative messages not intended for a specific user. *(Can be embedded within encrypted data section.)*
OnBehalfOfSubID	string	N	116	Assigned value used to identify specific message originator (desk, trader, etc.) if the message was delivered by a third party Trading partner SubID used when delivering messages via a third party. *(Can be embedded within encrypted data section.)*
DeliverToSubID	string	N	129	Assigned value used to identify specific message recipient (desk, trader, etc.) if the message is delivered by a third party Trading partner SubID used when delivering messages via a third party. *(Can be embedded within encrypted data section.)*
PossDupFlag	string	ND	43	Coppelia generated value (if the message is a Possible Duplicate) Indicates possible retransmission of message with this sequence number Valid values: Y = Possible duplicate N = Original transmission Always required for retransmissions, whether prompted by the sending system or as the result of a resend request. *(Can be embedded within encrypted data section.)*
PossResend	string	ND	97	Coppelia generated value (if the message is a Possible Resend) Indicates that message may contain information that has been sent under another sequence number. Required when message may be duplicate of another message sent under a different sequence number. *(Can be embedded within encrypted data section.)*
SendingTime	**string**	**Y**	**52**	Time of message transmission (always expressed in GMT) ***(Can be embedded within encrypted data section.)***
OrigSendingTime	string	ND	122	Original time of message transmission (always expressed in GMT) when transmitting orders as the result of a resend request. Required for message resends. If data is not available set to same value as SendingTime *(Can be embedded within encrypted data section.)*

2.5.0.5 Coppelia Trailer object

As with the Coppelia Header, each FIX message, regardless of type, requires a trailer. The Trailer is mainly used as a checksum for the FIX message, to ensure message validity.

Each message object in Coppelia has its own trailer object. If there is an Order object called ord1, to add the header information to that object, the programmer would use (in Java using standard CORBA)

```
ord1.trailer = new CTrailer();
```

All trailer objects are automatically generated and handled by Coppelia. Coppelia will fill in the only required field – CheckSum – by itself.

Signatures are very rare in FIX messages, and the user will normally not need to create them.

2.5.0.5.1 Trailer Object – Table of Fields

Field Name	Coppelia Data Type	Req'd	FIX Tag #	Description and Comments
SignatureLength	long	N	93	Coppelia generated value. Number of bytes in signature field. Required when trailer contains signature. *Note: Not to be included within SecureData field*
Signature	string	N	89	Coppelia generated value. Electronic signature *Note: Not to be included within SecureData field*
CheckSum	**string**	**Y**	**10**	Coppelia generated value. **(*Always unencrypted, always last field in message*)**

2.5.0.6 Special Data Types

2.5.0.6.1 Repeating fields

FIX allows for certain fields in FIX messages to repeat – a repeating field can be allowed to appear in a message more than once. For example, in the Allocation message, there is a field ClOrdID that is allowed to be repeated.

If a message is a repeating field, it is indicated by an (R) before the field name.

Repeating fields can also be dependent on another field. For example, the field ClOrdID field in the Allocations message – the number of times it repeats is dependant on the field NoOrders. If NoOrders is set to 5, the field ClOrdID will repeat 5 times.

In the field table, the dependent field will be italicized.

2.5.0.6.2 Sequences

Since FIX allows some fields to repeat, Coppelia uses the SEQUENCE_ data types as a special data type to allow easy creation of repeating fields. *Sequences must be used properly otherwise sending messages will not work properly*.

For example, in the Allocation object, there is the repeating field ClOrdID, and its data type is SEQUENCE_STRING, this means that it is an array of the STRING data type.

To create such a data type, the array must first be initialized and then each array element must be initialized independently.

```
/* First create the Allocation object with new CAllocation
*/

CAllocation alloc1 = new CAllocation();
```

As mentioned previously, the CAllocation object has a repeating field ClOrdID (Client Order ID.) Before any values in the field can be set, the array that will contain all the values must first be initialized. In Java, this is done by using 'new" and the data type of the sequence (in the case of the ClOrdID, it is a sequence of the String variable type)

```
/* Create an array that will contain 3 Client Order IDs,
but do not set any of the values of the array yet */

alloc1.ClOrdID = new String[3];


/* For reference, ClOrdID is a required field in the FIX
4.0 Allocation message. Its FIX tag number is 11 */

/* Now populate all three of the fields in the array. Be
sure to specify an array index when setting the field */

alloc1.ClOrdID[0] = "Order ID a";
alloc1.ClOrdID[1] = "Order ID b";
alloc1.ClOrdID[2] = "Order ID c";
```

Note: If a Coppelia Message Object has a sequence in it, it must be initialized, even if the field is not set, regardless of the FIX version! This can be determined by looking at the idl file for that particular message type – all fields that are of type sequence_ must be

intialized with a size of zero if not being used. If the field is not being used, it can be intialized to a size of zero without having that field appear in the FIX message.

For Example: in the Execution Report object, there is the field MiscFeeCurr that is of type SEQUENCE_STRING. If the programmer does not want this field to be set in the FIX message, it can be initialized this way:

```
/* Initialize MiscFeeCurr sequence to a size of zero */
exec1.MiscFeeCurr = new String[0].
```

As long as the sequence is initialized, the object can be sent properly.

2.5.0.7 Implementation

Languages:

C++:

C++ programmers interfacing to Coppelia have a choice of several different APIs depending on the platform.

On the Windows NT platform, several different flavors of CORBA are supported including Orbix, Visibroker and MICO; also Iona's COMET product can be used. We also have our own ActiveX API which can be used by C++ programmers.

On Solaris, we also support Orbix, Visibroker and MICO.

Sample programs have been set up so that they all have working Makefiles on Windows NT and Solaris.

Orbix Windows NT:

Building the sample programs on Windows NT using nmake

The sample programs are located in coppelia/demo/src. To build the examples using nmake:

Open a Command Prompt

- Run vcvars32.bat if necessary to set the Visual C++ environment variables
- cd to the directory where the package is installed
- cd to sample
- type "nmake /f Makefile.win32"

Visibroker Windows NT:

Building the sample programs on Windows NT using nmake

The sample programs are located in coppelia/demo/src. To build the examples using nmake:

- Open a Command Prompt
- Run vcvars32.bat if necessary to set the Visual C++ environment variables
- Make sure VBROKERDIR is set to the location of Visibroker
- cd to the directory where the package is installed
- cd to sample
- type "nmake /f Makefile.win32"

MICO:

MICO can be used as a no cost alternative to non-freeware CORBA products such as Orbix, Visibroker, for organizations that do not have existing CORBA installations and in-house knowledge of software development with CORBA.

Javelin Technologies ships the necessary MICO libraries, and C++ header files and libraries generated from the Coppelia CORBA IDL files, so that software developers can access Coppelia via C++.

Makefiles are provided to build the C++ header and source files from the IDL, and compile them into a library Coppelia.lib/Coppelia.dll on NT, libcoppelia.so on Solaris.

The MICO makefile has been modified to create a static library on Windows NT – the dynamic library created by default has conflicts with the Visual C++ libraries if included into a ATL or MFC application.

Solaris:

We provide a build of MICO for Solaris, which is built on Solaris 2.7. We also provide instructions on how to rebuild MICO on different versions of Solaris if required. We also provide sample makefiles and examples for Solaris.

The Package contains the libraries and includes necessary to build a C++ application. The MICO source is also included in the package as specified under the terms of the license - this is under the mico directory in the installed package - Javelin is currently using version 2.2.7.

The package contains makefiles and sample send and receive programs. There is a library libcoppelia.a and the corresponding includes CoppeliaServer.h and

UIRemote.h in the lib and include directory. These were built using GNU C++ 2.95.1 and GNU make version 3.77 on Solaris 2.7.

If you want to use the sample makefiles, you need to use GNU make – other versions of make will require changes to these makefiles as GNU Make is not compatible with other versions of make. If you want to use your own C++ compiler, you may have to build MICO with your C++ compiler since it may not support some of the C++ externsions, and consequently the header files generated by MICO may not work with your compiler.

Loading the MICO package:

The MICO package for Solaris is distributed as a compressed tar file JTMicoSolaris.tar.Z. To install this on your machine, uncompress the tar file and then un-tar the file using tar xvf JTMicoSolaris.tar.

The package contains the following directories:

- bin
- coppelia
- include
- lib
- mico

and a README and setvars.sh file.

The bin, lib and include directories contain the compiled MICO binaries, libraries and header files. The mico directory contains the MICO source code which Javelin Technologies, Inc. must distribute as part of the MICO license agreement.

The coppelia directory contains the Coppelia IDL, the compiled library and header files generated from the IDL and the sample code which is located under demo/src.

Building the sample programs on Solaris

The sample programs are located in coppelia/demo/src. To build the examples:

- cd to the directory where the package is installed
- run setvars.sh – in ksh type . setvars.sh
- cd to coppelia/demo
- type "configure"
- cd to src
- type "make"

Running the sample programs on Solaris

The receivemsg program expects a single parameter, which is the path of the CoppeliaIOR.str file for the Coppelia from which you want to receive.

The sendmsg program expects three parameters: the number of messages to be send, the target comp id and the path of the CoppeliaIOR.str file for the Coppelia which you want to send to.

Rebuilding MICO and the Coppelia libraries

You may need to rebuild MICO and the coppelia library on different versions of Solaris. You need to have GNU make installed in order to build these packages.

To build:

- go to the directory where you installed the package
- run "setvars.sh" script to set the PATH correctly
- go to the mico directory
- type "configure"
- type "make clean"
- type "make"
- go to the coppelia/idl directory
- type "configure"
- type "make clean"
- type "make" – this will build libcoppelia.a

Windows NT:

Loading the MICO package

The MICO package for Windows NT is distributed as a zip file JTMicoWINNT.zip. To install this on your machine, unzip the zip file.

The package contains the following directories:

- bin
- doc
- sample
- include
- lib
- dll
- src.

The bin, lib, dll and include directories contain the compiled MICO binaries, libraries and header files. The src directory contains the MICO source code which Javelin Technologies, Inc. must distribute as part of the MICO license agreement. The sample directory contains the sample code and nmake file, and sample Visual C++ 6 project.

Building the sample programs on Windows NT using nmake

The sample programs are located in coppelia/demo/src. To build the examples using nmake:

- Open a CommandPrompt
- Run vcvars32.bat if necessary to set the Visual C++ environment variables
- cd to the directory where the package is installed"
- cd to sample
- type "nmake /f Makefile.win32"

Building the sample programs on Windows NT using Visual C++ 6

The sample programs are located in coppelia/demo/src. To build the examples using nmake:

- Open a Visual C++ 6
- Open the project sendmsg.dsp or receivemsg.dsp
- Assuming you unzipped the package in D:\MICO,
- You must define _WINDOWS in the C++ pre-processor definitions - Project->Settings->Link
- Add the directories D:\MICO\include\windows and D:\MICO\include as the first two include directories in Tools->Options->Directories->Include Files.
- Add the directory D:\MICO\lib in Tools->Options->Directories->Library Files.
- You must also include mico230.lib and Coppelia.lib in Project Settings->Link->Object/library modules if not already present
- Replace the path in the fopen of CoppeliaIOR.str in sendmsg.cpp and receivemsg.cpp with the appropriate path on your machine.
- Build the project using - Build->Rebuild All

Running the sample programs on Windows NT

To execute the application, you need to make sure your path includes the directory D:\MICO\DLL assuming you installed in D:\MICO.

Visual Basic:

The Visual Basic example is included in the Coppelia ActiveX controls package.

Loading the ActiveX package

The ActiveX package for Coppelia is distributed as a zip tar file JTActiveX1.0.zip. To install this on your machine, unzip the file.

2.5.1 Common Object Request Broker Architecture (CORBA)

Common Object Request Broker Architecture (CORBA) is an architecture designed by the Object Management Group (OMG) to allow applications to communicate with each other no matter where they are located, what hardware platform is used, or what computer language the program was written in. The OMG is a non-profit organization formed in 1989 to create a standard object model. For more information on the OMG or CORBA, you can go to their web site http://www.omg.org/. The OMG's CORBA architecture is a based on an object based client-server model. CORBA 1.1 was introduced in 1991 defined the Interface Definition Language (IDL) and the Application Programming Interfaces (API) that enabled client/server object communications to a specific implementation of the Object Request Broker (ORB). In December of 1994, CORBA 2.0 was adopted. This version defines how ORBs from different vendors can interoperate.

Javelin Technologies, Inc. recommends that if you are programming for the CORBA interface of Coppelia, that your ORB adheres to CORBA 2.0 or higher. The reason is that CORBA 2.0 defined how different ORBs can communicate with each other, otherwise you will have to use the same ORB that Coppelia uses, which currently is OrbixWeb 3.1c from Iona Technologies, Inc.

Coppelia's CORBA implementation uses IDL stubs. Currently, we do not provide support for the Interface Repository service method of communicating with the ORB.

Javelin Technologies, Inc. compiles the IDL and ships the implementation code as well as the stub code together with Coppelia. Coppelia needs the implementation code to run. The included stub code will only be of use to you if you write your interface in Java using the same ORB as Coppelia.

If your interface will use a different ORB or will be written in a language other than Java, you will have to compile the IDL files that are provided with Coppelia yourself to generate stub code for your particular platform.

Coppelia also creates two files that contain the Interoperable Object Reference (IOR) string. The two files are CoppeliaIOR.str and RemoteUIIOR.str. The IOR string in CoppeliaIOR.str refers an object that allows you send messages, receive messages and execute commands. The IOR string in RemoteUIIOR.str refers to an object that allows you to execute commands, get statistical data about the connections, and get the uptime of Coppelia.

Note: The object reference by the IOR string in RemoteUIIOR.str is always available, even if the current interface setting is NOT set to CORBA.

2.5.1.1 IDL

Coppelia supplies the necessary IDL files for CORBA programs to communicate with Coppelia.

IMPORTANT: When writing any CORBA that communicates with Coppelia, IDL types that are sequences must be allocated even if the fields are not used.

2.5.1.2 CoppeliaServer Objects

The **CoppeliaServer** object is the workhorse of the CORBA interface. It contains methods for the sending of all FIX message-types, querying and retrieving information from Coppelia's interface queue, and general session layer control. Most CORBA interface programs will use the **CoppeliaServer** object.

The session layer methods that **CoppeliaServer** defines allow control over Coppelia's conenctions (*connectAll, connectId, disconnectAll, disconnectId, disconnectAllWithReason, disconnectIdWithReason)* , end-of-day (EOD) processing (*EID and EODId),* and interface message queue (*restore* and *restoreByMsgType*).

Coppelia spends the bulk of its time dealing with FIX messages. FIX messages, as they are sent from FIX engine to FIX engine, are Ascii plain text strings. Coppelia's CORBA interface, however, does not require the programmer to deal with FIX messages at this level. Every FIX message-type has an IDL object associated with it. Coppelia's convention is to call the object by the letter 'C' followed by the FIX message type name, hence COrder.idl and CIndicationOfInterest.idl. When an Order message is passed to the CORBA interface or when the CORBA interface wants to direct Coppelia to send an Order over a particular connection, a COrder object is used.

In general, for each FIX message field in a given message type, there is a corresponding data member in the Coppelia message object (e.g. COrder). A few fields warrant special mention.

All FIX messages have a required header and trailer. Rather than enumerate these data members repeatedly for each message type, the Coppelia IDL defines a CHeader and CTrailer object that is contained within a message object rather than the fields themselves.

Certain FIX messages define what is known as repeating fields. Repeating fields are used to express an array of values. FIX repeating fields always follow the same form. There is a field called NoXXX (meaning number of XXX) followed by one or more fields that repeat for the number of times based on the NoXXX field.

Other FIX messages have nested repeating fields (CAllocation object). This occurs when one of the repeating fields is itself a NoXXX-type object that has fields that repeat. The FIX protocol does not define a nesting greater than two layers.

Repeated fields are represented in the IDL objects as arrays. Nested repeating fields are represented as two-dimensional arrays, where the major dimension parallels the outer repeating group.

As stated earlier, the interface queue is the queue of those messages that have come into Coppelia. At the present time, CORBA interface programs are not notified when events come into this queue; they must proactively query Coppelia as to its contents. This querying is done with the various peek… methods defined by the **CoppeliaServer** object. The peek… methods return a CPeekType providing information as to the contents of the queue, specifically its current size and the oldest message on it. Retrieving an object requires another method call of the form getNext… Typically the interface program will "peek" determine the size of the queue, drain its contents., sleep for a short time, and start the process over again.

2.5.1.2.1 CHeader

The Message type for Coppelia is different than in FIX due to the fact that Coppelia represents all of the message objects with numeric values. FIX message types go from 1-9, and then start with the letter A. However, to ensure numeric values, for message types A and later, we use the following process to determine the value:

```
9 = 9
10 = A
11 = B
12 = C
13 = D
```

For example:

Message Type of Order (in FIX)	D
Message Type of Order (in Coppelia)	13

This is why the number 13 is used by Coppelia to identify order messages. However, the MessageType field in the FIX message will still reflect the actual FIX message type!

2.5.1.2.2 Administrative Commands

There is a set of functions that permits the USER to connect and disconnect Coppelia from or to targets or other FIX servers.

Note: Connectivity both on the network level and the FIX level is required between two servers in order to send and receive application messages.

These functions include:

`connectAll()`
connects all remote target FIX servers

`connectId()`
connects a specific server

`disconnectAll()`
disconnects all servers

`disconnectId()`
disconnects a specific target

`EOD()`
run End-Of-Day process for all targets

`EOD_Id()`
run End-Of-Day process for specified ID.

2.5.1.2.3 Message and Queue Operations

Coppelia is built with the following array of functions that permit manipulation of messages and allows a user to check what is on the Coppelia queue:

```
peek()
peekAll()
peekAllByMsgType()
peekByMsgType()
peekAllBySubId()
peekbySubId()
```

These functions are used by Coppelia to check the message queue. If there are messages in the queue, the user will be able to remove them from the queue and provide them to the application in use. If there are no messages on the queue, the queue_size element of the returned CPeekType variable from the function call would be set to zero. Also the

next_message_type element will be set to CoppeliaServer.EMPTYQUEUE or in numeric form it would -104.

`removeNext()`

This function will purge the top message from the queue.

`restore()`

This function will return to the queue messages that have already been removed from the queue by any of the getNext… methods or by the removeNext() function.

2.5.1.2.4 Application Messages

set… functions TBA

getNext… functions TBA

2.5.1.2.5 Return Codes

Nearly all of the Coppelia functions have a return code that specifies the result of the function call – success, failure, and other results. The return codes all have both a name and a numeric value.

The functions that return numeric values are:

All sendMessage functions (including sendOrder, sendEmail, sendAllocation)
All Connection functions (including connectAll, connectID, disconnectAll)
Exit
End Of Day functions (EOD, EODId)
Restore messages (restore, restoreByMsgType)
getNextRejectSent

The actual values and names are determined by entries in the CoppeliaServer.idl file, and are reproduced here with further detail.

2.5.1.2.5.1 Table of Return Codes

This table details the most common Coppelia numeric return codes. It gives information about the return code name, the numeric value of the return code, the exact data type for the return code, as well a detailed description of the Return Code.

Return Code Name	***Return Code Numeric Value***	***Coppelia Data Type***	Description and Comments
OK	0	int	Function executed successfully. Coppelia encountered no errors when calling function
INVALIDDATA	-100	int	User attempted to send data through Coppelia that was incorrect. Could indicate that required fields were missing from the message. Could also indicate that message was sent to an invalid TargetCompID. Also make sure that the message Header and Trailer have been properly initialized.
NOTAVAILABLE	-101	int	This message is not available from this Coppelia. For example, Coppelia does not allow the sending of Quote messages from a BUY side, and also does not allow sending of OrderCancelRejects from a BUY side. Please check the .dat file and make necessary modifications.
EODRUNNING	-111	int	Coppelia is in the process of running End Of Day on a connection, therefore it is unable to send any messages at this time. Coppelia must finish the End Of Day function before it can send more messages
REMOTEDOWN	-103	int	The remote connection that the user is attempting to send messages to is currently not connected.
REMOTELOGGING_ON	-106	int	Coppelia is in the process of logging on a remote connection and cannot send messages at this time.

2.5.1.2.6 Sample Java Programs

2.5.1.2.6.1 Sample Java Code for Creating a Coppelia Order Message

The following is sample Java code used to create and send an Order using Coppelia. This example includes all of the required FIX 4.0 fields.

```
/*

  Example program for processing sending an order message
using the
  CORBA interface to Coppelia.

*/

import java.util.*;
import java.io.*;
import org.omg.CORBA.ORB;
import org.omg.CORBA.SystemException;
import com.javtech.coppelia.*;

public class ExampleOrder
{
  private static CoppeliaServer remote_obj = null;

  public static void main(String args[])
  {
    try
    {
      /*
         read in the Interoperable Object Reference (IOR)
string
         that Coppelia creates
      */

      FileReader fr = new FileReader("CoppeliaIOR.str");
      BufferedReader br = new BufferedReader(fr);

      String newIOR =  br.readLine();

      /** create a remote object reference **/

      ORB orb = ORB.init (new String [] { "Coppelia" },
null);
```

```
      org.omg.CORBA.Object o =
orb.string_to_object(newIOR);

      remote_obj = CoppeliaServerHelper.narrow(o);
    }
    catch (SystemException ex)
    {
      System.out.println("Exception during bind");
      System.out.println(ex.toString());
      ex.printStackTrace();
      System.exit(1);
    }
    catch (IOException ioe)
    {
      System.out.println("IOException:" + ioe);
    }

    /*
    **** REPLACE with the correct TargetCompID that you
plan to use ****
    */
    String TargetCompID = "SBI";

    /* Create the new order object */
    COrder ord = new COrder();

    /* Set the fields in the order message */
    ord.ClOrdID = "1001";     //FIX tag #11
    ord.HandlInst = "1";      //FIX tag #21 - ** 1=best
execution
    ord.Symbol = "AAPL";      //FIX tag #55 - ** ticker
    ord.Side = "1";           //FIX tag #54 - ** 1=BUY,
2=SELL
    ord.OrderQty = 1000; //FIX tag #38
    ord.OrdType = "1";        //FIX tag #40 ** 1=MKT, 2=LMT


    /* Initialize the Header and Trailer for the Order
object */
    ord.header = new CHeader();
    ord.trailer = new CTrailer();

    /* Call the sendOrder function to send the Order
object. */
    int rc = remote_obj.sendOrder("JOE", TargetCompID,
"STAN", ord);
```

```
    switch(rc)
    {
      case CoppeliaServer.OK:
        System.out.println("Order Sent.");
        break;
      case CoppeliaServer.INVALIDDATA:
        System.out.println("Invalid Data.");
        break;
      case CoppeliaServer.NOTAVAILABLE:
        System.out.println("This message type is not
available to you.");
        break;
      case CoppeliaServer.EODRUNNING:
        System.out.println("EOD Running, messages cannot be
sent at this time.");
        break;
      case CoppeliaServer.REMOTEDOWN:
        System.out.println("The connection is down.");
        break;
      case CoppeliaServer.REMOTELOGGING_ON:
        System.out.println("Remote logging running,
messages cannot be sent at this time.");
        break;
      default:
        System.out.println("Unknown return code =" + rc);
        break;
    }
  }
}
```

2.5.1.2.6.2 – Sample Java code for processing received messages.

```
/*

  Example program for processing received messages using
the
  CORBA interface to Coppelia.

*/

import java.util.*;
import java.io.*;
import org.omg.CORBA.ORB;
import org.omg.CORBA.SystemException;
import com.javtech.coppelia.*;

public class ExampleReceive
{
  private static CoppeliaServer remote_obj = null;

  public static void main(String args[])
  {
    try
    {
      /*
         read in the Interoperable Object Reference (IOR)
string
         that Coppelia creates
      */

      FileReader fr = new FileReader("CoppeliaIOR.str");
      BufferedReader br = new BufferedReader(fr);

      String newIOR =  br.readLine();

      /** create a remote object reference **/

      ORB orb = ORB.init (new String [] { "Coppelia" },
null);

      org.omg.CORBA.Object o =
orb.string_to_object(newIOR);

      remote_obj = CoppeliaServerHelper.narrow(o);
    }
```

```
    catch (SystemException ex)
    {
      System.out.println("Exception during bind");
      System.out.println(ex.toString());
      ex.printStackTrace();
      System.exit(1);
    }
    catch (IOException ioe)
    {
      System.out.println("IOException:" + ioe);
    }

    try
    {
      for (;;)
      {
        try
        {
          Thread.sleep (1000);
        }
        catch (InterruptedException ie)
        {
        }

        int rc;
        CPeekType prc;

        prc = remote_obj.peekAll();

        int messages_left = 0;

        for (int queue_size = 0; queue_size <
prc.queue_size; queue_size++)
        {
          messages_left = prc.queue_size - queue_size;

          System.out.println("There are " + messages_left +
                             " messages left in the
queue");

          /*** process the message if there is one
available ***/

          switch(prc.next_message_type)
          {
            case CoppeliaServer.EXECUTIONREPORT:
```

```
              CExecutionReport e =
remote_obj.getNextExecutionReport("", prc.target_id);
              System.out.println("Execution Report received
from " + prc.target_id);
              break;
            case CoppeliaServer.QUOTE:
              CQuote q = remote_obj.getNextQuote ("",
prc.target_id);
              System.out.println ("Got quote and it's BidPx
is: " + q.BidPx);
              break;
            case CoppeliaServer.EMPTYQUEUE:
              System.out.println("The queue is empty.");
              break;
            default:
              System.out.println("Unhandled message type ="
+ prc.next_message_type);
              remote_obj.removeNext("", "");
              break;
          }
        } /** end of while queue has data **/
      }
    }
    catch (SystemException se)
    {
      System.out.println (se);
    }
  }
}
```

2.5.1.3 UIRemote Objects

The **UIRemote** object provides three methods: *doCommand*, *getRemoteData*, and *getSystemUpTime*.

The *doCommand* method gives programmatic access to the Coppelia command line interpreter (interface?), allowing the remote CORBA client to remotely "type" a command to Coppelia (see § 3.6). This only would be useful if your CORBA interface program would attempt to simulate the Coppelia command line interpreter (interface?). Its use is not recommended as it is difficult to discern the success or failure of issued commands.

getRemoteData is a useful method that returns an array of CSRemoteData objects, one for each connection described in Coppelia's configuration file. The CSRemoteData object contains a dump of all information currently available about a given connection.

getSystemTime returns the number of seconds that Coppelia has been running.

2.5.2 Java Observer / Observable

2.5.2.1 Introduction

The Coppelia server supports multiple interfaces. Observer / Observable is a native Java interface into the Coppelia server. This interface is only available in the java platform, and the document describes the procedures necessary to use this feature.

The Observer / Observable interface allows a client application to interact with the Coppelia process directly, in other words, in the same process space. This interface is only available in the Java platform, and it is actually using the JDK's Observer / Observable interface. For more details on this, please refer to your JDK's documentation.

Refer to the javadoc type documentation for interfaces of CoppeliaSrv and CoppeliaSrvFactory, which are the methods that the client application can invoke. Typically, the client application will obtain a handle to CoppeliaSrv via Factory, regardless of the interface.

2.5.2.2 .dat File Entry

To use this interface, you need the following entry in the your .dat file:

```
INTERFACE    OBSERVER
```

2.5.2.3 Methods to Use

There are a number of methods that a client application will would use in order to work with Coppelia's Observer / Observable interface:

Application Related

- `CoppeliaSrvFactory.getInProcObject()` to start the Coppelia process or obtain a handle to it subsequently.
- `addListener()` of CoppeliaSrv to register a callback for application messages and `addMonitor` for session connectivity and statistical information and notification.
- `post()` of CoppeliaSrv for outgoing messages.

Operations Related

- `Command()` of CoppeliaSrv
- `getOperatorData()` of CoppeliaSrv

Because the client application 'lives' in the same process space as the Coppelia process, the Coppelia process is actually started by the client application. This is usually the first thing to do in the client application. This is done through the use of the factory method:

```
GetInProcObject(String[]args)
```

Example:

```
CoppeliaSrv srv = CoppeliaSrvFactory.getInProcObject(args);
```

The 'args' here is actually the configuration file, for example buy.dat. The factory method will only create one instance of Coppelia, and therefore it is Singleton. Your client application could invoke this method the second time by providing an empty String[] to obtain the handle to CoppeliaSrv.

Because we are talking Observer/Observable, you need to develop a listener that will handle the incoming messages, and register that listener. For example,

```
AppObserver l = new AppObserver();
try {
    srv.addListener(l);
} catch(Exception e) {}
```

Once the listener is registered, and when there is an incoming FIX message, the listener will be notified with an object as the argument to update(Observable o, Object arg) in the implementation of the Observer. The object is of MessageObject type which is the FIX Application object. For FIX application objects, please refer the corresponding IDL files.

The `get()` method of CoppeliaSrv interface for Observer/Observable interface is not currently supported.

2.5.2.4 Session Connectivity

There is an addMonitor() method that allows a client application to register a listener for monitoring details about a FIX session. This listener will be notified when a session is down. The object passed in to this listener is of type OperatorData. For detail of the data structure of OperatorData, please refer to CSRemoteData.idl because it has the identical structure.

In order to use this feature, please add a flag in the .dat file:

```
REMOTE_DOWN_NOTIFICATION      ON
```

The connect_state in OperatorData is one of the following:

```
public static final int DISCONNECTED = 0;
public static final int SESSION_CONNECTION = 1;
public static final int APPLICATION_CONNECTION = 2;
public static final int LOGGING_ON_CONNECTION = 3;
```

Please refer to the examples for usage.

2.5.2.5 Outgoing Messages

For outgoing messages, the post() method of CoppeliaSrv is used. Since this is in the same process of Coppelia, and because Java uses references, you need to create a new application object everytime you post, which is the common practice, that is, no one Order is the same. This is only note for your information.

2.5.2.6 Operator's API

`operatorCommand()`

This method allows a client application to perform administrative operations such as connect, disconnect, exit, eod, etc. This works basically the same way as if you would type these commands at the command prompt of Coppelia.

`getOperatorData()`

This operator method allows a client application to query the statistics of a certain connection. The method returns an array of OperatorData. The structure of OperatorData is the same as described is CSRemoteData.idl. Please refer to this file for more details.

For detail of usage, please refer to the ObserverExample.

2.5.2.7 Further Process

To experiment with the example, do the following:

Compile the ObserverExample.java file as follows:

```
javac -classpath $CLASSPATH ObserverExample.java
```

where `$CLASSPATH` should include the installation of the coppelia.jar.

To run, use:

```
java -classpath $CLASSPATH ObserverExample buy.dat
```

Make sure you have added the `INTERFACE OBSERVER` line into your .dat file.

2.5.3 Java Remote Method Invocaton (RMI)

2.5.3.1 Introduction

This interface allows a remote client application to access the Coppelia engine via the Java Remote Method Invocation (RMI) API. RMI is a client/server model to allow Java programs to run methods running on another Java Virtual Machine (JVM). Currently, RMI has a few shortcomings, e.g. no good implementation to access the methods through a firewall. Programming in RMI is very similar to using Java Observable/Observable, except for the fact that the RMI has to get the object remotely using RMI Registry.

The client interface follows the CoppeliaSrv interface model. Please refer to the JavaDoc document CoppeliaSrv.html for a general description of the CoppeliaSrv interface. This document assumes that the reader is comfortable developing applications using the Java RMI API.

2.5.3.2 Configuration File Entries

To use the Coppelia Java RMI interface, the following lines are required in the .dat file:

```
INTERFACE       RMI
RMI_HOST        [hostname]      (default: localhost)
RMI_PORT        [port]          (default: 1099)
```

The RMI_HOST and RMI_PORT parameters are used to specify the URI the CoppeliaSrv remote object will bind to. The RMI_HOST parameter takes a string value that may be either a hostname or IP address. The default value is 'localhost'. The RMI_PORT parameter takes an integer value to specify the port the Coppelia RMI interface will listen on. The default port is 1099.

The configuration file may also contain the following optional information:

```
CALLBACKS                   [ON/OFF]    (default: OFF)
RAW                         [ON/OFF]    (default: OFF)
SESSION_NOTIFICATION        [ON/OFF]    (default: OFF)
RMI_ENCRYPTION              [method]    (default: none)
```

These options are described in sections 4.4 – Receiving Messages from Coppelia: Callbacks, and 5.0 – Using CoppeliaRMI with SSL Encryption.

2.5.3.3 Starting CoppeliaRMI

To start the Coppelia server using the Coppelia RMI interface there are two properties that need to be set:

- Java Security Policy (the location of a file describing the access rights for the server (grant all in our example)).
- Java RMI Server Codebase (the location of the stubs for object marshalling between client and server).

Example:

```
% java –Djava.security.polcy=d:/Coppelia/Rmi/policy.txt
–
Djava.rmi.server.codebase=file:///d:/Coppelia/classes/coppe
lia.jar Coppelia rmi_buy.dat
```

In the above example, the security policy is located in d:/Coppelia/Rmi/Policy.txt and the codebase is located in the local file system at file:///d:/Coppelia/classes/coppelia.jar. Note that the codebase must be specified as a URL and that it must point to the same coppelia.jar that is specified in the classpath.

2.5.3.3.1 RMI Registry

Since Java RMI requires an RMI Registry to provide a lookup for remote objects, CoppeliaRMI automatically creates and initializes at startup an instance of the RMI Registry that listens at the port specified by the RMI_PORT configuration parameter. There is no need to start an RMI Registry before starting CoppeliaRMI.

2.5.3.4 Using the CoppeliaSrv Client API with CoppeliaRMI

The CoppeliaRMI client interface follows the CoppeliaSrv model for communicating with Coppelia. The CoppeliaSrv description is an internal JavaDoc reference. The CoppeliaSrv.HTML document can be requested at support@javtech.com.

2.5.3.4.1 Initializing the Client

The very first thing that a client is required to do is to obtain a handle for a remote CoppeliaSrv object. The following methods are available:

```
public CoppeliaSrv CoppeliaSrvFactory.getRmiObject(host)
public CoppeliaSrv CoppeliaSrvFactory.getRmiObject(host,
port)
```

The CoppeliaSrvFactory methods take care of forming the URI and looking up the remote object. The methods throw any exceptions that are caught in this process.

2.5.3.4.2 – Sending Messages to Coppelia

The CoppeliaSrv remote object exposes several methods or functions for sending messages to Coppelia. These messages are divided into two categories: Application Level Messages and Administrative Messages.

For Application Level Messages, the following methods are available:

```
public String post(MessageObject message)
public String postRawString(String rawFIX)
```

post()

This method is used to send a FIX message object (Order, ExecutionReport, etc.) to Coppelia. It returns a string which is used to report error messages or reject messages from Coppelia.

postRawString()

This method is used to send a raw FIX string to Coppelia. It returns a string which is used to report error messages or reject messages from Coppelia.

For Administrative Messages, the following methods are available:

public String operatorCommand(String command)
public OperatorData[] getOperatorData()

operatorCommand()

This method is used to send command strings to Coppelia.

getOperatorData()

This method is used to query the status of Coppelia's FIX connections. getOperatorData() returns an array of OperatorData objects. The OperatorData object structure is discussed in detail in another section of the Coppelia Guide.

2.5.3.4.3 Receiving Messages from Coppelia: Polling

The original CoppeliaRMI interface was designed to exclusively support receiving messages on a polling basis. It is now recommended that clients use the Callback model as the industry standard. However, if the Polling model is desired, the following methods are available:

```
public boolean available()
public MessageObject get()
public String getRawString()
```

available()

This method is called to query the remote CoppeliaSrv object to see if any incoming application messages are on the queue.

get()

This method removes a MessageObject representing an incoming application message from the queue and returns it to the client.

getRawString()

This method removes a String representing an incoming application message from the queue and returns it to the client.

2.5.3.4.4 Receiving Messages from Coppelia: Callbacks

To enable the CoppeliaRMI Callback model, the following line is required in the configuration file:

```
CALLBACKS ON
```

To configure Coppelia to deliver raw FIX strings instead of MessageObjects to the client, the following line must appear in the configuration file:

```
RAW        ON
```

```
public abstract class CoppeliaRMIListener {
  public int update(Object data) throws RemoteException {
  }
}
```

The update() method is overridden to handle incoming messages from Coppelia. These will either be MessageObjects, Strings (for `RAW ON`), or OperatorData (see 4.5

Notification of Changes in Connection Status) objects. The update() method is invoked by the remote object when Coppelia receives an incoming FIX message.

After creating a listener, the client must add it to the CoppeliaSrv object:

```
CoppeliaSrv srv =
CoppeliaSrvFactory.getRmiObject("localhost", 1099);
MyListener listener = new MyListener();
srv.addListener(listener);
```

Please refer to the RMIListen.java in the Examples section for a more elaborate example.

In the RMI interface, when Coppelia receives a FIX message from the counter party FIX engine, it will try to publish the message to the RMI callback listener. It there's no callback or listener associated at that moment, Coppelia will store it in a memory queue(Please note that all these messages are already persisted in the database). In the case of recovering messages, the client application would recover them by first registering the callback, invoking the available() method to see if there is anything to restore from the memory queue, and then invoke the restore2Callback() method. restore2Callback would try to publish the message again (and in the case it fails again for whatever reason, it will be stored in the memory queue again).

Two more important considerations when using callbacks in RMI are:

- More than one application can register for callbacks in RMI. In fact, the timer call happens every 10 seconds, not every minute.
- So the maximum time the engine can be out of sync with the system time is 10 seconds. This is not configurable.

2.5.3.4.5 Notification of Changes in Connection Status

To enable the CoppeliaRMI Session Notification, the following line is required in the configuration file:

```
SESSION_NOTIFICATION     ON
```

When a FIX session is established or terminated with one of Coppelia's targets, the client can be notified, if it has set up a Monitor. Like Listeners, Monitors will extend the CoppeliaRmiListener class.

After creating a listener, the client must add it to the CoppeliaSrv remote object:

```
MySessionMonitor monitor = new MySessionMonitor();
srv.addListener(monitor);
```

When a connection changes state, the Monitor's update() method will be invoked with an OperatorData object representing the particular connection. Please refer to the SimpleClientRMI.java in the Examples section for a demonstration of using Coppelia RMI Session Notification.

In addition, there is a method called isListenerAlive(). This method serves two purposes. First, it checks if a listener is still alive (thus providing a way to find out if your client application callback or listener is still healthy). When Coppelia fails to publish a message to a callback, there are normally two scenarios: 1) No callback is registered, or 2) Some type of exception occurs in the client's callback. If an exception occurs when Coppelia invokes the callback, Coppelia will remove that callback from the callback registry. Normally, the client application would run another backgroud thread that checks the health of the callback. If a broken callback is detected on the client side, the recovery should be performed as mentioned above.

The second purpose of the isListenerAlive() method is to detect the livelihood of a Coppelia server. If, when invoking the method, an exception is thrown, it means that the Coppelia RMI server is no longer alive. This normally indicates that the Coppelia server is no longer alive. This is a convenient mechanism one can use to both check the health of the client callback, and server availability in a callback environment.

2.5.3.5 Using CoppeliaRMI with SSL Encryption

The Coppelia RMI interface can use a custom socket factory to provide SSL encryption. The interface currentlly requires the Phaos SSLava library from Phaos Technology. To use the Coppelia RMI interface with SSL encryption, the following lines are required in the .dat file:

```
RMI_ENCRYPTION SSL
SSL_server_cert     [server cert]
SSL_ca_cert         [ca cert]
```

The latter two options tell Coppelia which certificates to use for the SSL encryption. The `SSL_server_cert` parameter specifies the location of a server certificate and the `SSL_ca_cert` parameter specifies the location of a certificate from a Certification Authority.

To initialize the Coppelia RMI Interface with SSL encryption, the following method is used:

```
public CoppeliaSrv CoppeliaSrvFactory.getRmiSSLObject(host,
port)
```

The `getRmiSSLObject()` method behaves the same as the two getRmiObject() methods in section 4.1 with the exception that the default RMI socket factories are substituted with custom SSL socket factories.

To start the Coppelia server using the Coppelia RMI interface with SSL encryption, the Phaos SSLava library must be included in the classpath.

2.5.3.6 Examples

2.5.3.6.1 Example: Sending Messages to Coppelia – SendOrderRMI.java

This example shows how to send Administrative and Application Level messages to Coppelia using the CoppeliaRMI Client API.

```
public class SendOrderRMI {

  public SendOrderRMI(String host, int port) {
    CoppeliaSrv srv = null;
    boolean connect = false;
    while (!connect) {
      try {
        srv = CoppeliaSrvFactory.getRmiObject(host, port);
        connect = true;
      }
      catch(Exception e) {
        System.out.println("connect " + host + ":" + port + "
unsuccessful.");
        try { Thread.sleep(500); } catch(Exception ie) {}
      }
    }
  }

  public void connect(String target) {
    try {
      srv.operatorCommand(“connect “ + target);
    }
    catch(Exception e) {
      System.out.println(“unable to connect to “ + target + “: “ +
                         e.getMessage());
    }
  }

  public void sendOrder(String target) {
    Order order = new Order();
    order.header.TargetCompID = target;
    order.header.SenderCompID = "SLGM";
    order.ClOrdID = "j";
    order.HandlInst = "1";
    order.Symbol = "IBM";
    order.Side = "1";
    order.OrdType = "1";
    try {
      srv.post(order);
    }
   catch(Exception e) {
      System.out.println(“unable to send order to “ + target + “: “ +
                         e.getMessage());
   }
```

```
  }

  public static void main(String[] args) {
    if (args.length != 2) {
      System.out.println("Usage: java SendOrderRMI <host> <port>");
      System.exit(1);
    }
    String host = args[0];
    int port = Integer.parseInt(args[1]);
    SendOrderRMI rmi = new SendOrderRMI(host, port);
    rmi.connect(“SBI”);
    rmi.sendOrder(“SBI”);
  }
}
```

2.5.3.6.2 Example: Receiving Messages from Coppelia – RMIListen.java

The following example demonstrates using Callbacks and Session Notification to receive messages from Coppelia. It requires three command line arguments, a hostname, a port number, and a TraderID to filter incoming messages.

```
import com.javtech.coppelia.*;
import com.javtech.coppelia.interfaces.*;
import com.javtech.coppelia.interfaces.rmi.*;

class RMIListener extends CoppeliaRmiListener {

  public int update(Object obj) {
    if (obj instanceof MessageObject)
      return handleMessageObject((MessageObject)obj);
    else if (obj instanceof OperatorData)
      return handleOperatorData((OperatorData)obj);
  }

  public int handleMessageObject(MessageObject message) {
    // Extract the MsgType string from the MessageObject
    String msgType = message.header.MsgType;

    // Cast the MessageObject to the appropriate type and print its
    // corresponding ID string
    if (msgType.equals("8")) {
      ExecutionReport exec = (ExecutionReport)message;
      System.out.println("Received an ExecutionReport, ExecID = " +
                         exec.ExecID);
    }
    else if (msgType.equals("6")) {
      IndicationOfInterest ioi = (IndicationOfInterest)message;
      System.out.println("Received an IndicationOfInterest, IOIID = " +
                         ioi.IOIID);
    }
    else {
      System.out.println("Received an unhandled MessageObject, MsgType
= " +
```

```
                          msgType);
    }
    return 0;
  }

  public OperatorData handleOperatorData(OperatorData odata) {
    // Print the type of connection change and the target's ID
    if (odata.connect_state != ConnectState.DISCONNECT)
      System.out.println("Target " + odata.firm_id + " connected.");
    else
      System.out.println("Target " + odata.firm_id + " disconnected.");

    return 0;
  }

}

public class RMIListen {

  private CoppeliaSrv srv_;

  public RMIListen(String host, int port, String targetSubID) {
    // Get the RMI implementation of CoppeliaSrv via the
CoppeliaSrvFactory
    try {
      srv_ = CoppeliaSrvFactory.getRmiObject(host, port);
    }
    catch(Exception e) {
      System.err.println("Could not get CoppeliaSrv Object: " +
                         e.getMessage());
      System.exit(1);
    }

    // Create a listener object
    RMIListener l = new RMIListener();

    // Add a monitor and a listener based on the targetSubID.
    // If it is "all", receive all incoming messages.
    try {
      if (targetSubID.equals("all"))
        srv_.addListener(l);
      else
        srv_.addListener(l, targetSubID);
      System.out.println("Added CoppeliaRmiListener on subject " +
                         TargetSubID + "as a listener");
      srv_.addMonitor(l);
      System.out.println("Added CoppeliaRmiListener as a monitor");
    }
    catch(Exception e) {
      System.err.println("Could not add listener to CoppeliaSrv Object:
" +
                         e.getMessage());
      System.exit(1);
    }

  }
```

```
  public static void main(String[] args) {
    if (args.length != 3) {
      System.err.println("Usage: java RMIListen <host> <port>
<targetSubID>");
      System.exit(1);
    }

    // Extract parameters from args string
    String host = args[0];
    int port = Integer.parseInt(args[1]);
    String targetSubID = args[2];

    // Create a RMIListen object
    RMIListen l = new RMIListen(host, port, targetSubID);
  }

}
```

2.5.4 TIBCO TIB/Rendezvous

2.5.4.1 Introduction

This interface allows a remote client application to access the Coppelia engine via TIBCO Rendezvous. This section of the document assumes that the reader is comfortable developing applications using TIBCO TIB/Rendezvous.

IT IS HIGHLY RECOMMENDED THAT RV VERSION 5.0 OR HIGHER BE USED.

2.5.4.2 Configuration File Entries

To use this interface, the following lines are required in the .dat file:

```
1. INTERFACE         TIBCORV
2. TIBCO_SERVER_ID   [serverID]
3. TIBCO_API         [RV5X | RV5XCM]      (default: RV5X)
```

The `TIBCO_SERVER_ID` parameter takes a string value to specify the subject Server ID used by Coppelia for communication with client applications. The `TIBCO_API` parameter takes a string value specifying the RV API to use. This parameter allows you to specify that Coppelia is to use RV Certified Messaging.

The .dat file may also contain the following pertinent information:

```
1. TIBCO_CMLEDGERFILE    [filename]      (default: none)
2. TIBCO_SERVICE         [service]
3. TIBCO_NETWORK         [network]
4. TIBCO_DAEMON          [daemon]
5. RAW                   [ON/OFF]        (default: OFF)
6. FULL_OBJECT           [ON/OFF]        (default: OFF)
7. SESSION_NOTIFICATION  [ON/OFF]        (default: OFF)
```

The `TIBCO_CMLEDGERFILE` parameter takes a string value to specify the location of Coppelia's ledgerfile if RV Certified Messaging is enabled. By default, Coppelia will use an in-memory ledgerfile.

The `TIBCO_SERVICE, TIBCO_NETWORK,` and `TIBCO_DAEMON` parameters take string values to override the default RV service, network, and daemon for Coppelia.

The other options are described later in this document in section 2.5.4.5.3 – Receiving Messages from Coppelia

2.5.4.3 Starting Coppelia with TIBCO Rendezvous

To start the Coppelia server using the TIBCO Rendezvous interface the classpath needs to be set to include the RV libraries:

Example go_buy.sh:

```
#!/bin/sh
LIBDIR=/usr/local/Coppelia/classes
CLASSPATH=$LIBDIR/coppelia.jar:$LIBDIR/mct3_0.zip:$LIBDIR/O
rbixWeb31c.jar:$LIBDIR/pro.jar:$LIBDIR/rogue.zip:$LIBDIR/rv
jpro.jar
java Coppelia tibco_buy.dat
```

2.5.4.4 Detailed Description of Interface Implementation

2.5.4.4.1 Subject Namespace

Below is a description of the subjects used by Coppelia to interact with client applications via TIBCO Rendezvous. The [`ServerID`] field corresponds to the `TIBCO_SERVER_ID` value set in the config file.

Coppelia Server generated events:

1. Incoming FIX message received.

```
Coppelia.[ServerID].IN.[SenderCompID].[MsgType]
```

2. Outgoing FIX message sent acknowledgement.

```
_Coppelia.[ServerID].ACK.[TargetCompID].[MsgType]
```

3. Session Up.Down Notification.

```
_Coppelia.[ServerID].SESSION.NOTIFICATION.[TargetCompID]
```

Client Application generated events:

1. Outgoing FIX message sent.

```
Coppelia.[ServerID].OUT.[TargetCompID].[MsgType]
```

2. FIX session list requested.

```
_Coppelia.[ServerID].REQ.SESSION.LIST
```

3. FIX session's information requested.

```
_Coppelia.[ServerID].REQ.SESSION.INFO
```

4. Operator command issued.

```
_Coppelia.[ServerID].REQ.COMMAND
```

Coppelia subscribes to the following subjects:

```
Coppelia.[ServerID].OUT.>
_Coppelia.[ServerID].REQ.>
```

Client applications will generally subscribe to the following subjects:

```
Coppelia.[ServerID].IN.>
_Coppelia.[ServerID].ACK.>
_Coppelia.[ServerID].SESSION.>
```

2.5.4.4.2 Coppelia's TIBCO Rendezvous Message Format

Coppelia uses the TIBCO Rendezvous standard message format. For FIX messages, the message is a sequence of the Tag/Value pairs included in the header and body of the FIX message. The contents of the FIX trailer are not sent in the RV message. Tags are specified as a string representing their tag name. Values are always represented as strings.

IMPORTANT: The FIX tags are required to be in a specific order in the RV message, matching the required order specified in the FIX protocol documentation. The most notable requirement to be aware of is that tags comprising a repeating group must appear in the RV message as a contiguous block.

The only tags that have to be specified from the FIX standard header is MsgType (Tag #35), SenderCompID (Tag #49), and TargetCompID (Tag #56). The following tags will automatically be filled in by Coppelia, BeginString (Tag #8), BodyLength (Tag #9), MsgSeqNum (Tag #34), and SendingTime (Tag #52).

Incoming messages will appear on the RV message bus with the tag/value format. If the RAW ON option is enabled in the configuration file, the message will consist of a single RV Datum element with the raw FIX message string. If the FULL_OBJECT ON option is enabled, then the message will have the tag/value format with the raw FIX message string appended to the end of the RV message.

Example:

The following Incoming FIX Message:

```
8=FIX.4.1|9=0075|35=D|56=SBI|57=Daniel|49=SLGM|34=3|52=
19991022-
14:01:17|11=one|21=1|55=IBM|54=1|38=1000|40=1|10=124|
```

is by default represented with the following RV message:

BeginString FIX.4.1
BodyLength 75
MsgType D
TargetCompID SBI
TargetSubId Daniel
SenderCompID SLGM
MsgSeqNum 3
SendingTime 19991022-14:01:17
ClOrdID One
HandlInst

1
Symbol IBM
Side 1
OrderQty 1000
OrdType 1

For Admin requests, the message contains one element, a string pair consisting of a request type and an optional command field.

Example:

The operator command connect SBI is represented with the following RV message:

COMMAND connect SBI

For session up/down notification or session info requests, the fields of an OperatorData object comprise the RV message.

Example:

The following RV messagecontains the fields of an OperatorData object. The connect_state field indicates that the FIX session with SBI is up.

firm_id SBI
local_firm_id SLGM
local_trader_id GEORGE
net_address 127.0.0.1
port 7000
description Salomon Brothers Inc
contact Tech Support (212) 555-1212
heartbeat_interval 30
connect_state 2
encryption 0
version 410
msg_sequence_num_out 2
msg_sequence_num_in 1
last_message_time_in Wed Jan 12 14:35:02 EST 2000
last_message_time_out Wed Jan 12 14:35:17 EST 2000
messages_in

0
messages_out 1
orders 0
executions 0
rejects 0
execution_acks 0
allocations 0
iois 0
heartbeats 0
cancels 0
corrects 0
logons 0
bytes_in 0
bytes_out 82

2.5.4.5 Client Interaction with Coppelia using TIBCO Rendezvous

2.5.4.5.1 Sending FIX messages to Coppelia

In order to request that Coppelia send a FIX message to a counter party, it is necessary to generate a RV message according to the format described in 2.5.4.4.2 TIBCO Rendezvous Message Format. The message must then be sent to Coppelia on the appropriate subject for that counterparty and message type. Outgoing FIX messages are published to a Coppelia engine with Server ID [ServerID] on the subject "Coppelia.[ServerID].OUT.[TargetCompID].[MsgType]" where [TargetCompID] and [MsgType] are the corresponding fields in the FIX message header. Coppelia will send a response to the reply subject of the RvMsg indicating the status of the request to send the message.

Refer to the following example of sending an order to Coppelia:

```
MyOrder order;
String serverID;
RvSession sess;
...
String subj = "Coppelia." + serverID + ".OUT." +
order.TargetCompID + ".D";
String replySubj = "_INBOX.msg.reply.subject";
try {
  RvMsg msg = convertMyOrder(order);
  RvSender sender = sess.newSender(subj);
  sender.setReplySubject(replysubj);
  sender.send(rv);
}
catch(Exception e) {
  e.printStackTrace();
}
```

2.5.4.5.2 Sending Administrative Commands to Coppelia

Administrative commands are published to a Coppelia engine with Server ID [ServerID] on the subject "_Coppelia.[ServerID].REQ.COMMAND.[Command Name]" where [Command Name] is an optional string reperesenting the name of the administrative command. Coppelia will send a response to the command on the reply subject specified in the RvMsg.

Refer to the following example of sending Coppelia a command to connect to the target "SBI":

```
String serverID;
RvSession sess;
...
String subj = "_Coppelia." + serverID + ".REQ.COMMAND";
String replySubj = "_INBOX.cmd.reply.subject";
try {
  RvMsg msg = new RvMsg();
  msg.append("COMMAND", "connect SBI");
  RvSender sender = sess.newSender(subj);
  sender.setReplySubject(replysubj);
  sender.send(rv);
}
catch(Exception e) {
  e.printStackTrace();
}
```

2.5.4.5.3 Receiving FIX messages from Coppelia

When Coppelia receives an incoming FIX message, it is published on the subject "Coppelia.[ServerID].IN.[SenderCompID].[MsgType]" where [SenderCompID] and [MsgType] are the corresponding fields in the FIX message header. The RvMsg object can be parsed according to the format specified in 2.5.4.4.2 TIBCO Rendezvous Message Format.

Refer to the following example of parsing a FIX message:

```
synchronized public void onData(String subject, RvSender
reply, Object data, RvListener invoker) {
  String sender = "";
  String msgType = "";
  String trader = "";
  StringTokenizer tokenizer = new StringTokenizer(subject,
".");
  for (int i=0; i<tokenizer.countTokens(); i++) {
    String tok = tokenizer.nextToken();
    if (i==3) sender = tok;
    else if (i==4) msgType = tok;
  }
  RvMsg rm = (RvMsg) data;
  Enumeration e = rm.elements();
  while (e.hasMoreElements()) {
    RvDatum rd = (RvDatum) e.nextElement();
    String fieldName = rd.name;
    String value = rd.data;
    if (fieldName.equals("TargetSubID"))
```

```
      trader = value;
    System.out.println("Message for " + trader + "received
from " +
                        sender + " of type " + msgType);
  }
}
```

2.5.4.5.4 Receiving Session Up/Down Notification from Coppelia

When a FIX session is created or terminated, Coppelia publishes notification of the event on the subject "_Coppelia.[ServerID].SESSION.NOTIFICATION.[TargetCompID]" where [TargetCompID] is the name of the FIX target. The RV message can be parsed according to the format specified in 2.5.4.4.2 TIBCO Rendezvous Message Format.

Refer to the following example of parsing session notification message:

```
import com.javtech.coppelia.*;
synchronized public void onData(String subject, RvSender
reply, Object data, RvListener invoker) {
  String firmID = "";
  int state;
  StringTokenizer tokenizer = new StringTokenizer(subject,
".");
  for (int i=0; i<tokenizer.countTokens(); i++) {
    String tok = tokenizer.nextToken();
    if (i==4) firmID = tok;
  }
  RvMsg rm = (RvMsg) data;
  Enumeration e = rm.elements();
  while (e.hasMoreElements()) {
    RvDatum rd = (RvDatum) e.nextElement();
    String fieldName = rd.name;
    String value = rd.data;
    if (fieldName.equals("connect_state")) {
       state = Integer.parseInt(value);
    }
  }
  switch(state) {
    case ConnectState.DISCONNECTED:
      System.out.println(firmID + " disconnected");
    case ConnectState.LOGGING_ON_CONNECTION:
      System.out.println(firmID + " logging on");
    case ConnectState.APPLICATION_CONNECTION:
      System.out.println(firmID + " connected");
  };
}
```

2.5.5 Talarian SmartSockets

2.5.5.1 Introduction

This interface allows a remote client application to access the Coppelia engine via Talarian SmartSockets. The Java client interface follows the CoppeliaSrv interface model. Please refer to the JavaDoc document CoppeliaSrv.html for a general description of the CoppeliaSrv interface. This document assumes that the reader is comfortable developing applications using Talarian SmartSockets.

2.5.5.2 Configuration File Entries

To use this interface, the following lines are required in the .dat file:

```
INTERFACE                TALARIAN
TALARIAN_SERVER_ID       [serverID]
TALARAIAN_HOST           [host]
```

The TALARIAN_SERVER_ID parameter takes a string value to specify the subject Server ID used by Coppelia for communication with client applications. The TALARIAN_HOST parameter takes a string value that may be either a hostname or ip address describing the location of the Talarian SmartSockets RTServer on the network.

The .dat file may also contain the following pertinent information:

```
TALARIAN_PROJECT         [project] (default: coppelia)
RAW                      [ON/OFF]  (default: OFF)
FULL_OBJECT              [ON/OFF]  (default: OFF)
SESSION_NOTIFICATION     [ON/OFF]  (default: OFF)
```

The TALARIAN_PROJECT parameter takes a string value to override the default Talarian project that Coppelia will bind to.

The other options are described later in this document in section 5.3 – Receiving Messages from Coppelia

2.5.5.3 Starting Coppelia with Talarian SmartSockets

To start the Coppelia server using the Talarian SmartSockets interface the classpath needs to be set to include the SmartSockets libraries:

Example:

```
run_tss.sh:

#!/bin/sh
LIBDIR=/usr/local/Coppelia/classes
CLASSPATH=$LIBDIR/coppelia.jar:$LIBDIR/mct3_0.zip:$LIBDIR/OrbixWeb31c.j
ar:$LIBDIR/pro.jar:$LIBDIR/rogue.zip:$LIBDIR/ss.jar
java Coppelia talarian_buy.dat
```

2.5.5.4 Detailed Description of Interface Implementation

2.5.5.4.1 Subject Namespace

Below is a description of the subjects used by CoppeliaTSS to interact with client applications. The [ServerID] field corresponds to the TALARIAN_SERVER_ID value set in the config file.

Coppelia Server generated events:

1. Incoming FIX message received.

```
/Coppelia/[ServerID]/IN/[SenderCompID]/[MsgType]
```

2. Outgoing FIX message sent acknowledgement.

```
/_Coppelia/[ServerID]/ACK/[TargetCompID]/[MsgType]
```

3. Session Up/Down Notification.

```
/_Coppelia/[ServerID]/SESSION/NOTIFICATION/[TargetCompID]
```

Client Application generated events:

1. Outgoing FIX message sent.

```
/Coppelia/[ServerID]/OUT/[TargetCompID]/[MsgType]
```

2. FIX session list requested.

```
/_Coppelia/[ServerID]/REQ/SESSION/LIST
```

3. FIX session's information requested.

```
/_Coppelia/[ServerID]/REQ/SESSION/INFO
```

4. Operator command issued.

```
/_Coppelia/[ServerID]/REQ/COMMAND
```

Coppelia subscribes to the following subjects:

```
/Coppelia/[ServerID]/OUT/...
/_Coppelia/[ServerID]/REQ/...
```

Client applications will generally subscribe to the following subjects:

```
/Coppelia/[ServerID]/IN/...
/_Coppelia/[ServerID]/ACK/...
/_Coppelia/[ServerID]/SESSION/...
```

2.5.5.4.2 Coppelia's Talarian SmartSockets Message Format

Coppelia uses the SmartSockets STRING_DATA message type. For FIX messages, the data part of the SmartSockets message is a sequence of the Tag/Value pairs included in the header and body of the FIX message. The contents of the FIX trailer are not sent in the SmartSockets message. Tags are specified as a string representing their tag name. Values are always represented as strings.

Example:

The following FIX Message:

```
8=FIX.4.1|9=0075|35=D|56=SBI|57=Daniel|49=SLGM|34=3|52=
19991022-
14:01:17|11=one|21=1|55=IBM|54=1|38=1000|40=1|10=124|
```

is represented with the following SmartSockets data elements:

BeginString FIX.4.1
BodyLength 75
MsgType D
TargetCompID SBI
TargetSubId Daniel
SenderCompID SLGM
MsgSeqNum 3
SendingTime 19991022- 14:01:17
ClOrdID One
HandlInst

1
Symbol IBM
Side 1
OrderQty 1000
OrdType 1

For Admin requests, the data part of the message contains one element, a string pair consisting of a request type and an optional command field.

Example:

The operator command connect SBI is represented with the following SmartSockets data element:

COMMAND connect SBI

For session up/down notification or session info requests, the fields of an OperatorData object comprise the TipcMsg.

Example:

The following TipcMsg contains the fields of an OperatorData object. The connect_state field indicates that the FIX session with SBI is up.

firm_id SBI
local_firm_id SLGM
local_trader_id GEORGE
net_address 127.0.0.1
port 7000
description Salomon Brothers Inc
contact Tech Support (212) 555-1212
heartbeat_interval 30
connect_state 3
encryption 0
version

410
msg_sequence_num_out 2
msg_sequence_num_in 1
last_message_time_in Wed Jan 12 14:35:02 EST 2000
last_message_time_out Wed Jan 12 14:35:17 EST 2000
messages_in 0
messages_out 1
orders 0
executions 0
rejects 0
execution_acks 0
qllocations 0
iois 0
heartbeats 0
cancels 0
corrects 0
logons 0
bytes_in 0
bytes_out 82

2.5.5.5 Client Interaction with Coppelia using Talarian SmartSockets

2.5.5.5.1 Sending FIX messages to Coppelia

After generating a TipcMsg object according to the format described in 4.2 Talarian SmartSockets Message Format, the message can be sent to Coppelia on the appropriate subject. Outgoing FIX messages are published to a Coppelia engine with Server ID [ServerID] on the subject "/Coppelia/[ServerID]/OUT/[TargetCompID]/[MsgType]" where [TargetCompID] and [MsgType] are the corresponding fields in the FIX message header.

Refer to the following example of sending an order to Coppelia:

```
MyOrder order;
String serverID;
...
String subj = "/Coppelia/" + serverID + "/OUT/" + order.TargetCompID +
"/D";
try {
  Tut.setOption("ss.project", "coppelia");
  TipcSrv srv = TipcSvc.getSrv();
  TipcMsg msg = convertMyOrder(order);
  msg.setDest(subj);
  srv.send(msg);
  srv.flush();
  srv.destroy();
}
catch(Exception e) {
  e.printStackTrace();
}
```

2.5.5.5.2 Sending Administrative Commands to Coppelia

Administrative commands are published to a Coppelia engine with Server ID [ServerID] on the subject "/_Coppelia/[ServerID]/REQ/COMMAND/[Command Name]" where [Command Name] is an optional string reperesenting the name of the administrative command.

Refer to the following example of sending Coppelia a command to connect to the target "SBI":

```
String serverID;
...
String subj = "/_Coppelia/" + serverID + "/REQ/COMMAND";
try {
  Tut.setOption("ss.project", "coppelia");
  TipcSrv srv = TipcSvc.getSrv();
  TipcMsg msg = TipcSvc.createMsg(TipcMt.STRING_DATA);
  tssMsg.appendStr("COMMAND");
```

```
  tssMsg.appendStr("connect SBI");
  msg.setDest(subj);
  srv.send(msg);
  srv.flush();
  srv.destroy();
}
catch(Exception e) {
  e.printStackTrace();
}
```

2.5.5.5.3 Receiving FIX messages from Coppelia

When Coppelia receives and incoming FIX message, it is published on the subject "/Coppelia/[ServerID]/IN/[TargetCompID]/[MsgType]" where [TargetCompID] and [MsgType] are the corresponding fields in the FIX message header. The TipcMsg object can be parsed according to the format specified in 4.2 Talarian SmartSockets Message Format.

Refer to the following example of parsing a FIX message:

```
public void process(TipcMsg msg, Object obj) {
  String sender = "";
  String msgType = "";
  String trader = "";
  StringTokenizer tokenizer = new StringTokenizer(msg.getDest(), "/");
  for (int i=0; i<tokenizer.countTokens(); i++) {
    String tok = tokenizer.nextToken();
    if (i==3) sender = tok;
    else if (i==4) msgType = tok;
  }
  try {
    msg.setCurrent(0);
    for (int i=0; i < msg.getNumFields()/2; i++) {
      String fieldName = msg.nextStr();
      String value = msg.nextStr();
      if (fieldName.equals("TargetSubID"))
        trader = value;
    }
    System.out.println("Message for " + trader + "received from " +
                       sender + " of type " + msgType);
  }
  catch(TipcException te) {
    te.printStackTrace();
  }
}
```

2.5.5.5.4 Receiving Session Up/Down Notification from Coppelia

When a FIX session is created or terminated, Coppelia publishes notification of the event on the subject "/_Coppelia/[ServerID]/SESSION/NOTIFICATION/[TargetCompID]" where [TargetCompID]is the name of the FIX target. The TipcMsg object can be parsed according to the format specified in 4.2 Talarian SmartSockets Message Format.

Refer to the following example of parsing session notification message:

```
import com.javtech.coppelia.*;
public void process(TipcMsg msg, Object obj) {
  String firmID = "";
  boolean down;
  StringTokenizer tokenizer = new StringTokenizer(msg.getDest(), "/");
  for (int i=0; i<tokenizer.countTokens(); i++) {
    String tok = tokenizer.nextToken();
    if (i==4) firmID = tok;
  }
  try {
    msg.setCurrent(0);
    for (int i=0; i < msg.getNumFields()/2; i++) {
      String fieldName = msg.nextStr();
      String value = msg.nextStr();
      if (fieldName.equals("connect_state")) {
        int state = Integer.parseInt(value);
        down = (state == ConnectState.DISCONNECTED);
      }
    }
    if (down) {
      System.out.println(firmID + " disconnected");
    }
    else {
      System.out.println(firmID + " connected");
    }
  }
  catch(TipcException te) {
    te.printStackTrace();
  }
}
```

2.5.6 IBM MQSeries

2.5.6.1 Introduction

The Coppelia server supports multiple interfaces. IBM's MQSeries product (a middleware messaging solution) is one of them. This quick guide describes the usage of Coppelia with IBM MQSeries.

2.5.6.2 Pre-Requisites

Make sure you have MQSeries (either the client or server portion) installed on the machine or directory Coppelia is installed in.

There must be a (remote) MQServer for Coppelia to communicate with. This MQServer must have the appropriate channels, queues and queue manager setup for Coppelia to use. These are needed for Coppelia to be able to send messages to and and receive messages from the server.

2.5.6.3 Coppelia Setup

You can have either the buy side, sell side, or both communicate with MQSeries. The first step you should take is to include MQ Java libraries in Coppelia's classpath (in either the buy or sell side startup scripts, or both if you want both to use MQSeries.

The following is an example sell side startup script on WinNT:

```
SET PATH=../JRE/BIN
set CLASSPATH=../classes;../classes/Coppelia.zip;../classes/
pro.zip;../classes/mct_3.zip; ../classes/OrbixWeb_3.0.zip;
../classes/rogue.zip;../jre/lib;%Install Path%:\MQM\JAVA\LIB
jre –classpath %CLASSPATH% Coppelia sell.dat
```

%Install Path% represents the path where you installed MQSeries. If it is installed on your C:\ drive, the this would be "C:\MQM\JAVA\LIB". In UNIX, it could be (for example) "/opt/mqm/java/lib", depending on your installation path.

2.5.6.4 .dat file options

The next thing for you to set-up is the *.dat files. This is where you tell Coppelia to interface with MQSeries, which queues to use, who's the MQManager, etc. Take a look at the sell.dat example below:

```
PORT                        7000
IIOP_PORT                   7100
IIOP_IP                     192.168.129.20
CONNECT                     SERVER
LOG_HEARTBEAT               ON
TITLE                       Coppelia v4.1
GUI                         OFF
TRADETABLE                  ON
DESCRIPTION                 Sell Side configuration
INTERFACE                   MQSERIES
MQ_HOST_NAME                192.168.129.24
MQ_MANAGER_NAME             COPPELIA.QUEUE.MANAGER
MQ_CHANNEL_NAME             COPPELIA.CHANNEL
MQ_SOURCE_QUEUE_NAME        COPPELIA.SOURCE.QUEUE
MQ_TARGET_QUEUE_NAME        COPPELIA.TARGET.QUEUE
MQ_AUDIT_QUEUE_NAME         SYSTEM.DEAD.LETTER.QUEUE

#ID; TargetCompID; SenderCompID; SenderSubID; network
address; description; contact; heart beat; encryption type;
version number+start seq num
ID; DTC; SLGM; GEORGE; 192.168.129.20;7000; Salomon
Brothers Inc; Tech Support (212) 555-1212;30;0;401
ID; NWM; SLGM;LISA;192.168.129.88;7000; NatWest Markets;
Girard Roux (212) 555-1212;30;0;401
ID; MONT; SLGM;GEORGE;192.168.129.85;7000; Montgomery
Securities; Bruce Harris (415) 555-1212;30;0;300
TRADER_IDS; DTC; JOE, HARRY, SAM
```

The INTERFACE settings changes Coppelia's interface from the default (CORBA) to MQSERIES. This initiates Coppelia's communication to the MQServer. MQ_HOST_NAME is the IP Address or the DNS (Domain Name Server, if supported) on which the MQSeries Server is installed. MQ_MANAGER_NAME is the queue manager that controls the queues. MQ_CHANNEL_NAME is a Server-Connection Channel (SVRCON) that acts as a communication bridge between Coppelia and MQ. MQ_SOURCE_QUEUE_NAME is the name of the queue which will contain messages from FIX (Coppelia) to MQ. In other words, this is the queue that Coppelia writes to. MQ_ TARGET_QUEUE_NAME is the queue which will contain messages from MQSeries to FIX or the queue from which Coppelia reads. MQ_AUDIT_QUEUE_NAME is the queue where all undelivered messages are sent (rejects, invalid data, etc.) You could use the default dead letter queue of MQSeries or you can set-up a queue specifically for auditing undelivered FIX messages.

This is a basic set-up for MQ-Coppelia integration. Even with this kind of set-up you can start using the power of MQSeries Communications with Coppelia. Later, we will discuss some more advanced set-ups and example Java programs.

2.5.6.4.1 Other MQ Settings for Coppelia

You can customize Coppelia to write to different queues and not just one source queue. You can sort the messages by fix versions and message types so they could be placed in their appropriate queue. To do this, set your source queue to a simple default name. For this example, we'll set it to "Coppelia". Here are some additional settings you can add to the *.dat files:

```
MQ_SOURCE_QUEUE_NAME          COPPELIA
MQ_BY_VERSION                 ON
MQ_BY_FIX_MSG_TYPE            ON
```

With these settings, you will append the FIX version and FIX message type to the Source Queue name. So, if the FIX message was an order "Msg type D" using FIX version 4.0, then the message will be written to the queue named "COPPELIA_40_D". If the message was an Indication of Interest using FIX 4.1, it will be stored in "COPPELIA_41_6". Make sure that these queues exist, otherwise you would get an error message (which will be discussed later).

You could also customize Coppelia to place prefixes and suffixes after the queue name. This is used when there are naming conventions regarding queues in some organizations. Enter either one of these settings (or both, if your organization requires this):

```
MQ_QUEUE_NAME_SUFFIX          <suffix name>
MQ_QUEUE_NAME_PREFIX          <prefix name>
```

If you want to use the prefix, "FIX", then set the MQ_QUEUE_NAME_PREFIX to FIX. (period included). This would append the prefix to the queue name, telling Coppelia to send all messages to the source queues with the prefix "FIX". Using the source queue name, "COPPELIA" and the prefix "FIX" as example, Coppelia will be writing to the queue "FIX.COPPELIA". Likewise, if the suffix is used instead of the prefix, Coppelia will be writing to "COPPELIA.FIX".

You could also sort the messages to queues specific to the TargetCompID's or companies you are communicating with. Let's say you have Merrill Lynch, JP Morgan, Smith Barney and Bear Sterns as the companies that Coppelia is communicating with. These names are already set-up in your *.dat file. You want to have Coppelia sort out the messages into queues accordingly. In your *.dat file, you have Merrill Lynch set up as MLCO, JP Morgan as JPM, Smith Barney as SBI and Bear Sterns as BSC. Use this setting:

```
MQ_BY_TARGET_ID          ON
```

This will tell Coppelia to sort out the messages and place them into their appropriate queues (provided, of course, that these queues do exist). So, having "COPPELIA" as our source queue and the MQ_BY_TARGET_ID setting on, the messages will be sorted out as follows:

TargetCompID	*Queue Name*
Merrill Lynch	COPPELIA_MLCO
JP Morgan	COPPELIA_JPM
Smith Barney	COPPELIA_SBI
Bear Sterns	COPPELIA_BSC

If you want to accept only certain message types and reject others, you can set that up by simply using the `REJECT_MSGTYPES` parameter. Use this setting as follows:

```
REJECT_MSGTYPES     <msgtype>, <msgtype>, <msgtype>
```

Where <msgtype> represents the type of FIX message to be rejected (8=Execution reports, D=Orders, 6=IOI, etc.). You input the message types that you want to reject, separated by a comma (,).

2.5.6.5 Examples

The following example will place a message in the Target Queue and Coppelia will read the Queue and shoot it out as a FIX message to the client it is communicating with. It is presumed that the sell side is communicating with MQ.

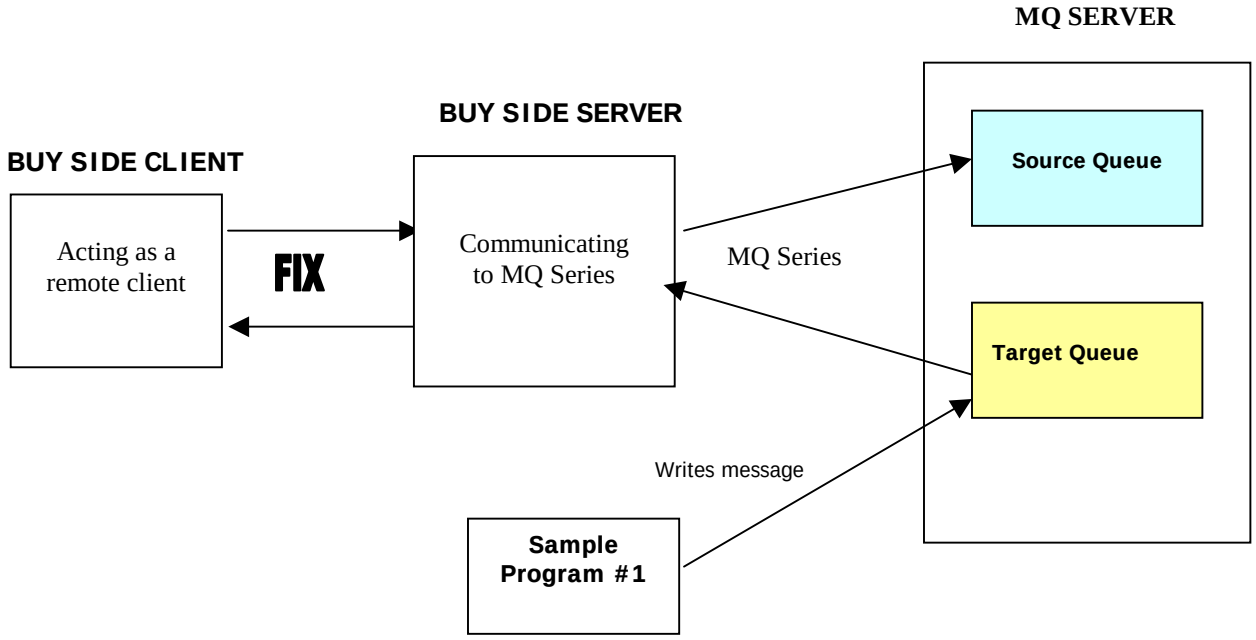
```
import com.ibm.mq.*;

/**
 * This example demonstrates how a client application uses MQSeries
 * interface, instructing the Coppelia server to send FIX application
 * messages to other connecting FIX engines. The underlying mechanism
 * is that a tag-value string that represents FIX-format messages
 * is put to MQ's queue and retrieved by Coppelia server which
 * turns it into Coppelia's out-bound message.
 *
 * This is a generic example for doing all the sending, for both
 * buy and sell side configuration. It is the message string that
 * determines what action(such as sending orders, IOI, etc.) is
 * to be taken by Coppelia server. Depending on what your goal,
 * all you need to do is construct the proper message string and
 * the rest is plain trivial stuff for an MQ developer.
 *
 * In this example, we demonstrate the send order scenario. Of course
 * you could use this example to send IOI's or Execution Reports, etc.
 * by simply constructing the string message for IOI's, Execution Reports,
 * etc. FIX application messages can be referenced in FIX protocol paper
 * or Javelin's FIXionary at www.javtech.com. In the abstract FIX messages
 * are stream of tag-value fields, each delimited by the "\u0001" character.
 * When constructing the FIX message string, you need to include all
 * required fields stated in the protocol plus the TargetCompID
 * and SenderCompID. Refer to the code for more detail.
 *
 * As you will notice as a MQ developer, the thing you need to
 * learn to develop a FIX application via MQ are the FIX application
 * messages and how to construct those message strings, which represents
 * the business perpective of FIX usage. And the rest is same old MQ.
 */
public class MQClientSend
{
    public static void main(String[] args)
    {
        String host = null;      // Name of the host or machine that runs MQ
        String channel = null;   // Name of the channel for client connection
        String qManager = null;  // Name of the Q manager
        MQQueueManager mqQManager = null;  // queue manager object.
        String fixMsg = null;

        // Put the order message here, hard-coded for this example.
        // Each field is delimited by the "\u0001" character. You must
        // put all the required fields specified in the FIX protocol
        // plus the TargetCompID(tag 56) and SenderCompID(tag 49)
        // in order to instruct Coppelia who the message is directed
        // to and who is it originated from and most importantly,
        // the MsgType(tag 35) so Coppelia knows what action
        // is to be taken. Other fields are optional.
        // Please refer FIX protocol or Javelin's FIXionary for
        // detail on this message. This is an order message.
        String delim = "\u0001";

        fixMsg = "35=6" + delim + "23=12" + delim + "28=N" +
            delim + "56=FIX46265" + delim + "49=DTCY" +
            delim + "55=AAPL" + delim + "54=2" +
            delim + "27=10000" + delim + "44=12" + delim;
```

```
        // Alternative for hardcoded host, channel and qManager.
        /*
        if(3 != args.length) {
            System.out.println("Usage:  java MQClientSend <host>" +
                               " <channel> <qManager>");
            return;
        }
        host = args[0];
        channel = args[1];
        qManager = args[2];
        */

        // Hardcoded. Could be passed in as args as shown above.
        // Make sure you provide the proper name used in your
        // system. The following are the setup at Javelin.
        host = "192.168.129.24";  // Name if DNS is supported.
        channel = "PLN.COP.SVRCONN";
        qManager = "COPPELIA.QUEUE.MANAGER";
        // Setup MQ environment
        MQEnvironment.hostname = host;
        MQEnvironment.channel = channel;
        //MQEnvironment.port = port;  // if neccessary

        // Get a q manager then get the queue.
        // Create the message and put the message
        // to the queue and done!
        try{
            // Creating a connection to queue manager.
            mqQManager = new MQQueueManager(qManager);

            // Setup the option of the queue we are to open.
            int openOptions = MQC.MQOO_INPUT_AS_Q_DEF | MQC.MQOO_OUTPUT;

            // Define the queue where you will put the msg in.
            // This queue is the queue that Coppelia server
            // will retrieve the msg from and turn it to
            // Coppelia out-bound FIX message.
                    MQQueue localQueue =
                mqQManager.accessQueue(//Name of the queue on your system
                                       "FIX.PLN_ALL.Q01",
                                       openOptions,
                                       null,
                                       null,
                                       null);

            // Define a  message Queue, order in this case.
            MQMessage order = new MQMessage();

            // Write the string into Msg object.
            order.writeBytes(fixMsg);

            // Specify the message option.
            MQPutMessageOptions pmo = new MQPutMessageOptions();

            // Put the message on the queue
            localQueue.put(order, pmo);

            // Done, close the queue manager connection
            System.out.println("Done sending FIX message : " +
                               fixMsg);
            mqQManager.disconnect();
        }
        catch(MQException e) {
            e.printStackTrace();
        }
        catch(java.io.IOException e) {
            e.printStackTrace();
        }
    }
}
```

When you compile the program, make sure you have the /mqm/java/lib directory within your classpath. To run the program, create a shell script (for UNIX) or batch file (for NT). Here is an example:

```
SET PATH=c:\jdk1.1.7a\bin;
SET
CLASSPATH=c:\jdk1.1.7a\lib\classes.zip;..\classes\pro.zip;..\classes\Co
ppelia.jar;..
\classes\OrbixWeb31c.jar;D:\MQM\JAVA\LIB;.
java MQClientSend
```

Below is a diagram of how this program integrates with Coppelia-MQ:



The next program will read FIX messages from a text file, send to a buy side client (not communicating with MQ) who will, in turn, deliver it to the sell side (communicating to MQ). The sell side will sort the message into its appropriate queue (provided that they were set-up properly):

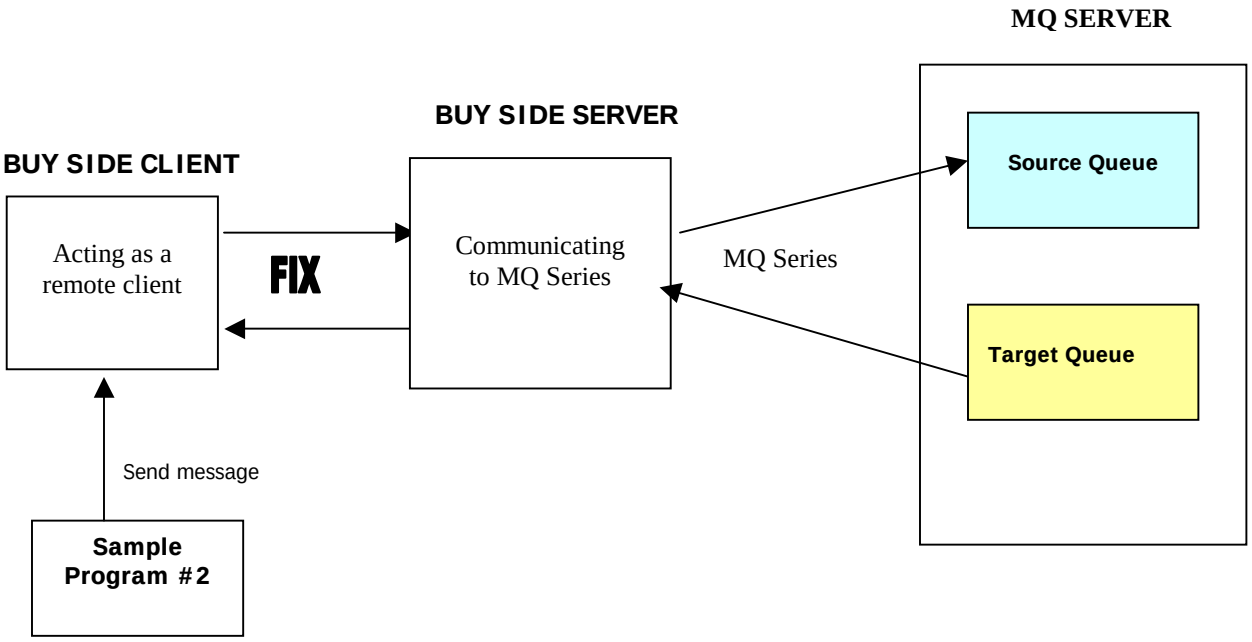
```
import java.util.*;
import java.lang.*;
import java.io.*;
import IE.Iona.OrbixWeb._CORBA;
import org.omg.CORBA.*;
import IE.Iona.OrbixWeb._OrbixWeb;
import IE.Iona.OrbixWeb.Features.Config;
import org.omg.CORBA.ORB;


public class rcvmsg {


 public static CoppeliaServer remote_obj = null;

 public static void main(String args[]) {

   timestamp("Looking up server:");

   try {

      /** read in the InterOp Reference (IOR) string that Coppelia creates **/

       FileReader fr = new FileReader("CoppeliaIOR.str");
       BufferedReader br = new BufferedReader(fr);

      String newIOR =  br.readLine();

      /** create a remote object reference **/

      org.omg.CORBA.Object o = ORB.init(new String []{“Coppelia”},
null).string_to_object(newIOR.trim());
      remote_obj = CoppeliaServerHelper.narrow(o);

   }
   catch (SystemException ex) {

      System.out.println("Exception during bind");
          ex.printStackTrace();
      System.exit(1);

   }
   catch (IOException ioe) { System.out.println("IOExcpetion:" + ioe); }

   if (remote_obj == null) System.exit(0); // exit gracefully

    timestamp("Starting loop");

     /** go into a forever loop, looking for messages or errors **/
     try {
        int rc;
        CRawDataType prc;
        while (true) {

            /** the first time this is called, a CORBA connection is created **/
            prc = remote_obj.getNextRawData();

               if (prc.data.length()>0)
                           System.out.println(prc.data);
                    else {
                        System.err.println("Waiting for data..." + new Date());
                       try { Thread.sleep(2000); } catch (InterruptedException i) {  }
                 }
            } /** end of while **/
```

```
     }
     catch (SystemException e) {

            System.out.println("Exception: remote call failed:" + e);
               e.printStackTrace();

    }

}


static void timestamp(String s) {

    Date d = new Date();
    System.err.println(s + ": " + d);

 }

}
```

Compile the Java program with the coppelia.jar, OrbixWeb31c.jar within your classpath. To run the program, copy the CoppeliaIOR.str file from the buy side client which the program will send the messages to. Then, run a script or batch file to execute the program. Below is sample batch file to run the program:

```
copy ..\buy2\CoppeliaIOR.str .
SET PATH=c:\jdk1.1.7a\bin;
SET
CLASSPATH=../classes;..\classes\pro.zip;..\classes\coppelia.jar;..\clas
ses\
mct3_0.zip;..\classes\rogue.zip;..\classes\ie.zip;..\classes\OrbixWeb31
c.jar;c:\jdk1.1.7a\lib\classes.zip;.
java -classpath %CLASSPATH% sendmsg -n %1 %2
```

Here is a diagram of this program working with Coppelia-MQ:



2.5.6.7 HINTS

```
MQRC_UNKNOWN_OBJECT_NAME      2085        X'00000825'
```

This error occurs when Coppelia tries to look for a defined object but cannot find it. This object could be either a queue manager or a local/remote queue. To avoid this, make sure all the objects defined in the *.dat files of Coppelia exist and are spelled correctly. Typos could be a nasty culprit of this problem.

```
MQRC_Q_MGR_NOT_AVAILABLE      2059        X'0000080B'
```

This means that the Qmanager is not available for communication. It is either the Qmanager or the channel listener. Make sure that they both have been started.

```
java.lang.NullPointerException
      at Interface.doMQSeries(Compile Code)
      at Interface.run(Interface.java:76)
      at java.lang.Thread.run(Thread.java:475)
```

This shows up when you start up Coppelia but does not have the mqm\java\lib directory within the classpath. Double-check your startup script or batch file.

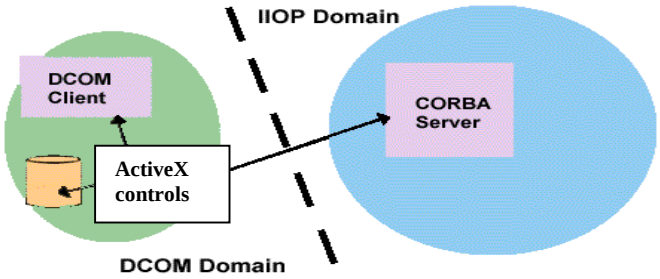
For more information on MQSeries and Programming for MQSeries, please refer to the "MQSeries Application Programming Guide" and the "MQSeries Application Programming Reference."

2.5.7 Active X /COM/DCOM/OLE

2.5.7.1 Introduction

Coppelia is designed to support an array of interfaces and middleware products. COM/ActiveX users implement Coppelia through Coppelia Active X Controls. This section of the document describes the methods to send and receive FIX messages with the Coppelia ActiveX controls.

The Coppelia Server is implemented using CORBA technology with IIOP as the natural client-server connection. To allow ActiveX/COM developers to integrate Coppelia into their trading systems, the Coppelia ActiveX controls are used to provide the inter-domain (DCOM/IIOP) bridging. The ActiveX control is written in C++ with ATL for portability and high performance.



Coppelia Active X controls allow construction of messages in two principal forms:

FIX Tags or FIX Fields Names	Tag Value Pairs
FIX Raw Data String	Raw Data Format

The primary tasks of the Coppelia Active X controls are:

Post - to send messages to Coppelia

Get - to receive messages from Coppelia.

Messages between the client and server are passed as FIX raw data messages.

2.5.7.2 Coppelia Active X Control

The Coppelia ActiveX Control Package consists of the FIXMessage Control and the CoppeliaSrv Control.

The FIXMessage Control provides an efficient and user-friendly interface to build FIX raw data messages.

The CoppeliaSrv provides post and get methods to send and receive messages.

2.5.7.2.1 JavTech.FIXMessage Methods

The JavTech.FIXMessage methods specified below are used for operations upon a FIX message. They are used to manipulate tags and messages, as well as return tags from existing messages.

`addByTag(int, String)`

Adds a tag value pair to message, where *int* is the FIX tag number, and *String* is the value of the field.

example:

```
addByTag 55, "IBM"  'where 55 = ticker symbol
```

adds tag 55 and the value of "IBM"

For tags that are doubles and longs, simply place the value in quotation marks in the function.

example:

```
addByTag 44, "98.125" 'where 44 = price
```

adds tag 44 the value of 98.125 (price of share: 98 1/8)

`addByFieldName(String, String)`

Adds a tag value pair to the message using the field name as identifier.

example:

```
addByFieldName "Symbol", "IBM"
```

adds the tag Symbol (tag 55) and the value of "IBM"

For tags that are doubles and longs, simply place the value in quotation marks in the function.

example:

```
addByFieldName "Price", "98.125"
```

adds tag Price (tag 44) and the value of 98.125

`fromString(String)`

Populates a FIXMessage object from a FIX raw data string

example:

```
fromString "8=FIX.4.0|9=0102|35=D|56=SBI|49=SLGM|34=5|
50=JOE|57=STAN|52=1999092815:03:28|11=1|21=1|55=ADVS|
54=1|38=1000|40=1|59=0|10=115|"
```

Sets the raw data string in the FIXMessage object to the argument.

`String toString()`

Gets a FIX raw data string from a FIXMessage object.

example:

```
Dim msg1 As Object
Dim FIXString As String
FIXString = msg1.toString()
```

Gets the FIX raw data string from the FIXMessage object into a string.

`short first()`

The `first()` method sets the pointer to the FIX message to the first field. It does not return a value. To pick up the first field in the message the `getTag()` method is used immediately following the `first()` method.

example:

```
Dim msg1 As Object
Dim Tag1 As Int
msg1.first()
Tag1 = msg1.getTag()
```

`int getTag()`

Returns the current Tag from a FIXMessage object.

example:

```
Dim msg1 As Object
Dim Tag1 As Int
Tag1 = msg1.getTag()
```

If msg1 was the raw data message specified in the `fromString` example, and the program was looking at the first tag in the message (done by using the `first()` function), `Tag1` would be equal to 8 (8=FIX4.0 -it is the first field in the message that denotes which version of FIX we are speaking).

`String getFieldName()`

Returns the current field name from a FIXMessage object.

example:

```
Dim msg1 As Object
Dim Field1 As String
msg1.first
Field1 = msg1.getFieldName()
```

If `msg1` was the raw data message specified in the `fromString` example, and the location was set by using the `first()` function to point to the first field, `Field1` would be equal to `BeginString` (tag 8 – the first field – is called BeginString)

`String getValue()`

Returns the current value from a FIXMessage object.

example:

```
Dim msg1 As Object
msg1.first
String Value1 = msg1.getValue()
```

If `msg1` was the raw data message specified in the `fromString` example, and the location was set by using the `first()` function to point to the first field, `Value1` would be equal to FIX4.0 – (the value of the first field `BeginString` is "FIX4.0")

`short next()`

The next() function sets the location to the next field in the FIX message.

String getByTag(short tag)

Returns the value of a tag (specified by tag number) in a message. If there are multiple occurrences of the same tag in a message, this will return the value of the first occurrence. To get the values of subsequent occurrences in this version of the ActiveX control, you need to use first and next to process the fields in order.

example:

```
Symbol = msg1.getByTag(55)
```

String getByFieldName(String name)

Returns the value of a field (specified by the field name) in a message. If there are multiple occurrences of the same field in a message, this will return the value of the first occurrence. To get the values of subsequent occurrences in this version of the ActiveX control, you need to use first and next to process the fields in order.

example:

```
Symbol = msg1.getByFieldName("Symbol")
```

2.5.7.2.2 JavTech.CoppeliaSrv

The JavTech.CoppeliaSrv methods specified below are used to communicate with the Coppelia FIX server. These functions are also used for Coppelia queue manipulation.

`initialize ( String initString )`

This initializes the connection to Coppelia. The initialization string is a CORBA IOR string, which can be stored in a file and read by the client application.

example:

```
initialize
"IOR:000000000000001749444c3a436f7070656c6961536572766572
3a312e3000000000000100000000000000470001000000000000f31393
22e3136382e3132392e3235000013ec0000000000273a5c3139322e31
36382e3132392e32353a3a3a3a4946523a436f7070656c69615365727
6657200"
```

`String get (String sender_comp_id, String target_sub_id)`

This checks Coppelia for messages for a given comp id and/or sub id and it returns the message as a FIX message object. A null string is returned if there are no messages. Either or both of the arguments can be null strings – in this case all comp id or all sub id.

example:

If you are sub id "JDOE" at "ACME_BUY_SIDE", and you want to get the next message sent to you from "ACME_SELL_SIDE" only, the syntax would be as follows:

```
String Msg = CoppeliaSrv.get "ACME_SELL_SIDE", "JDOE"
```

If you want to get the next messages from any source, you would call:

```
String Msg = CoppeliaSrv.get "", "JDOE"
```

If you had access to messages for other sub ids, to get the next messages from "ACME_SELL_SIDE"

```
String Msg = CoppeliaSrv.get "ACME_SELL_SIDE", ""
```

`String getAll ()`

This gets the next message from Coppelia for any trader id – it returns the message as a string.

`int post ( String rawDataMsg )`

Sends a FIX message using a raw data string.

The most common return codes for this function are:

0	OK – message was sent properly
-100	Invalid Data – a field in the message was incorrect or invalid or there were missing fields, or fields that shouldn't be in the message. Often times this is caused by trying to send to an invalid target.
-103	Remote Down – the Coppelia server is not connected to the target. Make sure that the connection is up.

Note: there is no separate method to request resend of messages. This is done by building a resend request FIX message and using post to send it.

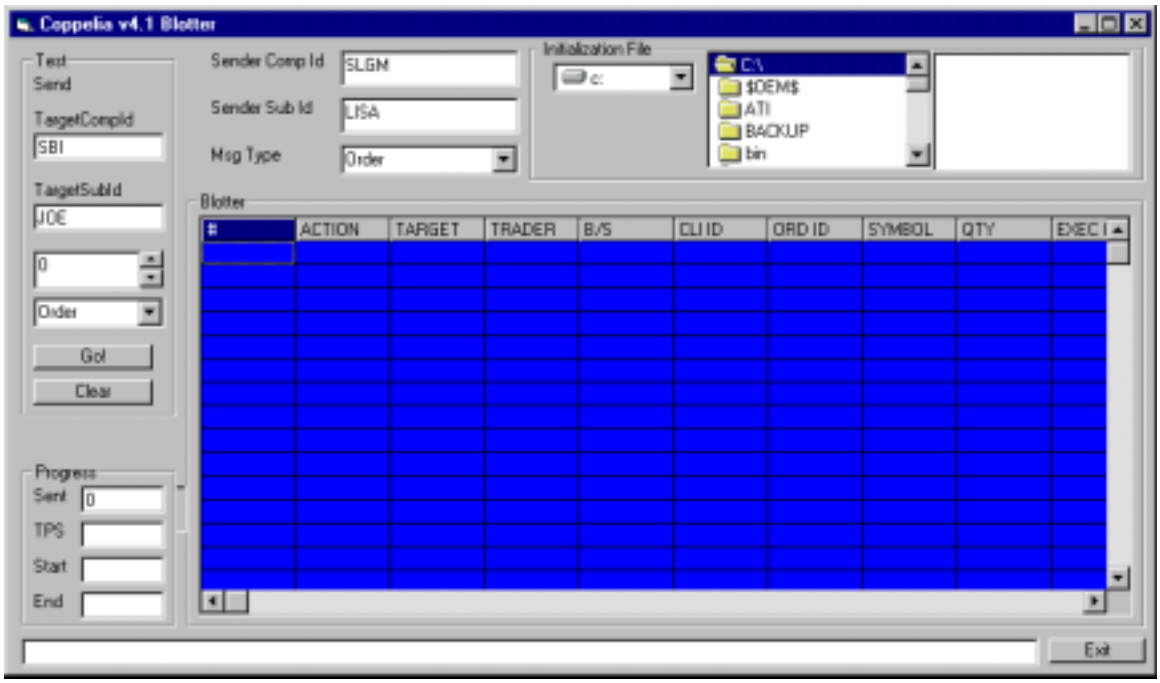
2.5.7.3 Examples

2.5.7.3.1 Visual Basic Sample Project

A Visual Basic example, which demonstrates how a simple application can be built using the Coppelia ActiveX controls, is included in the distribution package. This section of the document describes the main methods of this application to guide the reader through building a simple blotter application.

The application provides a file browser that can be used to specify the location of the IOR String file and a blotter screen that can be used to send orders and IOIs to Coppelia and can receive IOIs, Orders and Execution Reports. The application runs as a buy-side or sell-side blotter.

The application can also be used to send blocks of Orders, IOIs and Quotes using the "Test" sub-panel on the left of the blotter window.



2.5.7.3.2 Code Excerpts

Below are the excerpts from the Visual Basic sample application. The seven examples given comprise: *Initialize, Creating a message, Posting a message, Getting a message, and Getting Fields.* The application uses a timer to check if there are any incoming messages to process. This version of the ActiveX control does not support callbacks.

2.5.7.3.2.1 Initialize

This excerpt initializes a Coppelia Session by opening the IOR string from the CoppeliaIOR.str file.

```
Sub Session_Init()
Dim iorString As String
On Error GoTo ErrorHandler:
Open "D:\testing\Coppelia\buy\CoppeliaIOR.str" For Input As
#1
Line Input #1, iorString
Set CoppeliaSession = CreateObject("JavTech.CoppeliaSrv")
CoppeliaSession.Initialize iorString
Close #1
Exit Sub
ErrorHandler:
    MsgBox "ERROR in Session_Init: " & Error(Err)
End Sub
```

2.5.7.3.2.2 Creating a FIXMessage by tag number

This excerpt creates a FIXMessage. In this example, we create a simple FIX Order message by setting tag number values.

```
Dim aMessage As Object
On Error GoTo CreateErrorHandler
Set aMessage = CreateObject("JavTech.FIXMessage")
'=================================================
' build a FIX Message by setting values for tags
'=================================================
aMessage.addByTag 35, "D"
aMessage.addByTag 49, "SLGM"
aMessage.addByTag 50, "SenderSubID"
aMessage.addByTag 56, "SBI"
aMessage.addByTag 57, "ActiveX Order test"
aMessage.addByTag 55, "MSFT"
aMessage.addByTag 40, "1"
aMessage.addByTag 21, "1"
aMessage.addByTag 11, "Client ID 1"
aMessage.addByTag 38, "1000"
aMessage.addByTag 54, "1"
```

2.5.7.3.2.3 Creating a FIXMessage by field name

This excerpt creates a FIXMessage. In this example, we create a simple FIX Order message by setting field name values.

```
Dim aMessage As Object
        Set aMessage = CreateObject("JavTech.FIXMessage")
        aMessage.addByFieldName "BeginString", "FIX.4.1"
        aMessage.addByFieldName "MsgType", "D"
        aMessage.addByFieldName "SenderCompID", SenderCompId
        aMessage.addByFieldName "SenderSubID", SenderSubId
        aMessage.addByFieldName "TargetCompID", TargetCompId
        aMessage.addByFieldName "TargetSubID", TargetSubId
        aMessage.addByFieldName "Symbol", "SUNW"
        aMessage.addByFieldName "Side", "2"
        aMessage.addByFieldName "OrderQty", "1000"
        aMessage.addByFieldName "HandlInst", "1"
        aMessage.addByFieldName "OrdType", "1"
        aMessage.addByFieldName "ClOrdID", j + 1
        aMessage.addByFieldName "TimeInForce", "0"
```

2.5.7.3.2.4 Post a message

This method shows how to request Coppelia to send a message.

```
Dim returnCode As Integer
returnCode = CoppeliaSession.post(aMessage.toString())
```

2.5.7.3.2.5 Get a FIXMessage

This example shows how to get the next FIX message.

```
   Dim m as String
   '=====================================================
   ' use getAll to get the next incoming fix message
   '=====================================================
   m = CoppeliaSession.getAll
   If m <> "" Then
     '=====================================================
' if there's a message, then call fromString to
' populate FIX Message
     '=====================================================
     Set fm = CreateObject("JavTech.FIXMessage")
     fm.fromString m
  End If
```

2.5.7.3.2.6 Get a FIXMessage

This example shows how to get a FIX message

```
       Dim m as String
   '=====================================================
   ' use get to get the next incoming fix message from MLY
   ' for sub id "JOE"
  '=====================================================
   m = CoppeliaSession.get "MLY","JOE"
   If m <> "" Then
     '=====================================================
     ' if there's a message, then call fromString to
' populate FIX Message
'=====================================================
     Set fm = CreateObject("JavTech.FIXMessage")
     fm.fromString m
  End If
```

2.5.7.3.2.7 Get fields from FIXMessage

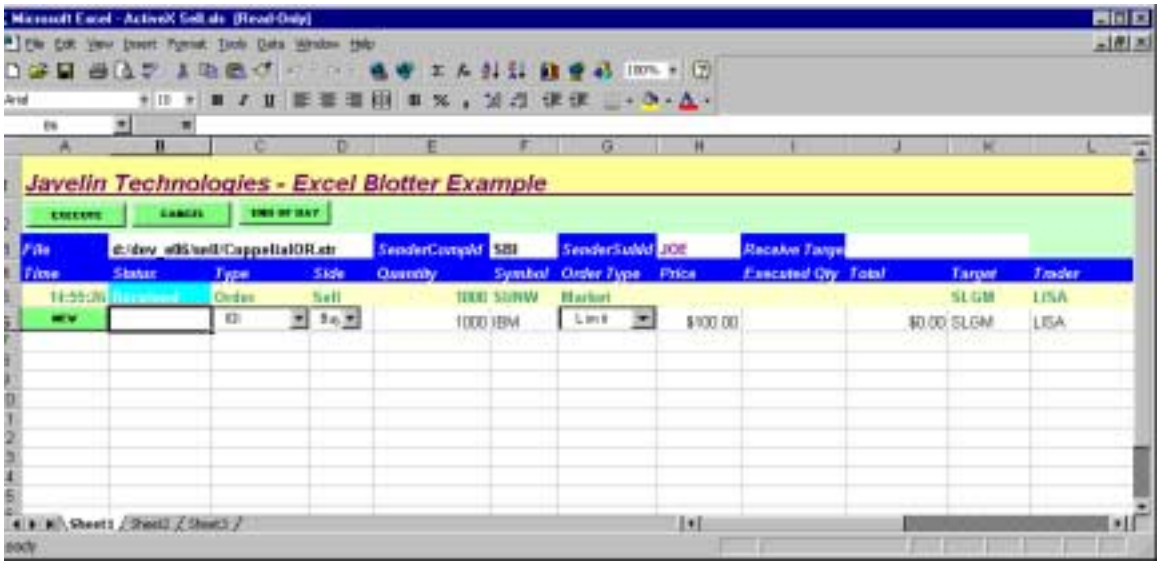
This example shows how to search for a particular field (by tag number) in a FIX message.

```
ClientOrderId = fm.getByTag(11)
```

You can also search a field by FIX 4.1 field name:

```
ClientOrderId = fm.getByFieldName("ClOrdID")
```

2.5.7.3.3 Excel Example



An Excel VBA blotter example is provided in zip file format. The ActiveX Sell.xls example demonstrates a how a simple application can be built in Excel using the Coppelia ActiveX Controls.

The application provides a blotter screen in Excel that can be set as either a buy side or sell side application to send and receive messages from Coppelia.

2.5.7.4 Visual Basic Example Complete Source Code

The following is the complete source of the Visual Basic Example:

```
Private Sub Form_Load()
Session_Init
Send_Order
End Sub

Sub Session_Init()
Dim iorString As String
On Error GoTo ErrorHandler:
Open "D:\testing\Coppelia\buy\CoppeliaIOR.str" For Input As #1
Line Input #1, iorString
Set CoppeliaSession = CreateObject("JavTech.CoppeliaSrv")
CoppeliaSession.Initialize iorString
Close #1
End Sub

Sub Send_Order()

Dim aMessage As Object
On Error GoTo CreateErrorHandler
Set aMessage = CreateObject("JavTech.FIXMessage")
'================================================
' build a FIX Message by setting values for tags
'================================================
aMessage.addByTag 35, "D"               'MsgType
aMessage.addByTag 49, "SLGM"
aMessage.addByTag 50, "SenderSubID"
aMessage.addByTag 56, "SBI"
aMessage.addByTag 57, "ActiveX Order test"
aMessage.addByTag 55, "MSFT"
aMessage.addByTag 40, "1"           'OrdType

aMessage.addByTag 21, "1"
aMessage.addByTag 11, "Client ID 1"
aMessage.addByTag 38, "1000"
aMessage.addByTag 54, "1"           'Side


Dim returnCode As Integer

returnCode = CoppeliaSession.post(aMessage.toString())

Exit Sub

CreateErrorHandler:
    MsgBox "ERROR creating Order Object: " & Error(Err)

'Exit Sub

ErrorHandler:
    MsgBox "ERROR in Session_Init: " & Error(Err)
End Sub
```

2.5.7.5 List of Dependencies

The Coppelia ActiveX Controls depend on the following DLLS:

```
msvcp60d.dll    6.00.8168.0
msvcrt.dll      6.00.8397.0
msvcrtd.dll     6.00.8447.0
msvcirt.dll     6.00.8168.0
msvcirtd.dll    6.00.8168.0
msvcp60.dll     6.00.8168.0
atl.dll         3.00.8449
```

The versions specified are those of the DLLs included in the install.

2.5.8 Advanced Programming

2.5.8.1 User-defined Fields

(Section TBA)

2.5.8.2 User-defined Messages

2.5.8.2.1 Introduction

The Coppelia server is capable of supporting new, user-defined message types. The new message types are defined by the user in a simple text format, and run on top of a FIX session. This document explains how to use this feature in Coppelia from the user's point of view. In order to run this feature, both communicating parties need to run the Coppelia servers, since other engine may not be aware of the user defined custom messages. Therefore, the user needs to enable the custom message parameter in both sides of the FIX connection.

This feature is only supported in FIX 4.0 and later. Earlier versions are not supported.

Follow these easy steps to enable your Coppelia server:

- Define new fields and new messages in a specific format in a text file. This format may be in XML format in the future release.
- Generate the java source files for the user defined custom messages using the utility provided in the Coppelia package.
- Compile the java source files. This is done by the user, and requires the user site to have the JDK environment. Coppelia will not compile the generated source files because the user environments are different.
- Package the generated byte code or .class files using the java jar utility and include it into the classpath of Coppelia environment.
- Declare the location and name of the text file to Coppelia in the .dat file using the parameter USER_DEFINED_FILE, to enable the custom message capability in the Coppelia server.

2.5.8.2.2 Defining new fields and messages

Before you can start defining new fields and new message types, the following items need to be considered:

- Fields and message types that are already defined inside Coppelia. You must make sure there are no duplicate descriptions and / or definitions. In addition to that,
- You need to know what the format of each field you are defining will be

Field numbers that are NOT available for user defined fields

- Field numbers 1 to 5000 are reserved for FIX.
- Field numbers 9500-9599 are used by NQB.
- Field numbers 7800-7900 are used internally by Coppelia.

To avoid any conflict, you should use any field number outside of these ranges. Javelin Technologies, Inc, does not know all custom field definitions, therefore, we cannot guarantee that you will not re-use a field tag number that already exists somewhere. Please check with all your counter parties. In addition, you might want to consult the FIX Protocol Website at

http://www.fixprotocol.org/cgi-bin/rbox/Fields.cgi?menu=51

or select Tech Resources, and then Custom Fields on the FIX website. We encourage every FIX user to register their custom fields on the FIX website.

Message types that are NOT available for user defined messages:

- 0-9
- A-Z
- U1-U9
- COA

Any other message type is acceptable.

Supported Formats:

a)	FORMAT_INTEGER	=	0
b)	FORMAT_FLOAT	=	1
c)	FORMAT_STRING	=	2
d)	FORMAT_DATE	=	3
e)	FORMAT_TIME	=	4

Text File Format:

The format of the text file you define for new fields and new message types is as follows:

```
[Field]
fieldNum, fieldName, format
fieldNum, fieldName, format
fieldNum, fieldName, format
.
.
```

Example:

```
[Field]
8001, Name, 2
8002, Age, 0
8003, CashInPocket, 1
8004, DateOfBirth, 3
8005, TimeOfBirth, 4
8006, MothersName, 2
8007, FathersName, 2
8008, MothersAge, 0
8009, HourlyWage, 1
```

Please try to use existing FIX fields or tag numbers wherever possible.

New Message Type Definition Format:

```
[Message]
:MsgType
className
fields
required fields

:MsgType
className
fields
required fields
.
.
```

Example:

```
[Message]
:Per
Person1
8001, 8002, 8005, 8008, 100, 55, 22
8001, 8008, 22

:Frnd
Friend2
8003, 8004, 8006, 8007, 109, 11, 33
8003, 11, 33
```

Save the file as a regular text file, for example external.txt. This file will be used to generate the java source code and as the value for the USER_DEFINED_FILE flag in .dat file. The complete example would look like the following (in the same file named external.txt):

```
[Field]
8001, Name, 2
8002, Age, 0
8003, CashInPocket, 1
8004, DateOfBirth, 3
8005, TimeOfBirth, 4
8006, MothersName, 2
8007, FathersName, 2
8008, MothersAge, 0
8009, HourlyWage, 1

[Message]
:Per
Person1
8001, 8002, 8005, 8008, 100, 55, 22
8001, 8008, 22

:Frnd
Friend2
8003, 8004, 8006, 8007, 109, 11, 33
8003, 11, 33
```

2.5.8.2.3 Generating java source files from the text file

Once the user defined fields and message types are properly formatted in the text file, you can run a utility to generate the java source files from it. In order to do that, you need to have a JDK environment installed on your machine. You also need to include coppelia.jar into your jdk classpath environment. For example, in NT, you would include coppelia.jar file into the classpath environment as follows:

```
SET CLASSPATH=%CLASSPATH%;d:\coppelia\classes\coppelia.jar
```

assuming coppelia.jar file is installed in D:\coppelia\classes of your local machine.

To generate the java source file:

```
java com.javtech.coppelia.utils.MsgGenerator external.txt
```

This assumes that the text file is at the current directory and you want the output java source file to be generated to be in the current directory as well.

The above will create a file with a .java extension, depending on the name of the message that has been created. If a message classname called OrderReport was created, the resulting name of the java file will be OrderReport.java. Note that the name of the java source file generated is based on the classname attribute.

2.5.8.2.4 Compiling the java source files

At any time, you can open the generated java source file using any text editor to view it. DO NOT make any changes to the generated source files. Any changes made will not be guaranteed to function as expected.

To compile the java source file, you need to have a JDK development environment Assuming all the generated java source files are in the current directory, to compile, do the following:

```
javac –d . *.java
```

The –d option indicates that the directory structure for the class files will remain intact, and will be the same as all the other Coppelia class files. Currently all Coppelia class files are packaged with a directory structure of /com/javtech/Coppelia. When you compile the java file, that directory structure will be created off of the directory that you are currently in, and the compiled .class file will be located in /com/javtech/Coppelia.

The "." indicates that the directory path will be built off of the current directory.

This will generate java byte code or *.class files.

2.5.8.2.5 Packaging the *.class file into a jar file

Once the java source files are compiled successfully, you will need to package them using the jar utility. Assuming you are in the directory where all the *.class files are:

```
jar –cvf0 user.jar *.*
```

This will create a user.jar file that contains all the *.class files.

You need to include this jar file into the classpath in where Coppelia will run. To do that in NT,

```
SET CLASSPATH=%CLASSPATH%;d:\coppelia\classes\user.jar
```

assuming you put user.jar in the coppelia installation directory.

2.5.8.2.6 Adding the USER_DEFINED_FILE parameter in the .dat file

Finally, you need to alter your .dat file to include the following flag:

```
USER_DEFINED_FILE   external.txt
```

assuming the external.txt is in the same directory of the .dat file. If it is not in the same directory, you can specify the path for this directory – i.e.

```
D:\Coppelia\classes\external.txt
```

Note to NT Users: Make sure that none of the directory level names have a space in their names, otherwise the file cannot be read. For example, a directory that ***will not work*** is:

```
D:\Coppelia\User Defined\external.txt
```

To successfully send and receive messages between two or more Coppelia servers, this parameter has to be set in every .dat file that is being used. Every server also needs to have its classpath correctly defined with the location of the jar file that was created with the User Defined Message classes.

If you have done all the above steps, you can start running Coppelia with the new message types you defined. This feature is not supported in the CORBA interface at the moment and is demonstrated using the Observer/Observable interface in the example file named PassThrough.java.

2.7 Troubleshooting

2.7.1 Troubleshooting Introduction

Javelin's Coppelia development effort is focused on ease of use and implementation into numerous platforms and middlewares. Coppelia is a robust application with many dynamic parts, all of which must be working correctly in order for Coppelia to function. As with any engine, Coppelia is not an exception to this rule, all the moving parts must be in order, or the engine will fail or not function optimally.

The key to troubleshooting Coppelia is to be fully aware of all the moving parts – know what each part does and how it relates to successful Coppelia operation. All parts must be in order for Coppelia to start – if there is a file missing from the classpath, the .dat file is missing, the .dat file is misconfigured, and many other potential problems – Coppelia will not start.

Usually, the error messages and prompts are informative enough to give the user some guidance into what causes the problem. However, the answers are not always obvious.

Some of the most common Coppelia problems are explained here, and hopefully the examples here will offer the user some clear ideas on what to look for when problems arise.

2.7.2 Troubleshooting at Startup

The evaluation copy of the Coppelia FIX engine is ready to run after it is installed. However, making changes to the go_buy or go_sell files, or changes to the sell.dat or buy.dat files can cause Coppelia to fail to start.

The most common source of problems at startup is due to the Classpath setting in the go_buy and go_sell files. The Classpath lets Coppelia know where to find all the parts of the software. Not just the Coppelia software itself, but its subsets, including OrbixWeb, the database (usually PSE PRO), and a few other necessities. There are many potential mistakes that can be made in the classpath – if a necessary jar file's name is mispelled, or a jar file is missing, or the location of the jar file is incorrect – all these can lead to Coppelia failing to start.

Classpath errors usually cause errors that begin with: Class not found. For some reason the Coppelia engine could not find a necessary file.

Other sources of errors at startup are due to problems in the dat files. Often times a misconfigured .dat file will cause Coppelia to fail to start.

Problems are also caused by misconfiguration due to Java 1.2. Running Coppelia with Java 1.2 requires additional parameters to be set.

2.7.2.1 Example #1 – Class not found: Coppelia

A Coppelia user with this classpath:

```
SET CLASSPATH=..\classes\pro.zip;..\classes\mct3_0.zip;
..\classes\rogue.zip;..\classes\OrbixWeb31c.jar;..\lib;
```

sees this error upon Startup:

```
     Class not found: Coppelia
```

Which is caused by a missing link to the coppelia.jar file in the classpath.

2.7.2.2 Example #2 – NoClassDefFoundError: OrbixWeb

A Coppelia user with this classpath:

```
SET CLASSPATH=..\classes\pro.zip;..\classes\coppelia.jar;
..\classes\mct3_0.zip;..\classes\rogue.zip;..\classes\OrbxW
eb31c.jar;
..\lib;
```

sees this error upon startup

```
java.lang.NoClassDefFoundError: IE/Iona/OrbixWeb/CORBA/ORB
at
at com.javtech.coppelia.Interface.run(Interface.java:73)
```

Again, there appears to be a class missing, judging by the error it is the OrbixWeb class – which is in the classpath. After a closer look, the link to the OrbixWeb jar file is misspelled in the classpath – notice the missing 'i' in OrbxWeb31c.jar.

2.7.2.3 Example #3 – Failure to create CORBA implementation object

When starting up a Coppelia, a user sees an error:

```
Failed to create CORBA implementation object:
org.omg.CORBA.COMM_FAILURE: Communication failure
    select error
    Reason: (unknown)
Verify that another copy of Coppelia is not running.
System exiting. org.omg.CO
RBA.COMM_FAILURE: Communication failure
    select error
    Reason: (unknown)
```

org.omg.CORBA.COMM_FAILURE: Communication failure

This error is caused when two different Coppelia engines are running on the same machine and have the same IIOP Port. IIOP Ports must be unique among Coppelia engines. This can also be caused if the port is being used by another application (not necessarily Coppelia) Check available ports by using the

```
netstat -n
```

command.

2.7.2.4 Example #4 - Solaris and Java 1.1 problems - dirname: not found

A user with a client program on a Solaris machine that uses Java 1.1, and has this PATH

export PATH=/usr/java1.1/bin

but gets this error upon start up

```
%39 quad>run_receive
:../classes/coppelia.jar:../classes/OrbixWeb31c.jar:/usr/ja
va1.1/lib:.
/usr/java1.1/bin/java[15]: dirname:  not found
/usr/java1.1/bin/java[16]: basename:  not found
/usr/java1.1/bin/java[65]: test: argument expected
 was not found in
/usr/java1.1/bin/../bin/sparc/native_threads/
```

This is caused when the **/usr/bin** directory is not in the PATH setting. Please make sure that the PATH setting looks like this:

export PATH=/usr/java1.1/bin:/usr/bin

2.7.3 Network Connectivity Troubleshooting

Another very common cause of trouble is connecting a Coppelia engine to another FIX engine.

There are a few simple principals to remember when connecting a Coppelia engine to a remote engine. This does not include FIX configurations (Firm IDs, FIX versions, etc – those will be discussed in the next section)

#1 – Know the remote party's IP address

Without the proper IP address, Coppelia will not be able to connect.

#2 – Know the remote party's Port Number

The remote party will be listening on a specific port number for connections. Make sure that you have the correct port number. As a reference, for incoming connections to a Coppelia server, they will be coming in through the Local Port.

#3 – Configure the ID line properly

Even if the Coppelia engine that you are running is receiving connections, the IP address and Port Number must be set. The IP address must be set to the proper IP address of the counterparty, regardless if they are connecting or if they are being connected to (unless NO_IP_CHECK) is used.

If a counterparty is doing the connection, the Port Number can be set to any value, as the value is irrelevant, but there must be a value in the ID line for Port Number.

#4 – Do not confuse Local Port with IIOP Port

Remember that Local Port is the port a counterparty will connect to you with. Local Port is used for external Coppelia communications over IP.

IIOP Port is used for internal Coppelia communications and for communications with a client program (a SendOrder program will run over the IIOP Port)

However, there are still some common problems when trying to connect to remote counterparty's using Coppelia, and here are some examples.

2.7.3.1 Example #1

After typing "connect" and an ID in the Coppelia window, a user receives this message:

```
:)connect SBI
Trying connection to SBI...

:)07-Jan-00 5:37:43 PM: Comm: Connecting to SBI...
07-Jan-00 5:37:43 PM: FIXCommConnection: Trying Client
socket, net address 192.168.129.25 port 9876
javtech dbg - Socket creation fails : Connection refused :
FIXCommConnection. Validate counter party IP and port.
```

The message "socket creation fails" means that the remote party was not available for a connection. Either the remote party is not up, or the wrong IP address or port number was used.

Also, make sure that the counterparty is configured to listen for connections. If the counterparty is also a Coppelia server, but is configured as a Client, they will not be listening for connections.

2.7.3.2 Example #2

A User is waiting for a connection from a counterparty, and this message appears in the Coppelia window:

```
:)07-Jan-00 5:40:40 PM: CommServer: Connection from an
unspecified host : egils1/192.168.129.25 rejected.
```

This error is caused when a connection is coming in from an IP address that is different from the one in the ID line. Make sure you have the correct IP address for the counterparty.

2.7.3.3 Example #3

After a Coppelia session has successfully connected, and has been up for a while, if a user sees a message like this:

```
:)07-Jan-00 6:01:50 PM: CommStuff: Coppelia-FIX received
disconnect, FIXCommConnection
07-Jan-00 6:01:50 PM: CommStuff: Connection to SBI is down
```

This indicates an abnormal disconnect. The remote side went down unexpectedly and without sending a FIX logout message.

A proper disconnect message would look like:

```
07-Jan-00 6:03:47 PM: CommStuff: Disconnecting SBI
07-Jan-00 6:03:47 PM: CommStuff: Connection to SBI is down
```

2.7.4 FIX Connectivity Troubleshooting

Another source of many potential errors is an improperly configured FIX session. Again, there are some certain rules that apply for these kinds of errors:

#1 – Make sure your IDs are correct

This is very important – your Target Firm ID and Local Firm ID must match with what your counterparty has – an incorrect ID of any kind will cause a FIX disconnect.

#2 – Make sure your FIX versions are the same

This must be agreed upon prior to connectivity – Coppelia will not allow a FIX engine running FIX 4.0 to communicate with a Coppelia engine running FIX 4.1.

#3 – Make sure that your Heartbeats are at the same interval

Having the heartbeat interval at different intervals will not cause any errors, but will cause extra network traffic as the side with the shorter interval will send many extra FIX Test Request messages, thereby causing unnecessary load on the Coppelia Server.

#4 – Make sure the sequence numbers for the connection match up

This is also important – misconfigured sequence numbers can cause at the worst dropped connections and at the best excess network traffic.

#5 - If you are testing multiple Coppelia connections on two Coppelia engines running on the same machine, be sure to use

NO_SERVER_CHECK ON

on the side that is receiving connections.

This is because having multiple connections between two Coppelia engines on the same machine often causes confusion within the Coppelia engine.

A quick review of the FIX sequence number rules:

a – if a connection comes in with a sequence number higher than expected, Coppelia will issue a resend request and reset sequence numbers accordingly

b – if a connection comes in with a sequence number lower than expected, Coppelia will drop the connection and not reset any sequence numbers. This will require manual intervention!

2.7.4.1 Example #1

When trying to connect to a remote ID JAVTECH, a user receives this message:

```
:)connect JAVTECH
Trying connection to JAVTECH...

:)07-Jan-00 6:18:52 PM: Comm: Connecting to JAVTECH...
07-Jan-00 6:18:52 PM: FIXCommConnection: Trying Client
socket, net address 192.168.129.25 port 9876
07-Jan-00 6:18:52 PM: CommStuff: Connection to JAVTECH accepted
07-Jan-00 6:18:52 PM: MainThread: Initialized Interface
thread
07-Jan-00 6:18:52 PM: Pump: Remote id SBI did not match
expected JAVTECH
07-Jan-00 6:18:54 PM: CommStuff: Disconnecting JAVTECH
07-Jan-00 6:18:54 PM: CommStuff: Connection to JAVTECH is
down
```

This message indicates that the remote side's Firm ID was SBI, though the user tried connecting with the ID JAVTECH. The user must change the TargetCompID to SBI in the ID line in the .dat file.

2.7.4.2 Example #2

A user receives a connection from a counterparty, but sees this message:

```
:)07-Jan-00 6:25:04 PM: CommServer: Connected to SLGM
07-Jan-00 6:25:04 PM: CommStuff: Connection to SLGM
accepted
07-Jan-00 6:25:04 PM: CommServer: Connection to egils1/192.168.129.25
established.
07-Jan-00 6:25:04 PM: CommServer: Waiting for connection
07-Jan-00 6:25:04 PM: Pump: Remote id JAVTECH did not match
expected SLGM
07-Jan-00 6:25:06 PM: CommStuff: Disconnecting SLGM
07-Jan-00 6:25:06 PM: CommStuff: Connection to SLGM is down
07-Jan-00 6:25:06 PM: CommStuff: SLGM is not connected
```

This indicates that the user received a connection from a user that had a Firm ID of JAVTECH, but the user was expecting a Firm ID of SLGM.

The user must change the SenderCompID in the ID line in the .dat file to SLGM to connect successfully.

2.7.4.3 Example #2

When connecting to a counterparty, a user sees a message

```
:)connect SBI
07-Jan-00 6:28:37 PM: Comm: Connecting to SBI...
Trying connection to SBI...

:)07-Jan-00 6:28:37 PM: FIXCommConnection: Trying Client
socket, net address 192.168.129.25 port 9876
07-Jan-00 6:28:37 PM: CommStuff: Connection to SBI accepted
07-Jan-00 6:28:38 PM: Pump: Wrong FIX version for remote id
SBI did not match expected version 410
07-Jan-00 6:28:39 PM: MainThread: Initialized Interface
thread
07-Jan-00 6:28:40 PM: CommStuff: Disconnecting SBI
07-Jan-00 6:28:40 PM: CommStuff: Connection to SBI is down
```

This indicates that the two parties are expecting different FIX versions. The user expects version 4.1, but the target firm is expecting something else. Verify FIX versions between the two parties.

2.7.4.3 Example #3

When a counterparty connects to you, and you see this message:

```
21-Mar-00 1:26:34 PM: CommServer: Connected to SLGM2
21-Mar-00 1:26:34 PM: CommStuff: Connection to SLGM2
accepted
21-Mar-00 1:26:34 PM: CommServer: Connection to
egils1/192.168.129.25 established.
21-Mar-00 1:26:34 PM: CommServer: Waiting for connection
21-Mar-00 1:26:34 PM: Pump: Remote id SLGM did not match
expected SLGM2
21-Mar-00 1:26:36 PM: MainThread: Initialized Interface
thread
21-Mar-00 1:26:36 PM: CommStuff: Disconnecting SLGM2
21-Mar-00 1:26:36 PM: CommStuff: Connection to SLGM2 is
down
21-Mar-00 1:26:36 PM: CommStuff: SLGM2 is not connected
```

and you have these ID lines in your sell .dat

```
ID; SLGM; SBI; GEORGE; 192.168.129.27;5200; Seligman Funds;
Tech Support (212) 555-1212;30;0;401

ID; SLGM2; SBI2; GEORGE; 192.168.129.25;5200; Seligman
Funds; Tech Support (212) 555-1212;30;0;401
```

This means that your Coppelia Engine is getting confused about the incoming connections.

The solution to this problem is to add this to this parameter to the side that is receiving connections:

NO_SERVER_CHECK ON

This will prevent the different connections from getting confused.

3.0 FIXionary

While working on our implementation of a FIX engine, it quickly became apparent that flipping through the pages of four versions of the FIX protocol was not an efficient way to look up specific information. In addition, there is no easy way to cross-reference the information contained in four or more different documents of considerable size.

In realization of these facts, the idea for FIXionary was born. FIXionary is an online FIX Dictionary or reference, available at no charge on Javelin Technologies' website as part of Javelin's dedication to educate and promote FIX, or, for local installation on your machine, as a floppy disk.

FIXionary allows a user to easily look up the corresponding values for each tag, by number, or by tag name. In addition, a user can choose which version of FIX to look at, or cross-reference information across message types and FIX versions. FIXionary is part of the Coppelia tool kit provided to our clients and evaluators for free to help them understand and implement FIX.

http://javtech.com/

4.0 FIXometer User's Guide

4.1 Introduction

This section of the document details the usage of FIXometer version 3.01. The FIXometer is the network monitor used to remotely affect the Coppelia Server process. It functions to gather statistics(such as the number of Orders or Rejects), check line status, manipulate sequence numbers, run end of day, and remotely connect or disconnect. The FIXometer can also be used to view multiple intances of Coppelia.

4.2 Installing FIXometer

The FIXometer is automatically installed with the Coppelia software package, or it can be downloaded from our web site at http://javtech.com/FIXometer.zip . The initialization file is place in the directory – coppelia\FIXometer, and the class file, fixometer.jar, is copied into coppelia\classes.

Also required for FIXometer 3.01 is OrbixWeb 3.1. The necessary file can be downloadedOrbixWeb31c.jar, which can also be downloaded from our web site at http://javtech.com/OrbixWeb31c.jar .

4.3 Configuring FIXometer

The FIXometer can be configured either of two ways, by copying the appropriate RemoteUIIOR.str that is generated by Coppelia into the FIXometer directory or by creating a remote.dat file in the FIXometer directory, which is described in section 3.2.

4.3.1 Copying RemoteUIIOR.str

The RemoteUIIOR.str file contains connection information for a Coppelia server, and each instance of a Coppelia server will have one. This file needs to be copied into the coppelia/FIXometer directory. This can be done manually or by creating a line in the FIXometer.bat file such as:

```
copy ..\buy\RemoteUIIOR.str .
```

The disadvantage to using this method is that it only allows you to monitor connections to and from this single location. Using the remote.dat method allows you to monitor connections from completely separate Coppelias.

4.3.2 Remote.dat file

The other method to configure the FIXometer is by creating a remote.dat file in the coppelia/FIXometer directory. This method is far more powerful than the RemoteUIIOR.str method in that with creating a remote.dat file, the user can monitor multiple and separate Coppelia servers.

To use this method, create a file called remote.dat in the Coppelia\FIXometer directory only containing lines that have this information:

```
Server name          IP Address          IIOP Port

For example:

BUY_SIDE             127.0.0.1           1234
```

with each line detaling an instance of Coppelia you wish to monitor.

A line beginning with # indicates a comment line – these lines will be ignored by FIXometer.

4.4 Running and Using FIXometer

You can use the FIXometer to remotely monitor Coppelia servers.

To start using FIXometer, run the FIXometer.bat file.

In the FIXometer screen, the fields that are monitored for each connection are as follows:

Field	Description
#	Identification of the connection
TargetCompID	ID of the target company
SenderCompID	ID of the sending company
Status	Status of the connection – either UP or DOWN. ????? indicates that information about this connection is unavailable (most likely caused by the Coppelia instance not running at the time)
Description	**Description of the connection – taken from the Description field in the Vendor Identification line in the buy.dat file**
Contact	Contact name and number for the connection – taken from the Contact field in the Vendor Identification line in the buy.dat file.
Net Addr	The IP address of the target machine

Port	The port number of the target computer
Version	FIX version being used by the connection
Seq_Num_In	The incoming message sequence number

Seq_Num_Out	The outgoing message sequence number
Msgs_In	Number of messages received
Msgs_Out	Number of messages sent out
Bytes_In	Number of bytes received
Bytes_Out	Number of bytes sent out
Last_In	Time the last message was received
Last_Out	Time the last message was sent
Rejects	Number of rejected messages sent and received
Orders	Number of orders received sent and received
Exec_Acks	Number of execution acknowledgements sent and received
Executions	Number of executions sent and received
Cancels	Number of cancels sent and received
Indications	Number of indication of interests sent and received
Allocations	Number of allocations sent and received

Totals are also summed up for the following fields: Msgs_In, Msgs_Out, Bytes_In, Bytes_Out, Rejects, Orders, Exec_Acks, Executions, Cancels, Indications, and Allocations.

FIXometer also allows the sorting of columns. Clicking on the column name will cause the column to be sorted, from greatest to least. An asterisk will indicate the curenntly active, sorted column.

4.5 Menu Options

FIXometer also has the following menu options.

Under File –

Exit – exits the FIXometer (does not disconnect an existing connection)

Under Options –

Connect/Disconnect – allows a user to make or break a connection to a server.

Send Test Request – allows a user to send a test request

Send Sequence Reset – allows a user to send a sequence reset message. The user is asked to type in the new sequence number, and select the server which to send the reset message to.

Reset Incoming Sequence – allows the user to reset the incoming sequence number. The user is asked to enter in the new sequence number that incoming messages should be reset to.

Reset Outgoing Sequence - allows the user to reset the outgoing sequence number. The user is asked to enter in the new sequence number that outgoing messages should be reset to.

Run End of Day (EOD) – allows the user to run the End of Day command on a connection. The user selects the connection to run the process on, and then either clicks on the Process button to run End of Day or clicks on Cancel to cancel the operation.

Download File – allows the user to download a file from any target server being monitored to the local FIXometer directory.

Upload File – allows user to upload file from any location to any target server.

Re-configure – allows the user to re-read in the buy.dat configuration file, in case there have been changes in it since the FIXometer was first run.

4.6 Reconfiguring a Remote Coppelia

FIXometer allows a user to reconfigure a remote Coppelia server by using the upload and download features. The process is as follows:

1 – First download the remote .dat file to be reconfigured
2 – Make changes in the file
3 – Upload it back to the remote site
4 – Using the re-configure menu option, re-read in the .dat file on the remote site

5.0 Coppelia Broker Simulator

5.1 Installation and Operation

The Coppelia Broker Simulator offers a quick and easy way to simulate order execution. The Simulator will automatically fill orders and send FIX Execution Reports in response. This is especially useful for testing purposes, as it can simulate a real-time trading environment. It is also useful for those developing order entry front ends. The Simulator also accepts FIX Cancel Request and FIX Cancel Replace messages.

To run the Broker Simulator, Coppelia must be installed on your computer. If you do not have Coppelia, you may obtain a copy from Javelin Technologies.

Download access to our website requires a password. A password can be obtained by contacting Javelin Technologies Support at (212) 422 6000, or by e-mailing support@javtech.com.

The Simulator software is packaged in the Coppelia product kit. The Simulator is also available separately, for those who would like to download the latest Simulator, but do not want to download the entire Coppelia package. It is available at http://www.javtech.com/downloads/Simulator.zip for downloading. Your browser will prompt you to save the file "Simulator.zip."

Extract the file with WinZip or any similar program. Make sure you click or set the option "Use Folder Names" in your un-zip program. Make sure the files get extracted in your Coppelia directory i.e. C:\Coppelia, as the extraction process will extract files to the Coppelia\classes and Coppelia\simulator directory. Five files will be extracted.

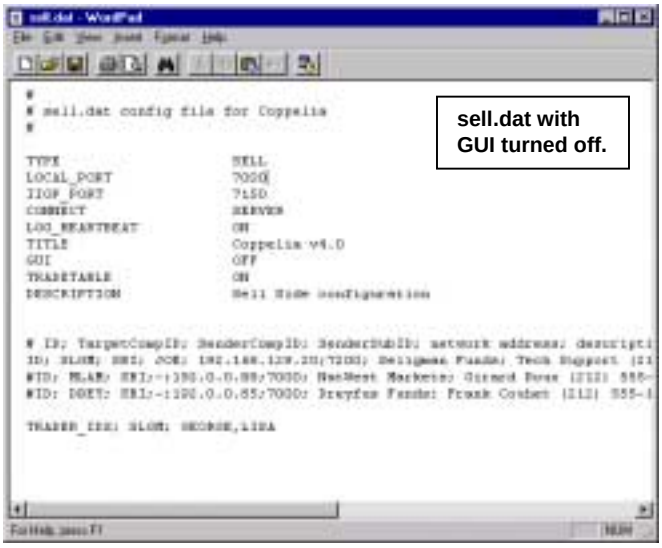
- Swingall.jar
- Simulator.jar
- Simulator.bat – (batch file for Windows NT Users)
- Simulator_java12.bat (batch file for Java 1.2 users)
- Simulator.unix (batch file for UNIX users)

The extraction process will place the files in their appropriate directories (swingall.jar, Simulator.jar within C:\Coppelia\classes\, Simulator.bat in C:\Coppelia\Simulator\).
Make certain that these files are extracted properly and in the proper directories.

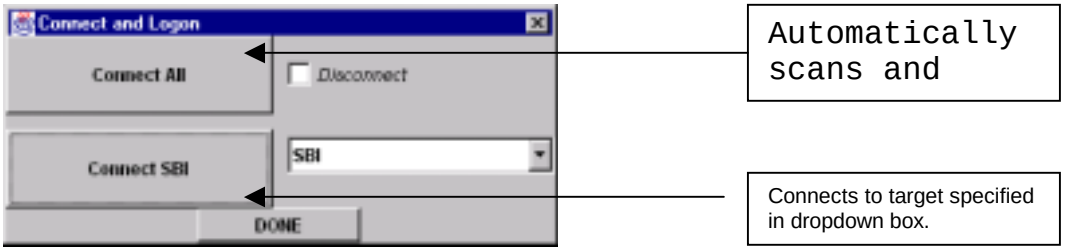
Important: Turn off the GUI or Blotter of Coppelia's Sell side.
To do this: edit the sell.dat file located in C:\Coppelia\Sell\sell.dat. Set the GUI parameter to OFF, save, then exit.

Please note: If the Coppelia Sell Side's GUI is NOT turned off, the Simulator will not be able to pick up any messages sent to that sell side Coppelia server. This is a direct result of the GUI's programmatic logic that orders received by the Sell side are automatically removed from the queue by the Sell Side GUI.

Run both buy side and sell side Coppelia by going to their respective directories and running their batch files (C:\Coppelia\Buy\go_buy.bat and C:\Copplia\Sell\go_sell.bat).

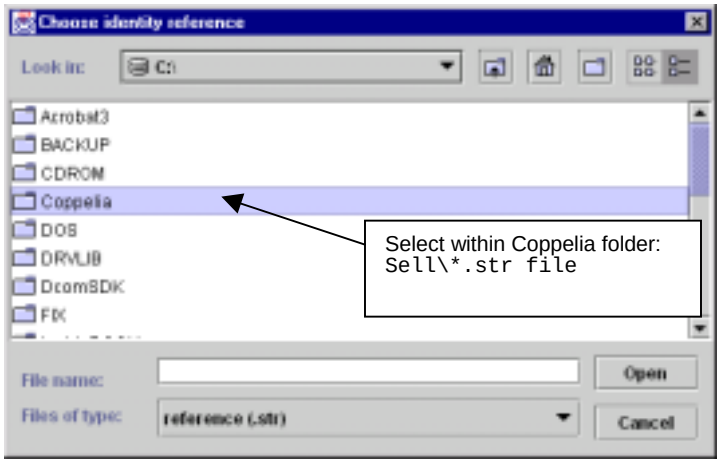


Click on the "Connect" button on the Buy side GUI to establish communication between the two servers. You may connect in two ways. YOU may click the "Connect All" button and it will scan for possible ID's to establish connection. In this example picture, specifying the Target ID of SBI will cause Coppelia to connect to this target.



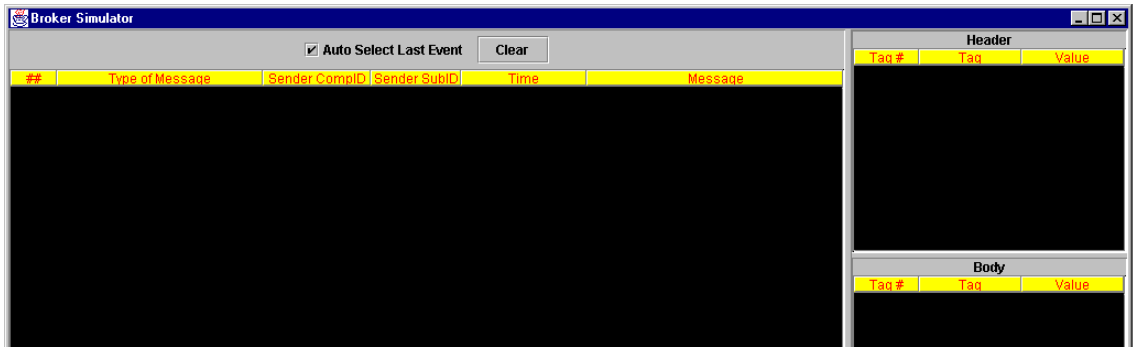
Note: Make sure the Disconnect checkbox is NOT checked! If the Disconnect box is checked, Coppelia will disconnect from the Target ID specified in the dropdown box.

Go into the Coppelia\Simulator directory and start the Simulator.bat by double-clicking on it. If a dialog box appears looking for a reference file or *.str file, navigate to the Coppelia\sell directory (as the Simulator simulates behavior of a Sell side). The simulator will be using this reference file to collect the data the sell side receives from the buy side and execute the orders.



You will see two files: CoppeliaIOR.str and RemoteIOR.str. Select the CoppeliaIOR.str file and click OPEN.

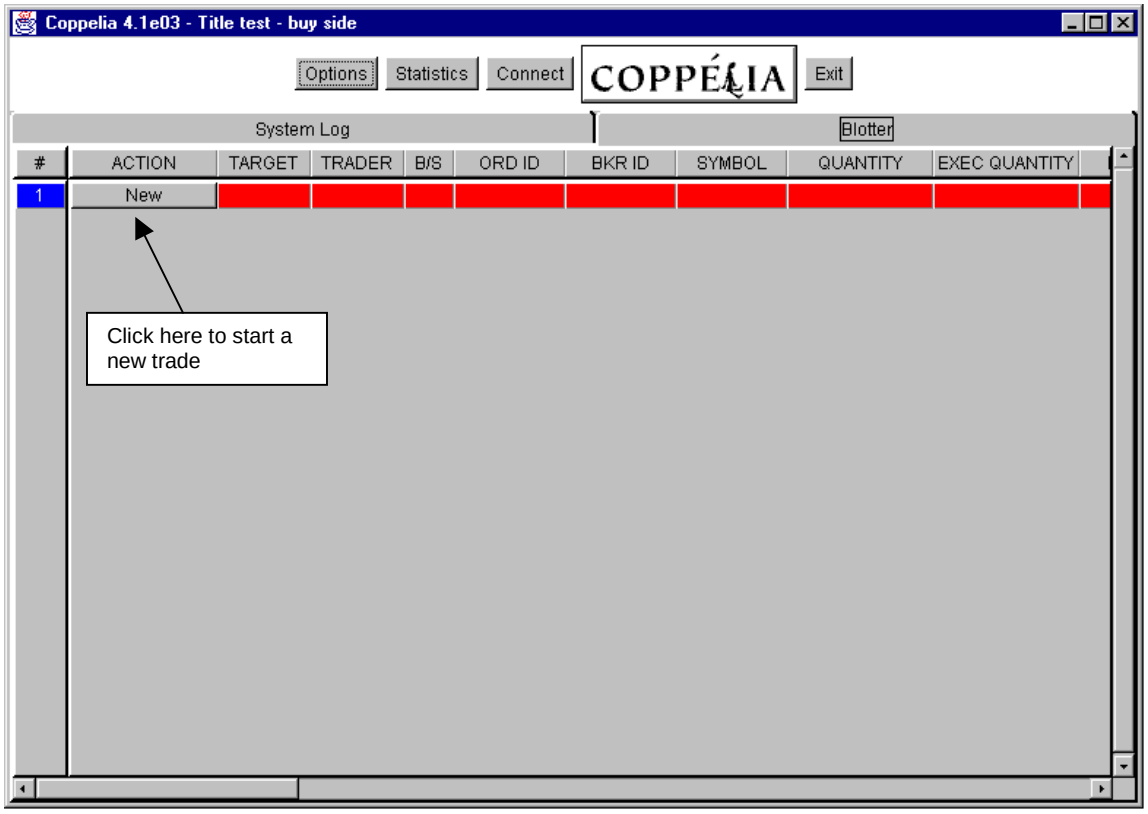
The Broker Simulator GUI will appear.



To avoid going through the process of selecting the CoppeliaIOR.str file using the dialog box, the location of the CoppeliaIOR.str file can be directly specified within the simulator batch file. The location must be specified at the end of the java execution line. For example, to specify to the Simulator that the CoppeliaIOR.str file to be opened is located in C:\coppelia\sell\, modify the java execution line (in a Windows NT environment) to have the following syntax:

```
jre -cp %CLASSPATH% Simulator
C:\Coppelia\sell\CoppeliaIOR.str
```

To now test the Simulator, send a sample order from the Buy Side GUI to the sell side. Go to the Buy Side GUI and click on the Blotter tab. and click on "New" within the "Action" column to start a trade.

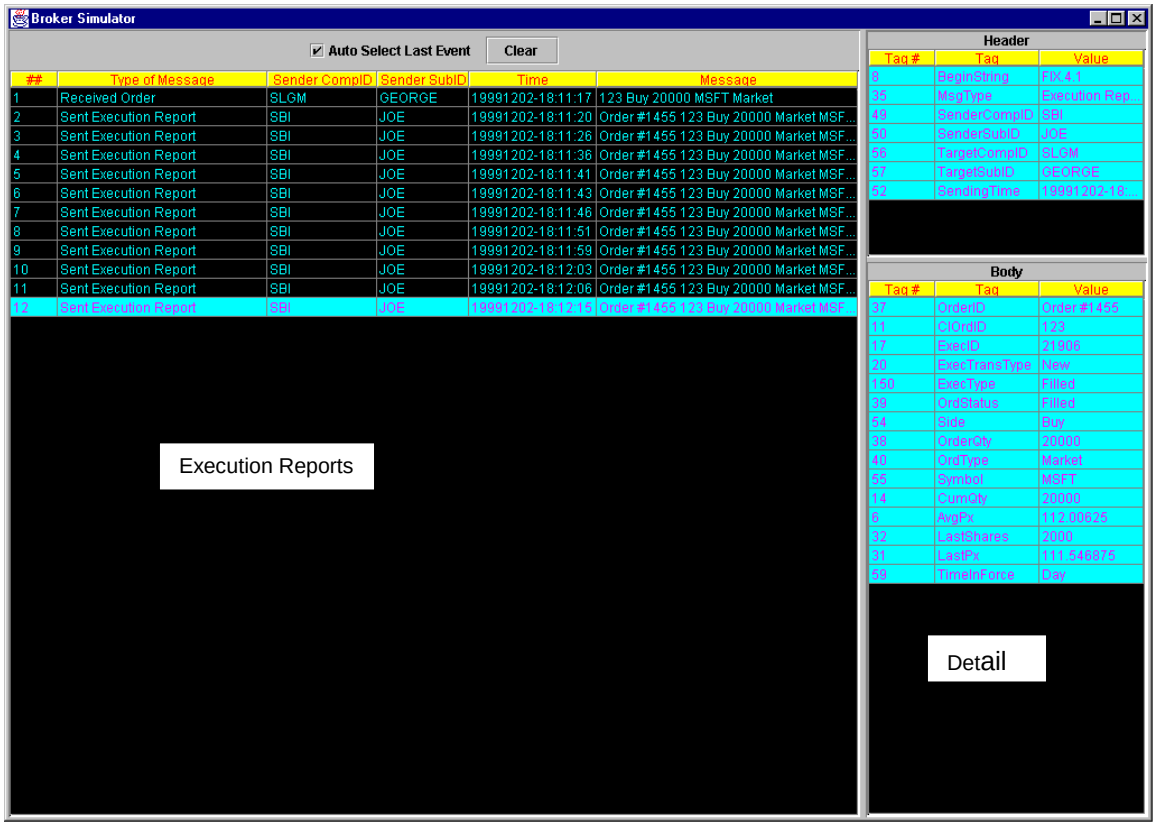


Enter the appropriate fields in the "Entry" dialog box.

Return to the Simulator GUI. The GUI should now begin displaying messages. The original FIX order message will be displayed as the first message. The Simulator will also begin filling the order that it received. The execution fill rate is determined this way: If the order was for a NASDAQ listed security, the Simulator will fill the order 2000 shares at a time – so, if the order was for 30,000 shares, there would 15 Execution Reports that fill 2000 shares each. The FIX Execution Reports will be sent every 1 to 2 seconds. If the symbol in the Order message is a NYSE listed security, the fill rate will be 3000 shares per execution every 1 to 5 seconds. Also, there is a limit of 5000 shares per order for NYSE stocks. If you put a stock that has a symbol of "XYZ", you will get a reject message.

Currently, the execution speed and execution share size cannot be configured.

To view any message in detail, click on the message in the main Simulator blotter, and the details are displayed on the right side of the GUI. The simulator is defaulted to highlight every new execution or trade it send/receives. To keep the highlight on one item, uncheck the "Auto Select Last Event" at the top of the window.



The simulator sends back at least two execution reports for each order received. This is standard behavior The first execution report is an acknowledgement. The second specifies be the first fill for the order – the first share quantity (2000 – NASDAQ, 3000 – NYSE) (or the full amount of shares of the order, if the order had a lower share quantity). The Simulator will continue filling the order until it is completed or the order is canceled.

The simulator also accepts FIX CancelRequest (outright cancellation of the order) and CancelReplace (modification of the order – change in number of shares, change of Price) messages. Be sure to have the correct OrigClOrdID set in the CancelRequest and CancelReplace message. The OrigClOrdID must match the ClOrdID of the original order for the Simulator to process orders correctly. To properly test this, you will need to use NASDAQ listed securities, or any security that has at least four characters in its symbol. As mentioned previously, NYSE listed stocks have a 5000 share minimum. With NASDAQ stocks, there is no limit on the number of shares in the order.

In case you may have trouble with the Simulator (i.e. Events stop before the order is filled), download the latest Coppelia class file archive (coppelia.jar) located at: http://www.javtech.com/downloads. For a password to this area of our wqebsite, please contact Javelin Technologies.

6.0 High Availability

6.1 Introduction

6.1.1 What is Coppelia HA?

Coppelia HA (for High Availability) was designed as an upgrade option to Javelin's Coppelia FIX Server technogy. Coppelia HA provides the following features over Coppelia:

6.1.2 Terms and Definitions

Users: Any external process that attaches to a CoppeliaHA engine.

Coppelia Cluster: Two or more CoppeliaHA engines working in unison on independent platforms to implement a highly available service.

Logical IP Address: A single address that represents a Coppelia Cluster.

6.1.3 Reference Documentation

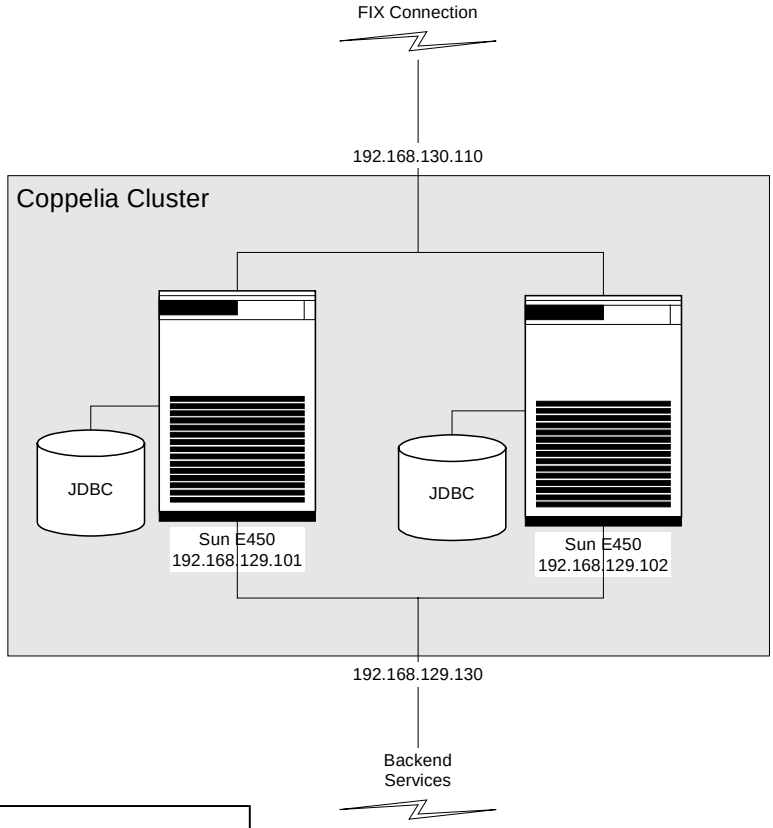
Coppelia User's Guide: Configuration Section
Attributes Section
Starting Coppelia

6.2 HA Concepts

6.2.1 Clustering (Logical Addresses)

The purpose of the Coppelia HA system is to present users with a single, highly available view of a Coppelia FIX server. This would shield users from any of the internal working of the system and any failure inside the cluster would simply result in a disconnection from the service followed by a reconnection. This is achieved by assigning a logical IP address for a Coppelia Cluster.

In the example provided we have two machines in the cluster. For simplicity, each machine is configured with two independent network cards that are internally connected to two different subnets. In a real world environment, it is recommended that each machine have four network cards, two for each segment.



Server Blocks

```
[BUY1]
server_block = [BUY1]
file_path=./buy1/
rmi_port = 9051
rmi_ip = 192.168.129.101

[BUY2]
server_block = [BUY2]
file_path=./buy2/
rmi_port = 9052
rmi_ip = 192.168.129.102

[HA_SERVERS]
HA_SERVER = [BUY1]
HA_SERVER = [BUY2]
WELL_KNOWN_ADDRESS = 192.168.129.001

[MAIN]
high_availability = [HA_SERVERS]
type = BUY
title = Coppelia v4.1, BUY SIDE
database = [DATABASE]
connect = CLIENT
local_port = 7200
connection = [SBI0]
interface_block = [INTERFACE]

[INTERFACE]
interface_type=INPROC
iiop_ip = 127.0.0.1
iiop_port = 8013

[DATABASE]

[SBI0]
net_address = 192.168.130.120
heartbeat_interval = 30
local_firm_id = SLGM0
remote_firm_id = SBI0
version = 411
contact = Benjamin Lai (212) 555-1212
user = ben
password = password
port = 9055
```

The important thing to note is that both the external FIX connections and the Backend Services have their own IP address to connect to the Coppelia HA service.

6.2.2 Coppelia HA Server Names

Each CoppeliaHA Server has a unique name, this not only helps to easily identify the server but it also used by the configuration file to distinguish attributes for each server allowing one configuration file to service n number of Coppelia HA engines. The example provided on the left is a typical configuration file for a pair of

Coppelia HA servers. The first thing to note is that all configuration attributes above the **MAIN** block are used to configure HA, anything below the **MAIN** block is used by a standard Coppelia engine excluding the **high_availability** attribute. In fact, you could delete the top part and create a valid config file for a standard Coppelia engine. Please see the Coppelia User's Guide for more configuration options. At the top of the file there are two blocks called **[BUY1]** and **[BUY2].** These represent the server blocks. On startup, Coppelia reads in the configuration file and copies all of the attributes in the server block to the **MAIN** block. This enables customers to configure server specific items in the server block.

6.3 Starting up Coppelia HA

To facilitate the above mentioned functionality, Coppelia requires two arguments at startup:

```
Coppelia <config file> <server name>
```

This methodology of startup has two major advantages over using one configuration file for each instance of the FIX engine:

- It guarantees that items such as connection data and server blocks are exactly the same across all servers.
- You only have to modify one file instead of n.

6.4 Inter-Process Communication

6.4.1 Java Remote Method Invocation (RMI)

Coppelia HA uses RMI to connect and communicate with other Coppelia HA servers in the cluster. Traditionally, Java applications that use RMI require an `rmregistry` to do the lookup and object binding. To reduce the chance of failure or errors, Coppelia HA incorporates this server into its JVM.

6.4.2 Pinging a Well Known Address (WKA)

Coppelia HA has many internal features to ensure the system is working correctly. As an extension, Coppelia HA also checks that the external Ethernet devices are working by pinging WKA. No Coppelia HA server can fully start up or become the primary server until it can successfully ping a WKA. An example of a WKA is a router or DNS server. When configuring your system there, is no limit to the number of WKA. So long Coppelia HA is connected to the subnet, its Ethernet card is working and at least one of the WKA returns a reply, the server will start up.

6.5 Configuration

6.5.1 Config File

As mentioned above, the configuration for a CoppeliaHA Cluster can and is recommended to be stored into a single file.

6.5.1.1 HIGH_AVAILABILITY Block

Attribute	Type	Default	Req	Description
HA_SERVER	Block		ON	This block contains a reference to every Server block in the cluster.
PING_COMMAND	String	ping	OFF	This is the system dependant ping command. It is required to send a single ping and exit with a 0 if successful.
PING_COMMAND_SUFFIX	String		OFF	In the unlikely case a parameter is required after the IP address. E.g. the "-n 1" attribute on HP/UX.
WELL_KNOWN_ADDRESS	Address	please set	ON	A list of WKA you are required to have at least one otherwise the server will not start.

6.5.1.2 Server/Main Block

As mentioned above, any attribute in the Server Block is copied to the Main block at startup. The following attributes are used by CoppeliaHA to configure the servers:

Attribute	Type	Default	Req	Description
HIGH_AVAILABILITY	Block		OFF	This points to the HA block.
RMI_IP	Address		OFF	The IP address of the server, this should be the real IP address of the server, not 127.0.0.1 since external servers use this IP address to communicate.
RMI_PORT	int		OFF	RMI port.
SERVER_BLOCK	Block		OFF	This points to the server block itself.
SERVER_NAME	String		OFF	This matches the arguments passed in to CoppeliaHA at startup with the associated server block.

6.6 Clustering

(Section TBA)

7.0 ACT Reporting – RESTRICTED ACCESS

8.0 FIX-to-CMS (Lolita) – RESTRICTED ACCESS

9.0 Appendices

9.1 Coppelia Errors, Warning and Information Messages

This section describes all the error, warning and informational messages generated by Coppelia. Coppelia does not currently assign error numbers to errors, those given in the following tables are assigned for reference purposes only. The information in this document is organized into sections for different components of Coppelia: Communication, Database, Message Formatting, End of Day and interface specific messages.

9.1.1 Communication Messages

These are errors indicating that either Coppelia has detected problems with a connection to a counterparties FIX engine.

Error Number	Text	Description	Severity
1	Connection to [***Computer ID***] is down	Computer ID is already down when trying to do a disconnect	ERROR
2	Failed to run disconnect script	A DISCONNECT_SCRIPT specified in the .dat file failed to run on disconnect.	ERROR
3	[***Computer ID***] is not connected	During attempt to send, the Computer ID is not connected.	ERROR
4	Not updating the currently active session : [***Computer ID***]	Displayed during reconfigure – indicates that active connections cannot be updated.	INFO
5	Connection from an unspecified host : [***IP Address***] rejected	Connection attempt from host with IP Address not found in configuration file.	WARNING
6	Configured as Server; listening for connections on port [***Port***]	Coppelia configured as a Server	INFO
7	Configured as Client	Coppelia configured as a Client	INFO
8	Connecting to [***Computer ID]***	Attempting to connect to Computer ID	INFO
9	Already Connected to [***Computer ID***]	Connect attempted when already connected to a Computer ID	INFO
10	Automatic Backup Recovery for id [***Computer ID***] setting sequence numbers IN\\OUT to [***In Sequence/Out Sequence***]	Switching to backup connections	INFO
11	Waiting for connection	Waiting for response to Login from Computer	INFO
12	Connected to [***Computer ID***]	Connection succeeded – Login request sent	INFO
13	Trying Client socket, net address [***IP Address]*** port [***Port***]	Attempting to establish connection	INFO
14	Re-connected to [***Computer ID***]		INFO
15	Connection to [***IP Address***] established.	Comm Connection to Computer ID successful	INFO
16	Start: Running Disconnect Script:	DISCONNECT_SCRIPT specified in	INFO

	[***Disconnect Script Name***]	the .dat file is running.	
17	Connection to [***Computer ID***] accepted	Logon request acknowledged successfully	INFO

9.1.2 Message Format Messages

These errors indicate that a message sent to Coppelia has is not formatted correctly or contains invalid tags or field values.

Error Number	Text	Description	Severity
18	Rejecting data on id [***Computer ID***] : reject reason is [***Reject Reason]*** : [***Message Data***]	Rejecting a message sent by Computer ID – reason is included in the message	ERROR
19	Wrong FIX version for remote id [***Computer ID***] did not match expected version " + [***FIX Version***]	FIX Version specified in the Coppelia configuration file for a Computer ID does not match FIX Version of message sent by Computer ID	ERROR
20	Received (PossDup) message from [***Computer ID***] with seq num [***Sequence Number***] will not be processed because sequence number [***Sequence Number***] is greater than the first un-processed queued message sequence number [***Sequence Number***]	Possible Duplicate received from on a connection with a sequence number higher than the sequence number of the first unprocessed message. This condition will be corrected automatically by Coppelia.	INFO
21	Received (PossDup) message from [***Computer ID***] with seq num [***Sequence Number***] and will not be processed because sequence number [***Sequence Number***] is greater than the next expected sequence number [***Sequence Number***]	Possible Duplicate received from on a connection with a sequence number higher than the sequence number expected. This condition will be corrected automatically by Coppelia.	INFO
22	Received (PossDup) message from [***Computer ID***] with seq num [***Sequence Number***] will not be processed because sequence number [***Sequence Number***] does not match the next expected sequence number [***Sequence Number***]	Possible Duplicate received from on a connection with a sequence number higher than the sequence number expected. This condition will be corrected automatically by Coppelia.	INFO
23	Process queued data on id ***[Computer ID] [Data***]	Heartbeat message written to Log file if LOG_HEARTBEAT is ON	INFO
24	Sending reject to ***[Computer ID]***	Notification of Reject Message	WARNING
25	Sending data on id [***Computer ID***] : [***Data***]	This is the log file rendition of a FIX message sent by Coppelia.	INFO
26	Received data on id ***[Computer ID] [Data]***	This is the log file rendition of a FIX message received by Coppelia.	INFO
27	Setting ***(@[1|2|3])*** incoming sequence number to ***[Sequence Number]*** for id ***[Computer ID]***	These messages are displayed when sequence reset is received – this can be received as a result of three different conditions – hence 1/2/3	INFO
28	Remote id ***[Computer ID]*** did not	Computer ID in the FIX message	ERROR

	match expected ***[Computer ID]***	does match the Computer ID of the connection	
29	Received (PossDup) message from ***[Computer ID]*** with seq num ***[Sequence Number]*** and will process	Notification that a Possible Duplicate has been received and it will be processed – it was not already received by Coppelia.	INFO
30	Received (new) message from [***Computer ID***] with seq num [***Sequence Number***] will be stored and processed after resend request has been fulfilled	Notification that a message has been received a while a resendis in progress. It will be stored and processed after the resend has completed.	INFO
31	REJECT RECEIVED - PLEASE CONTACT SUPPORT REJECT: SenderComputer ID=[Computer ID] SenderSubID=[Sub ID] REJECT: RefSeqNum= [Sequence Number] Message:[Message Text]	A message sent by Coppelia has been rejected by the couterparties FIX engine.	ERROR
32	Remote timed out, disconnecting [***Computer ID***]	Remote timed out by Coppelia – did not respond to heartbeat within required time.	ERROR
33	In message type ***[Message Type]*** from [***Computer ID***] ignoring tag ***[Tag Number]***	Ignoring a bad tag number	WARNING
34	In message type ***[Message Type]*** from ***[Computer ID]*** cannot assign field ***[Field Name]***	Ignoring an invalid field.	WARNING
35	Remote timed out for logging on, disconnecting	Remote did not respond to logon request within required time.	ERROR

9.1.3 Interface Messages

These errors indicate problems with configuration or usage of Coppelia's external interfaces.

Error Number	Text	Description	Severity
36	Setting interface to [CORBA/ TIBCO TIB/Rendezvous/ MQSERIES/ Observer/Observable/ Java RMI/Talarian SmartSockets]	Describes which interface used by Coppelia	INFO
37	Unknown interface. Check .dat file	Invalid Interface specified in configuration file	ERROR

9.1.4 RMI Interface Messages

Error Number	Text	Description	Severity
38	CoppeliaSrv fails binding to RMI registry ***[RMI Error]***	RMI Registry Port already in use, bad policy file, or RMI stub file missing.	ERROR
39	Callback with subject : ***[Subject]*** removed due to RMI RemoteException.	Callback cannot communicate to client application – client implemented callback is no longer alive.	WARNING
40	Global callback removed due to RMI RemoteException.	Callback for all inbound messages cannot communicate to client application – see above.	WARNING
41	Incoming message with TargetSubID : ***[Subject]*** and seq# ***[Sequence Number]*** not handled by any callback, delivered to the queue	No subscribers for this message – in normal operation, this should not occur so it needs to investigated.	WARNING

9.1.5 TIBCO RendezVous Interface Messages

Error Number	Text	Description	Severity
42	Invalid connection	Can't connect to RendezVous	ERROR
43	Invalid data, reject message : " + vd.reject_message , "CoppeliaRV");	Callback cannot communicate to client application – client implemented callback is no longer alive.	WARNING
44	Callback with subject : ***[Subject]*** removed due to RVException.	Callback cannot communicate to client application – client implemented callback is no longer alive.	WARNING
45	Global callback removed due to RVException.	Callback for all inbound messages cannot communicate to client application – see above.	WARNING
46	Incoming message with TargetSubID : ***[Subject]*** and seq# ***[Sequence Number]*** not handled by any callback, delivered to the queue	No subscribers for this message – in normal operation, this should not occur so it needs to investigated.	WARNING
47	listening on subject " + sub, "CoppeliaRV");	Indicates which subjects Coppelia is subscribing to..	INFO
48	Bad subject : ***[Subject]***	Invalid Subject	ERROR
49	Invalid field: ***[Field]***	Invalid Field in RendezVous Message	ERROR
50	Invalid token in RV sender subject :	Invalid Subject Token	ERROR

9.1.6 Database Messages

These errors indicate that a database error has occurred during a write to the Coppelia persistent database.

Error Number	Text	Description	Severity
51	Warning : JDBC driver not found in classpath. Please correct the problem and re-start.	Can't find JDBC jar file in the class path.	ERROR
52	Warning : Fail connecting to server : url, driver, user and password information may be invalid! Please correct the problem and re-start	JDBC Connection parameters in config file are invalid.	ERROR
53	Warning : Fail creating the database tables ***[Error Message].*** Please correct the problem and re-start or contact Javelin.	Problem creating database tables for Coppelia – could be a permission problem. Contact DBA	ERROR
54	Warning : Fail setting up the prepared statements ***-[Error Message]*** Please contact Javelin.	Problem creating JDBC prepared statements.	ERROR
55	Re-connecting after Persistent Database Error	Coppelia is reconnecting to the database after the connection has been dropped. This is only done if RECONNECT_PERSISTENTDB is ON	INFO
56	JdbcPersistentDB.java: Warning : No sql connection or connection broken, Coppelia will run without persistent database. Please correct the problem and re-start.	Coppelia's connection to the database has been dropped. This only appears if RECONNECT_PERSISTENTDB is OFF	ERROR
57	Warning : Committing to persistent database encounters exception, Coppelia will run without persistent database. Please correct the problem and re-start	Coppelia's connection to the database has been dropped. This only appears if RECONNECT_PERSISTENTDB is OFF	ERROR
58	Warning : Closing sql connection encouters exception - ***-[Error Message]***	Error disconnecting from database on shutdown.	WARNING
59	Warning : No sql connection or connection broken while trying to close the connection.	Error disconnecting from database on shutdown.	WARNING
60	Warning : There is no database connection/broken. Please correct the problem	Coppelia's connection to the database has been dropped. This only appears if RECONNECT_PERSISTENTDB is OFF	ERROR

61	JdbcPersistentDB.java: Warning : Database access error ***-[Error Message]*** Coppelia will run without persistent database, please correct the problem and re-start.	Coppelia's connection to the database has been dropped. This only appears if RECONNECT_PERSISTENTDB is OFF	ERROR
62	Warning : Error restoring.	Database error during Restore	ERROR

9.1.7 End of Day Messages

These errors indicate that a database error has occurred during end of day.

Error Number	Text	Description	Severity
63	Warning : No known Computer ID. End of Day will NOT be succesful for ***[Computer ID]***	Attempt to run End of Day for a Computer ID not found in the configuration file.	ERROR
64	Warning : Error accessing database during EOD.	Database connection down while trying to run End of Day	ERROR
65	Warning : No database connection/broken.	Database problems while running End of Day.	ERROR
66	End Of Day complete and successful; Coppelia ready for next trading day	End of Day successful!	INFO
67	End Of Day complete and successful for ***[Computer ID]*** Coppelia ready for next trading day.	End of Day successful for given Computer ID.	INFO

9.2 Glossary of Terms

. / #	
.bat file	A batch file. Batch files are executable files which execute a series of commands prior to running a larger **application**.
.class file	A filetype that contains compiled **Java code**.
.cpp file	A filetype associated with **code** created in the **C++ programming language**. *see* **C++**.
.dat file	A filetype which typically contains configuration options used by Javelin products.
.dll file	A filetype which refers to files containing **Dynamic Linked Libraries**. *see* **Dynamic Linked Library**.
.exe file	A filetype which refers to Executable files.
.h file	A filetype which refers to **header files**. *see* **header files**.
.idl file	A filetype which contains methods that describe objects that are used for exchange in **CORBA** environments. *see* **IDL**.
.in file	A filetype that refers to files produced by Lolita converter.
.ior file	A filetype which typcially contains **IOR** strings that are used by **CORBA** to map a references to a **CORBA** objects. *see* **IOR**.
.jar file	A filetype which typically contains an archived collection of **Java class**es. In Java, this term is analagous to a **library**. *see* **JAR**.
.java file	A filetype assocated with **code** created in the **Java programming language**. *see* **Java**.
.log file	A filetype which typically contains a text-based **log** of all the activity contained in a given **FIX session**. This includes **messages**, **heartbeats,** and other **connection**-related information.
.od* file	A filetype that refers to files used by eXcelon **databases** (e.g. **Pse/Pro** & **Objectstore**).
.out file	A filetype that refers to files produced by Lolita converter.
.rej file	A filetype that contains information pertaining to messages that are rejected. Created by Coppelia.
.srl file	A filetype that refers to files produced by Lolita converter.
.txt file	A filetype that refers to files containing plain **ASCII** text.
.zip file	A filetype which contains **zip**ped information. *see* **zip**.
24x7 environment	A Coppelia-related term which refers to the maintenance of Coppelia **connection**s longer than the traditional business day. Speicfically, the term applies to an environment where Coppelia connections are, in fact, *never* brought down.
A	
absolute path	A term which refers to the complete **path** of a **directory** or file. For example, *D:\Coppelia\buy\buy.dat* is the absolute path for the file *buy.dat*. *see also* **relative path**.
access rights	A term which refers to the list of rights and privileges granted to a **user** or users.
ACT	*acronym*. **A**utomated **C**onfirmation **T**ransaction Service, a post-execution service offered by **NASDAQ**.
ACT Reporting	**NASDAQ NWII** reporting facility. *see* **ACT**.
ACT Router	A **Coppelia** add-on that allows sending reports of executions to **ACT**.
ActiveX	A Microsoft-developed technology framework, referring to a light-weight implementation of **MFC**. A popular Coppelia **interface**. *other*: **CORBA; Talarian SmartSockets; Observer/Observable; TIBCO Rendezvous; IBM MQ Series; COM/DCOM; RMI; FIXML/HTTP**.
address	A number of bit pattern that uniquely identifies a location in computer memory. Every location in memory has a unique address. *see* **Network address, IP address**.
Administrative Data	**Data** or other information pertaining to a **FIX Administrative message** or other function.
Administrative message	A class of **FIX message** which deals with information pertaining to the state of a **connection** between two parties.
AIX	A **UNIX**-based **operating system** for computers, mainly used on **machines** built by **IBM**. *others:* **UNIX; HP/UX; WindowsNT; SUN Solaris**.
algorithm	A formula or set of steps for solving a particular problem. To be an algorithm, a set of rules must be unambiguous and have a clear stopping point. Algorithms can be expressed in any **language**, from natural languages (like English) to **programming languages** like **Java** or **C++**.

alias	In **UNIX**, an alias is a user-defined alternative to typing commands of various complexity. For example, a user can create an **alias** which will change **directory** to a predefined **path**. To get to that path, the user need only type the name of the alias, and **UNIX** will interpret it as the directory change specified in the alias's definition.
Allocation Message	A **buy-side FIX message** whose purpose is to instruct the **broker** to allocate the shares of a previously-executed **order** between two or more accounts.
AMEX	*acronym*. **Am**erican **S**tock **E**xchange.
API	*acronym*. **A**plication **P**rogram **I**nterface. *see* **Application Program Interface**.
application	A general computing term that refers to a **program** or group of programs designed for end **users**. Software can be divided into two general classes: systems software and applications software. Systems software consists of low-level programs that interact with the computer at a very basic level. This includes **operating systems**, **compile**rs, and utilities for managing computer **resource**s. In contrast, applications software includes **database** programs, word processors, and spreadsheets. Figuratively speaking, applications software sits on top of systems software because it is unable to run without the operating system and system utilities. Coppelia itself is an application.
Application Data	**Data** or other information pertaining to a **FIX Application message** or other function.
Application Layer	A **layer** within a **FIX session** whose purpose is to handle the actual business-related content in the **message** flow (e.g. **Orders, Execution Reports, Allocations**, etc.)
Application message	A class of **FIX message** which deals with information pertaining to business-related content (e.g. **Orders, Execution Reports, Allocations,** etc.) Application messages are handled in the **Application Layer**, and delivered via the **Session Layer**.
Application Program Interface	A term that is used when referring to a set of **routines**, **protocols**, and tools for building software **application**s. A good **API** makes it easier to develop a **program** by providing all the building blocks. A **programmer** puts the blocks together. Most operating environments, such as **WindowsNT**, provide an API so that programmers can write applications consistent with the operating **environment**. Although APIs are designed for programmers, they are ultimately good for **users** because they guarantee that all programs using a common API will have similar **interfaces**. This makes it easier for users to learn new programs.
architecture	In general computing terms, architecture refers to the overall design or **structure** of a computer system, including the hardware and the software required to run it, especially the internal structure of the **microprocessor**.
archive	A file containing **compress**ed contents of other files, which can later be **expanded** to original form for use. *see also* **compression**.
argument	A general computing term that refers to a value used to evaluate a **procedure** or **subroutine**.
array	A collection of **data** items that are given a single name and distinguished by numerical subscripts. Each item in an array is known as an element.
array element	A single item in an **array**. *see* **array**.
As Of Date	**Flag** contained in an **As Of FIX message** that indicates the day a trade was made. *see also* **As Of Time; As Of Flag; As Of Trading**.
As Of Flag	**Flag** contained in a **FIX message** that indicates an **As Of Trade**. *see also* **As Of Date; As Of Time; As Of Trading**.
As Of Time	**Flag** contained in an **As Of FIX message** that indicates the time the trade was made. *see also* **As Of Date; As Of Flag; As Of Trading**.
As Of Trading	Business term that refers to a **trade** made on a listed **stock** that is done before market hours. In these cases, the trade has to be saved and reported at any time the next day, As Of.
ASCII Characters	*acronym*. **A**merican **S**tandard **C**ode for **I**nformation **I**nterchange. A standard code for representing characters as numbers that is used on most computers. ASCII text is the format in which all **FIX messages** are coded.
ASCII SOH delimiter	*see* **field delimiter**.
asynchronous	Not synchronized; that is, not occurring at predetermined or regular intervals. The term asynchronous is usually used to describe communications in which data can be transmitted intermittently rather than in a steady stream. The difficulty with asynchronous communications is that the receiver must have a way to distinguish between valid **data** and **noise**. In computer communications, this is usually accomplished through a special start **bit** and stop bit at the beginning and end of each piece of data. For this reason, asynchronous communication is sometimes called *start-stop transmission*. Most communications between computers and devices are asynchronous.

asynchronous mode	Refers to a mode of **data** transfer where it is legal to send and receive information at the same time.
ATL	*acronym*. **A**ctive **T**emplate **L**ibrary. Lightweight **ActiveX** control.
attribute	In general computing terminology, an attribute refers to a characteristic. In **database** management systems, the term is sometimes used as a synonym for **field**.
audit queue	The **queue** used in **MQ Series** where all undelivered **messages** are sent (rejected messages, invalid **data**, etc.). It could be a customized queue or the **default dead letter queue**. *see also:* **dead letter queue**.
Autex	A company that provides a wide array of financial **program**s and services.
autoconnect	A configuration option set in the buy or sell side **.dat** file which, when activated, will automatically connect to the **site**s listed in the ID section of the .dat file.
average price	Term that indicates the average price of all the partial order fills that constitute a complete trade.
B	
back-end	The part of a computer system not directly interacting with the **user**. For example, the **database** system running on a mainframe computer is known as the back-end of a system, whereas the microcomputers used by those accessing the system are the **front-end** of the system. *see also* **front-end**.
backslash	The **ASCII** character "\". Typically used in the listing of **directory** paths.
batch file	*see* **.bat file**.
binaries	
binary code	The basic number scheme on which all modern computers operate. Binary code consists of two values, 0 and 1, and correspond electronically to the two unique states a switch can occupy.
binding	Refers to the **map**ping of a **program** to a specific **network port**.
Bi-Sync	
bits	Shorthand term for **Bi**nary Digi**t**, which is the smallest unit of information on a **machine**. A bit can occupy two legal states: 0 and 1. More meaningful information is obtained by combining consecutive bits into larger units (such as **bytes**). *see also* **bytes**.
block name	
blocking queue mechanism	*see* **blocking-queue**.
blocking-queue	This is the heart of the **thread** communication. It is a mechanism for inter-thread communication that has the ability to **filter messages** so as to avoid **queue** oversize, which eventually results in memory leakage and decreased performance.
blocks	A logical grouping of elements within a **configuration file**.
Blotter	A **table**-oriented **user interface** used to enter **trades**.
bond	In general business terminology, a bond is simply defined as an obligation to pay.
boolean	A type of **variable** that has only two possible values: True (0) and False (1).
branch	An instruction that tells the computer to jump to another part of a **program**. Also, in a decision tree, a branch refers to a link connecting one node to another.
bridge	**Code** which **interface**s the Coppelia **engine** with various **middleware**. Also refers to a hardware **device** that connects two **network**s, or two segments of the same network. The two networks being connected, it should be mentioned, can be alike or dissimilar. Unlike **routers**, bridges are **protocol**-independent. They simply forward **packet**s without analyzing and re-routing **message**s. Consequently, they're faster than routers, but also less versatile. *see also* **interface; router**.
broker	An individual or firm that acts as an intermediary between a buyer and a seller, usually charging a commission for doing so. For securities and other products, a license is required.
broker simulator	The Coppelia **Broker** Simulator is a part of the **Coppelia Tool Kit**, and offers a quick and easy way to simulate **order** execution. The simulator will automatically fill orders and send **FIX execution reports** in response. This is especially useful for testing purposes, as it can simulate a real-time trading environment. It is also useful for developers implementing order entry **front end**s. In addition to accepting FIX execution reports, the simulator also accepts FIX **cancel request** and FIX **cancel replace** messages.
bug	A generic computing term which indicates, at the most basic level, a mistake in a piece of **code** that causes the **program** to function improperly. This could range anywhere from an erroneously computed value, to an error which has serious system-wide ramifications.
build	A term which refers to the process in which a program is **compile**d in such a

	fashion as to create a final product, usually in the form of an executable **program**.
Build Process	Development-related term which refers to the process by which a software **application** is **compiled** into executable format.
Business Flow Model	The **FIX Protocol** defines certain logic with respect to the order of **application messages**, and the expected reactions of connected applications. Appendix D of the FIX specification documents outlines this logic.
Business Logic	Certain logical processes deciding what is to be done to a certain **FIX message**, or its business content.
Buy Side	A summary term referring to all non-**broker**age firms, typically the larger money management firms that purchase securities for their own accounts. Note: A buy-side firm can also sell securities. *see also* **sell side**.
Byte	The number of **bits** (8) that stand for one character. Bytes are the standard upon which concepts of **CPU** memory and performance are based.
Byte sum	The sum of all the **bytes** in a given computation. Used when deriving the **checksum**. *see also* **checksum**.
bytecode	The concise set of instructions produced by **compiling** a **Java program**. This bytecode is the same for all **platforms**; a feature which makes Java a truly platform-independent programming **language**. It is executed by a **Java Virtual Machine**. *see also* **Java; Java Virtual Machine**.
C	
C++	A **programming language**, extended from the C language developed at Bell Laboratories. *other:* **Java; GNU C++; VB; Machine Language**.
C++ w/ ATL	Defined as **C++** with **Active Template Library**. *see* **C++; ATL; Active Template Library**.
CA	*acronym*. **C**ertification **A**uthority. *see* **Certification Authority**.
cache manager	
call	*see* **invoke**.
callback	A **function** that is passed to another **function** and is called when an **event** is complete.
callback model	
Cancel Replace	A **buy-side FIX message** whose purpose is to request that changes be made to the parameters of a previously-placed order.
Cancel Request	A **buy-side FIX message** whose purpose is to request a cancel of an order previously placed.
case-sensitive	General computing term which refers to distinguishing between **uppercase** and **lowercase** letters, such as *G* and *g*. For example, **UNIX** filenames are case-sensitive, so *example* and *EXAMPLE* would denote two different files. **DOS** filenames, on the other hand, are not case-sensitive, so those two files would, in fact, be equivalent. In general, names typed in **programming languages** such as **C++** and **Java** are case-sensitive as well.
certificate	A certificate is a small file given to a **user** whose purpose is to guarantee that the individual granted the certificate is, in fact, who he or she claims to be. *see also* **Certification Authority**.
Certification Authority	A trusted third-party organization or company that issues digital **certificates**. The role of the **CA** in this process is to guarantee that the individual granted the unique certificate is, in fact, who he or she claims to be. Usually, this means that the CA has an arrangement with a financial institution, such as a credit card company, which provides it with information to confirm an individual's claimed identity. CAs are a critical component in **data** security and electronic commerce because they guarantee that the two parties exchanging information are really who they claim to be. *see also* **certificate**.
check memory	
checksum	In **FIX,** the **modulo** (256) of the simple **byte sum** of characters in a FIX **message**, up to and including the **delimiter** that precedes the CheckSum **field**. In general, a checksum can be defined as a simple error-detection scheme in which each transmitted message is accompanied by a numerical value based on the number of set **bits** in the message. The receiving station then applies the same formula to the message and checks to make sure the accompanying numerical value is the same. If not, the receiver can assume that the message has been garbled.
class	An **object** type in an **object-oriented programming language**.
class file	A file which contains a particular **class** or **library** of **class**es. Typically used by **object-oriented programs**. *see also* **.class file**.
CLASSPATH	An **environment variable** on a computer that pre-defines the **path** to certain **.class** files or **libraries** containing such .class files. Also used by **Java** to determine which **class**es to use to run Java **application**s. This is set up either

	in a **startup script** or in the system **environment**.
clear text	**Unencrypted** content of a **FIX message**. *see* **Encryption**.
Client	In **FIX**, the **session initiator**.
client application	In the context of Coppelia, this is an **application** interacting with Coppelia on a non-**FIX** level.
Client ID	Defined as the unique ID assigned to the **Client** on a particular side of a **FIX Connection**.
client/server model	A model which uses more than one computer to offload and distribute work so tasks are completed more efficiently than if the entire load were handled by one **machine**.
client-server model	*see* **client/server model**.
cluster software	Software whose purpose is to combine several **machine**s to make one **Virtual Machine**.
CMS	*acronym*. **C**ommon **M**essage **S**witch, a **protocol** used to connect to US exchanges; specifically **NYSE** (SIAC). *other:* **FIX**.
CMS Engine	*see* **Lolita**.
CMS messaging	*acronym*. **C**ommon **M**essage **S**witch protocol. *see* **CMS; Lolita**.
code	A general computing term which refers to a **program**, **algorithm**, **routine**, or any fragment thereof, written in a particular **programming language**.
codebase	
COM/DCOM	*acronym*. **D**istributed **C**ommon **O**bject **M**odel. Windows **API** for object messaging. *other:* **CORBA; Talarian SmartSockets; Observer/Observable; TIBCO Rendezvous; IBM MQ Series; ActiveX; RMI; FIXML/HTTP**.
COMet	A **COM** to **CORBA bridge** produced by **Iona Corp**. It is a **middleware** product whosse purpose is to integrate **WindowsNT** solutions with **UNIX** solutions, and uses **Microsoft**'s **COM** as a **bridge**. *other:* **WindowsNT; UNIX; Microsoft; COM**
Comm	This **thread** handles all the outbound **messages**. Outbound messages are put into the Comm thread **queue**, and Comm is then responsible for getting the message from the queue and **streaming** it to the **counter party** via **TCP/IP**. *other:* **main thread; manager thread; system thread; logger; SysLog; UIThread;Comm; Interface**.
command	An instruction sent to a computer which triggers the execution of a **program** or **process**.
Command Line Interface	In Coppelia, this refers to the **DOS**-based interface where commands are issued to Coppelia via the **command prompt**.
Command prompt	A visual representation of a point of **user** input into a computer system.
CommConnection	This **thread** basically listens on the **TCP socket** and reads any **incoming messages**. When a message arrives, this thread passes it to the **pump object** for validation. *other*: **main thread; manager thread; system thread; logger; SysLog; UIThread; CommConnection; Interface**.
comment	Term which refers to information contained in a computer **program** which is ignored by the computer, and is included only for the benefit for human readers. Ideally, comments should elucidate a program's function to the person trying to understand it, through the use of clear, concise explanations after critical lines of **code**. *see* **commented out**.
comment line	A line in a piece of computer **code** whose purpose is to explain how a line, or block of code functions. *see* **comment**.
commented out	A **programming** term which refers to a line or block of **code** which is ignored by the **compile**r. Nearly every programming **language** recognizes a short character string (such as "**/* ... */**") and ignores everything typed in between them. This text can either be code itself (code that the developer wants the compiler to ignore for one reason or another) or descriptive text, the purpose of which is to help other programmers better understand the structure of the code. *see* **comment**.
Common Object Request Broker Architecture	An **object-oriented message** passing **protocol**, the purpose of which is to define a standard by which disparate **architecture**s and **device**s can programatically communicate with each other. **CORBA** is the default **interface** through which Coppelia runs. *other:* **Talarian SmartSockets; Observer/Observable; TIBCO Rendezvous; IBM MQ Series; ActiveX; COM/DCOM; RMI**.
communication bridge	Term that defines something which converts one type of network communication to another type (e.g. **IPX** -> **TCP/IP**).
comp id	The ID of your **local machine**.
compile	A computing term which refers to the assembling of **source code** to produce executable **program**s. *other:* **build**.
complex instruments	A financial instrument or **security** with a high level of complexity, such as a multi-legged option strategy.
component	A section of software **code** that can be merged with other code to create an

	application. Usually, in order for such components to work in truly modular fashion, they must adhere to a fairly strict set of standards which allow them to merge easily with other code.
compression	The storage of **data** in a way that makes it occupy less space in memory than if it were stored in its original form.
config file	*see* **configuration file**
configuration file	A file (usually text) which provides the **program** that reads it in with necessary settings or **parameter**s.
configuration parameter	A term that refers to a value or set of values which serve to define a particular configuration. In Coppelia, for instance, configuration parameters can be found in the buy and sell **.dat** files, and serve to determine the particular configuration for each side of the transaction.
Connect	In Coppelia, a command whose purpose is to establish a **connection** to a specified **counter-party**.
connection	A generic term that refers to a link between two parties, the purpose of which is to typically transmit information. in **FIX** and Coppelia, a connection refers to the link between two or more parties over which FIX **message**s are transmitted.
connection status	Term which refers to the status of a particular **FIX connection** within a **FIX session**.
ConnectionData	This is a **data structure** that contains all the information regarding a **connection**. Every detail regarding a session or connection is recorded here. This structure is kept in the memory at **runtime** and also persisted to the database. **FIX message error recovery** depends greatly on this data structure, as it keeps all details of incoming and outgoing sequence numbers. In addition to helping Coppelia deal with error recovery, this data structure also acts as the source for administrative or operation information.
connectivity	A computer term that refers to a **program** or **device**'s ability to link with other programs and devices.
console	Term which refers to the **keyboard** and **monitor**, the standard input-output pair which makes interacting with the computer possible. *see also* **keyboard; monitor**.
Control Panel	A **program** that gives you the ability to adjust certain features of your computer **environment**.
conversion parameter	
Coppelia	Javelin Technologies' **FIX Server** and **FIX Engine**.
Coppelia Client	Refers to a Coppelia **engine** running as a **client** in regards to the communications **protocol** (i.e. **FIX**, **NWII**, etc.).
Coppelia Enhanced Performance	*see* **Coppelia EP; EP; Enhanced-Performance**.
Coppelia EP	*acronym*. Coppelia **E**nhanced **P**erformance. Coppelia **engine** designed with **connectivity** to **databases** for improved performance. *other:* **Coppelia HA; Standard Coppelia; Coppelia; Coppelia Single Connect**.
Coppelia HA	*acronym*. Coppelia **H**igh **A**vailability. Refers to any Coppelia **server** that achieves **High Availability** with a **PersistentDBPipe component**. CoppeliaHA is implemented with a **fault tolerant architecture**. *other:* **Coppelia EP; Standard Coppelia; Coppelia; Coppelia Single Connect**.
Coppelia High Availability	*see* **Coppelia HA; HA; High-Availability**.
Coppelia Message Object	
Coppelia Module	Refers to specific Coppelia products, each with a different **architecture** and purpose. Coppelia Modules include: **Coppelia Single Connect; Standard Coppelia; Coppelia Enhanced Performance; Coppelia High Availability.**
Coppelia queue	Coppelia's **message** storage **queue**. Messages that cannot be sent right away are stored here.
Coppelia RMI	*see* **RMI**.
Coppelia Server	The Coppelia **engine** that acts as the **server** in regards to the communications **protocol**.
Coppelia Single Connect	A single **FIX connection Coppelia Server**. *other*: **Coppelia EP; Standard Coppelia; Coppelia HA; Coppelia**.
Coppelia Tool Kit	Refers to the packaged suite of Coppelia-related products, distributed by Javelin Technologies, which contains the following **module**s: **Broker Simulator; FIXometer; FIXionary;** and a compendium of test **script**s.
CORBA	*acronym*. **C**ommon **O**bject **R**equest **B**roker **A**rchitecture, a standard which describes the **architecture** of a **middleware platform** which supports the implementation of **application**s in distributed and heterogeneous environments. *other:* **Talarian SmartSockets; Observer/Observable; TIBCO Rendezvous; IBM**

	MQ Series; ActiveX; COM/DCOM; RMI.
counter	*see* **Sequence Counter**.
counter-parties	A term which refers to parties on either side of a **FIX connection** within a **FIX session**. *see* **FIX Session**.
CPU	*acronym*. **C**entral **P**rocessing **U**nit. Generally defined as the heart of the computer. The entity which contains all necessary constituent **component**s for basic computer functionality. *see* **motherboard**.
crash	The sudden, complete failure of a computer because of a hardware failure or **program** error.
current directory	Defined as the directory a **user** currently has the computer pointing to.
custom instruments	A non-standard financial instrument or **security**.
D	
daemon	Generally speaking, a **UNIX program** that runs continuously in the background, or in some cases is activated by a particular **event**. Within the context of Coppelia, a daemon is a background **process** that monitors and services **network connections** in various ways. *other:* **RV Daemon**.
data	Information.
data structure	In most basic computing terminology, a data structure is simply defined as a way of arranging information in computer memory. In **programming**, it is often necessary to store large amounts of information in such a manner as to reflect a relationship between them. Common types of data structures are ***arrays***, ***records***, *and linked lists***.**
data type	Ranges of possible values that a possible **data** item might have. Some data types include: integers, character strings, real numbers, **Boolean** numbers, etc.
database	A collection of **data** stored on some sort of computer storage medium (such as a hard drive), that can be displayed, queried, or otherwise manipulated, usually for more than one purpose. All versions of Coppelia come with a database from Object Design, Inc. for replication and **persistence** capabilities. Coppelia's flexibility allows integration with other databases via **JDBC**.
database maintenance	The task of storing information in a **database** and retrieving information from that **data**.
DB2	A **database** product produced by **IBM** Corp. *see* **IBM DB2**.
debug	A term which describes a general process of removing **bugs**, or errors, from computer **code**. *see also* **bugs**.
dedicated machine	A computer whose exclusive duty is to perform a limited number of tasks.
default dead letter queue	Used by **MQ Series**. This is the default **queue** where all **messages** that are not processed are sent. It's default queue name is *SYSTEM.DEAD.LETTER.QUEUE*.
default interface	Defined as the **interface** which a **program** or **application** is **configure**d to operate on by default. The default interface for Coppelia is **CORBA**. *see* **CORBA**.
delimiter	Symbols that mark the beginning or end of a special part of a **program**.
delimiting routine	A part of a computer **program** that inserts **delimiter**s into a **data structure**. *see also* **delimiter**.
dependencies	
desktop	A **directory** that is associated with the **GUI**. This directory is "revealed" and shows it's contents on the screen by default. *see* **GUI**.
development	The process of creating software **application**s.
Development Process	Term which refers to the fairly complex process by which the software **development team** creates a functioning software **application**.
development team	The team of computer **programmers** who collectively create a piece of software.
device	Any **machine** or **component** that attaches to a computer. Most devices require a program called a device **driver** that acts as a translator, converting general commands from an **application** into specific commands that the device understands.
directory	A memory location that holds the **address** of a file and its name. The file can recursively contain other files. In internet parlance, the tern can also refer to a **URL**. *see* **URL**.
directory structure	A general computing term which refers to a heirarchical organization of nested directories. *see* **directory**.
disconnect	In the context of Coppelia, this is a command whose purpose is to terminate a **connection** to a specified **counter-party**. *see* **connect**.
diskette	A type of removable media which stores a limited amount of information in a particular **format**.
distribute	To parcel out tasks to various **subsystems** in order to solve problems and complete work more efficiently. Also, it can be defined as a company's process for making their software product(s) available to the market.
distribution package	Defined as the collection of executable files, supplementary files, and documentation that comprise the entirety of a company's software product.

DNS	*acronym*. **D**omain **N**ame **S**ervice. A DNS is commonly used to map **IP Addresses** with names and vice versa.
Domain Name Server	A **server** that runs a **DNS** service. *see* **DNS**.
double	**Java, VB** & **C++** code for a **variable** of type double.
download	To transmit a file or **program** from a central computer to a computer at a remote site.
downtime	Term that refers to the span of time a computer or **connection** between computers is down or inactive.
driver	A **program** that extends the **operating system** to support a specific **device**.
dynamic library	*see* **Dynamic Link Library**.
Dynamic Link Library	A **library** that can be used during **runtime** rather than when the **program** is **compile**d. This has the practical advantage of making programs smaller, because more commonly used **functions** can be placed in the **.dll** file, thereby reducing overall program size and **compile** time.
E	
ECN	*acronym*. **E**lectronic **C**ommerce **N**etwork. *see* **Electronic Commerce Network**.
Eicon Card	A **network card** that communicates over the **X.25 protocol**.
Electronic Commerce Network	A **network** that is specifically created and maintained for the purpose of carrying out business transactions.
element	*see* **array element**.
Email message	
Embedded delimiter character	In **FIX**, a **delimiter** that is part of the contents of a **field**.
encrypted data section	In **FIX**, the part of the FIX **message** that has undergone **encryption**.
encryption	The act of converting information into code or cipher so that unauthorized parties are unable to read it. *see also* **encryption key**.
encryption key	A term that refers to the *key* that provides the necessary information to decipher encrypted **data**. *see also* **encryption**.
End of Day	In the context of **Coppelia**, the **process** that truncates **database table**s, and resets **sequence number**s back to one (1).
engine	In general computing terminology, an engine is a special part of a computer **program** that implements a special technique. *see* **Coppelia; FIX Engine**.
environment	The display and **interface** provided by the software.
EOD	*acronym*. **E**nd **O**f **D**ay. *see* **End-Of-Day**.
EP	*acronym*. **E**xtended **P**erformance. *see* **Coppelia EP**.
error message	A message generated by a program which indicates an error, either of the input parameters, or of the code syntax.
error recovery	A term that refers to the general process by which a computer or **program** is designed to handle, interpret, manage, and recover from errors.
ethernet	A type of **Local Area Network** which uses radio frequency signals carried by coaxial cable. *see also* **network**.
ethernet address	*see* **network address; IP address**.
evaluation version	A special version of a company's software, the purpose of which is to highlight the software's function for prospective clients. Evaluation versions of software typcially contain limited functionality, and quite often expire after a specified period of time.
evaluator	A person or firm evaluating something. In this context, Javelin products.
event	A general computing term defined as an action or occurrence detected by a **program**. Events can be **user** actions, such as clicking a **mouse** button or pressing a key, or system occurrences, such as running out of memory. Most modern **application**s, particularly those that run in Macintosh and Windows **environment**s, are said to be *event-driven*, because they are designed to respond to **event**s.
exception	An **event** that cannot be handled in a normal **process**. Exceptions usually cause **program**s to halt operation, or to return a specific message.
exchange	Any organization, association or group which provides or maintains a marketplace where **securities**, options, futures, or commodities can be **trade**d. The term can also refer to the marketplace itself.
executable shell script	*see* **shell script**.
Execution Report	A **sell-side FIX message** whose purpose is to relay information pertaining to the status of an **order**, for example: partially filled, filled, canceled, or done for the day.
exit	To terminate operation. In Coppelia, the exit command has the effect of **kill**ing the window in which it is currently running, thereby terminating the **connection** (if not already **disconnect**ed) from the other party.
extension	*see* **file extension**.
extraction	The restoration of **compressed data** to its original form. Alternately, the removal

	of individual files from an **archive**.
F	
factory	
fault tolerant	Term that describes a method to provide uninterrupted system service after the failure of one of the system's components. s*ee also:* **High Availability; Coppelia HA**.
field	In **FIX**, the smallest element of a FIX **message**, containing a **tag-value pair** in the format <TAG=VALUE>. *see also* d**elimiter; record**.
field delimiter	In **FIX**, a non-printable **ASCII** character (SOH) that presents the border between one FIX **field** to another.
field name	In **FIX**, this term refers to the name associated with a particular **field**.
file extension	A computing term that refers to the three-letter **suffix** found after the "." in most filenames. *see* **suffix**.
file format	General computing term which refers to the **format** of a particular file. *see* **format**.
Fill (full)	A financial term that refers to an execution that leaves no more **share**s to be executed on an **order**.
Fill (partial)	A financial term that refers to an execution that does not execute the entire **share** quantity.
filter	A **program** that accepts a certain type of **data** as input, transforms it in some manner, and then outputs the transformed data. For example, a program that sorts names is a filter because it accepts the names in unsorted order, sorts them, and then outputs the sorted names. Utilities that allow you to import or export data are also sometimes called filters. Also, a filter can refer to a pattern through which data is passed. Only data that matches the pattern is allowed to pass through the filter.
firewall	A **device** of software package designed to detect and prevent un-authorized access to a **network** of computers.
firewall configuration	Refers to the software configuration of a **firewall**. *see also* **firewall**.
firm	A general business term that refers to a business, corporation, partnership, or proprietorship of some kind.
Firm ID	In **FIX**, this term refers to the unique identifier given to a particular **firm**.
FIX	*acronym*. **F**inancial **I**nformation e**X**change. *see* **FIX Protocol**. *other:* **CMS**.
FIX Application object	
FIX Engine	*see* **Coppelia**.
FIX Protocol	A **message protocol** used to transmit and receive information related to financial transactions.
FIX Session	A **session** established between two parties for the purpose of transmitting and receiving **FIX messages**.
fixed income securities	
FIX-enabled	A **machine** or **network** that is capable of handling and processing **FIX messages**.
FIXionary	Javelin's online **FIX** reference tool. FIXionary allows a user to easily look up the corresponding values for each **tag**, by number, or by tag name. In addition, a user can choose which version of FIX to look at, or cross-reference information across **message** types and FIX versions. FIXionary is part of the **Coppelia Tool Kit** provided to our clients and evaluators for free to help them understand and implement FIX. *see also* **Coppelia Tool Kit.**
FIXML	*acronym*. A hybrid of **FIX** and **XML**, combining the advantages of these two standards. An **interface** with which Coppelia interacts. *other:* **Talarian SmartSockets; Observer/Observable; TIBCO Rendezvous; IBM MQ Series; ActiveX; COM/DCOM; RMI; CORBA**.
FIXometer	Javelin's **network monitor**. It is used to remotely affect the **Coppelia Server** process. Its **functions** gather statistics, check **connection** status, manipulate **sequence numbers**, run **end of day**, and remotely **connect** or **disconnect**. The FIXometer can also be used to view multiple instances of Coppelia. FIXiometer is part of the **Coppelia Tool Kit** provided to our clients and evaluators for free to help them understand and implement FIX. see also **Coppelia Tool Kit**.
flag	A general computing term which indicates a **variable** whose purpose is to indicate whether something is active or inactive.
flavor	Refers to a particular version of **UNIX**. Typically, different flavors of UNIX share some basic commands, but usually have commands unique to that flavor. Popular UNIX flavors are **SUN Solaris, HP/UX,** and **AIX**. *see also* **UNIX**.
float	**Java, VB** & **C++** code for a **variable** of type floating point decimal.
fopen	A **function** available in **C/C++** used to open files. It can also be used by extension in some **UNIX shell** command languages. *see* **UNIX; shell**.
format	Any method of arranging information that is to be stored, displayed, or

	manipulated.
formatter	An internal Coppelia command which is responsible for formatting the outbound **message** from other **middleware objects**. It uses an interface and delegation pattern in order to allow for different formatters to be plugged in at **runtime**. Because of this design, Coppelia is capable of supporting various **protocol**s such as **NASDAQ NWII,** etc. The protocol-specific formatter is determined at runtime based on the **connection**. Once sent, **pump** does the reverse operation on the receiving side. *see also* **pump**.
free form text	*see* **clear text**.
freeware	A term which refers to software that can be obtained and run at no cost to the **user**. In other words, it's absoutely free. *see also*: **shareware**.
front-end	A computer or **program** that helps you communicate with another computer or program on the **back-end**. *see also* **back-end**.
function call	*see* **procedure**.
function call	A computing term that refers to a **program**, **routine**, or other **function** that sends a command, and often times **parameters,** to a **function** or **method**, thereby executing that function and typically returning a value, or values, to the routine that called it.
G	
gap detection	In **FIX**, the process of determining that **sequence numbers** (FIX **messages**) have been missed.
garbage collection	Refers to the process of clearing out **objects** that are taking up space in memory but are no longer in use by the **program**.
garbage collection time	Refers to the time when a **garbage collection process** executes, thereby clearing out **objects** that are taking up space in memory but are no longer in use by the **program**.
gateway	Refers to a link between two different **network**s.
gateway server	**Refers to a server** that acts as a **gateway** between two different **network**s.
global attributes	Refers to a system-wide set of **attributes**. *see also* **local attributes; attribute**.
GNU C++	*acronym*. **G** **N**ot **U**NIX. A version of the **C++ programming language** created and maintained by the GNU Foundation (http://www.gnu.org) *see* **C++**.
Graphical User Interface	A way of communicating with the computer by manipulating icons, pictures, and windows with a **mouse**. *see* **GUI**.
GUI	*acronym*. **G**raphical **U**ser **I**nterface. *see* **GUI**.
H	
HA	*acronym*. **H**igh **A**vailability. *see* **Coppelia HA**.
handle	In **programming** parlance, a handle is a token, typically a **pointer**, that enables the **program** to access a **resource**, such as a **library function**.
hardcoding	
hash	A function that converts a string of characters to a number or a shorter string.
hash reference	The reference of a particular **hash** function. *see* **hash**.
HAWK	**TIBCO** Finance management **API** for their **Rendezvous** messaging system.
header files	Header files are commonly defined as **source** or **object** files that are to be included in a **program**. They are typically prepended to a file before compilation, and usually contain **libraries** critical to the function of the program. *see also* **.h files; .dll files**.
header object	An **object** which defines and initializes the **Standard Header.** *see* **Standard Header**.
heartbeat	A **FIX message** whose purpose is to frequently relay the status of the **FIX session** to the **counter-party**.
heartbeat interval	In **FIX** and Coppelia, this refers to the amount of time, in seconds, between **heartbeat**s. *see also* **heartbeat**.
High Availability	The characteristic of a system or process pertaining to its availability 99.99% of all times. s*ee* **fault tolerant; Coppelia HA**.
home directory	Defined as the **directory** a computer or **program** will look or go to by default. *see also* **directory; current directory**.
hostname	The name of the host **machine** to which another computer wishes to **connect**.
HP/UX	A **UNIX**-based operating system for computers, mainly used on **machine**s built by Hewlett Packard. http://www.hp.com *other:* **UNIX; Windows NT; SUN Solaris**.
HTML	*acronym*. **H**yper **T**ext **M**arkup **L**anguage. A set of codes that can be inserted into text files to indicate special typefaces, insterted images, and links to other **hypertext** documents.
http://	*acronym*. **H**yper**T**ext **T**ransfer **P**rotocol. A standard method of publishing information as hypertext in **HTML format** on the internet. see **hypertext; HTML**.
hypertext	Electronic documents that present information that can be read by following

	many different connections, or links, to other parts of the document. This differs from a book, for instance, whose flow is strictly sequential.
I	
IBM	*acronym*. **I**nternational **B**usiness **M**achines. http://www.ibm.com.
IBM DB2	**Database** product by **IBM** Corp. Also known as **Universal Database** or **UDB**. *other:* **Objectstore; Oracle; PsePro; SQL database; Sybase SQL Server; MS SQL Server; IBM DB2; JDBC.**
IBM enterprise database	*see* **IBM DB2**.
IBM MQSeries	A **middleware** package with which Javelin's Coppelia **interface**s. Produced by **IBM** Corp. An interface through which Coppelia interacts. Delivers **messages** by reading and writing to different **queues**. *other:* **Talarian SmartSockets; Observer/Observable; TIBCO Rendezvous; IBM MQ Series; ActiveX; COM/DCOM; RMI; FIXML/HTTP**.
IDL file	*see* **.idl file**.
IIOP	*acronym*. **I**nternet **I**nter-**O**RB **P**rotocol. A protocol that extends **TCP/IP** by adding **CORBA** defined messages for objects to connect to each other over the network. *see* **iiop port, iiop ip**. *see also* **TCP/IP; CORBA**.
iiop ip	The **IP Address** of a particular **IIOP connection**.
iiop port	The **port** of a particular **IIOP connection**.
implementation	General **programming** term that refers to the specific means by which a computer **program** solves a particular problem.
inbound messages	*see* **incoming messages**.
incoming messages	
Indication of Interest	A **FIX message** used to convey to the **buy side** a non-binding interest of a **broker** or other **sell side** firm to buy or sell securities.
Infinite loop detection	**Program**matic detection of a potentially harmful condition where the same **process** or action repeats infinitely.
initialization	To store a value in a **variable** for the first time. In most cases, if a **program** tries to use an uninitialized variable, it will get a random value, and will therefore behave unpredictably. In some cases, deleteriously.
install	The process of copying all necessary executable, registry, and other files onto a computer system, usually in a specialized **path** that the **installer** creates.
install packages	Refers to a suite of executable **program**s that are **install**ed on a computer.
installation path	The **path** where a **user** specifies where a **program** or **application** is **install**ed.
installer	A **program** which **install**s a software package on a computer system.
instance	Refers to an **object** of a particular type. For example, if we had a **Java class** called *foo*, and we created a new *foo* object called *bar*, then the *bar* object is said to be a new instance of the object type *foo*.
instantiated	Refers to the creation or initialization of an **instance** of an **object** type. Once created, the new object is said to be *instantiated*. *see also* **instance**.
Instinet	An **ECN** created by Reuters. *see* **ECN**.
int	**Java, VB** & **C++ code** for a **variable** of type integer.
inter-domain bridging	Refers to the process of **bridging** between **subdomains** for purposes of communication between the two.
interface	The **connection** between two computers through which information is exchanged. In Coppelia, this term refers to the method by which Coppelia communicates with various client **architecture**s. *see also* **CORBA**; **Talarian SmartSockets; Observer/Observable; TIBCO Rendezvous; IBM MQ Series; ActiveX; COM/DCOM; RM; FIXML/HTTP**.
Interface	Thread which handles the delivery of **incoming messages** to a **client application** via the **middleware interfaces** that Coppelia supports. *other:* **main thread; manager thread; system thread; logger; SysLog; UIThread; CommConnection; Comm**.
interface queue	**Queue** used by Coppelia to process **messages** from its current **interface**.
Interface Repository	
Internet Inter ORB Protocol	Refers to a method of communication commonly used by **CORBA**. *see* **IIOP**.
Interoperable Object Reference	
interrupt	An instruction that tells a **microprocessor** to put aside what it is currently doing and call a specified **routine**. The **processor** resumes its original work when the interrupt service routine finishes. Interrupts are used for two main purposes: To deal with hardware **event**s that cannot be ignored, such as a key is pressed; or to call **subroutines** that are provided by the hardware or **operating system**.
invalid data	**Data** that is not valid, or is otherwise in conflict with some aspect of a computer **program**. In **FIX** and Coppelia, invalid data usually arises as a result of

	attempting to assign a bad value to a **variable** that accepts input only within a certain accepted range.
invalid target	Refers to a target **machine** whose ID or other necessary information is not valid in some way.
invoke	In general computing terms, to invoke means to activate. One usually speaks of invoking a **function** or **routine** in a **program**. In this sense, the term invoke is synonymous with **call**.
IOI	*acronym*. **I**ndication **O**f **I**nterest. *see* **Indication of Interest**
IOI id	In **FIX**, this **field** contains the unique ID of an **IOI**. *see also* **IOI; Indication Of Interest**.
Iona	Company that produces **CORBA** solutions, including **ORBIX** and **ORBIX-Web** (both supported by Javelin's products). http://www.iona.com. *see* **ORBIX; ORBIX-Web**.
IOR	*acronym*. **I**nteroperable **O**bject **R**eference. *see* **Interoperable Object Reference**.
IP address	An identification of a computer on the **network**, in the format of 4 octets (for example, 123.255.255.1).
J	
JAR	*acronym*. **J**ava **AR**chive. *see* **.jar file**.
Java	A **platform**-independent **programming language** developed by Sun Microsystems**.** http://www.sun.com. Coppelia is written 100% in Java. *other:* **C++; GNU C++; Machine Language; VB**.
Java Virtual Machine	A kind of translator that turns general **Java** instructions into **machine**-specific commands.
JBM Box Gateway 50	A hardware box that **bridge**s **BiSync** -> **TCP/IP network protocols**.
JBM Box Gateway 80	A hardware box that **bridge**s **X.25** -> **TCP/IP network protocols**.
JDBC	*acronym*. **J**ava **D**ata**b**ase **C**onnectivity, a solution that allows **Java application**s to connect to non-Java **database**s.
JDK	*acronym*. **J**ava **D**evelopment **K**it. System provided free by **Sun Microsystems** which can be used to develop **program**s in **Java**.
JDK environment	*see* **JDK**.
JIT	*acronym*. **J**ust **I**n **T**ime. *see* **Just In Time Compiler**.
JRE	*acronym*. **J**ava **R**untime **E**nvironment. ????????????????
Just In Time Compiler	A **Java VM** that converts Java **bytecode** into native **machine language** the first time it is encountered; this allows subsequent execution of that code to occur faster. *see* **Java VM; Machine Language**.
K	
Key change request	
keyboard	A computer's standard input **device**.
kill command	A **UNIX** administration command used to force quit an **application**. Used by **root** or **superuser**.
L	
language	A general term, which in the computing sense, refers to a formal means of issuing simple and complex commands to a **CPU** for processing. *see also* **Java; C++; GNU C++; Machine Language; VB**)
Layer	A constituent component of a **FIX session** which processes a particular class of **message**.
legacy interface	Refers to a largely outdated and unsupported **interface**.
libraries	A computing term which refers to a set of files that contain text or **compiled objects**. Their contents are usually specific to a certain task, and are used often and repeatedly by the **program**s that rely on them.
library	A collection of files, computer **program**s, or **subroutines**.
license agreement	An agreement between a company and an individual or group for software use and distribution. Usually the purchaser does not own the software, but rather, licenses it from the provider.
limit order	An order to a **broker** to buy a specified quantity of a security at or below a specified price, or to sell it at or above a specified price (known as the limit price).
listen port	A **TCP/IP port** which is devoted to listening for **incoming messages**.
listener	**A listener** is a process or a **deamon**, in **UNIX** parlance, that will wait in memory and detect information coming from a particular source. It can do this with an inter-process communication **protcol**. In **Java, Observer/Observable** can listen and react to a changes of state.
local attributes	Refers to **attributes** specific to a particular **program**, **function**, **routine**, **machine**, or **environment**. *see also* **global attributes; attribute**.
local database	A **database** stored on a local **machine**.
local machine	A local computer physically located at the present site.
local port	A **connection** on a local computer whose purpose is to connect the computer to an external **device** or **network**.

localhost	The local host is usually in the context of the an **IP address**. In that sense, the local host is the IP address of the local **machine.** Usually (but not always) the default local host adress uses the universal 127.0.0.1, a constant for all machines that use IP.
log file	*see* **.log file**.
logger	This **thread** handles the transactional **message log**ging and retrieving in Coppelia. All **inbound** and **outbound messages**, once validated, are processed by the logger thread. *other:* **main thread; manager thread; system thread; SysLog; UIThread; CommConnection; Comm; Interface**.
logical address	**IP address** that is created, but one that is not specifically assigned to one **machine**. In Coppelia terms, this refers to a single **ethernet address** that represents all of the **Coppelia Servers**. It enables both **clients** to connect to a single address that is guaranteed to be the master **server**.
logical IP address	*see* **logical network address**.
Logoff	In FIX, Logoff refers to the process of (gracefully) terminating a **FIX session**.
Logon	In FIX, Logon refers to the process of establishing a **FIX session**.
logon protocol	
Lolita	Javelin Technologies' **CMS Engine**.
long	**Java, VB** & **C++ code** for a **variable** of type long integer.
lookup	
lowercase	A general computing term indicating one or more uncapitalized **ASCII** characters. *see also* **uppercase; case-sensitive**.
M	
machine	A general term which, in the computing sense, refers to a computer, or **CPU**. *see* **CPU**.
Machine Language	Instructions that a computer can execute directly, with no other intermediate interpretation necessary. Machine Language statements are written in **binary code**, and each statement corresponds to one **machine** action. Instructions in all higher-level **programming languages** are eventually converted to Machine Language for execution by the computer. *other*: **Java; C++; GNU C++; VB**.
main thread	This is the first **thread** that gets started in Coppelia. It acts as the **manager thread** for all other threads in the **Coppelia engine**. During **startup**, it is responsible for **initializing** and starting all the necessary **subsystems** and threads. Likewise, when a **shutdown** command is issued to Coppelia, the main thread signals all other threads to stop. It waits for all threads to completely **shut down**, before shutting down the Coppelia **engine** itself. Because of this nature, this component is referred to as the **state engine**. *see also* **manager thread; state engine**. *other*: **manager thread; system thread; logger; SysLog; UIThread; CommConnection; Comm; Interface**.
makefile	A file (most often used in **UNIX**) that tells the **CPU** how to **compile** and **link program**s.
manager thread	In Coppelia, the **thread** that manages the **startup**, **initialization**, and **shutdown** of certain **subsystems** and threads. *see* **main thread**. *other*: **main thread; system thread; logger; SysLog; UIThread; CommConnection; Comm; Interface**.
map	A general term which refers to the direct 1:1 translation (or *mapping)* of a character, number, code, or value to another.
Market feed data	**Data** pertaining to dissemination of prices and quotes in the financial markets.
Market Maker	A **broker** or bank that maintains a firm bid and ask price in a given over-the-counter security by standing ready, willing, and able to buy or sell at publicly quoted prices (this is known as *making a market*).
Market Maker ID	An identifier used by a **Market Maker**, mainly in **NASDAQ** and other **exchange**s. *see also:* **Market Maker**.
market order	A buy or sell **order** in which the **broker** is to execute the order at the best price currently available.
marshalling	Defined as a client/server assignment of communication exchange.
master	Refers to any **device** that controls another device. *see also* **slave**.
matrix	*see* **array**.
Message	A general term referring to information sent from one party to another.
message format	Refers to the **format** of a **FIX message**.
Message Header	The first part of a **FIX message**, consisting of at least these elements: *BeginString, Body Length, MsgType, SenderCompID, TargetCompID, MsgSeqNum, and SendingTime*.
Message length	In **FIX**, the length of the body of a FIX **message**, excluding *BeginString, BodyLength itself, and CheckSum*, including the first **delimiter** following the *BodyLength* **field**, and the **delimiter** preceding the **CheckSum** field.
Message Oriented	???

Model	
Message Trailer	The last part of a **FIX message**, consisting of at least these elements: **CheckSum**.
message validation	
method	*see* **procedure**.
method call	*see* **function call**.
MFC	*acronym*. **M**icrosoft **F**oundation **C**lasses.
MICO	A **CORBA ORB** produced by Free Source, Inc., compatible with CORBA 2.2 at the time of this writing. http://www.mico.org. *other:* **ORBIX; ORBIX-Web; Visibroker**.
microprocessor	An integrated circuit containing the entire **CPU** on a single wafer of silicon. *see also* **CPU**.
middleware	In a **three-tier system architecture**, the system that is between the **user interface** and the **database** access software.
ML	*acronym*. **M**achine **L**anguage. *see* **Machine Language**.
module	In software parlance, a module is simply defined as part of a **program**. Programs are composed of one or more independently developed modules that are not combined until the program is linked. A single module, in turn, can contain one or several **routines**. When speaking of hardware, a module is a self-contained **component**.
Modulo	With respect to ta given **modulus**. *see* **modulus.**
modulus	*abbr.* **mod.** A number by which two given numbers can be divided and produce the same remainder.
MOM	*acronym*. **M**essage **O**riented **M**odel. *see* **Message Oriented Model**.
monitor	A computer's standard output **device**.
motherboard	The main circuit board in a computer, where all the circuitry, chips, **CPU**, and memory are located. *see also* **CPU**.
mouse	A popular input **device** that interacts with the **GUI** and **application**s via event-driven **process**es.
MQ Series	*see* **IBM MQ Series**.
MS SQL Server	**Sequel Server database** product by Microsoft Corp. http://www.microsoft.com. *other:* **Objectstore; Oracle; PsePro; SQL database; Sybase SQL Server; MS SQL Server; IBM DB2; JDBC**.
multicast communication	A way direct broadcasts of **network traffic** to specific sites.
multitasking	The apparent execution of more than one **program** at the same time. Multitasking is a basic tenet on which many popular **operating systems** operate.
multi-threaded	Multithreaded **process**es are instructions within a **procedure** which can be executed by a one or many physical **processor**s (in a manner of parallel management determined by the **operating system**) or by one or more virtual processors. A **thread** is an ensemble of the following: a start location, and a sequence of code that is to be performed independently of other threads in the **program**. Independence and cooperation among threads is programmed into the code itself with the assistance of thread management libraries. The purpose of this threading is to increase execution speed. Should not be confused with **multitasking** processes, which are different. *see also:* **thread**.
N	
NASDAQ	*acronym*. **N**ational **A**ssociation of **S**ecurities **D**ealers **A**utomated **T**ransactions
NASDAQ NWII	*see* **NASDAQ; NWII**.
Netscape Navigator	*see* **web browser**.
network	A set of computers connected together.
network address	*see* **IP Address**.
network card	Refers to an expansion board you insert into a computer so the computer can be connected to a **network**.
network controller	*see* **Network Card**.
network monitor	A network monitor's **primary function is to** gather network statistics, check **connection** statuses, manipulate **sequence numbers**, run **end of day**, and remotely **connect** or **disconnect**. Javelin's proprietary network monitor is known as **FIXometer**. *see* **FIXometer**.
Network Protocol	A standard way of regulating **data** transmission between computers on a **network**. **FIX** is designed to be independent of the network **protocol** used to transport the FIX **message** itself. *see* **Network**.
News message	
Next expected sequence number	The next **sequence number** expected to be sent FROM the local **FIX engine**, or to be received FROM the remote FIX engine, and vice versa.
Next expected value	*see* **Next expected sequence number**.

nightly build tree	Term used to indicate the **source tree** used for the automated **building** of **program**s on a nightly basis.
noise	In communications parlance, noise refers to interference (static) that destroys the integrity of signals on a line.
non-threaded object	
non-volatile stock	Business term which refers to a traded stock whose price isn't subject to a large degree of fluctuation. Strictly speaking, non-volatile stocks don't exist, because ALL stocks inherently contain *some* volatility as a part of their price adjustments. If a stock is truly non-volatile, then almost by definition, no one is trading it. The term generally refers to a security (equity) for which prices or quotes do not move many times during a day, or not significantly with respect to the amount of the movement. *see also* **volatile stock**.
null	A character that has all its **bits** set to 0. A null character, therefore, has a numeric value of 0, but it has a special meaning when interpreted as text. In some **programming languages**, notably **C++**, a null character is used to mark the end of a character string. In many **database application**s, null characters are often used as padding and are displayed as spaces.
null string	A character **string** that terminates with the **ASCII** code "0". Null strings have many **C++ language** and **UNIX** system uses.
NWII	*acronym*. **N**et**W**ork Station II. Refers to a level of service provided by **NASDAQ**.
NYSE	*acronym*. **N**ew **Y**ork **S**tock **E**xchange.
O	
object	A **data** item that has **procedure**s associated with it.
object linking and embedding	*see* **OLE**.
object marshalling	
object oriented programming	*see* **OOP**.
object reference	
object structure	
Objectstore	**Database** software provided by eXcelon Corporation, formerly Object Design, Inc. http://www.excelon.com. *other:* **Objectstore; Oracle; PsePro; SQL database; Sybase SQL Server; MS SQL Server; IBM DB2; JDBC**.
Observer/Observable	A term used to describe **objects** that can view other objects. Commonly, the term refers to **Java program**s that can call other Java programs. An **interface** through which Coppelia interacts. *other:* **Talarian SmartSockets; TIBCO Rendezvous; IBM MQ Series; ActiveX; COM/DCOM; RMI; FIXML/HTTP**.
ODBC	*acronym*. **O**pen **D**ata**b**ase **C**onnectivity, a standard **database** access method developed by Microsoft Corporation. The goal of ODBC is to make it possible to access any **data** from any **application**, regardless of which database management system (DBMS) is handling the data. ODBC manages this by inserting a middle **layer**, called a database **driver**, between an application and the DBMS. The purpose of this layer is to translate the application's data queries into commands that the DBMS understands. For this to work, both the application and the DBMS must be *ODBC-compliant.* That is, the application must be capable of issuing ODBC commands and the DBMS must be capable of responding to them.
ODI	*acronym*. **O**bject **D**esign, **I**nc. The previous name of eXcelon Corp, *see also* **Objectstore**.
OLE	*acronym*. **O**bject **L**inking and **E**mbedding. A method of combining information that is processed by different application **program**s. An example would be inserting a drawing or a slide from one application into a word processing document. The main document is called the *client*, and the document or application that supplies the embedded information is the *server*.
OMG	*acronym*. **O**bject **M**anagement **G**roup. Organization which acts as the caretakers and maintainers of the **CORBA** and **XML** specifications. http://www.omg.org. *see also* **CORBA; XML**.
OMS	*acronym*. **O**rder **M**anagement **S**ystem. *see* **Order Management System**.
OOP	*acronym*. **O**bject **O**riented **P**rogramming. Defined as a **programming** methodology in which the **programmer** can define not only **data** types, but also **procedure**s that are automatically associated with them. Examples of OOP languages are **Java, VB,** and **C++**. *see also* **Java; C++; GNU C++; VB**.
Open Link Request Broker service	
OpenLink	
operating system	A (very large, very complex) **program** that controls a computer and makes it possible for **users** to run their own programs. *see also* **WindowsNT; UNIX;**

	HP/UX; AIX.
operator	A general computing term that refers to a **user** of a system or a piece of software. *see also* **user**.
Operator's API	*see* **API**.
optional field	Refers to a **FIX field** that is not required for the proper processing of a FIX **message.**
Oracle	Oracle Corporation, provider of **databases** and related software products. http://www.oracle.com.
ORB	*acronym*. **O**bject **R**equest **B**roker. ORB is a **daemon** that handles **object message** passing between **interfaces**. *see also* **ORBIX; ORBIX-Web; CORBA**.
ORBIX	**C++ CORBA ORB** produced by **IONA** Corp. Compatible with CORBA 2.2 at the time of this writing. http://www.iona.com. *other:* **ORBIX-Web; MICO; Visibroker**.
ORBIX-Web	**Java CORBA ORB** produced by **IONA** Corp. Compatible with CORBA 2.2 at the time of this writing. http://www.iona.com *other:* **ORBIX; MICO; Visibroker**.
Order Cancel Reject	A **FIX message** whose purpose is to reject a previously-received **Order Cancel Request** or **Order Cancel Replace** for business related reasons.
Order Cancel Replace	*see* **Cancel Replace**
Order Cancel Request	*see* **Cancel Request**
Order Entry	
Order flow model	*see* **Business Flow Model**.
Order ID	A **FIX** term which refers to the unique identification number assigned to an **order**.
Order Management System	
Order Message	A buy-side **FIX message** whose purpose is to transmit information pertaining to an **order** to be executed.
Order state change matrix	*see* **Business Flow Model**.
order status	A financial term that refers to the current status of an order, such as New, Filled, Canceled, Done, for example. In **FIX,** this refers to a **field** containing coded values expressing such order status information.
Order Status message	*see* **order status**.
OSJI	*acronym*. **O**bject**S**tore **J**ava **I**nterface. An **environment variable** used by eXcelon **databases**. This is used in order for **Java application**s such as Coppelia, to be able to use **Objectstore**. *see also:* **Objectstore**.
OTC	*acronym*. **O**ver-**T**he-**C**ounter
other service	
outbound messages	*see* **outgoing messgaes**.
outgoing messages	
P	
packet	A computing term that refers to a continuous stream of characters of finite length which is sent from one computer to another.
parameter	A characteristic. For example, specifying parameters means defining the characteristics of something. In general, parameters are used to customize a **program**. *see also* **attribute**.
parse	General computing term which refers to the analysis, by the computer, of the structure of statements in a **language**. This is typically accomplished by comparing the string to a specific grammar, which defines legal and possible structures. **Compile**rs and command interpreters regularly have to parse statements in **programming languages** to make useful sense of them
ParserData	An intermediate **object** within Coppelia.
password	A secret sequence of typed characters that is required to gain access to a computer system, thus preventing unauthorized persons from gaining access to the computer.
password protected	A term that indicates a computer system, or certain parts of a computer system, require a **password** for access.
PATH	An environment name in **UNIX** and **DOS**-based **operating systems** that permits the **shell** to search for files and **directories**. A path is set to a list of **delimited** directory names, for example, *PATH=.;/usr/bin;/usr/home/myname/mysubdirectory*. The operating sytem (UNIX) will first search locally (specified by the "."); then, it checks the */usr/bin*; then finally, */usr/home/myname/mysubdirectory*, in that order.
peek	
persistence	The process of storing **data** or state information for retrieval and use at a later time, outside of memory to prevent loss of that information after the process terminates.

persistent database	**Database** used to maintain **persistent** data for a **FIX Session**. *see* **persistence**.
persistentDB	*see* **persistent database**.
PersistentDBPipe	
Phaos SSLava	An **SSL library** created by Phaos Corp. http://phaos.com. *see* **SSL**.
physical line	
ping	A **TCP/IP network** command which sends out a test message to another site and waits for a response. The amount of time it takes the remote site to respond is known as its *ping time*.
pipe	In **UNIX** and **DOS**, a pipe is a way of stringing two **program**s together so that the output of one is fed into the other as input.
pipeline	
platform	*see* **operating system**.
pointer	A **data** item containing an **address** to a specific memory location. Used primarily to tell computer **program**s where in memory to find a particular item.
Point-to-point communication	A term which refers to a communication link between two specific parties.
polling	Polling is a methodology in which a **process** waits or "**listen**s" for an **event** to occur. The process will cycle through the list of processes, **device driver ports,** or **object** testng methods, if they need service. This is a **synchronous determnistic** method of servicing requests. Another case is **event-**driven models such as **interrupts**. *see also* **interrupt**.
polling model	Any process that adheres to a **polling** regime. *see* **polling**.
port	A **connection** where a computer can be connected to an external **device**. Also, a unique number used by a **microprocessor** to identify an input-output device. Or, a number identifying the type of connection requested by a **remote** computer.
port number	
portability	Refers to a (desireable) property of a **program** to be able to run on more than one type of computer or **platform**.
possible values	
prefix	
price	The price of one share of a particular stock.
private	
procedure	A general computing term that refers to a smaller **program** contained within a larger program. It is typically executed when the main program calls it. Procedures serve the advantage of not having to write the same instructions multiple times through the course of a program. *see also* **function; method; algorithm**.
process	A series of instructions that a computer executes in a **multitasking operating system**. Many processes execute concurrently.
process space	The amount of space in a computer's **Virtual Memory** store that a given **program** occupies.
processor	A chip that is largely dedicated to taking **data** and instructions for **process**ing the data either serially or in parallel. *see also* **microprocessor; motherboard; CPU**.
profile	A file of saved information that indicates how the **user** normally wants something done.
program	General computing term that refers to a finite set of instructions sent to a computer for execution. Programs can implement solutions ranging from the simplest concepts to problems of nearly infinite complexity.
programmatic logic	
programmer	*see* **software developer**.
programmer's interface	*see* **Application Program Interface; API**.
programming	General computing term that refers to the practice of writing **programs**. *see also* **development; development team; software developer**.
protected	
protocol	A standard way of regulating **data** transmission between computers.
proxy	To complete a task, request, or other action on behalf of another party.
proxy server	A computer that saves information acquired from elsewhere on a network and makes it available to other computers in the immediate area.
PsePro	Lightweight **database** supported by Javelin's Coppelia. Created by eXcelon Corp. http://www.excelon.com. *other:* **Objectstore; Oracle; SQL database; Sybase SQL Server; MS SQL Server; IBM DB2; JDBC**.
public	
publish/subscribe	Otherwise known as the **Broadcast** method of sending out **data**.

model	
pump	An internal Coppelia command which is responsible for validating the **incoming messages** and converting them to other **middleware objects**. It is exactly the same as **formatter**, but the reverse side. *see also* **formatter**.
purge	
Q	
QSR	related to ACT reporting
QSROFF	related to ACT reporting
QSRON	related to ACT reporting
quality	
quantity	The number of shares of a stock transacted.
query	A general computing term that refers to a request for information from a **database**.
queue	A list, maintained by an **application** or **operating system**, of jobs or **messages** waiting to be processed.
queue element	A single element within a **queue**.
queue prefix	
queue size	Refers to how many elements are currently stored in a particular **queue**.
queue state information	
queue suffix	
R	
Raw Data Messages	
raw data string	
raw FIX string	
RDM	*acronym*. **R**aw **D**ata **M**essages. *see* **Raw Data Messages**.
Receiver	In **FIX,** the target of a FIX **message**.
reconfigure	To rearrange the elements or settings that feed into a **program** or **process**.
Reconnect	The act of re-establishing a previously-terminated **connection**.
record	Defined as a collection of **field**s. *see also* **field**.
recursion	A computing term that refers to the calling of a **procedure** by itself, thereby creating a new copy of the procedure.
References	
registry	The part of an **operating system** that stores setup information for the hardware, software, and operating system itself.
regression test	A test whose purpose is to verify that what worked previously still works.
Reject message	A **FIX message** whose purpose is to reject a previously-received FIX message for reasons of non-compliance with respect to the **FIX Prototol**.
relative path	A term which refers to the **path** to a **directory** or file, relative to the **current path**. For example, when in the *D:\Coppelia* directory, the relative path to the file *buy.dat* would be *\buy\buy.dat*. *see also* **absolute path**.
Release Process	
remote client	
remote database	A **database** stored on a **remote machine**.
remote link	
remote machine	A computer physically located at a remote site.
Remote Method Invocation	A **Java method** that accesses Java objects on a remote **machine**. A popular Coppelia **interface**. *other:* **CORBA; Talarian SmartSockets; Observer/Observable; TIBCO Rendezvous; IBM MQ Series; ActiveX; COM/DCOM; FIXML/HTTP**.
remote object	
remote port	The **TCP/IP** port which is assigned to the remote machine you want to connect to.
Remote Procedure Calls	Defined as the invocation of a **procedure** on a **remote machine**.
remote server	
repeating fields	In **FIX**, repeating fields are defined as **field**s that are allowed to repeat for one purpose or another. A good example of this occurs in an *Allocation* **message**, where certain fields are required to repeat in order to properly communicate all the relevant information pertaining to the trade allocation.
required field	In **FIX,** a **field** that is required for proper **message** handling.
Resend Request	A **FIX message** whose purpose is to recover eventually lost **messages** from the **counter-party**.
resource	Generally, any item that can be used. **Device**s such as printers and disk drives are resources, as is memory. In many **operating systems**, including **WindowsNT** operating system, the term resource refers specifically to **data** or

	routines that are available to **programs**. These are also called system resources.
return code	A code returned by a **function,** usually due to a failure in execution.
RMI	*acronym*. **R**emote **M**ethod **I**nvocation, the **Java**-equivalent of **RPC** (**R**emote **P**rocedure **C**all). *see* **Remote Method Invocation**.
RMI encryption	
RMI registry	
root	Usually, root is the account name used by the system administrator under **UNIX**. Also refers to the **root directory** of a disk. *see also* **root directory**.
root directory	A term which refers to the top-most directory in a **heirarchy**. It is the directory in which all **subdirectories** are located. In this sense, it is the *root*, or base, of a tree-like **directory** structure.
router	A **device** designed to route **TCP/IP**-formatted information (**packets**) from a point of origin to a particular destination.
router configuration	Refers to the software configuration of a **router**. *see* **router**.
routine	*see* **procedure; function; method; algorithm**.
RPC	*acronym*. **R**emote **P**rocedure **C**alls. *see* **Remote Procedure Calls**.
running	A **program** or **algorithm** that is in a state of being executed.
runtime	A computing term that refers to the time when a **program** is executed.
runtime directory	A computing term that refers to the **directory** in which a **program** is typically executed.
Run-Time for Java	*see* **Java Virtual Machine**.
RV daemon	**Daemon** created by **TIBCO** Finance for their **Rendezvous** messaging system.
S	
sample code	A template, usually in restricted form, illustrating a working example of a process in **code** form.
scalability	The ability of an **application** or **process** to grow in size, capacity or number to accommodate an application's increasing demand for processing power, **data** throughput, or performance.
scenario	Defined generally as an outline or a model of an expected or a supposed sequence of events.
script	A file containing commands to be executed. *see* **shell script**.
securities	In general business parlance, a security is defined as evidence of a secured indebtedness or of a right created in the holder to participate in the profits or assets distribution of a profit-making enterprise. Also, securities can refer to those written assurances for the return or payment of money. Additionally, securities are often defined as instruments giving to their legal holders a right to money or other property.
Sell Side	A summary term referring to all **broker** firms. Note: A sell-side firm can also buy securities. *see also* **buy side**.
send date	In **FIX,** the date a **message** was sent.
send time	In **FIX,** the time a **message** was sent.
Sender	In **FIX,** the originator of a FIX **message**.
sender	Refers to the party on the sending **side** of a transaction or **message**.
Sequel	*see* **SQL database**.
Sequence Counter	A process within a **FIX engine** that keeps track of all incoming and outgoing **sequence numbers**.
Sequence number	An incrementing integer number assigned to all **FIX messages** (FIX version 4.0 and above), or to certain FIX messages (FIX 3.0 and below).
Sequence Reset Message	A **FIX message** whose purpose is to reset the outgoing **sequence number**, or (FIX 4.0 and above) to fill over a gap in **sequence numbers** after a **Resend Request**.
sequence string	**Java, VB** & **C++ code** for a **variable** of type **sequence string**.
sequences	
Server	In **FIX**, the **session acceptor**.
server application	
server block	
server certificate	
Services	
Session	*see* **FIX Session.**
Session acceptor	In **FIX,** the party that listens on a **port** for incoming requests to open a **socket**, and replies to the first **logon**.
Session Data	**Data** or other information pertaining to a **FIX** Session Layer or other function. *see also* **administrative message; Session Layer**.
Session initiator	In **FIX,** the party that establishes the **socket connection**, and sends the first **logon**.
Session Layer	A **layer** within a **FIX session** whose purpose is to handle **administrative**

	messages and to ensure message delivery.
session level integrity	
Session level logic	In **FIX,** refers to the entirety of specified rules pertaining to the delivery and recovery of FIX **messages**.
session listener	*see* **listener**.
session monitor	
session notification	
Session Termination	*see* **Logout**.
setup	The process of **installi**ng and configuring a software package to run properly on a computer system. *see* **install**.
shareware	Software that is initially **download**able for free, but usually with limited functionality. After a trial period, the user can either pay a registration fee to unlock the **program**'s complete functionality, or delete it. Alternatively, some shareware (either with full or limited functionality) will expire after a specified amount of time on your system, at which point you can register or delete it.
shell	A program that accepts **operating system** commands and causes them to be executed.
shell script	A file of commands to be executed by the **shell**. In **UNIX,** a shell script can begin with a line to indicate which of several shells should process it.
short	A financial term describing a transaction in which someone sells securities he or she doesn't own. Also, the term describes the resulting position after such a transaction.
shutdown	General computing term referring to the closing of all running **program**s (including, most importantly, the **operating system**) in preparation for the turning off of the computer. In Coppelia, this term can refer both to the aforementioned description, or to the process of terminating a **FIX session**. *see also* **startup**.
SIAC	Industry Utility that creates, implements, and maintains a wide array of computer systems for **NYSE** and **AMEX.**
side	Refers to the a particular party on one side of a connection
Single-process	Software that consists of only one **process** as opposed to multiple processes.
singleton	
site	General term which refers to a specific location, usually one where a computer is present.
slash	Refers to the **ASCII** character "/".
slave	Refers to any **device** that is controlled by another device. *see also* **master**.
slave thread	
SLD modifier	related to ACT reporting
sleep	
socket	A communication path between two computer **program**s not necessarily running on the same **machine**. In **UNIX** and some other **operating systems**, a socket refers to a software **object** that connects an **application** to a **network protocol**. In UNIX, for example, a program can send and receive **TCP/IP messages** by opening a socket and reading and writing **data** to and from the socket. This simplifies program **development** because the **programmer** need only worry about manipulating the socket and can rely on the operating system to actually transport messages across the network correctly. Note that a socket in this sense is completely soft - it's a software object, not a physical component.
socket connection	
socket factory	
socket level	
software developer	A person who develops software **application**s within a certain **programming language**.
Solaris	A **UNIX**-based **operating system** for computers, mainly used on machines built by Sun Microsystems. *see* **SUN Solaris**.
Solaris OS	*see* **SUN Solaris**.
Soltice	**Solaris** product used for **X.25 connectivity** through Solaris. No longer supported by Lolita.
Source Control	Also known as Version control. version control of program source
source file	
source queue	**MQ Series queue** for inbound **messages**.
source tree	term used for version control which indicates the branch that a cobase is stored at
SQL database	*acronym*. **S**tructured **Q**uery **L**anguage, formerly known as **Sequel**. A standard **query language** used by many **program**s that manipulate large **databases**.
SSL encryption	*acronym*. **S**ecure **S**ocket **L**ayer Encryption. A method of **encryption** which

	protects the privacy of data exchanged between websites and individual **users**.
Standard Coppelia	Multiple **FIX Connection Coppelia Server**. *other*: **Coppelia EP; Coppelia Single Connect; Coppelia HA; Coppelia**.
Standard header	*see* **Message Header**.
standard object model	
Standard trailer	*see* **Message Trailer**.
startup	General computing term referring to the process by which a computer loads the operating system upon initially turning the computer power on. In Coppelia, startup refers to the process of launching an **instance** of a Coppelia **engine**.
startup script	Defined generally as a **script** which is executed upon a computer's, or **program**'s **startup**.
State change diagram	*see* **Business Flow Model**.
State Engine	engine that uses stateful connections
static library	A **library** that is statically linked into a **program**, and must be included in the **build** for it to work. *see also* **dynamic library**.
statistics	A Coppelia term which refers to various **buy-side** and **sell-side connection**-related information. *see* **stats**.
stats	A Coppelia command issued from either **side** which displays various **connection**-related information and statistics
stderr	
stock	A business term which refers to the partial ownership of a corporation represented by **shares** that are a claim on the corporation's earnings.
string	**Java, VB** & **C++ code** for a **variable** of type string.
string pair	
structure	General computing term which refers to an organized framework around which something is built. *see* **data structure**.
stub code	Term that refers to **code** that only contains **function** declarations. The tactic is used primarily by **programmer**s when designing **development** solutions.
sub id	The ID of a person (**trader**, **broker**) who is affiliated with a particular **side** of a trade.
Sub routing field	In **FIX,** refers to any of *TargetSubID, SenderSubID, OnBehalfOfSubID, or DeliverToSubID*.
subdirectory	A **directory** that lies within another directory. For example, if *D:\foo* is a directory, then any directories contained within *foo* are considered subdirectories of it. Note that this definition is recursive in nature. That is to say, directories can have subdirectories, and those subdirectories themselves can have subdirectories, and so forth, ad infinitum.
subdomain	
subroutine	A set of instructions, given a particular name or **address,** that will be executed when the main **program** calls for it.
subsystem	
sub-system	*see* **subsystem**.
suffix	Refers to the three-letter **extension** after the "." in most filenames.
SUN Solaris	A **UNIX** based **operating system,** created, distributed, and maintained by Sun Microsystems. http://www.sun.com *other:* **WindowsNT; HP/UX; UNIX; AIX**.
superuser	A **UNIX** login name which grants the **user** special **access rights**.
Sybase	Company that produces **databases**. http://www.sybase.com. *see* **SQL Database; Sybase SQL Server**.
Sybase SQL Server	**Sequel Server database** product by Sybase, Inc. *see* **SQL Database; Sybase**.
symbol	The 3- or 4-letter **NYSE, AMEX,** or **NASDAQ** code which refers to a company trading on that exchange.
syntax	General computing term which refers to a set of rules that specify how the symbols of a **language** can be put together to form meaningful statements. In **FIX,** syntax is defined by the FIX specification for the version being run; in Coppelia, there are many **program**matic syntaxes to which the **developer**, and **user**, must adhere.
SysLog	This **thread log**s all the activities in a Coppelia **session** into a plain **ASCII text file**. These activities include all **inbound** and **outbound messages**, **error messages**, **debug** information, etc. *other:* **main thread; manager thread; system thread; logger; UIThread; CommConnection; Comm; Interface**.
syslogd	**UNIX log**ging **daemon**.
system administration	The process of managing a multi-user computer or **network** of computers.
System DSN	
System Log	Term used to describe the **log** that contains system-related **messages**.
system thread	*other*: **main thread; manager thread; logger; SysLog; UIThread; CommConnection; Comm; Interface**.

T

T modifier	related to ACT reporting
table	An arrangement of **data** in a **database** where each row defines a relationship between the items in that row.
Table of Fields	A **table** that contains information pertaining to required and optional **fields**.
Table of Return Codes	A **table** that contains information pertaining to codes returned by Coppelia.
tag	In **FIX**, a term meaning the smallest meaningful element of a **FIX message**.
tag/value pair	In **FIX**, a term describing the structure of a **tag**. *see also* **tag**.
Talarian SmartSockets	A **middleware** package with which Javelin's Coppelia **interface**s. Produced by Talarian, Inc. http://www.talarian.com *other*: **CORBA; Observer/Observable; TIBCO Rendezvous; IBM MQ Series; ActiveX; COM/DCOM; RMI; FIXML/HTTP**.
target	Refers to the party on the receiving **side** of a transaction or **message**.
target id	The ID of the target **machine** you wish to connect to.
target queue	**MQ Series queue** for outbound **messages**.
TCP/IP	*acronym*. **T**ransmission **C**ontrol **P**rotocol / **I**nternet **P**rotocol. *see* **TCP/IP Protocol.** *other:* **X.25.**
TCP/IP packet	*see* **packet**.
TCP/IP Protocol	A standard format for transmitting **data** in **packets** from one computer to another. It is used on the internet and various other **networks**. TCP deals with the construction of packets, and IP deals with routing them from machine to machine. Coppelia uses the TCP/IP protocol exclusively. *other:* **X.25**.
telco	*see* **telecommunication**
telecommunication	The electronic systems used in transmitting **message**s.
Telnet	A **UNIX** command that lets an individual use a computer as a **terminal** on another computer through a **network.**
terminal	An input-output **device** for communicating with a computer, typically consisting of a **monitor**, **keyboard,** and **mouse.**
terminal window	A terminal screen. *see* **terminal**.
test environment	A terminal screen. *see* **terminal**.
test request	
test suite	A collection of tests designed to test critical **application** functionality, exception handling, and **usability**.
Third-party	In **FIX,** when a business-to-business transaction involves an intermediary FIX **engine** between the actual trading partners, such as a FIX network vendor.
third-party software	Software that does not come from the manufacturer, nor the end user.
thread	Refers to a task or **process** in a computer **program** that uses **multitasking**. In Coppelia, threading provides the **engine** with a significant amount of its processing flexibility and power. *see* **main thread; manager thread; system thread; logger; SysLog; UIThread; CommConnection; Comm; Interface**.
threaded object	
threads interaction	Refers to the interaction of different **threads** to create a smooth program flow.
three-tier architecture	A **database** system with a **user interface** running on a microcomputer, a database **engine** running on a mainframe computer, and a **layer** of **middleware** that connects these two tiers.
throw an exception	Term which refers to a program encountering an **event** that it cannot handle in its normal processes. *see* **exception**.
TIB/Rendezvous	A **middleware** package with which Coppelia **interface**s. Provided by Tibco Software. http://www.tibco.com. *other:* **CORBA; Talarian SmartSockets; Observer/Observable; IBM MQ Series; ActiveX; COM/DCOM; RMI; FIXML/HTTP.**
TIBCO TIB/Rendezvous (RV)	A popular messaging system from **TIBCO** Finance which uses a **publish/subscribe** architecture. *see* **TIB/Rendezvous**.
Tier 1 support	Defined as top level support. The highest level of support available.
time in force	A financial term describing the time span for which a certain **order** is to be valid.
timer	
trade	A transaction of a security or commodity, also referred to as barter.
Trade Entry message	In **ACT**, the **message** that transmits the **data** necessary to create an entry in **NASDAQ's ACT** system.
trade log file	
Trade Time	The time a **trade** is effected.
trade type	In **FIX**, this term refers to the type of trade taking place in a given FIX **message** (typically, a **market order** or **limit order**).
trader	Generally, one who buys and sells securities for his/her personal account, not on behalf of clients.
tradetable	An option in Coppelia related to the demo **GUI**.

trading system	
trading volume	Business term which refers to the number of **shares** of a particular security that are traded within a certain time frame.
traffic	General computing term that refers to the amount of information traveling across a **network** at any given time.
trailer object	An **object** which defines and initializes the **Standard Trailer**. see **Standard Trailer**.
trailing backslash	Refers to the addition of the "\" character at the end of a **directory** path. For example: *D:\Coppelia\buy\buy.dat*
trailing slash	Refers to the addition of the "/" character at the end of a **directory** path. For example: *http://javtech.com/*
transaction pipeline	
two-dimensional array	Defined as an **array** with two dimensional indices. For example, the array *foo[]* is one dimensional, whereas *bar[][]* is two dimensional. That is to say, there are two indexes associated with it
U	
UDB	*acronym*: **U**niversal **D**ata**b**ase. *see* **IBM DB2**.
UIRemote object	An **object** that refers to the Admin Object provided by Coppelia.
UIThread	This **thread** provides the **console interface** to Coppelia, handling all available console commands that are issued. *other:* **main thread; manager thread; system thread; logger; SysLog; CommConnection; Comm; Interface.**
UML	*acronym*. **U**nified **M**odeling **L**anguage. A **language** designed to describe **object-oriented** interactions.
undelivered messages	
unencrypted	Refers to **data** that is not encrypted. In Coppelia, unencrypted **messages** travel to their destination free of **encryption**, and are therefore more susceptible to observation or tampering from unauthorized parties.
unit tests	Tests that only test a specific **module** of a larger **program** or **application**.
UNIX	An **operating system** for computers. *other*: **WindowsNT; HP/UX; AIX; SUN Solaris**.
unzip	The process of expanding a **zip**-compressed archive into it's original, constituent parts. *see* **zip; zip file**.
upload	Defined as the transfer of a file from a local (usually smaller) computer to a computer at a remote location.
uppercase	A general computing term indicating one or more capitalized **ASCII** characters. *see also* **lowercase; case-sensitive**.
uptime	Term that refers to the span of time a computer or **connection** between computers is up or active.
URI	*see* **URL**.
URL	*acronym*. **U**niform/**U**niveral **R**esource **L**ocator. A way of specifying the location of publicly available information on the internet.
user	General term referring to an indiviudal using a software **application** on a computer or **network**. *see* **operator**.
user interface	A general computing term which refers to any means by which a **user** can communicate with a computer. *see also* **interface; GUI**.
user site	
user-defined message	
username	A text string entered by the **user** which, if recognized by the **host** system, serves to log in the user to that system. In some cases, the username corresponds to a user **profile**, which is used to grant the user certain access privileges on the system.
V	
ValidatedData	This is equivalent to a validated **object**.
valuation	A Business term which refers to an estimated worth or price. Also, valuation can be defined as the *act* of estimating or appraising the value of a particular stock or commodity.
variable	A computing term that refers to a symbol or name that stands for a value.
VB	*acronym*. **V**isual **B**asic. *see* **Visual Basic**.
vector	A memory location containing the **address** of some **code**, often some kind of **exception** handler or other operating system service. By changing the vector to point to a different piece of code it is possible to modify the behavior of the operating system.
vendor	Refers to a company which provides a certain product or service.
version	Refers to the release version of a software package. Coppelia, for instance, recently released version 4.1e08.
version number	Refers to a particular release **version** of a piece of software. *see* **version**.
Visibroker	A **CORBA ORB** produced by Visigenics/Inprise Corp. It is **CORBA** 2.0 compliant

	as of this writing. http://www.inprise.com. *other:* **MICO; ORBIX; ORBIX-Web**.
Visual Basic	A Microsoft **language** which is an extension of BASIC. Microsoft extends the language with a usful **event-driven syntax** for a **graphical user interface**. *see* **VB**. *other*: **C++; GNU C++; Machine Language; Java**.
volatile stock	Business term which refers to a traded stock whose price is subject to large degrees of fluctuation. Strictly speaking, ALL stocks inherently contain *some* volatility as a part of their price adjustments. The term generally refers to a security (equity) for which prices or quotes move many times during a day, or in some other significant fashion with respect to the amount of the movement. *see also* **non-volatile stock**.
volatility	Financial term which refers to the relative amplitude with which the trading value of a particular stock fluctuates over a given time period.
volume	Business term which refers to the total number of **stock** shares, bonds, or commodities traded in a particular period.
W	
web browser	A **program**, such as **Netscape Navigator**, that enables the **user** to read **hypertext** in files or on the World Wide Web.
web-server	Defined generally as a computer that delivers (serves up) web pages.
white space	General computing term which defines characters that don't appear on the screen when entered (e.g. spaces, tabs, carriage returns, etc.)
WindowsNT	An popular and powerful 32-bit, Microsoft Windows-based operating system for computers. *other:* **UNIX; HP/UX; AIX; SUN Solaris**.
Witold	A rare species of Homo Sapiens, Phylum: SAMES, adept and highly specialized in FIX across all versions.
X	
X.25	A type of communication **protocol** that defines a standard way of transmitting **data** in **packets**. *see also* **protocol**. *other:* **TCP/IP**.
XML	*acronym*. E**x**tensible **M**arkup **L**anguage, a standard used to communicate information from computer to computer.
Z	
zip	A popular **format** of file **compression**. *see* **.zip file**.

9.3 FIX Specifications

9.3.1 FIX 2.7

http://www.fixprotocol.org/specification/spec_27a.doc

9.3.2 FIX 3.0

http://www.fixprotocol.org/specification/fix30a.doc

9.3.3 FIX 4.0

http://www.fixprotocol.org/specification/fix40.doc

9.3.4 FIX 4.1

http://www.fixprotocol.org/specification/fix41.doc

9.3.4.1 FIX 4.1 with Errata as of June 30, 1999

http://www.fixprotocol.org/specification/fix-41-with_errata_19990630.html

9.3.5 FIX 4.2

http://www.fixprotocol.org/specification/fix-42.html

9.4 Protocol Implementation Notes

(Section TBA)

9.4.1 Registered User-defined fields

http://www.fixprotocol.org/cgi-bin/rbox/Fields.cgi?menu=51

9.4.2 Encryption Implementation Notes

9.5 Coppelia Notes

9.5.1 List of Coppelia Files

(Section TBA)

9.5.2 List of Coppelia Configuration Options

(Section TBA)

9.5.3 FIX Performance

(Section TBA)

9.6 Test Scripts

(Section TBA)

9.6.1 Coppelia Router – RESTRICTED ACCESS

9.7 Java for the first Time

(Section TBA)

9.8 PGP Usage – RESTRICTED ACCESS