CHAPTER 3 APPLICATION PROGRAMMING INTERFACE – ADVANCED TOPICS 1

3.1. The Enhanced Services API 2
Services with acknowledgements 5
Synchronization 8
Incast 9
Duplicate Elimination 9
Best-effort Ordering 9

3.2. Stream API 10
API of the Stream Manager 13
API of HCastInputStream and HCastOutputStream 14

3.3. Event Notifications 15

3.4. Message Interception API 19

3.5. Configuring Overlay Sockets – Advanced topics 23
Configuration Attributes 23
Overlay servers 25
Creating configurations with *createOLConfig* 25
Programming with configuration objects 27

3.6. Secure Overlay Sockets 31
Authentication and Certificates 32
Neighborhood Keys 33
Shared Group Keys 36
SSL Security 37

3.7. References 38

This document is a draft. If you have comments or corrections, please mark this document up and send it to jorg@cs.virginia.edu. If you send your comments in plain text, please include the date of this draft (see upper left corner), the page number and the paragraph number. If you find discrepancies between this document and the most recent version of the HyperCast software, please give a detailed description of the problem.

Thank you,

Jörg Liebeherr

CHAPTER 3

Application Programming Interface – Advanced Topics

In this chapter we continue the discussion of the application programming interface of the overlay socket. We present several APIs that extend the basic best-effort delivery service for overlay messages. The services discussed in this chapter are:

- **Enhanced Services API:** The overlay sockets supports several services that enhance the basic delivery service for overlay messages, by adding services for in-sequence delivery, synchronization, and improved delivery assurance.
- **Stream API:** The Stream API offers a byte-stream programming interface. This API is similar to stream sockets in the Berkeley socket environment.
- **Secure overlay sockets:** Overlay sockets can be configured so that they ensure integrity or confidentiality for overlay messages, and integrity for messages of the overlay protocol.

The interactions between application programs and overlay sockets can be enriched by taking advantage of a message notification system and by intercepting messages:

- **Notifications:** The event notification system offers mechanisms by which an overlay socket can report the occurrence of preprogrammed events to an application program.
- **Interception** The interception API permits applications that forward a message but are neither the sender nor the intended receiver of the message to read and manipulate the message.

This chapter also includes a section about configuring an overlay socket from a server.

3.1. THE ENHANCED SERVICES API

The default delivery service of overlay sockets is a connectionless best-effort delivery of overlay messages. Messages may be delivered in a different sequence than they were transmitted, and messages may even get duplicated. This type of service is similar to the service provided by the IP protocol for the transmission of IP packets. In HyperCast, the overlay socket implements services that improve the default delivery service for overlay messages. This is different in IP networks, where improvements to the delivery service of IP are deferred to higher layer protocols, e.g., the transport-layer protocol TCP provides a reliable service, and the application-layer protocol RTP supplies information that can be used to achieve an in-sequence delivery of application data.

HyperCast overlay sockets support a variety of services that have a stronger semantics than the default best-effort delivery service. These services are implemented in the overlay socket and are made available to applications through the *enhanced services API*. This section presents the available services and discusses how application programmers use enhanced services.

Although the choice of protocols in the underlay network can noticeably improve the quality of the delivery service of overlay messages, it does not principally change the semantics of the best-effort service. For example, when TCP is used in an Internet underlay network, overlay messages will be delivered reliably between neighbors in the overlay network. However, relying on TCP in the underlay network does not result in a reliable delivery service for overlay messages. Even if the transmission between neighbors in the overlay network is reliable, numerous scenarios can result in a message loss. For example, an overlay socket could drop an overlay message after it has been, but before it could be forwarded to its neighbors. A likely cause for a message loss is a change of the overlay network topology while a message is on its route from the sender to its destination(s).

Overlay sockets distinguish between enhanced services that are message-oriented and that are stream-oriented. A message-oriented service is provided separately for each overlay message. A stream-oriented service is provided for a sequence of messages. Overlay messages that are delivered with an enhanced service, henceforth called *enhanced service message*, have additional fields in the message header. An application programmer should be aware of the following fields: the service identifier, the message identifier, the stream identifier, and the sequence number. All enhanced service messages have service identifier that specifies the service desired for this message.

For message-oriented services, a message identifier is used to distinguish messages. If two messages have the same message identifiers and the same service identifier they are treated as two copies of the same message, even if they have different payloads or if they have different source addresses. For stream-oriented services, an enhanced service message contains a stream identifier and a sequence number that determines the offset of the payload in the stream. The sequence number is either set by the application program or by the overlay socket. The identifiers are set when a message is transmitted, either by the application program or, if the application program does not specify some values, by the overlay socket. If the selection of the sequence number is left to the overlay socket, the sequence number is incremented by one for each message, with a randomly selected initial sequence number.

Application programmers use the enhanced services API by creating a message with a specified service identifier. Once created, the message is transmitted with an appropriate method to transmit a message, e.g., sendTo, SendToAll, SendFlood.

The services available through the enhanced services API are listed in Table 1. The table includes the service identifier and the available delivery modes. The services are as follows:

- **Hop-by-hop acknowledgement (H2HACK):** This service confirms the delivery of a message between neighbors in the overlay network. Whenever a socket receives a message, it sends a delivery confirmation (ACK) to the neighbor from which the message was received. If a neighbor does not receive a confirmation, it retransmits the message.
- **End-to-end acknowledgement (E2EACK):** The E2EACK service confirms delivery of a message to the source of a message. For unicast messages, the destination sends a delivery confirmation that is forwarded towards the source. For broadcast messages, each overlay socket that forwards a message waits until it has received an ACK from all downstream neighbors in the delivery tree and sends a single ACK to the upstream neighbor in the delivery tree. If an ACK is not received then the message is retransmitted. The application program is alerted through events when all ACKs arrive at the source (`E2EACK_RECEIVED`), or when some ACKs are not received after a certain number of retransmissions (`E2EPARTIALACK_RECEIVED`).
- **Duplicate Elimination (DELDUPS):** This service discards a message if it is a duplicate of an earlier received message.
- **Neighbor synchronization (SYNC):** The overlay socket stores each transmitted and received message and synchronizes the stored message with its neighbors in the overlay network.

- **Incast (INCAST):** This service merges the payload of unicast messages with identical destination address and message identifier.
- **Best-effort ordering (INORDER):** In this stream-oriented service, received messages are passed to the application program in the order of sequence numbers.

Table 1. Enhanced services in HyperCast.

Enhanced Service	***Service Identifier***	***Service Type***	***Available Delivery Modes***
Hop-by-hop acknowledgement	H2HACK	message-oriented	Unicast Broadcast
End-to-end acknowledgement	E2EACK	message-oriented	Unicast Broadcast
Synchronization	SYNC	message-oriented	Broadcast
Incast	INCAST	message-oriented	Unicast
Duplicate Elimination	DELDUPS	message-oriented	All
Best-effort ordering	INORDER	stream-oriented	All

In the following we describe the methods of the enhanced services API. With exception of the createMessage method, all methods are available through the I_OverlayMessage interface.

I_OverlayMessage createMessage(byte[] payload, short serviceID)
: This method belongs to the I_OverlaySocket interface. It creates an enhanced service message with a given service identifier. The service identifier must be one of the available services: H2HACK, E2EACK, DELDUPS, SYNC, INCAST, INORDER.

void setServiceIdentifier (short service)
: Sets the service identifier of the message to one of the available services. When a message is created with the signature

  OverlayMessage createMessage(byte[] payload)

  the message can be changed into an enhanced service message by setting the service identifier.

short GetServiceIdentifier ()
: Returns the service identifier of a message.

void setStreamIdentifier (short StreamID)
: Sets the stream identifier to one of the available services message if this is a service message with a stream-oriented service

short GetStreamIdentifier ()
: Returns the stream identifier if this is a service message with a stream-oriented service.

void setSequenceNumber(int getSequenceNumber)
: Sets the sequence number of an enhanced service message with a stream-oriented service. If the application does not select the sequence number, the overlay socket sets the sequence number. Sequence numbers set by the overlay socket are incremented by one for each subsequent message, starting with a randomly selected initial sequence number.

int getSequenceNumber()
: Returns the sequence number of the message if this is a message with a stream-oriented service.

void setMsgIdentifier (int MsgID)
: Sets the message identifier if this is a message with a message-oriented service.

short getMsgIdentifier()
: Returns the message identifier if this is a message with a message-oriented service.

static int generateMsgIdentifier ()
: Creates a message identifier that can be used by the application program.

static int generateStreamIdentifier ()
: Creates a message identifier that can be used by the application program.

We next give a more extensive description of the enhanced services in HyperCast, which gives enough information so that an application programmer can take advantage of these services. We refer to Chapter **[MStore]** for the details of the services.

Services with acknowledgements

The hop-by-hop acknowledgement (H2HACK) and end-to-end acknowledgement (E2EACK) services improve the reliability of message transmissions in an overlay network. The H2HACK service assures the delivery of a message between neighbor(s) in the overlay network. The E2EACK service confirms to the source whether or not all intended destinations have received a message.

Figures 1 and 2 show an example that illustrates these services work for the transmission of a broadcast message. Each node in the depicted graphs represents an application with an overlay socket that has joined the overlay network. The edges between nodes indicate that nodes are neighbors in the overlay network. We assume that there is a broadcast transmission from node A. Recall that broadcast messages are transmitted by forwarding a message to downstream neighbors in a rooted tree that is embedded in the overlay topology and that has the sender of the message as the root of the tree. The transmission of an overlay message is indicated by an arrow that has label *Msg*, and the transmission of a delivery confirmations are indicated by arrows that are labeled *H2H ACK* or *E2E ACK*. Figure 1 shows a transmission scenario for the H2HACK service, and Figure 2 shows a scenario for the E2EACK service.

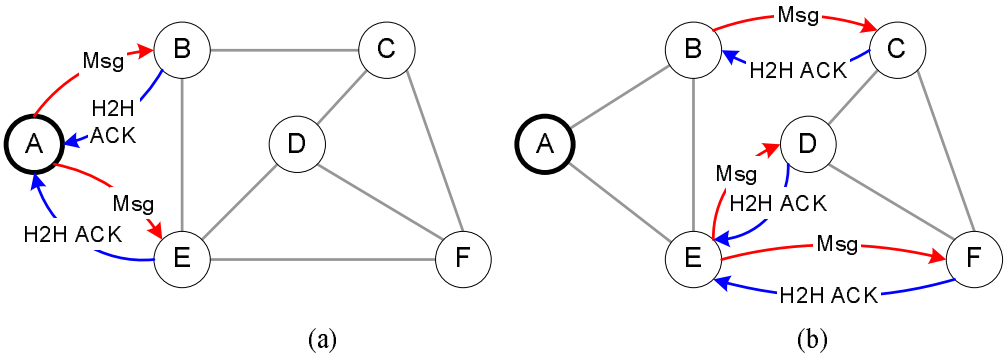

Figure 1. H2HACK service.

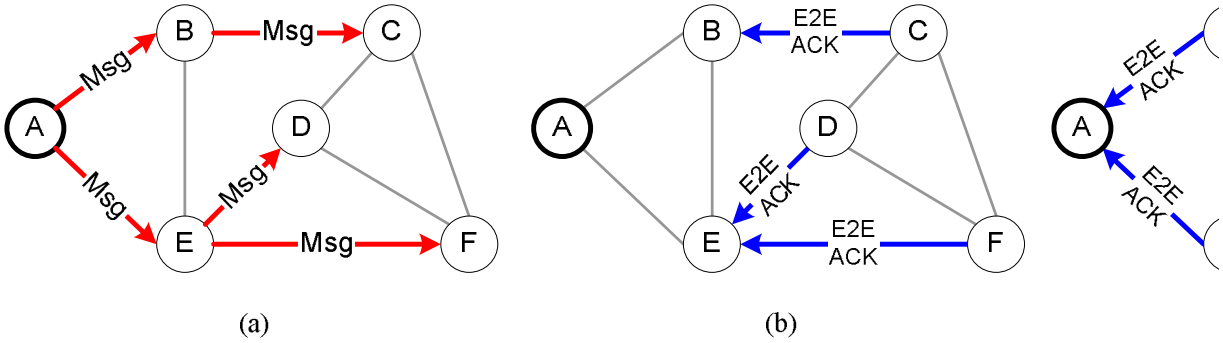

Figure 2. E2EACK service.

In the H2HACK service, node A forwards the message to its downstream neighbors in the tree that has A as root (Figure 1(a)). As soon as B and E receive the message, they each send an H2H ACK to A. Then, B and E forward the message to their downstream neighbors (Figure 1(b)), which respond with an H2H ACK. When a node does not send an acknowledgment, a message is retransmitted, but each message is only transmitted a finite number of times. [What is the default value?]

In the E2E ACK service, the transmission of delivery confirmations is realized in a recursive fashion. When a source transmits a message to its downstream, it expects an ACK from each of its neighbors. The neighbors send an ACK only after they have received an ACK from their own downstream neighbors to which they have forwarded the message. An overlay socket that does not have downstream neighbors sends an ACK immediately after receiving the message. In this way, the ACKs from the leaves trigger the transmission of ACKs at intermediate nodes. When the source of the message receives an ACK from each of its downstream neighbor, the E2EACK service for this message is completed. If that happens, the application receives an event notification of type `E2EACK_RECEIVED`. (The notification system in HyperCast is explained later in this chapter.) If an overlay socket does not receive an E2E ACK from all its downstream neighbors, it will, after a timeout period, send a *partial ACK* to its upstream neighbor. When the source of the data receives a partial ACK from one of its neighbors, it sends a notification of type `E2EPARTIALACK_RECEIVED` to the application. This ensures that some kind of acknowledgement is delivered to the source of the message, even if not all intended receivers have acknowledged the receipt of the message. In Figure 2(a), A sends a broadcast message with an E2EACK service, and then waits for an E2E ACK message. When B and E receive the message, they forward the message to their

downstream neighbors and wait for an ACK (Figure 2(b)). When C, D, and F receive the message, they immediately send an E2E ACK because they do not have downstream neighbors with respect to A (Figure 2(c)). Nodes B and E send their E2E ACK immediately after they have received E2E ACKs from all downstream neighbors. When A has received an ACK from both B and E, the message has been confirmed and a notification about this message is sent to application.

The above example explained the transmission of broadcast messages. The H2HACK and E2EACK services are also available for unicast messages. However, the implementation of the services for unicast messages is different, since unicast messages are transmitted upstream in a tree that is rooted at the destination of the message. In the H2HACK server, each overlay sockets that forwards a message to an upstream neighbor expects an H2H ACK from this server, and retransmits the message if no acknowledgement is sent. An overlay socket that receives a message, immediately sends an H2H ACK. In the E2EACK service, ~~there are two methods to send acknowledgements for unicast services. By default, each overlay socket that forwards a message to an upstream neighbor waits until an E2E ACK is received from that neighbor. An overlay socket sends an E2E ACK only if it has received an ACK from its upstream neighbor. If the receiver is the destination it immediately sends an E2E ACK. In an alternative method to realize acknowledgments in the E2E ACK service,~~ only the destination sends an E2E ACK for a unicast message. Here, the acknowledgement is sent as a unicast message to the source of the message.

The H2HACK and E2EACK services do not assume that an overlay topology is fixed when a message or an acknowledgement is transmitted. In fact, the implementation of the services accounts for overlay networks where the membership is changing frequently. In the H2HACK and E2EACK services, and all other enhanced services, each overlay socket always uses the latest information about its upstream and downstream neighbors to determine where to transmit messages and from where to expect ACKs.

The semantics of the H2HACK and E2EACK services are quite different. The H2HACK service provides a reliable transfer of overlay messages between neighbors in the overlay network. If an overlay socket receives an H2H ACK for a message, it knows that an immediate neighbor has received the message. The socket does not know whether the neighbor has successfully forwarded the message to other overlay sockets. When an overlay socket receives a message and sends an H2H ACK, but then drops the message before it is forwarded to the next neighbor, the H2HACK service does not recover the message. The delivery service of H2HACK is similar to the service provided by a reliable underlay network protocols, e.g., TCP, between overlay sockets. In fact, when TCP is used as the underlay protocol the H2HACK service does not offer any additional benefits.

The level of delivery assurance provided by the E2EACK service is much stronger than in the H2HACK service. In most cases, when an application receives an `E2EACK_RECEIVED` event, all intended receivers have received the message. However, it is relatively easy to construct a scenario where the source receives the event, but not all receivers obtain the message. For example, when a new overlay socket joins an overlay network before the source has issues an `E2EACK_RECEIVED`, but after all its neighbors in the overlay network have transmitted an E2E ACK for a message, the message will not be received by the new sender. Without keeping track of the membership of overlay sockets in an overlay network, it may be difficult to significantly improved the delivery assurance of the E2EACK service. We note that in an overlay network where overlay

sockets frequently join and leave, there is an increased chance that the transmission of messages and E2E ACKs is not completed before the timeout for partial ACKs occurs, and a `E2EPARTIALACK_RECEIVED` is issued to the source of the message application.

The following program transmits the HelloWorld message using the E2E ACK service. The methods *handle_E2EACK_RECEIVED* and *handle_E2EPARTIALACK_RECEIVED* are event handler methods that are part of the *NotificationHandler* class. The methods are called when the corresponding events occur. In the example, when the application program receives a partial acknowledgement, it retransmits the message.

```
public class HelloWorld extends NotificationHandler implements
I_ApplicationCallBack I_ReceiveCallback {

 public static void main(String[] args) {
        String MyString = new String("Hello World");
        HelloWorld hw = new HelloWorld();
        HyperCastConfig ConfObj =
        HyperCastConfig.createOLConfig("hypercast.xml");
        I_OverlaySocket MySocket=ConfObj.createOverlaySocket(hw, hw);
        MySocket.joinOverlay();
        Thread.sleep(4000);

        I_OverlayMessage msg =
                MySocket.createMessage(MyString.getBytes(), E2EACK);
        MySocket.sendToAll(msg);
 }

 void handle_E2EACK_RECEIVED (Object event) {
        short MsgID = (byte []) event;
        System.out.println("Message " + MsgID+ “is acknowledged”);
 }

 void handle_E2EPARTIALACK_RECEIVED (Object event) {
        short MsgID = event;
        System.out.println("Message " + MsgID+ “partially acknowledged”);
        //Redo with different message ID;
        String MyString = new String("Hello World");
        I_OverlayMessage msg =
                        MySocket.createMessage(MyString.getBytes(), E2EACK);
        MySocket.sendToAll(msg);
 }
}
```

Synchronization

The synchronization service of HyperCast assures persistence of overlay messages that are transmitted in an overlay network. Each overlay socket stores all messages that have been transmitted or received in a local repository. Periodically, each overlay socket contacts its neighbors in the overlay network to synchronize the content of its repository with that of its neighbors. If a neighbor has a message that is missing in the local repository, the overlay socket requests the missing message from the neighbor. Whenever an overlay message receives a message for the first time, the message is delivered to the application program.

The synchronization service is useful for interactive shared collaboration applications, e.g., a shared whiteboard, where an application that joins an overlay network needs to receive data that has been transmitted before the application joined. The synchronization service can help with recovering messages that were transmitted while an application was disconnected from the overlay network.

Incast

An incast service supports many-to-one communication primitives, where multiple applications send data to a single application. The need for many-to-one communications appears in many application domains, such as, collection of sensor data from a sensor array, collection of responses to distributed queries, and transmission of network management information from managed devices to a monitoring application. Incast can be thought of as a reverse broadcast where multiple overlay sockets send data to a single overlay socket. An incast message is a unicast message. Each overlay socket that receives and forwards an incast message stores and holds the message for a finite period of time. If during this holding time, other messages arrive with the same message identifier and the same destination address, the overlay socket concatenates the payloads of these messages to form a single payload. When the holding time of a message expires, a single message is sent to the next upstream neighbor in the tree. In this sense, incast permits to merge messages that are sent from multiple sources to the same destination. The incast service only merges messages that have the same destination address and message identifier.

Duplicate Elimination

In this service, an overlay socket memorizes the message identifier of each message that is forwarded or delivered to the application program. If the overlay socket receives a message with a known message identifier, the message is dropped. The duplicate elimination service is particularly useful in conjunction with the flooding delivery mode, where it can prevent the transmission of duplicates of the same message.

Best-effort Ordering

The best-effort ordering message is a stream-oriented service that tries to ensure an in-sequence delivery of messages of messages with the same stream identifier. An overlay socket uses the sequence number in stream-oriented messages to determine if an arrived message is in sequence. The service assumes that messages are transmitted with increased sequence numbers. The service is useful in situations when frequent changes to the network topology cause out-of-sequence delivery of messages, i.e., messages are received in a different order than they were sent. The INORDER delivery does not recover lost or excessively delayed messages. A reliable in-sequence service that

combines a retransmission scheme of the H2HACK or E2EACK services with the INORDER service is currently not provided in HyperCast.

The INORDER service does not consider the source address of a message when ordering messages according to sequence numbers. Thus, when INORDER messages with identical stream identifier are received from different sources, the overlay socket treats them as belonging to the same sequence.

An overlay socket uses the concept of an expected sequence number to determine if a message is in-sequence. The expected sequence number is initialized with the first received message for a given stream identifier. The expected sequence number is incremented each time a message is delivered to the application program. After the first message, the overlay socket delivers the message with the expected sequence number to the application program. Received messages with a sequence number smaller than the expected sequence number are discarded. Messages with a sequence number larger than the expected sequence number are buffered for some time. When the timer goes off, the message and all buffered messages with a smaller sequence number are passed to the application in the order of the sequence numbers, and the expected sequence number is reset. Since an in-sequence delivery of messages to the application is violated when a timeout occurs on the buffering times, the service is said to provide a best-effort ordering.

3.2. STREAM API

The stream API for overlay sockets gives programmers the simplicity and convenience of the byte-oriented streams programming abstraction from the Java I/O package. Streams in Java are defined as ordered sequences of data from a source (*input streams*) or to a destination (*output streams*). The stream API is built on top of the message-oriented interface of the overlay socket. It permits programs to send or receive bytes by writing to a Java OutputStream or by reading from a Java InputStream. The stream API is provided through the StreamManager class, which translates the message-oriented best-effort ordering service (INORDER) into a byte-stream oriented API.

Input and output streams are bound to a stream identifier of INORDER messages. The binding occurs when the streams are created. When an application program creates an output stream and binds it to a stream identifier, the data transmitted on this stream are INORDER messages with the given stream identifier. There are no explicit operations for allocating or releasing stream identifiers. A stream identifier is allocated implicitly when an overlay socket is sending data with a given stream identifier for the first time. An overlay socket may delete information about a stream identifier if no data is sent to the stream for a long time. Stream identifiers are locally interpreted and do not have a scope where uniqueness is assumed or enforced. For unicast messages, messages with identical stream identifier can be in use for data streams between different senders and receivers, without interfering with each other. For broadcast messages, if two sources use the same stream identifier, the messages may become interleaved at the receivers.

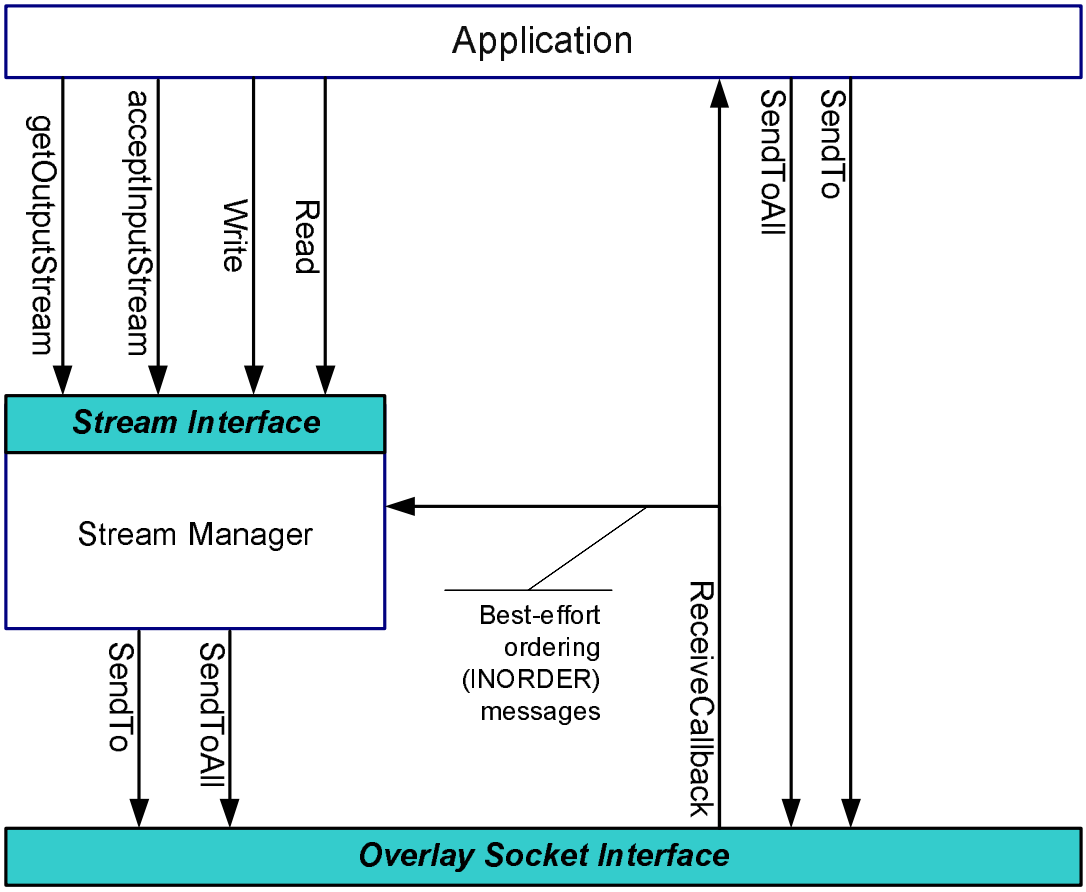

Figure 3. Stream Manager.

The following application program illustrates the use of the Stream API. We present a version of the HelloWorld program that transmit the “Hello World” string to an output stream, and a version that reads the string from an input stream. The implementation of the sender is as follows:

```
public class HelloWorld_StreamSender {
   public static void main(String[] args) {

        String MyString = new String("Hello World");
        HyperCastConfig ConfObj = HyperCastConfig.createOLConfig("hypercast.xml");
        I_OverlaySocket MySocket=ConfObj.createOverlaySocket(null);
        MySocket.joinOverlay();
        Thread.sleep(4000);

        StreamManager MyStreams =  MySocket.getStreamManager();
        try {
                HCastOutputStream out =  MyStreams.getOutputStream(1111);
                byte[] b = MyString.getBytes();
                for (int i=0; i < b.length; i++) {
                        out.write((int) b[i]);
                }
```



```
                out.flush();
        } catch(Exception e) { }
    }
}
```

The program creates an overlay socket that joins an overlay network, as seen many times before. Before sending data, the program waits for a few seconds to give the overlay protocol the opportunity to integrate the overlay socket in the overlay network. Then, the application program creates an object of type StreamManager. This is the stream manager that supports the creation of input and output streams for an overlay socket. The program requests from the stream manager an output steam that is bound to stream identifier 1111. The class HCastOutputstream extends the base class OutputStream by a few methods that give access to HyperCast specific information. Once the output stream is created, the program writes the string "HelloWorld" byte by byte. The stream manager is responsible for generating overlay messages, filling the payload of the message, and transmit the message. By calling flush, the application forces the stream manager to transmit an overlay message. By default, the stream manager broadcasts all data sent to an HCastOutputstream output stream, but the delivery mode can be modified by the application. In the above example, the delivery mode of the output stream can be set to unicast by invoking.

```
out.SetUnicast (LAddr);
```

Here, *LAddr* is the logical address of the intended destination in the overlay network. The *HCastOutputstream* can switch back to broadcast delivery mode by issuing

```
out.SetBroadcastl;
```

Next, we take a look at the corresponding program that receives a stream. This is implemented by the following program:

```
public class HelloWorld_StreamReceiver {
   public static void main(String[] args) {

        HyperCastConf ConfObj = HyperCastConf.createOLConfig("hypercast.xml");
        I_OverlaySocket MySocket=ConfObj.createOverlaySocket(null);
        MySocket.joinOverlay();

        StreamManager MyStreams =  MySocket.getStreamManager();
        HCastInputStream in = MyStreams.acceptInputStream(1111);
        byte[] a = new byte[1];
        int length;
        try {
                while((length = in.read(a)) > 0) {
                        for (int i=0; i<length; i++) {
                                System.out.println((char) a[i]);
                        }
                }
                in.close();
```



```
        } catch (Exception ee) {}
    }
}
```

This program creates a *StreamManager* object which creates an *HCastInputstream* that is bound to stream identifier 1111. The *acceptInputStream* method blocks the application program until the stream manager receives a message with the specified stream identifier. If a message with stream identifier 1111 arrives, the application reads the data byte by byte and displays the results. The stream manager buffers incoming traffic for each stream identifier, even if the application has not created an input stream for the identifier. However, the amount of data buffered is limited. In the example, when *acceptInputStream* is issued and data from the *HelloWorld_StreamSender* program has already arrived for stream identifier 1111, the application may receive data that has been received before the application has issues the *acceptInputStream* data. The stream manager does not distinguish different sources when receiving messages on an input stream. Therefore, when multiple applications use the same stream identifier to send data to the same overlay socket, the data of the stream from different sources may become interleaved in the stream manager.

In the HelloWorld program, the application program specifies a specific stream identifier. If data arrives to the stream manager, but not for stream identifier 1111, the application continues to wait. If the application wants to create an input stream for arriving data with any given stream identifier, it uses the statement.

```
HCastInputStream in = MyStreams.acceptInputStream();
```

API of the Stream Manager

Next, we give an overview of the API for programs that take advantage of the stream manager. With exception of the first method, all of the following methods are defined by the StreamManager class.

StreamManager getStreamManager()

A stream manager is created by the *getStreamManager* method of the overlay socket interface. The stream manager can be created at any time during the lifetime of the connections. Once created, the stream manager exists for the remainder of the lifetime of the overlay socket. After a stream manager is created, all INORDER messages that the overlay socket delivers to the application program are sent to the stream manager.

int getServiceIdentifier ()

Returns the service identifier of a message.

HCastInputStream acceptInputStream()

HCastInputStream acceptInputStream(short sid)

Block the calling thread until data arrives to the stream manager and creates an input stream. Without a parameter the calling thread is blocked until data arrives for a stream identifier for which the application does not yet have an input stream. If the call is issued and data is available in the stream manager for one or more streams for

which no input stream has been created, the stream manager creates an HCastInputStream for one of these streams. If a parameter is provided, the method blocks the calling thread until data is available with a matching stream identifier.

HCastOutputStream getOutputStream()

HCastOutputStream getOutputStream (short sid)

Creates an *HCastOutputStream* with a given stream identifier. If no parameter is provided the stream manager determines the identifier. After an output stream is created the application program can write data to the output stream.

API of HCastInputStream and HCastOutputStream

The HCastInputStream and HCastOutputStream classes extend the InputStream and OutputStream in the Java I/O package, and support all methods defined for these streams. For input streams, these include the following methods:

int available()

Returns the number of bytes that can be read (or skipped over) from this input stream without blocking the next caller of a method for this input stream.

void close()

Closes an input stream and releases any system resources associated with the stream.

int read()

Reads the next byte of data from the input stream.

int read(byte[] b)

Reads some number of bytes from the input stream and stores them in a buffer.

int read(byte[] b, int off, int len)

Reads up to *len* bytes of data from the input stream into an array of bytes.

long skip(long num)

Skips over and discards *num* bytes of data from this input stream.

The methods required by OutputStream are as follows:

void close()

Closes this output stream and releases any system resources associated with this stream.

void flush()

Flushes this output stream and forces any buffered output bytes to be written out.

void write (byte[] b)

Writes b.length bytes from the specified byte array to this output stream.

void write (byte[] b, int offset, int len)

Writes a given number of bytes an output stream. The written data is from a byte array starting at a specified offset.

void write (byte b)
Writes the specified byte to this output stream.

The following additional methods are defined in the HCastOutputStream class provides, in addition the following methods:

short getStreamIdentifier ()
Returns the stream identifier associated with an HCastInputStream or HCastOutputStream.

void setUnicast (I_LogicalAddress Laddr)
Sets the delivery mode for all outgoing messages to unicast with the specified logical address as destination.

void setBroadcast ()
Sets the delivery mode for all outgoing messages to broadcast.

void setFlood ()
Sets the delivery mode for all outgoing messages to flooding.

byte getDeliveryMode ()
Returns the current delivery mode for outgoing messages. The values are broadcast (0x1), flood (x02), or unicast (0x3).

void setHopLimit (short value)
Sets the hop limit field for outgoing messages of a given output stream. The hop limit is the maximum number of overlay sockets traversed before a message is dropped.

short getStreamIdentifer (InputStream)
Returns the stream identifier of an HCastInputStream and HCastOutputStream.

3.3. EVENT NOTIFICATIONS

HyperCast has a notification system by which the overlay socket can report events of interest to an application program. For example, when an application program has issued a *joinOverlay* it often wants to wait until the overlay socket has been integrated in the overlay network topology before transmitting data. In several of the earlier presented versions of the HelloWorld program, application programs paused for a few seconds before transmitting the "HelloWorld" string. However, this time may overestimate or underestimate the actual time until the overlay socket is integrated into the overlay network. With the event notification system of the overlay socket, an application program can be alerted when the overlay protocol has stabilized.

The event notification system for application is handled by the *NotificationHandler* class. Objects of this class have an event queue and a thread. The notification system is triggered when some object in the overlay socket fires an event. Firing an event corresponds to adding an event object to the event queue. The thread processes events in the queue in a FIFO order. For each event, a *NotificationHandler* has an event handler

method with an empty implementation. By overriding these empty handler methods an application programs can determine the code to be executed when an event occurs. All events in HyperCast are predefined. An application programmer can write custom event handlers, but cannot define new events. Adding new events requires changes to the HyperCast software and is explained in Chapter **[ChangeCode].**

To enable notifications of events, an application program creates an object that extends the HyperCast event handler class *NotificationHandler*. An application can use events in three ways: (1) It can specify a handler method that is executed when an event is fired; (2) It can block and wait until an event occurs; Or (3) it can implement a complete notification handler that processes all events.

We next show how an application program that uses the event notification system. The following code segments defines a customized event handler that extends the *NotificationHandler* class:

```
class MyNotificationHandler extends NotificationHandler {

   public void handle_NODE_HASBECOMESTABLE
                        (final NODE_HASBECOMESTABLE event) {
     System.out.println ("Socket is in a stable state at: " + event.getTimestamp());
   }

   public void handle_NODE_NEIGHBORHOODCHANGED
                        (final NODE_NEIGHBORHOODCHANGED event) {
     System.out.println ("Neighborhood change at: " + event.getTimestamp());
   }
}
```

This class overrides the handler methods for the events NODE_HASBECOMESTABLE and NODE_NEIGHBORHOODCHANGED. The event NODE_HASBECOMESTABLE is fired when the overlay protocol has reached a stable state. Often, when a protocol has become stable for the first time after an overlay socket has joined the overlay network, the overlay socket has been fully integrated into the overlay socket. The event NODE_NEIGHBORHOODCHANGED is fired when the neighborhood of the overlay socket has changed, that is, a new neighbor has arrived or an existing neighbor has left. Most overlay protocols support these events. For all other events, the empty default handlers are invoked.

The example illustrates the convention for naming event handler methods. For an event of type

NODE_HASBECOMESTABLE

the handler method is called

handle_NODE_HASBECOMESTABLE ()

When an event handler is invoked it contains as parameter an event event object, that carries information about the event. Each event object has a timestamp that can be accessed with the method *getTimestamp*. The timestamp is the time when the event has occurred. In addition, there is an event specific object that contains information about the event. To exploit this information, an application programmer needs to know the type of the event specific object.

The following version of the *HelloWorld* program uses the previously defined MyNotificationHandler class:

```
public class HelloWorld implements I_ApplicationCallBack I_ReceiveCallback {

   public static void main(String[] args) {
        String MyString = new String("Hello World");
        HelloWorld hw = new HelloWorld ();
        MyNotificationHandler nh = new MyNotificationHandler();

        HyperCastConfig ConfObj =  HyperCastConfig.createOLConfig("hypercast.xml");
        I_OverlaySocket MySocket=ConfObj.createOverlaySocket(hw, nh);
        MySocket.joinOverlay();

        hw. WaitUntilNODE_HASBECOMESTABLE();
        I_OverlayMessage msg =  MySocket.createMessage(MyString.getBytes());
        MySocket.sendToAll(msg);
   }
}
```

The program creates a notification handler object, which is passed as an argument when the overlay socket is created:

```
I_OverlaySocket  MySocket=ConfObj.createOverlaySocket(hw, nh);
```

This is a new signature of the *createOverlaySocket* method. The first argument sets the callback method for arriving messages, as seen many time before, and the second argument is the notification handler. If an overlay socket is created in this fashion, the event handlers defined in the class *MyNotificationHandler* are executed each time the corresponding events are fired. The HelloWorld program with E2EACK service (from Section **[Enhanced Service]** also contains an event notification handler. The implementation of the notification handler in that example is different, in that the application program extends the *NotificationHandler* class.

When the overlay socket is created, the application program blocks until the overlay protocol in the overlay socket has reached a stable state. This is done by calling:

```
nh. WaitUntilNODE_HASBECOMESTABLE ();
```

The WaitUntilNODE_HASBECOMESTABLE method is associated with the event NODE_HASBECOMESTABLE. The calling thread is blocked until the event is fired. If an application program does not want to wait indefinitely for an event, it can provide an argument that specifies the maximum waiting time in milliseconds. For example, by invoking

```
nh. WaitUntilNODE_HASBECOMESTABLE (1000);
```

the application program is blocked for at most one second. When the maximum waiting time is reached, the method unblocks the thread and returns a null object. If multiple threads are blocked on the same event, all threads waiting on the event become unblocked. This can be useful in multithreaded applications, where different threads may want to block on different events. The prefix WaitUntil is the convention used for all methods that block on an event. Note that the HelloWorld program blocks on the event

NODE_HASBECOMESTABLE and also receives an event notification via the event handler method handle_NODE_HASBECOMESTABLE.

The third method to deal with events is when the application program intercepts all events that are fired, instead of writing event specific handler methods. This is done by overriding the method handle of the NotificationHandler. The handle methods gives the application program the most flexibility with respect to processing events. However, overriding the handle method should be done with caution, since it disables the built-in event notification process of invoking event specific handler methods and unblocking of threads waiting on events. The following version of the notification handler contains an implememtation of the handle method. If this class is used for the HelloWorld program, then the call that blocks the program with WaitUntilNODE_HASBECOMESTABLE is not available.

```
class MyNotificationHandler extends NotificationHandler {

    public void handle (final NOTIFICATION_EVENT event) {
        if (event instanceof NODE_HASBECOMESTABLE) {
                NODE_HASBECOMESTABLE ev = (NODE_HASBECOMESTABLE) event;
                System.out.println ("NODE_HASBECOMESTABLE at: " +
                        ev.getTimestamp());
        }
        else {
                System.out.println ("Some other event at: " + event.getTimestamp());
        }
   }
}
```

At the end of the section, we give an overview of the available events. Refer to Table 2 for information how an event is fired and what the event object is. For each defined event, the NotificationHandler class has an event handling method (with prefix handle_) and a method on which a thread can block (with prefix WaitUntil). As discussed, applications can override the event handling methods, but the *WaitUntil* methods that block an application are immutable.

Table 2. Events defined in HyperCast.

Event	**Fired by:**	**Event object:**
NODE_HASBECOMESTABLE	Overlay protocols HC and DT	null
NODE_LEAVEOVERLAY	Overlay protocols HC, DT, and SPT	null
NODE_LOGICALADDRESSCHANGED	Overlay protocols HC, DT, and SPT	null
NODE_NEIGHBORHOODCHANGED	Overlay protocols HC, DT,	null

	and SPT	
E2EACK_RECEIVED, E2EPARTIALACK_RECEIVED	E2EACK enhanced service	I_OverlayMessage (currently it is a short containing a message identifier)
NEWSTREAM_ARRIVED~~_EVENT~~	Stream manager	short (containing a stream identifier)
NODE_SPT_CHILDRENCHANGED, NODE_SPT_PARENTCHANGED, NODE_SPT_ ROOTCHANGED	Overlay protocol SPT	????

- *NODE_HASBECOMESTABLE* (Note: currently it is NODE_ISSTABLE): The event is fired when the overlay node in the overlay socket satisfies the conditions for local stability as defined by the overlay network protocol. Normally, local stability means that the overlay protocol has converged after the local socket has joined the overlay or after a neighbor of the overlay socket has joined or left. The event may not be defined by all overlay protocols.
- *NODE_LEAVEOVERLAY* **(Note: Rename "NODE_LEAVEGROUP" to "NODE_LEAVEOVERLAY").** The event is fired by an overlay node when the node determines that it has left the overlay network.
- *NODE_LOGICALADDRESSCHANGED:* The event indicates that the logical address of the overlay socket has changed.
- *NODE_NEIGHBORHOODCHANGED:* The event is fired when the neighbor table of the overlay node has changed, that is, a new neighbor in the overlay network has appeared or an existing neighbor has left.
- *E2EACK_RECEIVED, E2EPARTIALACK_RECEIVED:* The events are fired at the source of messages that are sent with the E2EACK service. When an event E2EACK_RECEIVED is fired, the overlay socket has received all expected E2EACKs. This indicates that the delivery of an E2EACK message has been successful. When the event E2EPARTIALACK_RECEIVED is fired, the transmission of an E2EACK message has not been successful. The event object for these events contain the complete messages. With the message, the application can match the delivery confirmation with the transmitted message.
- *NEWSTREAM_ARRIVED~~_EVENT~~:* The event is fired by the stream manager, which implements the stream API, when data is received for a new stream identifier. An application program can react to this event by creating a new input stream.
- *NODE_SPT_CHILDRENCHANGED, NODE_SPT_PARENTCHANGED, NODE_SPT_ ROOTCHANGED:* These events are specific to the SPT protocol, and are explained in the context of the SPT protocol.

3.4. MESSAGE INTERCEPTION API

The message interception API consists of a callback method that permits applications participating in an overlay network to view and manipulate every incoming message

before the message is processed by the overlay socket. The interception API is an ideal access point for malicious or non-cooperating users in an overlay network, who want to filter or modify messages based on payload content. The interception API explicitly exposes the vulnerability of application overlay network, where a non-cooperating application may manipulate or filter data for which it is neither the sender nor the receiver. Application layer networks can discourage or limit the impact of non-cooperative or malicious users by providing appropriate incentives or by auditing the behavior of applications. HyperCast leaves it to the application programmer to supply such policies and mechanisms.

The role of the intercept callback is best explained by reviewing how an incoming overlay message is processed in an overlay socket. Refer to Figure 4 for an illustration. When a message arrives the overlay socket first verifies correctness of the message. Then the message is processed and the header of the message is updated. Next, the message is forwarded to neighbors as determined by the message header and the neighbor table of the overlay protocol. In the last step, the overlay socket determines if the message should be delivered to the application program. Broadcast and flooding messages are always passed to the application. Unicast messages are passed to the application if the destination address of the message matches the logical address of the overlay socket. A message is passed to an application in one of two ways. By default, the message is written to the application buffer from which it can be retrieved by the application program with the receive method. If the application program provides a receive callback method at the time the overlay socket is created, this callback is invoked on the message.

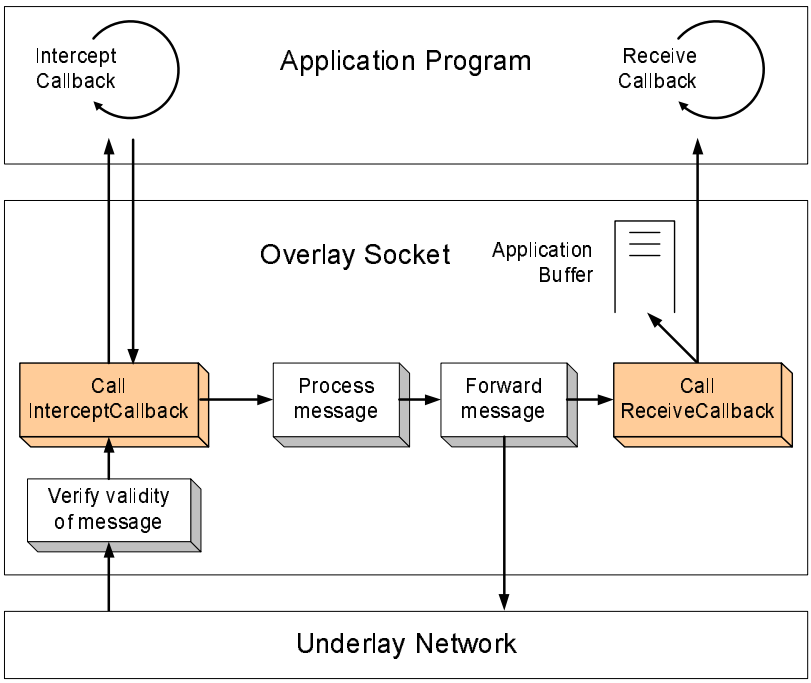

Figure 4. Processing an Incoming Packet.

Figure 4 illustrates that the intercept callback is invoked right after the correctness of the header is verified. When the callback is completed the overlay socket continues processing the message. The intercept callback can change the payload of the message, modify the source and destination addresses, change a unicast message to a broadcast message, force the message to be dropped by setting the hop limit field to zero, and so

forth. In comparison, the regular receive operation is safer, since the message is passed to the receive method only after the overlay socket is done with processing the message.

When an application program uses the intercept callback, the application program may see the same overlay message twice: Once by the intercept callback before the message is processed by the overlay socket, and once by the receive callback after the message is processed by the overlay socket. Figure 5 illustrates the traversal of a unicast message through the overlay network.

The figure shows the processing of a message at the source, at an intermediate application that forwards the unicast message, and at the destination the message. At the source the message is passed by the application program to the overlay socket, and is transmitted over the underlay network. When an overlay socket receives the unicast message, the message is normally only passed to the application if the overlay socket is the destination of the message. However, with the intercept callback, all intermediate applications see and possibly modify the unicast message. At the destination, the message is first passed to the intercept callback and then to the receive callback.

Figures 6 shows how the intercept callback operates for broadcast messages. Since a broadcast message is forwarded to all applications in the overlay network, each application in the overlay network intercepts the message.

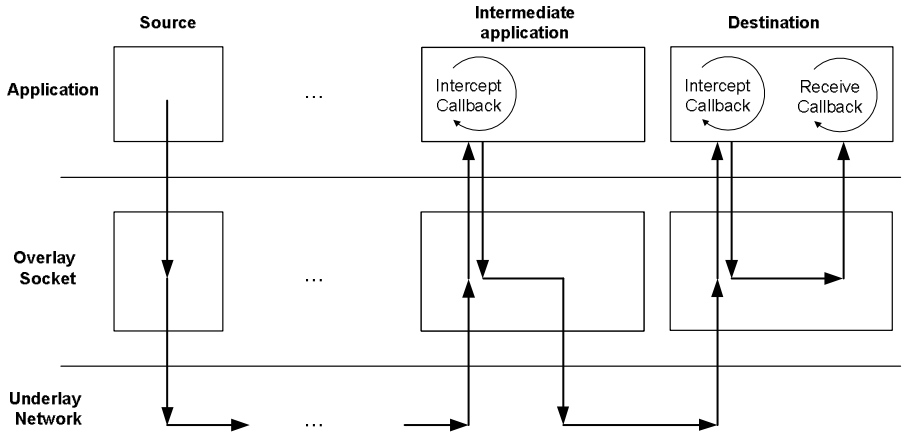

Figure 5. Interception of a unicast messages

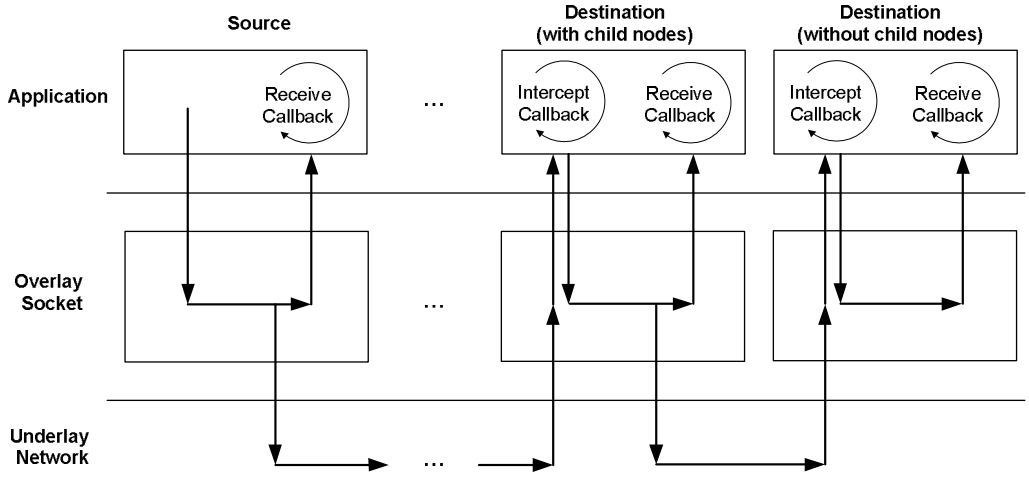

Figure 6. Interception of a broadcast message

Although it appears that the main role of the intercept callback is for malicious purposes, the interface can be useful in many scenarios. In the following version of the HelloWorld program, an overlay socket uses the intercept callback to add its own logical address to the payload. As a result, each transmitted overlay message carries information about its route through the overlay network. The following program extends the HelloWorld program to include the interception API:

```
public class HelloWorld implements I_ReceiveCallback, I_InterceptCallback {

  I_OverlayMessage  InterceptCallback (I_OverlayMessage msg) {
        byte[] data = msg.getPayload();
        String ChangedString = data.toString() +
                  "(" + MySocket.getLogicalAddress().toString()+ ","
                     System.currentTimeMillis() + ")";
        Msg.setPayload = ChangedString.toBytes()
        return msg;
 }

 public void ReceiveCallback (I_OverlayMessage msg) {
        byte[] data = msg.getPayload();
        System.out.println("Received message is " + new  String(data) + ".");
 }

 public static void main(String[] args) {
        HelloWorld hw = new HelloWorld ();
        String MyString = new String("Hello World");
        NotificationHandler nh = new NotificationHandler();

        HyperCastConfig ConfObj = HyperCastConfig.createOLConfig("hypercast.xml");
        I_OverlaySocket MySocket=ConfObj.createOverlaySocket(hw, nh, hw);
        MySocket.joinOverlay();
        nh.WaitUntilNODE_HASBECOMESTABLE();
        MyString = MyString + MySocket.getLogicalAddress().toString();
        I_OverlayMessage msg =  MySocket.createMessage(MyString.getBytes());
        MySocket.sendToAll(msg);
        for(;;);
 }
}
```

All application programs that use the intercept callback implement the I_*InterceptCallback* interface, which contains a single method *InterceptCallback*. This method is called when a message is intercepted. The implementation of the intercept callback in the extracts the payload of the message, and then changes the payload by adding the logical address of the overlay socket to the message. The result of the interception can be seen when the receive message displays the payload.

The intercept callback is set when the overlay socket is created. The parameters in

```
I_OverlaySocket MySocket=ConfObj.createOverlaySocket(hw, nh, hw);
```


sets the receive callback in the first argument, the notification callback in the second interface, and the intercept callback in the third argument. Here, the HelloWorld class provides callbacks for both intercepting and receiving messages.

In the *createOverlaySocket* method, the object that provides the intercept callback is always the third argument. If no receive callback and no notification handler are set, the first and second argument are set to null:

```
I_OverlaySocket MySocket=ConfObj.createOverlaySocket(null, null, hw);
```

3.5. CONFIGURING OVERLAY SOCKETS – ADVANCED TOPICS

This section has additional information for application programmers on the configuration of overlay sockets. We discuss the organization of configuration files, we show how to download configuration information from a server, discuss how the method createOLConfig operates when interacting with a server, and discuss an API that interacts with the configuration of an overlay socket.

Configuration Attributes

Configuration attributes describe the configuration of an overlay socket. All configuration attributes discussed so far are stored in a configuration. However, some attributes cannot be obtained from the configuration file, and must be specified by the application program. Such attributes generally contain confidential information, such as a password to access a private key or a certificate. We refer to configuration attributes that must be configured by the application program as *private attributes*, and all other attributes as *public attributes*. Private attributes are never stored in a configuration file and are not retrievable from a server.

From Chapter **[API-basic]** you know that configuration files for overlay sockets are XML documents. The root element of the configuration file has the name *public*, indicating that the configuration file contains the public attributes. Figure 7 shows the hierarchical organization of public attributes. Attributes that contain subattributes are indicated as boxes. When a configuration file is loaded by an application program, either from a local file or an overlay server, it is represented as a DOM document. A DOM document has a tree structure consisting of a hierarchy of nodes, each representing an element of the XML document. Each node in the tree represents an element in the XML file. The configuration object contains a DOM representation of the XML configuration file. We use XPath expressions to refer to a single element or a subtree in the XML document. As an example, */Public/Node/DTBuddyList/BuddyList* is an XPath expression that denotes the attribute *BuddyList* in the overlay protocol *DTBuddyList.*

Although they are not stored in a file, private attributes are also organized as an XML document and are internally represented as a DOM document. The root element of the document has the name *private*. The number of private attributes is small, and only a few socket configurations require private attributes, mostly for setting security properties of an overlay socket.

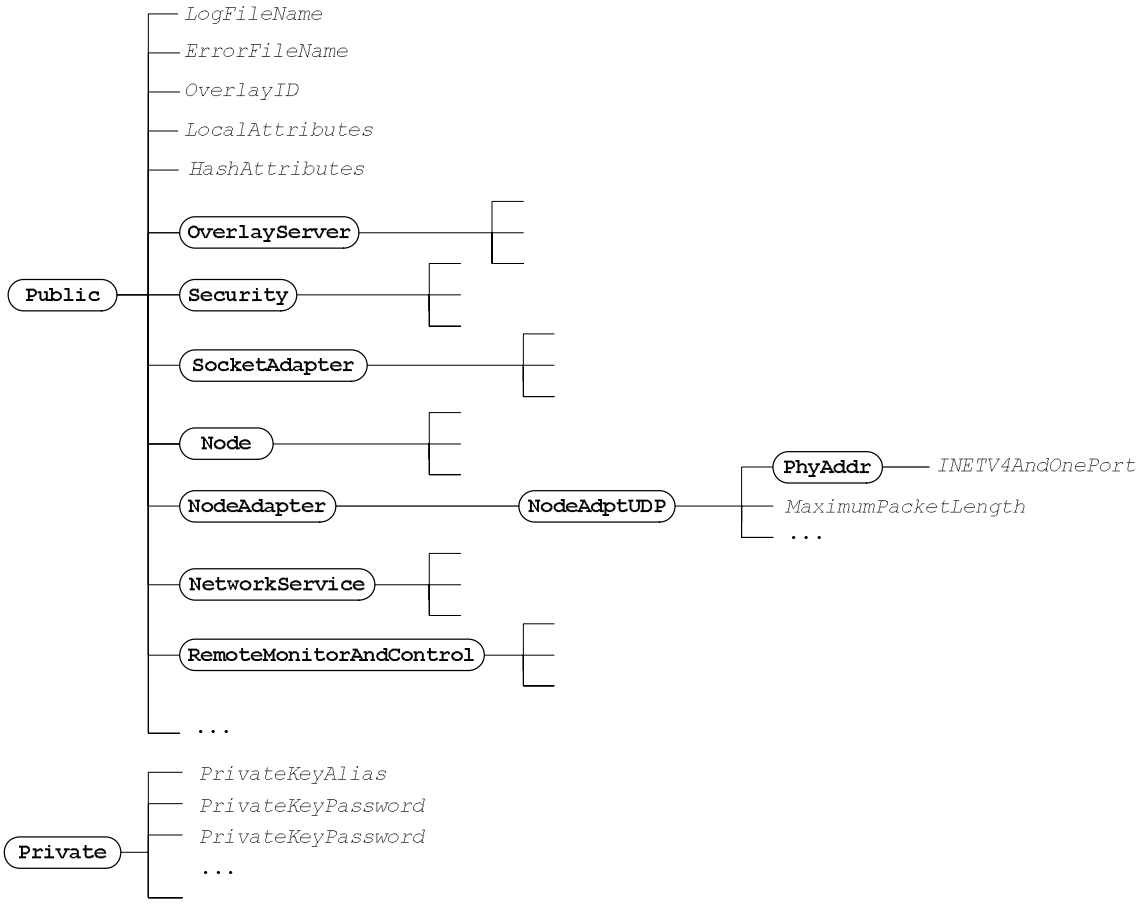

Figure 7. Structure of the configuration file.

The format and the default values of attributes are described by XML schema files. There is one XML schema file for the public attributes and another one for the private attributes. An XML configuration file document that is valid with respect to the schema file is likely to result in a usable configuration file. The schema files for the public and the private attributes are part of the HyperCast software distribution, and have names:

hypercast_*xxx*.xsd
hypercastPrivate_*xxx*.xsd

Here, *xxx* is the version of the HyperCast software. Each new distribution of the HyperCast socket contains new XML schema files. Since the schema files describe all valid configurations of overlay sockets, they are quite large. The API for configuration objects can validate a configuration against a schema definition.

When the configuration of an overlay socket requires an attribute that is not specified, either by the application program or the configuration file, then the overlay socket selects the default value for the attribute from the XML schema. If a default value is needed, but the schema file does not specify a default, then the configuration of the overlay socket fails.

Remark: To accelerate the lookup of default values when applications configure an overlay socket, the default values from the XML schema files are made available to application programs via the Java class file *ConfigurationDefault.class*. This avoids the

need to parse or lookup the XML schema file when a default value is needed. The HyperCast utility package *hypercast.util* contains a standalone program *ExtractConfigurationDefaults.class* which generates the *ConfigurationDefault.java* file. The program may be run as follows:

```
java hypercast.util.ExtractConfigurationDefaults
        ./hypercastMyVersion123.xsd
              ./hypercastPrivateMyVersion123.xsd
                    ConfigurationDefaults.java
```

Here, the name of the XML schema files are *hypercastMyVersion123.xsd* and *hypercastPrivateMyVersion123.xsd*. The *ConfigurationDefaults.java* file must be copied to the directory that contains the HyperCast source files. Since changes to the XML schema files of HyperCast are done by developers of HyperCast systems software, the creation of the default source file is transparent to application programmers.

Overlay servers

If HyperCast is running in an environment where it can access the Internet, an application can interact with a special HTTP server, called *overlay server*, that stores configuration files and generate new overlay network identifiers. An overlay server is an HTTP server that stores configuration files, where the configuration files are indexed by their local identifier. Given the URL of the configuration file, a user can download stored configuration files with a web browser. In addition to this obvious use of a web server, HyperCast provides methods to interact with an overlay server and automatically download or create configuration files.

The HyperCast software contains the implementation of an overlay server. This server is a minimal implementation of an HTTP server and is run as a standalone application. An overlay server is started with the command:

```
java overlay_server 8080
```

This command starts an overlay server at TCP port number 8080.

Creating configurations with *createOLConfig*

We next discuss in detail how the method createOLConfig creates a configuration object from a configuration file. The first step of the configuration of a socket is the creation of a HyperCastConfig object from the XML configuration file. The configuration object contains all information needed to configure an overlay socket. In all examples seen so far, an configuration object was created from a configuration file *hypercast.xml* by calling the static method:

```
HyperCastConfig ConfObj = HyperCastConfig.createOLConfig("hypercast.xml");
```

The createOLConfig method creates a configuration object from the configuration file and may also interact with an overlay server.

In the following we discuss the operations performed by the createOLConfig method, without and with an overlay server. Let us first assume that the configuration object does not involve an overlay server. This is indicated in the configuration file with one of the following selections:


```
<OverlayServer>None</Overlayserver>
<OverlayServer></Overlayserver>
<OverlayServer />
```

If the configuration file contains an overlay identifier, the overlay socket will try to join an overlay network with the given overlay identifier. If the file does not contain an overlay identifier, the configuration object picks a new overlay identifier, using a local address and a timestamp. The selection of a new overlay identifier results in the creation of a new overlay network.

Let us now consider that the configuration file specifies an overlay server. An overlay server is specified in the configuration file by providing the URL of the server, as follows:

```
<OverlayServer>
    <HTTPServer>143.71.22:8080/Groups</HTTPServer>
</OverlayServer>
```

If the server cannot be reached, the overlay socket is configured from the information in the local file.

Next we describe the actions performed by the createOLConfig method when the configuration file is executed. The method first creates a configuration object of type HyperCastConfig from the XML configuration file, and then contacts the overlay server. If the configuration file has specified an overlay identifier, the configuration object contacts the overlay server to test if the overlay server has a configuration file with the given identifier. If the overlay server has a configuration file for this overlay identifier, the configuration object will make an attempt to download this file. If the configuration file does not exist at the server, the configuration object uploads its own configuration file from the server, thereby creating a new configuration file at the overlay server. If some attribute values in the downloaded file are different from the local configuration file, the downloaded information takes precedence and overwrites the value in the configuration object. (The configuration file is not modified, however). There is one exception when overwriting attributes. Attributes listed in the LocalAttributes list in the local configuration file are not overwritten with downloaded information. Attributes in this list are assumed to be locally assigned.

Now let us suppose that the local configuration file does not specify an overlay identifier. Then, the configuration object uploads the configuration file (without an overlay identifier) to the overlay server. The server generates a new overlay identifier for this file, creates a new configuration file at the overlay server, and returns the complete file (with overlay identifier) to the configuration object. The overlay identifier in the returned file is used by the configuration object when the socket is created.

Figure 8 illustrates the interactions of the configuration object of type HyperCastConfig with the overlay server. The object is created from a configuration file. The configuration object interacts with the overlay server by testing if a given overlay identifier exists, and by downloading and uploading a configuration file. Once the interaction with the overlay socket is completed, the configuration object creates an overlay socket. Additionally, The HyperCastConfig object can also be accessed and with the methods of the configuration API discussed in the next subsection.

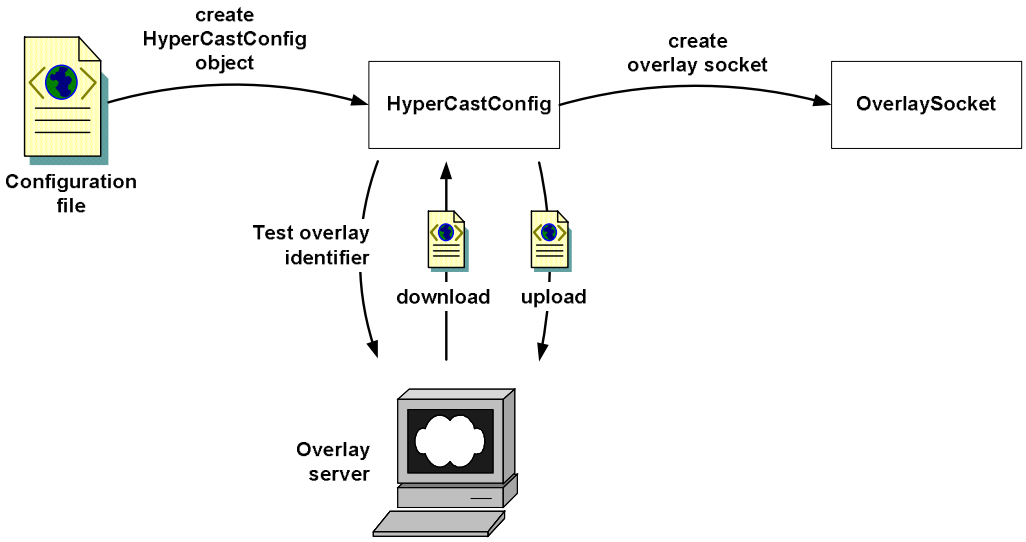

Figure 8. Interaction of the createOLConfig method with a configuration file and overlay server.

In all situations, when the overlay server cannot deal with a request, it returns an error message (that starts with "ERROR"). If that happens the configuration object throws a HyperCastConfigurationException and does create an overlay socket.

Programming with configuration objects

The method createOLConfig defines a default set of actions to be performed when configuring an overlay socket. While the createOLConfig method is likely sufficient for the vast majority of applications, the decisions made by createOLConfig, e.g., for selecting an overlay identifier when none is specified in the configuration file, may not be suitable for some application program.

In the following we describe an API for configuration objects that gives application programmers a great deal of flexibility for manipulating the configuration object. The API has a variety of methods for constructing a configuration, for uploading and downloading configuration files to and from an overlay server, and for reconciling configuration information from multiple sources.

Before discussing the methods of the API in detail, let us take a look at the implementation of the ` method. The method is implemented as follows:

```
public static HyperCastConfig createOLConfig(String filename) {
  HyperCastConfig ConfObj =new HyperCastConfig (filename);
  try {
        String overlayId =  ConfObject.getTextAttribute ("/OverlayID");
        String overlayServer = ConfObject.getTextAttribute ("/OverlayServer");

        Boolean existsOverlayId = ((overlayId == null) || (overlayId.equals ("")));
        Boolean existsOverlayServer =
                              (overlayServer == null) || (overlayServer.equals (""));
        if (! existsOverlayId && !  existsOverlayServer)
```



```
            ConfigObj.createandsetOverlayId();

        if ( existsOverlayServer ) {
            URL serverURL = ConfObject.getURIAttribute ("/OverlayServer");
            if (! existsOverlayId || ! ConfObject.doesOverlayExist (serverURI, overlayId))
                String OverlayId = ConfigObj.UploadConfig (serverURI);
            document DConfig
                        = ConfigObj.DownloadConfig(serverURI, OverlayId);
            ConfigObj.MergeConfig (document DConfig);
        }
  } catch (Exception e) {
        System.err.println("Exception when creating overlay  configuration object.");
        System.err.println("Exception: " + e.getMessage());
        System.err.println("Exiting.");
        e.printStackTrace(System.err);
        System.exit(1);
  }
return ConfObj;
}
```

The program first constructs a HyperCastConfig object from the XML configuration file whose name is passed as an argument. The configuration information is stored in a DOM document. Then the program determines if the configuration file contains an overlay identifier and an overlay server. If neither are specified then the configuration object sets a new overlay identifier with the method *createandsetOverlayId*. If an overlay server is specified, the URL of the server is extracted with the method *getURIAttribute*. If the configuration file does not have an overlay identifier or if the overlay identifier is not known at the server, then the configuration object uploads the entire DOM document with the configuration information to the overlay server. The method returns the new value of the overlay identifier that has been chosen by the overlay server. Before transmitting the document, it is converted into an XML document. Finally, the configuration object downloads the configuration file from the server (DownloadConfig) and then reconciles the downloaded information with the content of the configuration object.

Next we discuss the methods of the configuration API. Since the configuration object is needed for the configuration of an overlay socket, modifying the configuration object after a socket has been created has generally no effect.

HyperCastConfig (filename XMLfilename)

HyperCastConfig (URL serverURI, String OverlayId)

These are the two constructors for the configuration object. The first constructor uses an XML file, the second constructor downloads a configuration file from an overlay server, using the given overlay identifier. The constructors fail when the file does not exist or when the overlay server does not have a configuration file with the given overlay identifier.

The constructors also compute the overlay hash of the socket configuration. The overlay hash is built from the values of the hash attribute list, and serves as a signature for the configuration of an overlay socket. The overlay hash is included in

protocol messages sent by an overlay socket. An overlay socket rejects received protocol messages whose overlay hash is different from that of the overlay socket. In this way, an overlay socket can join an overlay network only if it has the same overlay hash as the sockets in the overlay network.

boolean testIfAttributeIsDefined (Xpath name)

Tests whether an attribute is defined. This method is generally not needed in application programs. When an overlay socket is configured and an attribute is not defined, then the configuration takes the default value for the attributes.

String getTextAttribute (String name)

Retrieves a public attribute from the configuration object. The attribute is specified as a string containing an XPath expression, and the type of the attribute is specified in the name of the method. There are many such methods, one for each data type used in the configuration file: getIntAttribute, getLongAttribute, getNonNegative-IntAttribute, getPositiveIntAttribute, getNonNegativeLongAttribute, getPositive-LongAttribute, getNonNegativeShortAttribute, getURIAttribute.

setTextAttribute (String name, String value)

Modifies a public attribute in a configuration object. The attribute is specified as a string containing an XPath expression, and the value is given as the second parameter. The attribute is specified in the name of the method. There is one method for each data type that can appear in the configuration value: setIntAttribute, setLongAttribute, setNonNegativeIntAttribute, setPositiveIntAttribute, setNon-NegativeLongAttribute, setPositiveLongAttribute, setNonNegativeShortAttribute, setURIAttribute.

String getPrivateTextAttribute (String name)

Retrieves a private attribute from the configuration object. Private attributes must be explicitly set by the application program and are never stored in a configuration file. As with public attributes, there are numerous versions of the methods for the different types of attributes.

setPrivateTextAttribute (String name, String value)

Sets a private attributes in the configuration object.

String UploadConfig (URL serverlocation)

This method uploads the public attributes in the configuration object as an XML document to an overlay server. The DOM document in the configuration object is transmitted as an XML document. The overlay server returns the overlay identifier in response to an upload operation, and the value is returned by the method. Private attributes are not uploaded.

How long does the program block to wait? What happens if server cannot be contacted?

document DownloadConfig (URL serverlocation, String OverlayIdentifier)

This method downloads a configuration file with public attributes from an overlay server and stores it as a DOM document. The parameters identify the location of the

overlay server and the overlay identifier of the configuration that is retrieved. If the overlay configuration cannot be retrieved a null object is returned. How long does the method wait?

void MergeConfig (document fromDOM)
: Merges a (downloaded) configuration file with public configuration attributes with a configuration object. The attributes from the configuration overwrite the attributes in the configuration object, with exception of the attributes in the local attribute list of the configuration object.

boolean doesOverlayExist(URL serverlocation, String OverlayIdentifier)
: This method tests if an overlay with the given overlay identifier exists at the specified overlay server. The method returns true if the overlay server has a configuration with the given overlay identifier, and returns false otherwise.

void createandsetOverlayId()
: Creates a new overlay identifier ands set the configuration to this identifier. The method is invoked when the local configuration file does not define an attribute for the overlay identifier. The overlay identifier that is created concatenates a local address (e.g., an IP address) and a local timestamp.

boolean validatePublicConfig (String XSDfilename)
: Validates the DOM document configuration object with the public attributes against an XML schema description. The name of the XML schema file is provided as a parameter. The method returns true if the validation is successful and false otherwise. This method is computationally intensive.

boolean validatePrivateConfig (String XSDfilename)
: Validates the private attributes in the configuration object with the XML schema file that is supplied as a parameter. The method returns true if the validation is successful and false otherwise.

document getPublicConfiguration()
: Returns the public configuration attributes as a DOM document.

document getPrivateConfiguration()
: Returns the public configuration attributes as a DOM document.

int getOverlayHash ()
: The methods retrieves the overlay hash from the configuration object.

void setOverlayHash ()
: The methods recomputes the overlay hash. The overlay is computed from the values of the attributes listed in the *HashAttributeList*.

The configuration object has several versions of the method createOverlaySocket that creates an overlay socket.

I_OverlaySocket createOverlaySocket (I_ReceiveCallback callback)

Creates an overlay sockets with a callback method to be executed when a message is received by the overlay socket. If the callback is set to null, then the overlay sockets writes messages for the application in the application buffer. The application obtains messages from this buffer by calling receive.

I_OverlaySocket createOverlaySocket (I_ReceiveCallback callback,
NotificationHandler nh)

Creates an overlay socket with a callback method for received messages, and a notification handler. With a notification handler, the application can indicate to block until an event occurs. The application can provide handler methods that are invoked when an event fires.

I_OverlaySocket createOverlaySocket (I_ReceiveCallback callback,
NotificationHandler nh, I_InterceptCallBack icb)

Creates an overlay sockets which, in addition to a callback method for received messages, and a notification handler, provides a callback for the intercept method described earlier in this chapter. By using null arguments, this method can emulate all of the above versions of createOverlaySocket.

3.6. SECURE OVERLAY SOCKETS

HyperCast overlay sockets can support authentication, integrity and confidentiality of data. The following discusses issues relevant to writing application programs with security features. The security features are described in full in a separate chapter.

Most importantly, there is no separate API to apply the security features of an overlay socket. Application programs that take advantage of security features can utilize the same API that is available to other overlay sockets. All security configurations are determined from configuration attributes. Configuration attributes that involve confidential information are designated as private. Recall that private attributes are not available through the configuration file, but must be defined by the application program.

Before we discuss security in HyperCast, we need to recall that overlay sockets exchange two types of messages. The overlay protocol which maintains the membership of an overlay socket in the overlay network exchanges protocol messages with other overlay sockets. Data from application program is transmitted in overlay messages.

An overlay socket can be configured with three different security levels: *plaintext*, *integrity*, and *confidentiality*. The security level is setting the attribute */Public/SecurityLevel* in the configuration file to either *plaintext*, *integrity* or *privacy*. Plaintext means that no security features are activated. Here protocol messages and overlay messages are transmitted in plaintext and the origin of a message is not authenticated. At the integrity level, overlay sockets authenticate neighbors in overlay network and check if a message has been subject to unauthorized modification. With integrity, all protocol messages and overlay messages are digitally signed with a message authentication code (MAC), or, in short, a hash. There are a total of three different types of hashes:

- The sender of a protocol message computes a MAC for the entire message, and adds it to the message. The MAC is verified by the receiver of the message. This ensures integrity for protocol messages.
- When an overlay socket sends or forwards an overlay message, it computes a MAC for the header of the message. The MAC is verified by the receiver of the

message. In this way, the integrity of the message header is verified at each hop on the route of the message. Ensuring integrity for message header protects, among others, against unauthorized changes of the route of a packet and the destination address of a message.

- The sender of an overlay message also computes a MAC for the payload of an overlay message. The MAC is computed at the source and verified at the destination(s) of the message. The MAC is not inspected or modified when a message is forwarded. This protects against unauthorized changes of application data while a message is transmitted in the overlay network.

At the privacy level, in addition to providing integrity, the payload of each overlay messages is encrypted. The encryption is done at the source of a message and the decryption occurs that the destination(s) of the message.

Authentication and Certificates

Authentication of overlay sockets is managed through X.509 digital certificates. When an overlay socket receives the protocol message from another overlay socket for the first time, it requests a certificate from this socket. The certificates must be signed by the CA whose certificate is specified in the configuration file.

Each application program is responsible for maintaining its own certificate, and the associated private and public keys. The certificate is managed through a *keystore*, an encrypted database of private keys and X.509 certificates authenticating the public keys.[1] Overlay sockets access private keys and certificates from the keystore. The configuration file declares the location of the certificates and the keystore. The attribute */Public/KeystoreLocation* stores the location of the keystore file. The attribute */Public/CertificateLocation* specifies the file that contains the X.509 certificate of the application program. (The certificate is obtained from the private and public key pair in the keystore file.) The attribute */Public/CACertificateLocation* specifies the file that contains the X.509 certificate of the Certificate Authority (CA) that granted the certificates. The default values of the files are *testcert.cer* for the certificate and the CA certificate, and *.keystore* for the keystore file.

Access to the keystore and the private key stored in the keystore is protected by passwords. This information is stored in private attributes. The password to access the keystore is kept in the attribute */Private/KeyStorePassword*. The private key is accessed with an alias and protected with a password. They are stored in the attributes */Private/PrivateKeyAlias* and */Private/PrivateKeyPassword*, respectively.

Each application program that uses the keystore must define the values of the private attributes. When the attributes are not defined, the overlay socket cannot be constructed. The private attributes can be set as follows:

```
setPrivateTextAttribute ("/Private/KeyStorePassword", "MyKeyStorePassword");
setPrivateTextAttribute ("/Private/PrivateKeyAlias", "MyPrivateKeyAlias");
setPrivateTextAttribute ("/Private/PrivateKeyPassword", "PrivateKeyPassword");
```

[1] HyperCast assumes that the keystore format uses the default type *JKS* by Sun Microsystems.

In most applications, the passwords must be entered by the user running the application. Entering a password may include a graphical user interface, the use of smart cards or biometric devices.

The keystore file is an encrypted database of private keys and certificates authenticating the corresponding public keys. The command **keytool**, included with JDK 1.4 or later, can create a keystore file and generate and manage keys for the keystore. As an alternative to the keytool command, keystores can be created and managed directly by the Java KeyStore API of the Java.

The keys generated by the *keytool* command are pairs of a public and a private key. Public keys are embedded in a X.509 certificate. The following describes some of the keytool commands to manipulate the keystore file. The following commands generates a key pair with a public key and a private key:

```
keytool –genkey –alias mykey –keypass mykeypasswd
```

The public key is wrapped in a X.509 certificate. By default, the command interactively requests information that are needed to create the certificate. The default location of the keystore is file *.keystore* in the home directory of the current user. The option *–keystore mykeystore* specifies a location that is different from the default.

The next command writes the certificate associated with the created key pair to a file.

```
keytool –export –alias mykey –file mycertfile.cer
```

The certificate stored in this file can now be submitted to and signed by a certificate authority. After the certificate has been signed or if a certificate has already been created and signed, it can be imported into the keystore with the command:

```
keytool –import –alias mykey –file mycertfile.cer
```

In some cases, it may be easier to give all overlay sockets the same self-signed certificate, that is, a certificate that has been signed by the user who created the certificate, and distribute the certificate to all application programs of the overlay network. A self-signed certificate is created with the following command:

```
keytool –selfcert –alias mykey
```

Finally, the command

```
keytool –list
```

lists the entire content of the keystore.

There are two methods for constructing and managing the keys that hash or encrypt information in an overlay socket. One method uses the novel concept of neighborhood keys and the second method is based on shared group keys. The method is determined by setting the attribute */Public/Security/KeyModeMethod*. to either *NeighborhoodKeys* or *GroupKeys*. There is an additional third method to realize security, which is based on SSL tunnels. This method is orthogonal to the first two methods, since it is entirely based on the configuration of adapters in the overlay socket. We next discuss each method in a separate section.

Neighborhood Keys

The neighborhood key method was developed for HyperCast overlay networks **[citeZaritsky]**, and presents a new method to ensure authentication, integrity, and

confidentiality in overlay networks. Here, each overlay socket authenticates every overlay socket that it communicates with by exchanging X.509 certificates. Each overlay socket maintains a symmetric key, called a *neighborhood key*, that it shares with its neighbors in the overlay network. Each time the neighborhood of an overlay socket changes, i.e., a new neighbor appears or an existing neighbor disappears, the overlay socket computes a new neighborhood key and sends the key to its current neighbors. Keys are computed with the algorithm specified by the attribute */Public/Security/CryptAlgorithm*. The neighborhood key is securely exchanged with the RSA algorithm using the public key of the neighbor.[2]

With the security level set to integrity, the overlay socket computes secure hashes for each outgoing message. Protocol messages and the header of overlay messages are hashed with the neighborhood key. To enforce integrity of the payload, the overlay socket generates a new key for each message, called the *message key*, and hashes the payload with the message key. The message key is encrypted with the neighborhood key and added to the header of the message. **[Double-check that this is indeed done. Neigbhors exchange certificates, but do not exchange keys.]** When security is enabled, in addition, the payload of each overlay message is encrypted with the message key.

When an overlay socket receives a hashed protocol messages from a neighbor, it uses the neighborhood key of that neighbor to verify integrity. When an overlay socket receives a hashed or encrypted overlay messages, it uses the neighborhood key of that neighbor to verify integrity of the overlay header and to decrypt the message key using the neighborhood key with which the payload was signed or encrypted. When the message needs to be passed to the application program, the decrypted message key is used to verify the integrity of the payload and decrypt the payload. When an overlay message must be forwarded to the next hop, the overlay socket computes a new hash for the header and re-encrypts the message key with its own neighborhood key.

Figure 9 illustrates the encryption of (the payload of) a message before it is transmitted by an overlay socket. The message is encrypted with a message key. The message key is encrypted with the neighborhood key of the overlay socket and added to the header of the message. The encrypted message key can be decrypted only by an overlay socket that holds the neighborhood key.

[2] HyperCast realizes the RSA implementation from *bouncycastle.org.*

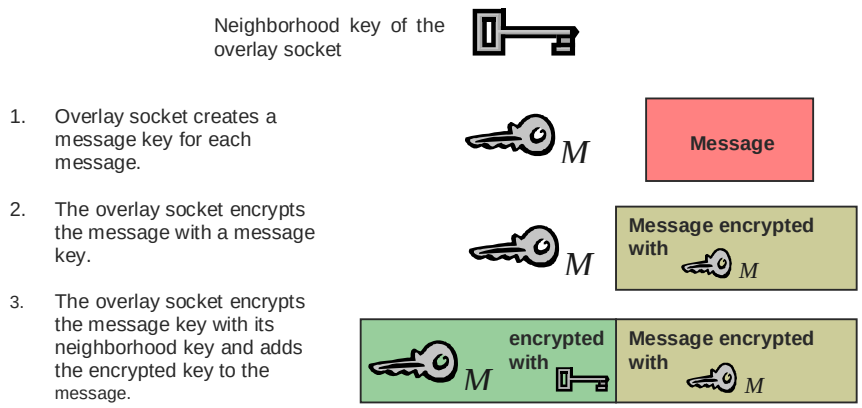

Figure 9. Transmission of a message with neigbhorhood keys.

Figure 10 illustrates the forwarding of an encrypted message. Again the figure only shows the payload of the message. When an encrypted message arrives, the overlay socket must have the neighborhood key of the overlay socket from which the message is received. The overlay socket first decrypts the message key in the message and then re-encrypts the message key with its own neighborhood key. Note that the payload of the message is not modified in the process.

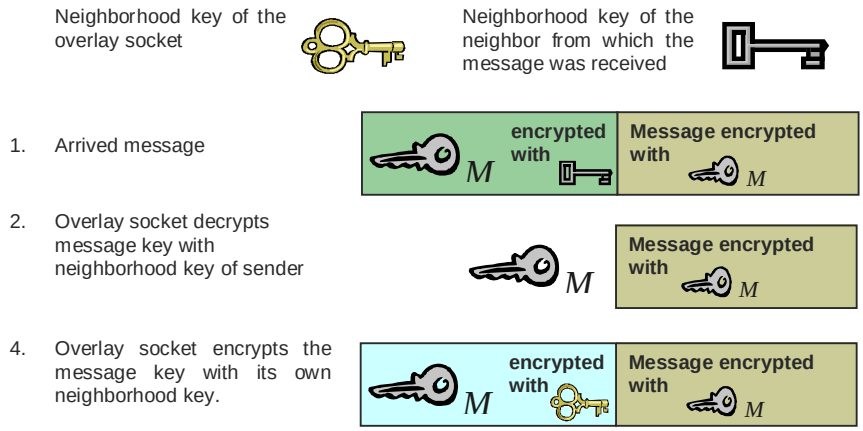

Figure 10. Forwarding a message with neigbhorhood keys.

The algorithms used for hashing and decryption are specified in the configuration file. The attribute */Public/Security/MacAlgorithm* specifies the hashing algorithm, */Public/Security/CryptAlgorithm* specifies the encryption method for encrypting the message key and the payload, and */Public/Security/SymmetricKeyLength* specifies the length of the encryption key. The algorithm specified as *CryptAlgorithm* is also applied when generating neighborhood keys. Since Java permits the inclusion of third party implementations of cryptographic algorithms, the Java implementation of HyperCast permits the incorporation of a large variety of algorithms. Tables 3 and 4 summarize the values of the attributes that will be supported in most Java installations.

Table 3. Values for the attributes *MacAlgorithm.*[3]

HmacMD5	The HMAC-MD5 keyed-hashing algorithm as defined in RFC 2104: "HMAC: Keyed-Hashing for Message Authentication".
HmacSHA1	The HMAC-SHA1 keyed-hashing algorithm as defined in RFC 2104: "HMAC: Keyed-Hashing for Message Authentication".

Table 4. Values for the attributes *MacAlgorithm CryptAlgorithm.*[4]

AES	Advanced Encryption Standard as specified by NIST in a draft FIPS. Based on the Rijndael algorithm by Joan Daemen and Vincent Rijmen. *SymmetricKeyLength* must be set to 128, 192, and 256 bits
Blowfish	The block cipher designed by Bruce Schneier. *SymmetricKeyLength* must be a multiple of `8`, and can only range from `32` to `448`.
DES	The Digital Encryption Standard as described in FIPS PUB 46-2. *SymmetricKeyLength* must be equal to `56`.
DESede	Triple DES Encryption (DES-EDE). *SymmetricKeyLength* must be equal to `112` or `168`

The neighborhood key method has a number of desirable properties. Since nodes exchange neighborhood keys only if they are neighbors in the overlay topology, and update their neighborhood key every time the neighborhood changes, only the current neighbors in the overlay topology can read encrypted information. The neighborhood scheme ensures forward secrecy, that is, a departing overlay socket cannot read messages that are transmitted after the socket has left, and backward secrecy, that is, a joining overlay socket cannot read messages that were transmitted before the overlay socket joined the overlay network.

When a signed or encrypted message is forwarded by an overlay socket, then the encrypted payload of the message and the hash of the payload need not be recomputed. This is important since decrypting and re-encrypting the complete payload each time a message is forwarded is not practical. Instead, the neighborhood key method merely decrypts and re-encrypts the field in the header that contains the message key.

When a new overlay socket joins or an existing overlay socket leaves, then only the neighbors of the coming or leaving socket update their neighborhood keys. This is different from many group key management schemes **[citeRFC??]** which require that all members of a group update a key each time the group membership changes.

Shared Group Keys

The symmetric key method assumes that there is a single symmetric key, called the session key, that is shared by all overlay sockets. The session key is stored in the private attribute */Private/GroupKey***.** The generation and distribution of new session keys is not

3 from: JavaTM Cryptography Extension (JCE) Reference Guide for the JavaTM 2 SDK, Standard Edition, v 1.4.

4 from: JavaTM Cryptography Extension (JCE) Reference Guide for the JavaTM 2 SDK, Standard Edition, v 1.4.

handled by the overlay socket, and must be implemented by the application programs. All encryption and signing of messages is done with the session key.

When the value of the shared key changes, the attribute storing the shared key must be updated. To ensure that changes to the attribute have an immediate impact in the use of the overlay socket, the methods that apply the group key look up the attribute whenever the group key is being used to encrypt or decrypt a message.

Overlay sockets with group keys authenticate each other with X.509 certificates as discussed previously. With integrity enabled, the overlay socket computes hashes for each protocol message, for the header of an overlay message and for the payload of an overlay message. All hashes are computed with the group key. When security is enabled, then, in addition, the payload of each overlay message is encrypted with the group key. The encryption and signing of the payload is done at the source of an overlay message. The integrity of the header of the overlay message is verified at each hop. In this way, any attempt to alter a header field can be detected.

The advantage of the group key method is that overlay sockets that forward an overlay message involve less processing than with the neighborhood key method. The main disadvantage of a shared key is that ensuring forward and backward secrecy requires generating a new group key each time a new overlay sockets joins an overlay network or an existing overlay socket leaves. Since Hypercast does not include procedures for generating and distributing new group keys, the group key must be managed by the application program or a separate protocol.

The available hashing and encryption algorithms are the same as in the neighborhood key method.

SSL Security

1. **SSL should read X509 certificate from keystore?**
2. **The implementation of SSL_UnicastAdapter has many attributes that are not in the configuration file:**

   .MaximumPacketLength

   .Timeout

   .MaxIdleTime
   .CipherSuite

   .keystore

With SSL security, overlay messages and protocol messages are transmitted over Secure Socket Layers (SSL) tunnels. An SSL tunnel is a secure communication channel that provides message privacy by encrypting all information exchanged using a session key, that is negotiated with the public key of the requestor of the SSL tunnel.

The configuration of SSL security is different from the configuration of the previously discussed security methods, and is done entirely by configuring adapters that transmit messages over SSL tunnels. SLL tunnels for protocol messages and overlay messages are configured independently. SSL security is established for overlay messages by selecting the socket adapter in the configuration file to be of type *SocketAdptSLL*, and for protocol messages by selecting the node adapter to be of type *NodeAdptSSL*. If an overlay socket transmits data over one of these adapters, it first establishes an SSL tunnel and then transmits the data over that tunnel.

SSL tunnels can be run with a variety of security protocol configurations, called cipher specifications **[citeRFC2249]**. SSL tunnels in HyperCast use the cipher specification TLS_RSA_WITH_AES_128_CBC_SHA. This specification designates that the protocol version is TLS, that the key exchange is done with the RSA protocol, that the hash algorithm to compute a message digest is computed with the SHA-1 algorithm, and that data is encrypted with the symmetric AES in cipher block chaining (CBC) mode with 128 bits **[citeAES]**.

SSL tunnels provide the strongest type of security available in HyperCast. The main drawback is that the establishment of a single SSL tunnel takes considerable time, generally in the order of 10 seconds or more. Since each overlay socket establishes an SSL tunnel to each neighbor to which it transmits data the delay to set up all necessary SSL connections can be considerable, especially when an overlay socket joins the overlay network. Another drawback of SSL security is that it assumes that the underlay network is an IPv4 network.

3.7. REFERENCES

[RFC2085] Oehler, M., and R. Glenn, "HMAC-MD5 IP Authentication with Replay Prevention", RFC 2085, February 1997.

[AES] Advanced Encryption Standard, FIPS-197. *National Institute of Standards and Technology*, November 2001.

