MOCKETS DOCUMENTATION

Public API and messages.

Last revision: 2016

Contact info: nsuri@ihmc.us rfronteddu@ihmc.us

Contents

1. **INTRODUCTION** 1
1.1. Mockets Behavior Summary: 1
1.2. Heartbeat 1
2. **MOCKET API** 3
2.1. Mocket.h 3
Mocket (3) 3
setIdentifier (1) 3
const char* **getIdentifier** (0) 3
int **registerPeerUnreachableWarningCallback** (2) 4
int **registerSuspendReceivedWarningCallback** (2) 4
int **registerPeerReachableCallback** (2) 4
MocketStats * **getStatistics** (0) 5
int **bind** (2) 5
int **connect** (2) 5
int **connect** (3) 6
int **connect** (4) 6
int **reEstablishConn** (1) 6
int **resumeAndRestoreState** (2) 7
int **connectAsync** (2) 7
int **finishConnect** (0) 8
uint32 **getRemoteAddress** (0) 8
uint16 **getRemotePort** (0) 8
uint32 **getLocalAddress** (0) 9
uint16 **getLocalPort** (0) 9
bool **isConnected** (0) 9
int **close** (0) 10
int **suspend** (2) 10
int **getState** (1) 10
int **enableCrossSequencing** (1) 11
bool **isCrossSequencingEnabled** (0) 11
MessageSender **getSender** (2) 11
uint32 **getOutgoingBufferSize** (0) 11
int **send** (8) 12
int **gsend** (8+) 12
Int **gsend** (10) 14
int **getNextMessageSize** (1) 14
uint32 **getCumulativeSizeOfAvailableMessages** (0) 15
int **receive** (3) 15
int **receive** (2) 16
int **sreceive** (3+) 16
int **replace** (9) 17
int **cancel** (3) 17
int **setConnectionLingerTime** (1) 18
uint32 **getConnectionLingerTime** (0) 18
uint16 **getMTU** (0) 18
static uint16 **getMaximumMTU** (0) 19
int **activateBandwidthEstimation** (1) 19
int **activateCongestionControl** (0) 19
void **debugStateCapture** (0) 20
void **useTwoWayHandshake** (0) 20
int **setTransmitRateLimit** (1) 20
API METHODS FOR EASY CONFIGURATION OF SATELLITE LINKS 21
void **setKeepAliveTimeout** (1) 21
void **disableKeepAlive** (0) 21
void **setInitialAssumedRTT** (1) 21
void **setMaximumRTO** (1) 22
void **setMinimumRTO** (1) 22
void **setRTOFactor** (1) 22
void **setRTOConstant** (1) 23
void **disableRetransmitCountFactorInRTO** (0) 23
void **setMaximumWindowSize** (1) 23
void **setSAckTransmitTimeout** (1) 24
void **setConnectTimeout** (1) 24
void **setUDPReceiveConnectionTimeout** (1) 24
void **setUDPReceiveTimeout** (1) 25
API METHODS FOR GENERAL UTILITIES 26
int **readConfigFile** (1) 26
void **resetTransmissionCounters** (0) 26
void **enableTransmitLogging** (1) 26
2.2. MessageSender.h 27
MessageSender (1) 27
int **send** (2) 27
int **gsend** (2+) 27
int **send** (4) 28
int **send** (3) 28
int **send** (6) 29
int **replace** (4 29
int **replace** (4) 30
int **replace** (7) 30
int **cancel** (1) 31
void **setDefaultEnqueueTimeout** (1) 31
void **setDefaultRetryTimeout** (1) 31
2.3. ServerMocket.h 32
ServerMocket (3) 32
int **listen** (1) 32
int **listen** (2) 32
Mocket * **accept** (1) 33
int **close** (0) 33
void **setIdentifier** (1) 33

const char * **getIdentifier** (0) 34

3. **MESSAGES DESCRIPTION** 35

3.1. Common message packet header 35

3.2. Version/flags 36

3.3. Delivery prerequisites option header 36

3.4. Chunk format: 37

3.5. Metadata Chunks description 39

CT_**SAck** 39

Acknowledge information 39

CT_**Heartbeat** 41

CT_**Cancelled** 41

CT_**Timestamp** 41

CT_**TimestampAck** 42

CT_**SAckRecBandEst** 42

3.6. StateChange Chunks description 43

Ct_**Init** 43

CT_**InitAck** 43

CT_**CookieEcho** 44

CT_**CookieAck** 44

State Cookie 44

CT_**Shutdown** 45

CT_**ShutdownAck** 45

CT_**ShutdownComplete** 45

CT_**Abort** 45

CT_**Suspend** 46

CT_**SuspendAck** 46

CT_**Resume** 46

CT_**ResumeAck** 46

CT_**ReEstablish** 46

CT_**ReEstablishAck** 46

CT_**SimpleSuspend** 46

CT_**SimpleSuspendAck** 47

CT_**SimpleConnect** 47

CT_**SimpleConnectAck** 47

3.7. Data Chunks description 48

CT_**Data** 48

1. INTRODUCTION

Problems that TCP experiences in tactical environments have been extensively studied and were the subject of many researches. Mockets were designed keeping in mind the specific problems of Tactical Edge Networks which are typically wireless and ad-hoc, with low bandwidth, intermittent connectivity, and variable latency. Therefore, communications in this type of networks are subject to highly variable conditions and exhibit significant reliability and performance problems. The Mockets framework is a communications middleware specifically designed to address the challenges of Mobile Ad hoc Network (**MANET**) scenarios, it provides the transport capability to applications and is a replacement for TCP and UDP sockets.

1.1. Mockets Behavior Summary:

Mockets adopts the traditional Client/Server programming paradigm of Sockets and provides a message-oriented communication API with advanced functionalities to manage endpoints mobility and monitor network conditions. All the messages exchanged by the Mockets communication protocol are encapsulated in UDP packets. Reliability and stream abstractions are provided by Mockets on top of the unreliable UDP packet delivery service.

Communications between two unicast datagram Mockets are established by connecting an active Mocket to a passive one listening on a peer that, apart from the case of explicit endpoint migration commands issued at middleware or application level, will not change during the entire communication.

On Server side, the communication is not established on the same port on which the server application listens for incoming connections, but on a new, system-assigned port. If a Server Mocket receives more than one Connection Request Message (the equivalent of a SYN packet in TCP) from the same peer application, it assumes that SYN_ACK replies are not reaching the sender and it automatically increases the frequency of acknowledgements for that connection. The self-regulating acknowledgements mechanism allows Mockets to dynamically adapt to specific network conditions on a per-connection basis, increasing the chances of a success and avoiding unnecessary additional traffic for protocol negotiation.

Mockets Middleware allows applications to exploit one or more delivery services on a per-message basis by choosing orthogonally between:

- Reliable or Unreliable;
- Sequenced or Unsequenced.

Sequenced Reliable provides semantics similar to TCP, S**equenced Unreliable** similar to RTP, **Reliable Unsequenced** it's suited for important but unrelated messages while **Unreliable Unsequenced** provides semantics similar to UDP.

1.2. Heartbeat

If no message has been sent to (or received) from the peer application during a configurable amount of time, Mockets will automatically send a Heartbeat message to the remote communication endpoint. This simple keep-alive mechanism allows quick detection of problems at the link and network layers. The transmitting endpoint will generate a Heartbeat every second if no data packets need to be sent; therefore, the receiving endpoint is able to detect a disconnection which is not due to a request from the remote application.

2. MOCKET API

2.1. Mocket.h

Mocket *(3)*

Return type

- void

Parameters

1. const char *pszConfigFile = NULL	Path to the configuration file.
2. CommInterface *pCI = NULL	
3. bool bDeleteCIWhenDone = false	

Description

The main class for a client application to use the Mockets communication library. Similar in functionality to a socket it is used by a client to establish a connection to a server and then communicate with the server.

setIdentifier (1)

Return type:

- void

Parameters

1. const char *pszIdentifier	User friendly identifier for this Mocket instance.

Description

Sets a string to use as the application or user friendly identifier for this Mocket instance. The identifier is used when sending out statistics and when logging information (some suggestions include the name of the application, the purpose for this Mocket, ...). The pszIdentifier may be set to NULL to clear a previously set identifier.

Note:

The string is copied internally, so the caller does not need to preserve it.

*const char** getIdentifier *(0)*

Return type:

- const char*	Mocket's instance identifier.

Parameters

1. void

Description

Returns the user friendly identifier for this Mocket instance, will return NULL if there is no identifier set.

Note:

int registerPeerUnreachableWarningCallback *(2)*

Return type:

•	int	Time in ms since last contact.

Parameters

1.	PeerUnreachableWarningCallbackFnPtr pCallbackFn	Callback function to be invoked.
2.	void *pCallbackArg	Pointer to callback argument.

Description

Register a callback function to be invoked when no data (or Keepalive) has been received from the peer's Mocket. The callback will indicate the time (in milliseconds) since last contact, if the callback returns true, the Mocket connection will be closed. An optional argument (which will be passed in during the callback) may be passed when setting the callback.

Note:

int registerSuspendReceivedWarningCallback *(2)*

Return type:

•	int	Time in ms since the connection has been suspended.

Parameters

1.	PeerUnreachableWarningCallbackFnPtr pCallbackFn	Callback function to be invoked.
2.	void *pCallbackArg	Pointer to callback argument.

Description

Register a callback function to be invoked when a suspend message has been received. The callback will indicate the time (in milliseconds) since the connection has been suspended. An optional argument may be passed when setting the callback which will be passed in during the callback.

Note:

int registerPeerReachableCallback *(2)*

Return type:

•	int	Time in ms since last contact.

Parameters

1.	PeerReachableCallbackFnPtr pCallbackFn	Callback function to be invoked.
2.	void *pCallbackArg	Pointer to callback argument.

Description

Register a callback function to be invoked once peerUnreachable has been invoked (and subsequently we have heard from the peer). The callback will indicate the time (in milliseconds) since last contact. An optional argument may be passed when setting the callback which will be passed in during the callback.

Note:

*MocketStats * *getStatistics *(0)*

Return type:

• MocketStats *	Pointer to Statistics class.

Parameters

1. void

Description

Returns a pointer to the Statistics class that maintains statistics about this Mocket connection.

Note:

This must not be deallocated by the caller.

int bind *(2)*

Return type:

• int	0	Success.
	<	Error code.

Parameters

1. const char *pszBindAddr	Address string
2. uint16 ui16BindPort	Port

Description

Binds the local end point to a particular address (interface) and port. Calls to this method will work if invoked before calling 'connect()', otherwise, it will return an error code.

Note:

Must be invoked before connect.

int connect (2)

Return type:

• int	0	Success.
	<0	Error code.

Parameters

1. const char *pszRemoteHost	String containing the remote host address.
2. uint16 ui16RemotePort	Remote host port.

Description

Attempt to connect to a Mocket Server at the specified remote host on the specified remote port. The host may be a hostname that can be resolved to an IP address or an IP address in string format (e.g. "127.0.0.1"). The default connect timeout is 30 seconds. Returns 0 if successful or a negative value in case of failure.

Note:

int connect *(3)*

Return type:

• int	0	Success
	<0	Error code

Parameters

1.	const char *pszRemoteHost	String containing the remote host address.
2.	uint16 ui16RemotePort	Remote host port.
3.	int64 i64Timeout	Timeout.

Description

Same as the connect method above with the additional capability of specifying an explicit timeout value. The timeout value must be in milliseconds.

Note:

int connect *(4)*

Return type:

• int	0	Success
	<0	Error code

Parameters

1.	const char *pszRemoteHost	String containing the remote host address.
2.	uint16 ui16RemotePort	Remote host port.
3.	bool bPreExchangeKeys	Secury key exchange flag.
4.	int64 i64Timeout = 0	Timeout.

Description

Same as the connect method above with the additional capability of specifying an explicit timeout value and to specify if security keys should be exchange at connection time. The timeout value must be in milliseconds. If security keys are exchanged reEstablishConn (supports change in the network attachment point) is supported both for client and server side while simpleSuspend only for client side. A default value can be used for the timeout parameter.

Note:

int reEstablishConn *(1)*

Return type:

• int	0	Success
	<0	Error code

Parameters

1. uint32 ui32ReEstablishTimeout =
   DEFAULT_RESUME_TIMEOUT

Description

Connect to the remote host after a change of the machine's IP address and/or port due to a change in the network attachment.
Returns 0 if successful or a negative value in case of failure.

Note:

int resumeAndRestoreState *(2)*

Return type:

- int

0	Success
<0	Error code

Parameters

1. NOMADSUtil::Reader *pr
2. uint32 ui32ReEstablishTimeout = DEFAULT_RESUME_TIMEOUT

Description

It initializes a new Mocket after a suspension, then it creates an objectDefroster to extract values from the previous node. Finally it Connects to the remote host and exchange the messages ack and resume.

Note:

int connectAsync *(2)*

Return type:

- int

0	Success
<0	Error code

Parameters

1. const char *pszRemoteHost	String containing the remote host address.
2. uint16 ui16RemotePort	Remote host port.

Description

Attempt to connect to a Mocket Server at the specified remote host on the specified remote port. The host may be a hostname that can be resolved to an IP address or an IP address in string format (e.g. "127.0.0.1"). The connection attempt is asynchronous, this call will return 0 on success and a callback will notify the application when the connection attempt succeeded or failed. Returns a negative value in case of failure.

Note:

int finishConnect *(0)*

Return type:

• int	1	Connection is established.
	0	Connection process is in progress.
	<0	Error code.

Parameters

1. void

Description

Check whether a connection has been established for this Mocket. To use with connectAsync. Returns 1 if the connection is established, 0 if the connection process is in progress, <0 (the error code returned by connect()) if the connection process failed (no connection was established).

Note:

uint32 getRemoteAddress *(0)*

Return type:

• uint32	Remote host address.

Parameters

1. void

Description

Return the remote host address to which the connection has been established.

Note:

uint16 getRemotePort *(0)*

Return type:

• uint16	Remote host port

Parameters

1. void

Description

Return the remote port to which the connection has been established.

Note:

uint32 getLocalAddress *(0)*

Return type:

- uint32 Local host connection bounded address.

Parameters

1. void

Description

Return the remote host address to which the connection has been established.

Note:

uint16 getLocalPort *(0)*

Return type:

- uint16 Local host connection bounded port.

Parameters

1. void

Description

Return the port on the local host to which this connection is bound to.

Note:

bool isConnected *(0)*

Return type:

- bool

True	Mocket is connected
False	Mocket is not connected

Parameters

1. void

Description

Returns true if the Mocket is currently connected.

Note:

int close *(0)*

Return type:

- int

0	Success
<0	Error code

Parameters

1. void

Description

Closes the current open connection to a remote endpoint. Returns 0 if the connection is being closed, <0 in case of error.

Note:

int suspend *(2)*

Return type:

1. int

0	Success
<0	Error code

Parameters

1. uint32 ui32FlushDataTimeout = DEFAULT_FLUSH_DATA_TIMEOUT   DEFAULT_FLUSH_DATA_TIMEOUT
2. uint32 ui32SuspendTimeout = DEFAULT_SUSPEND_TIMEOUT   DEFAULT_SUSPEND_TIMEOUT

Description

Invoked by the application to suspend the Mocket. Returns 0 in case of the connection being suspended, <0 in case of error.

Note:

int ***getState*** *(1)*

Return type:

- int

0	Success
<0	Error code

Parameters

1. NOMADSUtil::Writer *pw

Description

Invoked by the application if suspend ends with success. Create an ObjectFreezer that contains the state of the Mocket connection.

Note:

int enableCrossSequencing *(1)*

Return type:

•	int	0	Success
		<0	Error code

Parameters

1.	bool bEnable	True for enabling Cross Sequencing.

Description

Enables or disables Cross Sequencing across the reliable sequenced and unreliable sequenced packets.

Note:

bool isCrossSequencingEnabled *(0)*

Return type:

1.	bool	True	CrossSequencing is enabled.
		False	CrossSequencing not enabled.

Parameters

1. void

Description

Returns the current setting for cross sequencing.

Note:

MessageSender getSender *(2)*

Return type:

2.	MessageSender	Get instance of the MessageSender class

Parameters

1.	bool bReliable	True for reliable.
2.	bool bSequenced	True for sequenced.

Description

Obtains a new sender for the specified combination of reliability and sequencing parameters.

Note:

uint32 getOutgoingBufferSize *(0)*

Return type:

•	uint32	Space available in bytes

Parameters

1. void

Description

Returns the amount of space available in the outgoing (transmit) buffer, which implies that any call to send() or gsend() with a message size that is less than this value will not block.

Note:

Large messages may be fragmented, resulting in the message using up more space, therefore, do not assume this is an exact value.

int send *(8)*

Return type:

• int	0	Success
	<0	Error code

Parameters

1.	bool bReliable,	True for reliable.
2.	bool bSequenced	True for sequenced.
3.	const void *pBuf	Message buffer pointer.
4.	uint32 ui32BufSize	Buffer size.
5.	uint16 ui16Tag	Tag value.
6.	uint8 ui8Priority	Message priority.
7.	uint32 ui32EnqueueTimeout	Enqueueing timeout.
8.	uint32 ui32RetryTimeout	Retransmission timeout

Description

Enqueues the specified data for transmission using the specified reliability and sequencing requirements. The tag identifies the type of the packet and the priority indicates the priority for the packet. The enqueue timeout indicates the length of time in milliseconds for which the method will wait. If there is no room in the outgoing buffer (a zero value indicates wait forever). The retry timeout indicates the length of time for which the transmitter will retransmit the packet to ensure successful delivery (a zero value indicates retry with no time limit). Returns 0 if successful, <0 in case of error.

Note:

int gsend *(8+)*

Return type:

• int	0	Success
	<0	Error code

Parameters

1.	bool bReliable	True for reliable.
2.	bool bSequenced	True for sequenced.
3.	uint16 ui16Tag	Tag value.
4.	uint8 ui8Priority	Message priority.
5.	uint32 ui32EnqueueTimeout	Enqueueing timeout.
6.	uint32 ui32RetryTimeout	Retransmission timeout.
7.	const void *pBuf1	Message Buffer pointer.
8.	uint32 ui32BufSize1	Message Buffer size.
9.	…	

Description

Variable argument version of send (to handle a gather write). Caller can pass in any number of buffer and buffer size pairs.

Note:
The last argument, after all buffer and buffer size pairs, must be NULL.

Int gsend *(10)*

Return type:

• int	0	Success
	<0	Error code

Parameters

1.	bool bReliable	True for reliable.
2.	bool bSequenced,	True for sequenced.
3.	uint16 ui16Tag	Tag value.
4.	uint8 ui8Priority	Message priority.
5.	uint32 ui32EnqueueTimeout	Enqueueing timeout.
6.	uint32 ui32RetryTimeout	Retransmission timeout.
7.	const void *pBuf1	Message Buffer pointer.
8.	uint32 ui32BufSize1	Message Buffer size.
9.	va_list valist1	
10.	va_list valist2	

Description

Variable argument version of send (to handle a gather write).

Note:

The last argument, after all buffer and buffer size pairs, must be NULL.

int getNextMessageSize *(1)*

Return type:

• int	0	No data.
	-1	Connection being closed.
	>0	Size of next message.

Parameters

• int64 i64Timeout = 0	No data timeout

Description

If no message is available, the call will block based on the timeout parameter. Not specifying a timeout or a timeout of 0 implies that the default timeout should be used whereas a timeout of -1 implies wait indefinitely.

Note:

uint32 getCumulativeSizeOfAvailableMessages *(0)*

Return type:

• uint32	>0	Cumulative size of all messages that are ready to be delivered.
	0	No messages available

Parameters

1. void

Description

Returns the cumulative size of all messages that are ready to be delivered to the application or 0 in the case of no messages being available.

Note:

This method does not provide an indication that the connection has been closed.

int ***receive*** *(3)*

Return type:

• int	0	No data.
	-1	Connection being closed.
	>0	Number of bytes that were copied into the buffer.

Parameters

1.	void *pBuf	Message buffer.
2.	uint32 ui32BufSize	Message buffer size.
3.	int64 i64Timeout = 0	No data timeout.

Description

Retrieves the data from next message that is ready to be delivered to the application. At most ui32BufSize bytes are copied into the specified buffer. Not specifying a timeout or a timeout of 0 implies that the default timeout should be used whereas a timeout of -1 implies wait indefinitely. Returns the number of bytes that were copied into the buffer, -1 in case of the connection being closed, 0 in case no data is available within the specified timeout.

Note:

Any additional data in the packet that will not fit in the buffer is discarded.

int receive *(2)*

Return type:

• int	0	No data.
	-1	Connection being closed.
	>0	Number of bytes that were copied into the buffer.

Parameters

1. void **ppBuf,	Pointer to ppbuf.
2. int64 i64Timeout = 0	No data timeout.

Description

Retrieves the data from next message that is ready to be delivered to the application. A new buffer of the size necessary for the message is allocated and the pointer to the buffer is copied into ppBuf. Not specifying a timeout or a timeout of 0 implies that the default timeout should be used whereas a timeout of -1 implies wait indefinitely. Returns the number of bytes that were copied into the buffer, -1 in case of the connection being closed, 0 in case no data is available within the specified timeout.

Note:

- This method is inefficient because it results in a new memory allocation for every receive. Consider using getNextMessageSize and maintaining a single buffer in the application.
- The application is responsible for deallocating the memory by calling free().

int sreceive *(3+)*

Return type:

• int	0	No data.
	-1	Connection being closed.
	>0	Number of bytes that were copied into the buffer.

Parameters

1. int64 i64Timeout	No data dimeout.
2. void *pBuf1,	Message buffer.
3. uint32 ui32BufSize1	Message buffer size.
4. …	

Description

Retrieves the data from the next message that is ready to be delivered to the application. Not specifying a timeout or a timeout of 0 implies that the default timeout should be used whereas a timeout of -1 implies wait indefinitely. The data is scattered into the buffers that are passed into the method. The pointer to the buffer and the buffer size arguments must be passed in pairs. The last argument should be NULL. Returns the number of bytes that were copied into the buffers, -1 in case of the connection being closed, 0 in case no data is available within the specified timeout.

Note:

- If the caller passes in three buffers, (e.g., sreceive (-1, pBufOne, 8, pBufTwo, 1024, pBufThree, 4096)), and the method returns 4000, the implication is that 8 bytes were read into pBufOne, 1024 bytes into pBufTwo, and the remaining 2968 bytes into pBufThree.
- Any additional data in the packet that will not fit in the buffers is discarded.

int replace *(9)*

Return type:

• int	0	Success
	<0	Error code

Parameters

1.	bool bReliable,	True for reliable.
2.	bool bSequenced	True for sequenced.
3.	const void *pBuf	Pointer to data buffer.
4.	uint32 ui32BufSize	Data buffer size.
5.	uint16 ui16OldTag	Tag to look for.
6.	uint16 ui16NewTag	New tag.
7.	uint8 ui8Priority	New priority.
8.	uint32 ui32EnqueueTimeout	New enqueuer timeout.
9.	uint32 ui32RetryTimeout	New retry timeout.

Description

First cancels any previously enqueued messages that have been tagged with the specified OldTag value and then transmits the new message using the specified parameters. See documentation for cancel() and send() for more details.

Note:

- There may be no old messages to cancel - in which case this call behaves just like a send().

int cancel *(3)*

Return type:

• int	0	Success
	<0	Error code

Parameters

1.	bool bReliable	True for reliable.
2.	bool bSequenced	True for sequenced.
3.	uint16 ui16TagId	Tag to look for.

Description

Cancels (deletes) previously enqueued messages that have been tagged with the specified tag

Note:

- Note that the messages may be pending transmission (which applies to all flows) or may have already been transmitted but not yet acknowledged (which only applies to reliable flows).

int setConnectionLingerTime *(1)*

Return type:

• int	0	Success
	<0	Error code

Parameters

1. uint32 ui32LingerTime	Unsent data timeout.

Description

Sets the length of time (in milliseconds) for which a connection should linger before closing in case there is unsent data. A timeout value of 0 implies that the connection should wait indefinitely until all data has been sent.

Note:

uint32 getConnectionLingerTime *(0)*

Return type:

• uint32	Linger time.

Parameters

1. void

Description

Returns the current setting for the connection linger time.

Note:

uint16 getMTU *(0)*

Return type:

• uint16	MTU.

Parameters

1. void

Description

Returns the current MTU that is in effect

Note:

static uint16 getMaximumMTU *(0)*

Return type:

• uint16	Maximum MTU.

Parameters

1. void

Description

Returns the maximum MTU that may be used

Note:

int activateBandwidthEstimation *(1)*

Return type:

• int	0	Success
	<0	Error code

Parameters

1. uint16 ui16InitialAssumedBandwidth = DEFAULT_INITIAL_ASSUMED_BANDWIDTH	Assumed Bandwidth.

Description

Activates bandwidth estimation.

Note:

- Must be called after connect().

int activateCongestionControl *(0)*

Return type:

• int	0	Success
	<0	Error code

Parameters

1. void

Description

Activates congestion control

Note:

void debugStateCapture *(0)*

Return type:

- void

Parameters

1. void

Description

Used to activate debugging of the state capture during Mockets migration. This will disable sending of messages with odd sequence number in order to perform a migration with messages in the queues.

Note:

void useTwoWayHandshake *(0)*

Return type:

- void

Parameters

1. void

Description

Activates two-way handshake instead of default four-way handshake.

Note:

int setTransmitRateLimit *(1)*

Return type:

- int

int	0	Success
	<0	Error code

Parameters

1. uint32 ui32TransmitRateLimit — Outgoing Bandwidth limit.

Description

Set a bandwidth limit on the outgoing flow of data. ui32TransmitRateLimit is specified in bytes per second. A value of 0 indicates no limit.

Note:

void setKeepAliveTimeout *(1)*

Return type:

- void

Parameters

1.	uint16 ui16Timeout	Keep alive timeout.

Description

Set the keep Alive Timeout.

Note:

- Even when keepAlive are disabled this timeout is used to trigger peerUnreachable callbacks.

void disableKeepAlive *(0)*

Return type:

- void

Parameters

1. void

Description

Disable the keep alive function.

Note:

void setInitialAssumedRTT *(1)*

Return type:

- void

Parameters

1.	uint32 ui32RTT	RTT value.

Description

Set the initial assumed RTT.

Note:

- InitialAssumedRTT, minimumRTT and maximumRTT are used together to calculate the retransmission timeout (RTO) of packets.

void setMaximumRTO *(1)*

Return type:

- void

Parameters

1. uint32 `ui32RTO` RTO value.

Description

Set the maximum RTO.

Note:

- The value of the maximum RTO is zero by default. A value of zero means no maximum RTO

void setMinimumRTO *(1)*

Return type:

- void

Parameters

1. uint32 `ui32RTO` RTO value.

Description

Set minimum RTO.

Note:

void setRTOFactor *(1)*

Return type:

- void

Parameters

1. float fRTOFactor RTO factor value.

Description

Set RTO factor.

Note:

- RTO factor and RTO constant are used in the calculation of the RTO for the packet according to this formula:
  _fSRTT + RTOConstant) * RTOFactor * (1 + pWrapper->getRetransmitCount()

void setRTOConstant *(1)*

Return type:

- void

Parameters

1. uint16 ui16RTOConstant RTO constant value.

Description

Set RTO constant.

Note:

- RTO factor and RTO constant are used in the calculation of the RTO for the packet according to this formula:
  _fSRTT + RTOConstant) * RTOFactor * (1 + pWrapper->getRetransmitCount()

void disableRetransmitCountFactorInRTO *(0)*

Return type:

- void

Parameters

1. void

Description

Disables the the factor (1 + pWrapper->getRetransmitCount()) from RTO calculation.

Note:

void setMaximumWindowSize *(1)*

Return type:

- void

Parameters

1. uint32 ui32WindowSize Window size value.

Description

Set maximum window Size.

Note:

void setSAckTransmitTimeout *(1)*

Return type:

- void

Parameters

1. uint16 ui16SAckTransTO SACK transmit timeout value.

Description

Set SACK transmit timeout.

Note:

void setConnectTimeout *(1)*

Return type:

- void

Parameters

1. uint32 ui32ConnectTO Connection timeout value.

Description

Set connection timeout.

Note:

void setUDPReceiveConnectionTimeout *(1)*

Return type:

- void

Parameters

1. uint16 ui16UDPRecConTO Low level socket connection timeout value.

Description

Low level socket timeout at connection time.

Note:

At connection time.

void setUDPReceiveTimeout *(1)*

Return type:

- void

Parameters

1. uint16 ui16UDPRecTO	Low level socket connection timeout value.

Description

Low level socket timeout after connection is open**.**

Note:

After connection is open.

API METHODS FOR GENERAL UTILITIES

int readConfigFile *(1)*

Return type:

- int

0	Success.
<0	Error code.

Parameters

1. const char *pszConfigFile — Path to the configuration file.

Description

Different configuration files are defined for different type of networks. The application can call readConfigFile() and pass in the path to the configuration file that should be loaded.

Note:

void resetTransmissionCounters *(0)*

Return type:

- void

Parameters

1. void

Description

When the behavior of a node is to be connected for some time, then disconnected for some time and then connected again and so on, the application may wish to reset the transmission counters upon reconnection so the communication won't suffer from the period of unreachability.

The values reset with this function are estimated RTT and the transmit count and transmit timeout of the packets waiting for transmission in the unacknowledged packet queue.

Note:

void enableTransmitLogging *(1)*

Return type:

- void

Parameters

1. bool bEnableXMitLogging — True for packet transmit logging.

Description

Enable or disable the packet transmit log.

Note:

2.2. MessageSender.h

MessageSender (1)

Return type:

- void

Parameters

1. const MessageSender &src

Description

The class is used to send messages. Obtained by calling getSender() on a Mocket.

Note:

Message may also be sent using the Mocket class directly. This class makes it a little more convenient by having default or configurable values for some of the parameters to a socket - used by a client to establish a connection to a server and then communicate with the server.

*int **send** (2)*

Return type:

• int	0	Success.
	<0	Error code.

Parameters

1. const void *pBuf,	Pointer to message buffer.
2. uint32 ui32BufSize	Message buffer size.

Description

Send (enqueue for transmission) data to the remote endpoint. The pBuf must point to the data to be sent and ui32BufSize must specify the number of bytes to send. A tag value of 0 and a priority of 0 are used for the data, along with the default values for the enqueue and retry timeouts. Returns 0 if successful or a negative value in case of error.

Note:

*int **gsend** (2+)*

Return type:

• int	0	Success.
	<0	Error code.

Parameters

1. const void *pBuf1	Pointer to message buffer.
2. uint32 ui32BufSize1	Message buffer size.
3. ...	

Description

Gather write version of send. Caller can pass in any number of buffer and buffer size pairs.

Note:

The last argument, after all buffer and buffer size pairs, must be NULL.

int ***send*** *(4)*

Return type:

• int	0	Success
	<0	Error code

Parameters

1. const void *pBuf	Pointer to message buffer.
2. uint32 ui32BufSize	Message buffer size.
3. uint16 ui16Tag	Message tag.
4. uint8 ui8Priority	Message priority.

Description

Send (enqueue for transmission) data to the remote endpoint. The data is tagged with the specified tag value and sent using the specified priority,

pBuf must point to the data to be sent and ui32BufSize must specify the number of bytes to send. The default values for the enqueue and retry timeouts are used. Returns 0 if successful or a negative value in case of error.

Note:

int ***send*** *(3)*

Return type:

• int	0	Success.
	<0	Error code.

Parameters

1. const void *pBuf	Pointer to message buffer.
2. uint32 ui32BufSize	Message buffer size.
3. Params *pParams	Pointer to parameters.

Description

Send (enqueue for transmission) data to the remote endpoint. The tag, priority, enqueue timeout, and retry timeout values are specified via the params object. The pBuf must point to the data to be sent and ui32BufSize must specify the number of bytes to send. Returns 0 if successful or a negative value in case of error.

Note:

int send *(6)*

Return type:

• int	0	Success
	<0	Error code

Parameters

1. const void *pBuf	Pointer to message buffer.
2. uint32 ui32BufSize	Message buffer size.
3. uint16 ui16Tag	Message Tag.
4. uint8 ui8Priority	Message Priority.
5. uint32 ui32EnqueueTimeout	Message enqueue timeout.
6. uint32 ui32RetryTimeout	Message retry timeout.

Description

Send (enqueue for transmission) data to the remote endpoint. The data is tagged with the specified tag value and sent using the specified priority, using the specified enqueue and retry timeout values. The pBuf must point to the data to be sent and ui32BufSize must specifiy the number of bytes to send. Returns 0 if successful or a negative value in case of error.

Note:

*int **replace** (4*

Return type:

• int	0	Success
	<0	Error code

Parameters

1. const void *pBuf	Pointer to message buffer.
2. uint32 ui32BufSize	Message buffer size.
3. uint16 ui16OldTag	Old Tag to look for.
4. uint16 ui16NewTag	New Tag.

Description

First cancels any previously enqueued messages that have been tagged with the specified OldTag value and then transmits the new message using the specified parameters. Returns 0 if successful or a negative value in case of error.

Note:

- There may be no old messages to cancel - in which case this call behaves just like a send(). See documentation for cancel() and send() for more details.

int ***replace*** *(4)*

Return type:

• int	0	Success.
	<0	Error code.

Parameters

1.	const void *pBuf	Pointer to message buffer.
2.	uint32 ui32BufSize	Message buffer size.
3.	uint16 ui16OldTag	Old Tag to look for.
4.	Params *pParams	New parameters object.

Description

First cancels any previously enqueued messages that have been tagged with the specified OldTag value and then transmits the new message using the specified parameters. Returns 0 if successful or a negative value in case of error

Note:

There may be no old messages to cancel - in which case this call behaves just like a send(). See documentation for cancel() and send() for more details.

int ***replace*** *(7)*

Return type:

• int	0	Success.
	<0	Error code.

Parameters

1.	const void *pBuf	Pointer to message buffer.
2.	uint32 ui32BufSize	Message buffer size.
3.	uint16 ui16OldTag	Old Tag to look for.
4.	uint16 ui16NewTag	New Tag.
5.	uint8 ui8Priority	New priority.
6.	uint32 ui32EnqueueTimeout	New enqueuer timeout.
7.	uint32 ui32RetryTimeout	New retry timeout.

Description

First cancels any previously enqueued messages that have been tagged with the specified OldTag value and then transmits the new message using the specified parameters. Returns 0 if successful or a negative value in case of error.

Note:

There may be no old messages to cancel - in which case this call behaves just like a send(). See documentation for cancel() and send() for more details.

int ***cancel*** *(1)*

Return type:

- int

0	Success
<0	Error code

Parameters

1. uint16 ui16TagId

Description

Cancels (deletes) previously enqueued messages that have been tagged with the specified tag.

Note:

The messages may be pending transmission (which applies to all flows) or may have already been transmitted but not yet acknowledged (which only applies to reliable flows).

void ***setDefaultEnqueueTimeout*** *(1)*

Return type:

- void

Parameters

1. uint32 ui32EnqueueTimeout	New default enqueuer timeout.

Description

Set the default timeout for enqueuing data into the outgoing buffer If the outgoing buffer is full, a call to send will block until the timeout expires.

Note:

void ***setDefaultRetryTimeout*** *(1)*

Return type:

- void

Parameters

1. uint32 ui32RetryTimeout	New default retry timeout.

Description

Set the default timeout for retransmission of reliable packets that have not been acknowledged. If a timeout value of 0 is specified the timeout is disabled.

Note:

Setting a timeout would imply that packets may not be reliable!

2.3. ServerMocket.h

ServerMocket (3)

Return type:

1. void

Parameters

1. const char *pszConfigFile = NULL	Path to config file.
2. CommInterface *pCI = NULL	
3. bool bDeleteCIWhenDone = false	

Description

The main class for a server application to use the Mockets communication library. Similar in functionality to a server socket - used by a server to accept connections from client applications.

Note:

int ***listen*** *(1)*

Return type:

• int	>0	Port number.
	<0	Error code.

Parameters

1. uint16 ui16Port	Listen port.

Description

Initialize the server Mocket to accept incoming connections. Specifying a 0 for the port causes a random port to be allocated. Returns the port number that was assigned, or a negative value in case of error.

Note:

int ***listen*** *(2)*

Return type:

• int	>0	Port number.
	<0	Error code.

Parameters

1. uint16 ui16Port	Port number.
2. const char *pszListenAddr	Address to listen.

Description

Initialize the server Mocket to accept incoming connections. Specifying a 0 for the port causes a random port to be allocated. Returns the port number that was assigned, or a negative value in case of error.

Note:

*Mocket * **accept** (1)*

Return type:

- Mocket * Pointer to the Mocket instance.

Parameters

1. uint16 ui16PortForNewConnection = 0 New port for the connection.

Description

Accept an incoming connection request, if ui16PortForNewConnection is zero randomly chose a port.

Note:

*int **close** (0)*

Return type:

- int

0	Success
<0	Error code

Parameters

1. void

Description

Close the Mocket conection.

Note:

*void **setIdentifier** (1)*

Return type:

- void

Parameters

1. const char *pszIdentifier

Description

Note:

See Mocket.h.

*const char * **getIdentifier** (0)*

Return type:

- const char *

Parameters

1. void

Description

Note:

See Mocket.h.

3. MESSAGES DESCRIPTION

Considering Mocket's messages, each one consists of two basic section:

- A common header
- A data chunks which form the remaining portion of the packet.

The chunks are divided in three classes:

- Metadata.
- Statechange.
- Data.

3.1. Common message packet header

For each packet we need:

- **Protocol version**.
- **Packet type** (which can be one of: stream/message, reliable/unreliable, sequenced/unsequenced).
- **Window size**.
- **Validation tag** (which is a connection ID and not the Mocket ID).
- **Sequence number** (which must be interpreted as transmission sequence number for reliable sequenced, unreliable sequenced or control packet flows or flow ID number for reliable unsequenced flow according to message type).

Information about packet length, source/destination ports and checksum don't need to be included since it is already present in the IP/UDP headers.

00	**31**	**32**	**63**
Version/flags	Window size	Validation tag	
Sequence number			

3.2. Version / flags

The most significant 4 bits of the version/flags field contain the protocol version number.
The defined flags are:

- **HEADER_FLAG_RELIABLE** (0x001)
  Discriminates between reliable (flag set) and unreliable (flag unset) packets.

- **HEADER_FLAG_SEQUENCED** (0x002)
  Discriminates between sequenced (flag set) and unsequenced (flag unset) packets.

- **HEADER_FLAG_MSGPKT** (0x004)
  Discriminates between message (flag set) and stream (flag unset) packets.

- **HEADER_FLAG_CONTROL** (0x008)
  Set if this is a control packet. Conflicts with HEADER_FLAG_SEQUENCED and HEADER_FLAG_MSGPKT flags. If HEADER_FLAG_CONTROL is set the two flags must be unset.

- **HEADER_FLAG_DELIVERY_DEPS** (0x010)
  Set if this packet has a delivery prerequisites option header.

- **HEADER_FLAG_FIRST_FRAGMENT** (0x020)
  Set if this packet is the first fragment of a message.

- **HEADER_FLAG_INTERMEDIATE_FRAGMENT** (0x040)
  Set if this packet is an intermediate fragment of a message.

- **HEADER_FLAG_LAST_FRAGMENT** (0x080)
  Set if this packet is the last fragment of a message.

- **HEADER_FLAG_RETRANSMITTED** (0x100)
  Set if this packet fragment is retransmitted.

3.3. Delivery prerequisites option header

<table>
<tr><th>0
0</th><th>3
1</th><th>3
2</th><th>6
3</th></tr>
<tr><td colspan="2">Sequence number 1</td><td colspan="2">Sequence number 2</td></tr>
</table>

- For **Control** packet:

  Sequence number 1:= Reliable sequenced sequence number.

  Sequence number 2:= Unreliable sequenced sequence number.

- For **Reliable Sequenced** packet:

  Sequence number 1:= Control sequence number.

  Sequence number 2:= Unreliable sequenced sequence number.

- For **Unreliable Sequenced** packets it is:

  Sequence number 1:= Control sequence number.

  Sequence number 2:= Reliable sequenced sequence number.

Because of their unreliability, the receiver treats unreliable sequenced sequence number in a special way. To prevent from a deadlock in case of packet loss, if the destination endpoint hasn't received the unreliable sequenced packet specified by the delivery prerequisites option header (or an unreliable sequenced packet with a greater TSN) after a predefined time interval, then it will consider that packet lost and ignore the delivery prerequisite condition related to that packet.

3.4. Chunk format:

<table>
<tr><th>0
0</th><th>3
1</th><th>3
2</th><th>6
3</th></tr>
<tr><td>Chunk type</td><td>Chunk length</td><td>Tag value</td><td></td></tr>
<tr><td colspan="4">Chunk data</td></tr>
</table>

The chunk identifier is a 16 bit ID divided in 2 fields:

- A 4 bit field that specifies **which class the chunk belongs** to.
- A 12 bit field with the **actual ID** which must be unique in the chunk class.

We define the chunk classes:

CHUNK_CLASS_METADATA	= 0x1000.
CHUNK_CLASS_DATA	= 0x2000.
CHUNK_CLASS_STATECHANGE	= 0x4000.

The defined chunk types in the **CHUNK_CLASS_METADATA** class are:

CT_SAck	=	CHUNK_CLASS_METADATA	| 0x0001
CT_Heartbeat	=	CHUNK_CLASS_METADATA	| 0x0002
CT_Cancelled	=	CHUNK_CLASS_METADATA	| 0x0003
CT_Timestamp	=	CHUNK_CLASS_METADATA	| 0x0004
CT_TimestampAck	=	CHUNK_CLASS_METADATA	| 0x0005
CT_SAckRecBandEst	=	CHUNK_CLASS_METADATA	| 0x0006

The defined chunk types in the **CHUNK_TYPE_CLASS_DATA** class are:

CT_Data	=	CHUNK_CLASS_DATA	| 0x0001

The defined chunk types in the **CHUNK_CLASS_STATECHANGE** class are:

Ct_Init	=	CHUNK_CLASS_STATECHANGE	| 0x0001

- **CT_InitAck** = CHUNK_CLASS_STATECHANGE | 0x0002
- **CT_CookieEcho** = CHUNK_CLASS_STATECHANGE | 0x0003
- **CT_CookieAck** = CHUNK_CLASS_STATECHANGE | 0x0004
- **CT_Shutdown** = CHUNK_CLASS_STATECHANGE | 0x0005
- **CT_ShutdownAck** = CHUNK_CLASS_STATECHANGE | 0x0006
- **CT_ShutdownComplete** = CHUNK_CLASS_STATECHANGE | 0x0007
- **CT_Abort** = CHUNK_CLASS_STATECHANGE | 0x0008
- **CT_Suspend** = CHUNK_CLASS_STATECHANGE | 0x0009
- **CT_SuspendAck** = CHUNK_CLASS_STATECHANGE | 0x000A
- **CT_Resume** = CHUNK_CLASS_STATECHANGE | 0x000B
- **CT_ResumeAck** = CHUNK_CLASS_STATECHANGE | 0x000C
- **CT_ReEstablish** = CHUNK_CLASS_STATECHANGE | 0x000D
- **CT_ReEstablishAck** = CHUNK_CLASS_STATECHANGE | 0x000E
- **CT_SimpleSuspend** = CHUNK_CLASS_STATECHANGE | 0x000F
- **CT_SimpleSuspendAck** = CHUNK_CLASS_STATECHANGE | 0x0010
- **CT_SimpleConnect** = CHUNK_CLASS_STATECHANGE | 0x0011
- **CT_SimpleConnectAck** = CHUNK_CLASS_STATECHANGE | 0x0012

Note that chunk length is, just like for SCTP, the length (in octets) of the whole chunk, including the “chunk type” and “chunk length” fields. See the Packet class for more details.

3.5. Metadata Chunks description

CT_SAck

Each SACK chunk includes one (and only one) Acknowledgement Information. Notice that we acknowledge packets, not data.

0 0	3 1	3 2	6 3
Control cumulative TSN ACK		Rel. seq. cumulative TSN ACK	
Rel. unseq. cumulative TSN ACK			
Acknowledgement information			

Each SACK is composed by:

- **One cumulative acknowledgement** TSN for each of the Control Packet Reliable Sequenced, and Reliable Unsequenced flows.
- An acknowledge information.

Cumulative TSN (Transmission Sequence Number) acknowledgement is the highest consecutive packet TSN that the sender of the SACK chunk has seen.

Notice that since for Reliable Unsequenced packets we use sequential packet ID numbers, cumulative acknowledgement works for Reliable Unsequenced packets as well.

Acknowledge information

This is the Acknowledgement Information included in **SACK** and **Cancelled Packets** chunks. Each Acknowledge Information is composed by a sequence of acknowledgement information blocks.

0 0	3 1	3 2	6 3
1st acknowledgement info block			
………………………………………………………			
Nst acknowledgement info block			

Each Acknowledgement Information block has the following structure:

0 0		3 1	3 2	6 3
Flags	Length			
Data				

The **flags field** discriminates between several types of packet (or flows) and acknowledgement information blocks.
The defined flags are:

SACK_CHUNK_BLOCK_FOR_CONTROL_FLOW	=	0x01
SACK_CHUNK_BLOCK_FOR_RELIABLE_SEQUENCED_FLOW	=	0x02
SACK_CHUNK_BLOCK_FOR_RELIABLE_UNSEQUENCED_FLOW	=	0x04
SACK_CHUNK_BLOCK_TYPE_RANGE	=	0x10
SACK_CHUNK_BLOCK_TYPE_SINGLE	=	0x20

The **length field** contains the length (in octets) of the whole acknowledgement information block (including type and length fields). The size of the data portion of the acknowledgement information block is length-3. At the moment the only defined acknowledgement information blocks are:

- Acknowledgement Information Block Type 0: **RANGE OF SEQUENCE NUMBERS**

<table>
<tr><th>0</th><th>3</th><th>3</th><th>6</th><th></th></tr>
<tr><th>0</th><th>1</th><th>2</th><th>3</th><th></th></tr>
<tr><td>Flags</td><td>Length</td><td>Begin of 1st sequence number range</td><td>End of</td></tr>
<tr><td colspan="2">1st sequence number range</td><td>Begin of 2nd sequence number range</td><td></td></tr>
<tr><td colspan="4">....</td></tr>
<tr><td colspan="3"></td><td>End of</td></tr>
<tr><td colspan="2">Nth sequence number range</td><td colspan="2"></td></tr>
</table>

- Acknowledgement Information Block Type 1: **SINGLE SEQUENCE NUMBERS**

<table>
<tr><th>0</th><th>3</th><th>3</th><th>6</th></tr>
<tr><th>0</th><th>1</th><th>2</th><th>3</th></tr>
<tr><td>Flags</td><td>Length</td><td>1st sequence number</td><td></td></tr>
<tr><td colspan="4">....</td></tr>
<tr><td colspan="4"></td></tr>
<tr><td colspan="2"></td><td>Nth sequence number range</td><td></td></tr>
</table>

CT_Heartbeat

For each Heartbeat chunk we need:

- Heartbeat sender-specific information (generally includes system time)

0	3	3	6
0	1	2	3
Heartbeat sender-specific information			

For each Heartbeat Sender Specific Information we need:

- System time

0	3	3	6
0	1	2	3
System time			

CT_Cancelled

Each Cancelled Packets chunk includes one (and only one) acknowledgement information. Notice that we acknowledge packets, not data.

CT_Timestamp

0	3	3	6
0	1	2	3
Timestamp			

CT_TimestampAck

<table>
<tr><th>0</th><th>3</th><th>3</th><th>6</th></tr>
<tr><th>0</th><th>1</th><th>2</th><th>3</th></tr>
<tr><td colspan="4">Timestamp</td></tr>
</table>

CT_SAckRecBandEst

<table>
<tr><th>0</th><th>3</th><th>3</th><th>6</th></tr>
<tr><th>0</th><th>1</th><th>2</th><th>3</th></tr>
<tr><td colspan="2">Control Cumulative Ack</td><td colspan="2">Reliable Sequenced Cumulative Ack</td></tr>
<tr><td colspan="2">Reliable Unsequenced Cumulative Ack</td><td colspan="2">Bytes Received</td></tr>
<tr><td colspan="4">Timestamp</td></tr>
</table>

3.6. StateChange Chunks description

Ct_Init

Chunk to initialize the Mocket link. For each Init chunk we need:

- An **initiation tag** value Validation (!=0) which will have to be the same for the reply.
- **Initial Transmission Sequence Numbers** (TSNs) for Reliable Sequenced, Unreliable Sequenced and Control Packet flows and initial Reliable.
- Unsequenced **flow ID Number**.

<table>
<tr><th>0</th><th>3</th><th>3</th><th>6</th></tr>
<tr><th>0</th><th>1</th><th>2</th><th>3</th></tr>
<tr><td colspan="2">Validation</td><td colspan="2">Control Packet Flow TSN</td></tr>
<tr><td colspan="2">Reliable Sequenced Flow TSN</td><td colspan="2">Unreliable Sequenced Flow TSN</td></tr>
<tr><td colspan="2">Reliable Unsequenced Fl. ID Num</td><td colspan="2">Unreliable Unsequenced Fl. ID Num</td></tr>
</table>

CT_InitAck

Must be the first chunk of the reply. For each Init-Ack chunk we need:

- An **Initiation tag value** (!=0) must be the same of the Init request.
- **Initial Transmission Sequence Numbers** (TSNs) for Reliable Sequenced, Unreliable Sequenced, Control Packet flows.
- Initial Reliable Unsequenced **flow ID Number**.
- **A state cookie.**

<table>
<tr><th>0</th><th>3</th><th>3</th><th>6</th></tr>
<tr><th>0</th><th>1</th><th>2</th><th>3</th></tr>
<tr><td colspan="2">Validation</td><td colspan="2">Control Packet Flow TSN</td></tr>
<tr><td colspan="2">Reliable Sequenced Flow TSN</td><td colspan="2">Unreliable Sequenced Flow TSN</td></tr>
<tr><td colspan="2">Reliable Unsequenced Fl. ID Num</td><td colspan="2">Unreliable Unsequenced Fl. ID Num</td></tr>
<tr><td colspan="4">State Cookie</td></tr>
</table>

CT_CookieEcho

For each Cookie Echo chunk we need:

- **A state cookie**

0	**3**	**3**	**6**
0	**1**	**2**	**3**
State Cookie			

CT_CookieAck

For each Cookie Ack chunk we need:

- The **remote port** to which the local host has to connect.

0	**1**
0	**5**
Port	

State Cookie

For each State Cookie we need:

- Cookie generation **timestamp**.
- **Maximum cookie lifetime**.
- Local & remote **initial Transmission Sequence Numbers** (TSNs) for Reliable Sequenced, Unreliable Sequenced.
- Control Packet flows and initial Reliable Unsequenced **flow ID Number**.
- Local and remote port.
- HMAC-SHA1 checksum.

0	3	3	6
0	1	2	3
Cookie generation timestamp			
Cookie lifespan			
A validation tag		Z validation tag	
A Control Packet Flow TSN		Z Control Packet Flow TSN	
A Reliable Sequenced Flow TSN		Z Reliable Sequenced Flow TSN	
A Reliable UnSequenced Flow TSN		Z Reliable UnSequenced Flow TSN	
A Unrel. Unsequenced Fl. ID Num		Z Unrel. Unsequenced Fl. ID Num	
A port	Z port		
HMAC-SHA1 checksum			
Tie tags			

A is the endpoint that initiates the connection (by sending the Init message). Z is the passive endpoint of the connection (that waits for the Init Ack message). We probably should encrypt the cookie someway. In the future we will probably add tie-tags (see RFC2960).

CT_Shutdown

The Shutdown chunk carries no information - it is an empty chunk. The sender **MUST** bundle a SACK chunk in every message containing a Shutdown chunk to more completely describe the current view of what has been received. The sender expect a Shutdown ack chunk.

CT_ShutdownAck

The Shutdown ack chunk carries no information - it is an empty chunk.

CT_ShutdownComplete

The Shutdown complete chunk carries no information - it is an empty chunk.

CT_Abort

The Abort chunk carries no information - it is an empty chunk. In future we may decide to add some information related to the error cause.

CT_Suspend

The Suspend chunk carries no information - it is an empty chunk.

CT_SuspendAck

The SuspendAck chunk carries no information - it is an empty chunk.

CT_Resume

The Resume chunk carries no information - it is an empty chunk.

CT_ResumeAck

The ResumeAck chunk carries no information - it is an empty chunk.

CT_ReEstablish

The ReEstabilish chunk carries no information - it is an empty chunk.

CT_ReEstablishAck

The ReEstabilishAck chunk carries no information - it is an empty chunk.

CT_SimpleSuspend

The SimpleSuspend chunk carries no information - it is an empty chunk.

CT_SimpleSuspendAck

The SimpleSuspendAck chunk carries no information - it is an empty chunk.

CT_SimpleConnect

For each SimpleConnect chunk we need:

- An **Initiation tag value** (!=0) must be the same of the Init request.
- **Initial Transmission Sequence Numbers** (TSNs) for Reliable Sequenced, Unreliable Sequenced
- Initial Reliable/Unreliable Unsequenced **flow ID Number**.

0	**3**	**3**	**6**
0	**1**	**2**	**3**
Validation		ControlTSN	
ReliableSequencedTSN		UnReliableSequencedTSN	
ReliableUnsequencedID		UnreliableUnsequencedID	

CT_SimpleConnectAck

For each SimpleConnectAck chunk we need:

- An **Initiation tag value** (!=0) must be the same of the SimpleConnect request.
- **Initial Transmission Sequence Numbers** (TSNs) for Reliable Sequenced, Unreliable Sequenced.
- Initial Reliable/Unreliable Unsequenced **flow ID Number**.
- **Local port.**

0	**3**	**3**	**6**
0	**1**	**2**	**3**
Validation		ControlTSN	
ReliableSequencedTSN		UnReliableSequencedTSN	
ReliableUnsequencedID		UnreliableUnsequencedID	
Local Port			

3.7. Data Chunks description

CT_Data

For each Data chunk we need:

- **Tag ID** which can be one from:
    - ui32SentReliableSequencedMsgs.
    - ui32SentReliableUnsequencedMsgs.
    - ui32SentUnreliableSequencedMsgs.
    - ui32SentUnreliableUnsequencedMsgs.
    - ui32ReceivedReliableSequencedMsgs.
    - ui32ReceivedReliableUnsequencedMsgs.
    - ui32ReceivedUnreliableSequencedMsgs.
    - ui32ReceivedUnreliableUnsequencedMsgs.
    - ui32CancelledPackets.
- **Fragment ID** (zero for first fragment).
- **Data**.

0		**3**	**3**	**6**
0		**1**	**2**	**3**
Tag ID	Data			