This is an unfinished 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 (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 2

Application Programming Interface – The Basics

This chapter discusses the application programming interface (API) and other topics related to building application programs for HyperCast overlay networks. We use the term *overlay network programming* to refer to the software development process of building application programs that communicate with one another in an application-layer overlay network. The API of the overlay socket tries to make overlay network programming easy by hiding issues concerned with maintaining the overlay network topology from the application programmer.

The main abstraction for overlay network programming in HyperCast is the *overlay socket*. An overlay socket is an endpoint of communication in an overlay network. The API of the overlay socket does not require that an application programmer knows the type of overlay network topology that is joined by the overlay socket. The API is also entirely independent of the underlay network.

Since an application program that uses overlay sockets does not need to be concerned with building and maintaining the topology of the overlay network, the API of the overlay socket can be made relatively simple. The main role of the API is to enable applications to create a new overlay network, to join and leave an existing overlay network, and to send and receive data to and from other applications in the overlay network.

The API of the overlay socket is message-oriented, similar to the datagram Berkeley sockets available in most operating systems **[citeCalvertJava]**. The service provided through this API is a datagram-style delivery service (unreliable, unordered, possible duplication) for application messages, which are referred to as *overlay messages*. Overlay messages can be sent via unicast, broadcast or flooding. An overlay socket can receive an overlay message through a blocking read operations or a through a non-blocking callback function.

The *HyperCastAppl* class is an alternative API that provides a simplified interface for applications that only require a single overlay socket. When an application programmer instantiates a HyperCastAppl application, the HyperCastAppl class automatically creates an overlay socket that immediately tries to join an overlay network. The application programmer only needs to issue operations to send and receive messages. The tradeoff for the simplicity of the HyperCastAppl application is a certain loss of flexibility.

Before an overlay socket is created, the application program configures the components of the overlay socket. Although the configuration of an overlay socket is not part of the API, an application programmer who wants to exploit the features of the overlay sockets must have some knowledge how overlay sockets are configured.

Each overlay socket has a *logical address* and a *physical address*. The logical address of an overlay socket is a unique address of the socket in the overlay network. The format and the length of the logical address are dependent on the overlay protocol that is executed by the overlay socket. In some overlay topologies, a logical address may be a bit string, in other overlays it may consist of multidimensional tuple. In addition to a logical address each overlay socket also has a physical address. The physical address of an overlay socket contains the information to derive the underlay network addresses of both the socket adapter and the node adapter. If the underlay network is the Internet, then the physical address generally consists of an IP address and the port numbers of the node adapter and the socket adapter. Application programs never use the physical address of an overlay socket. The relationship between a logical address and a physical address in HyperCast is analogous to the terminology for describing IP interfaces, where the IP address is seen as a logical entity and a MAC address as a physical address. Overlay networks introduce a new layer of addressing, that is, an IP address would be a physical address. However, it is also possible that an overlay socket does not use the Internet protocols and a MAC address becomes physical address.

The following section given an overview of the message-oriented API of the overlay socket, the HyperCastAppl class, and explains the role of configuration files. Additional details on the API and the configuration of overlay sockets are given in Chapter **[API-Advanced]**.

The overlay socket API presented here is specific to the Java programming language. In fact, the message-based API of the overlay socket is quite similar to Java's datagram sockets. However, since the specification of HyperCast is not language-specific, the overlay socket API can be realized in other programming languages. The effort to develop an overlay sockets API in another programming languages should be similar to that of writing a Berkeley socket API in that language.

2.1. THE HELLOWORLD PROGRAM

Before getting into the details of the message-based API, we present a small program, called Hello World, that illustrates the main features of the API. The program creates an overlay socket that joins an overlay network and broadcasts the string "Hello World". Then, the application waits for messages sent to the overlay network and displays each received message. We will use different versions of the Hello World program to introduce new features of the overlay socket API. Here is the Java code of the program:

```
import java.io.InterruptedIOException;
import hypercast.*;
public class HelloWorld {
  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();

        String MyLogicalAddress= getLogicalAddress().toString()
        System.out.println("Logical address is "+new  String(MyLogicalAddress)+".");

        try {
        Thread.sleep(3000);
        } catch (InterruptedException e) {}
        I_OverlayMessage msg = MySocket.createMessage(MyString.getBytes());
        MySocket.sendToAll (msg);
        for(;;) {
                msg = MySocket.receive();
                byte[] data = msg.getPayload();
                System.out.println("Received message is "+new  String(data)+".");
        }
  }
}
```

The HelloWorld program is not specific to a particular overlay topology. Also, the program does not make assumptions on a specific underlay network, that is, the program does not assume that the underlay network is the Internet, or a specific transport protocol (UDP or TCP).

All programs that use HyperCast overlay socket need to import the classes of the HyperCast package. This is done with the following import statements.

```
import hypercast.*;
```

After defining the string “Hello World”, the next two lines of the main program are concerned with the configuration of the overlay socket. By calling `HyperCastConfig.createOLConfig`, the program reads the configuration parameters of the overlay socket from the configuration file *hypercast.xml*. The configuration file is an XML document that specifies all parameters necessary to configure an overlay socket, including information which overlay network to join and how to join it. The information from the configuration file is stored in a *configuration object* of type *HyperCastConfig*. The configuration object creates the overlay socket with the static method *createOverlaySocket*.

After the overlay socket has been created, the overlay socket makes an attempt to join an overlay network by calling the *joinOverlay* method. The overlay network to be joined is specified in the configuration file. Now, the program waits for a few seconds to give the overlay socket time to get integrated into the overlay network. Later, we will see that it is also possible to obtain a notification from the overlay socket when the process of joining an overlay network is completed.

The next steps of the program are the creation and transmission of an overlay message. The data to be transmitted is passed as a byte array. The *sendToAll* method transmits the message to all applications of the overlay network. If the socket calls *sendToAll* before it is fully integrated in the overlay network, then the transmission may not reach all overlay sockets in the overlay network.

After transmitting the overlay message, the program enters an infinite loop, where it reads an overlay message, extracts the payload, and displays the payload. The receive method retrieves an overlay message from the application buffer inside the overlay socket. When this buffer is empty, the receive method blocks the program until a message arrives.

The delivery service provided by the overlay socket in the above example is a message-oriented best-effort service. There is no assurance that the transmitted overlay message is correctly delivered to all overlay sockets. The likelihood that a message is correctly delivered can be improved through the choice of the protocol in the underlay network. For example, if overlay socket run TCP as the protocol in the underlay network, the TCP protocol will recover all packet losses between neighbors in the overlay network. However, this is not sufficient to guarantee that an overlay message is delivered to all intended receivers, since changes in the overlay network topology due to overlay sockets joining and leaving can prevent messages from reaching their destinations. An improved delivery assurances can be achieved with the enhanced service API's, discussed in Chapter **[API-Advanced]**.

In the previous example, when the HelloWorld program performs a receive operation and no data is available for the application, the receive method blocks the program until a message has arrived. This is called a synchronous receive operation. As an alternative, an application program can also receive messages from an overlay socket in an asynchronous fashion, without ever blocking the application program. Here, the overlay socket executes a callback function whenever a new message arrives. The callback function is provided by the application program when the socket is created. Here is a version of the *HelloWorld* program that contains a callback function:

```
import hypercast.*;
```

```
public class HelloWorld implements I_ReceiveCallback {

  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");

          HyperCastConfig ConfObj =
          HyperCastConfig.createOLConfig("hypercast.xml");
          I_OverlaySocket  MySocket=ConfObj.createOverlaySocket(hw);
          MySocket.joinGroup();
          try {
                    Thread.sleep(3000);
          } catch (InterruptedException e) {}

          I_OverlayMessage msg =
                    MySocket.createMessage(MyString.getBytes());

          MySocket.sendToAll(msg);
          for(;;);
   }
}
```

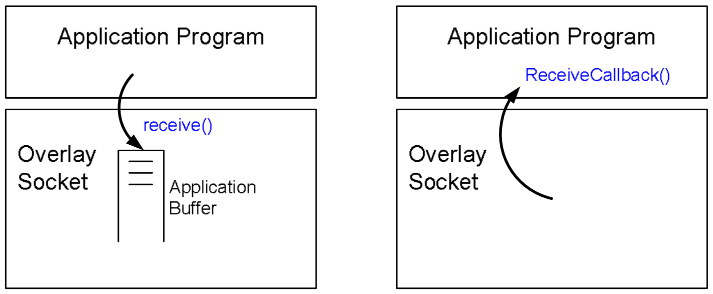


Figure 1. Receiving a message without and with callbacks..

Every class contains an overlay socket with asynchronous receive operations must implement the interface *I_ReceiveCallback*, which consists of the callback *ReceiveCallback*. The callback function is invoked by the overlay socket each time a new message for the application program arrives to the overlay socket. The operations performed by the *ReceiveCallback* method are identical to those in the original program.

The application program sets the *ReceiveCallback* method when the overlay socket is created, by passing a parameter:

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

The parameter hw is an object that implements the `I_ReceiveCallback` interface. In the first version of the HelloWorld program, the parameter of the *createOverlaysocket* was was a *null* object:

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

Here, when no callback method is made available, the overlay socket puts received messages into the application buffer, from where they are retrieved with the synchronous *receive* method.

The difference between the two versions for receiving a message is illustrated in Figure 1. In the synchronous version, the overlay socket writes messages in the application buffer and the application program retrieves messages from this buffer. When the buffer is empty the application program is blocked until a message arrives. In the asynchronous version, the overlay socket invokes the ReceiveCallback method independently of the threads of the application program. The choice between synchronous and asynchronous receive operations is found in many network programming environments. Callbacks are more convenient since the receive operation is non-blocking. Problems with callbacks arise when callback functions have a long execution time or may get blocked on an event. In HyperCast, callbacks are executed by a thread of the overlay socket. Keeping this thread busy with executing application code, will result in a backlog of unfinished work inside the overlay socket. If the application code for processing incoming messages is long are may block, the application programmers should create a separate thread for processing message arrivals and have this thread use the synchronous version for receiving messages.

Next we give a more complete overview of the methods available in the message-based API for creating an overlay socket, for joining and leaving an overlay network, for receiving overlay messages. Other parts of the API, including methods to send data, methods that provide information about the socket, and methods to manipulate a message are defined in later sections.

The API discussed here consists of methods from two classes. The HyperCastConfig configures and creates an overlay socket. The methods in the I_OverlaySocket interface, which is implemented by all overlay sockets, define how an application program interacts with an overlay socket. This includes methods to join an overlay network, to send and receive messages, and to create new messages.

Creating an Overlay Socket

An overlay socket is created with an object of type *HyperCastConfig,* which stores configuration information for an overlay socket. The configuration information is obtained from a configuration file. In Chapter **[API-Advanced]** we show how to configure an overlay socket from a remote server. The following two methods of the *HyperCastConfig* class will be sufficient for most purposes.

`static HyperCastConfig createOLConfig (String s)`

Creates a configuration object from an XML document. The name of the XML document is provided as a parameter.

`I_OverlaySocket  createOverlaySocket(I_ReceiveCallBack cb);`

Creates a new overlay socket. The argument supplies the callback function that is executed when a message is received. If the callback is set to *null*, then the overlay socket writes messages into the application buffer, and the application program retrieves messages from this buffer with the *receive* method.

Participation in an Overlay Network

The participation of an overlay socket in an overlay network is managed by the following methods of the *I_OverlaySocket* interface.

void joinOverlay()

This method starts the process of joining an overlay network. The overlay network to be joined is specified when the overlay socket is created. When the method returns, the process of joining the overlay network may not be complete. If an overlay socket wants to be notified when the process of joining an overlay network is completed, it can use the event notification system of HyperCast (see Chapter **[API-Advanced])**.

void leaveOverlay()

With this method an overlay socket departs from an overlay network. After leaving an overlay network, the overlay socket cannot send or receive messages. An overlay socket that has left an overlay network can re-join the network by issuing a *joinOverlay*.

Receiving Messages

As discussed, an application program can receive overlay messages by performing an asynchronous callback method or a blocking read operation. The specification of the two methods are as follows.

void ReceiveCallback (I_OverlayMessage msg)

This is the callback method for processing an incoming message. The ReceiveCallback method is the only method defined in the *I_ReceiveCallback* interface. The callback method is supplied to the overlay socket when the overlay socket is created. The overlay socket invokes the callback each time it receives an overlay message for the application program.

I_OverlayMessage receive()

Receives an overlay message from the overlay socket. The message is retrieved from the application buffer in the overlay socket. When the application buffer is empty, the application blocks until a message arrives in the buffer, or until a timeout occurs.

void setSoTimeout(int timeout)

This method limits the amount of time that an overlay socket stays blocked on an empty application buffer. The parameter defines the maximum waiting, measured in milliseconds.

int getSoTimeout()

This method returns the value of the maximum time that an overlay socket blocks on an empty application buffer. A negative value indicates that the option is disabled, i.e., there is no limit on the blocking time.[Check: According to JW 0 indicates that the there is no limit on blocking time. This seems inconsistent. “0” should be no waiting time at all.]

Exceptions in HyperCast

HyperCast defines a few runtime exceptions that can be caught by the application program. In Section **[HyperCastAppl]** we have seen that the *HyperCastAppl* class can intercept and possibly ignore the exceptions of HyperCast. The defined exceptions are as follows:

- *HyperCastFatalRuntimeException:* The overlay socket cannot recover from this exception. This type of exception is generally result of an error in the HyperCast software or a software design problem.
- *HyperCastWarningRuntimeException:* This exception is raised when an invalid operation is performed on an overlay socket or message, or when an invalid messge is received. Overlay sockets can recover from this exception.
- *HyperCastConfigException:* This exception is raised during the configuration phase of an overlay socket. When the configuration exception is thrown, the overlay socket cannot be configured due to incorrect or missing configuration parameters.
- *HyperCastStatsException:* This exception may be thrown as part of the monitor and control system in HyperCast. The monitor and control system, discussed in Chapter **[MonControl],** can occur when a remote monitor program tries to retrieve an invalid state variable, called *statistic,* or tries to set a statistic to a value that is not valid.

2.2. SENDING OVERLAY MESSAGES

An overlay socket can send an overlay message to a single destination overlay socket (unicast), to all overlay sockets in the overlay network (broadcast), or to all or a subset of neighbors of the overlay socket in the overlay topology.

When sending a unicast messages, an overlay socket must specify the logical address of the destination. Although the length and the format of a logical address is dependent on the overlay protocol executed in the overlay socket, an application can reference any logical address by the type I_LogicalAddress. We now give a few examples to illustrate how an application program works with logical addressees. If an application program has created an overlay socket, it can display the logical address of the overlay socket by executing

```
        I_LogicalAddress MyLaddr=MySocket.getLogicalAddress();
        System.out.println("Logical address is " + MyLaddr.toString() + ".");
```

The method getLogicalAddress() retrieves the logical address from the overlay socket. In many cases, the logical address is determined only when the overlay socket joins an overlay network. In other overlay protocols, the logical address of an overlay socket may change during the lifetime of a socket.

In the following implementation of the ReceiveCallback method, the application program extracts the source logical address from each received overlay message and returns the same message to the source as a unicast message.

```
 public void ReceiveCallback (I_OverlayMessage msg) {
        I_LogicalAddress SrcLaddr = msg.getSourceLogicalAddress();
        System.out.println("Message received from logical address"
                        + new  SrcLaddr.toString() + ".");
        MySocket.sendTo(msg, SrcLaddr);
```

```
  }
```

Finally, we give an example where an application program generates a logical address from a text string. In this case, the application program must necessarily know the format of the logical address. Application programmers can take advantage of the method *createLogicalAddress* to build a valid logical address from a text. This method, which is available for all logical address formats, expects that the string has a structure that can be transformed into a logical address. The application programmer must be familiar with this structure. Let us suppose that an application program in a Delaunay triangulations overlay network wants to send a unicast message to the overlay socket with logical address (1288, 1883). The

```
I_LogicalAddress dest = createLogicalAddress("1288,1883");
sendTo (msg, dest);
```

Before discussing the available methods for sending a message, let us review how overlay messages are forwarded in the overlay network. In HyperCast, all overlay messages are transmitted along rooted trees that are embedded in the topology of the overlay network. A broadcast message is forwarded downstream in a tree that has the source of the message at the root of the tree. A unicast message is forwarded upstream in a tree with the destination of the message at the root of the tree. These embedded trees are not explicitly constructed. Rather, each overlay socket locally computes the upstream neighbor (parent) or the downstream neighbors (children) in the embedded tree with a given logical address as the root.

The embedding of trees in an overlay network is illustrated in Figure 2. Figure 2(a) shows the topology of an overlay network with six overlay sockets, labeled A through F. Let us suppose that the labels are the logical addresses. Figure 2(b) shows the embedded tree for this network when A is the root of the tree. In this tree, B and C are the children of A, E and F are the children of E, and C is the child of B. Likewise, B is the parent of B with respect to the embedded tree with A as root.

In most overlay networks, a different root generally results a different embedded tree. Figure 2(c) shows an embedded tree with D as the root of the tree. We will refer to Figure 2 when we discuss the different send operations available for overlay sockets.

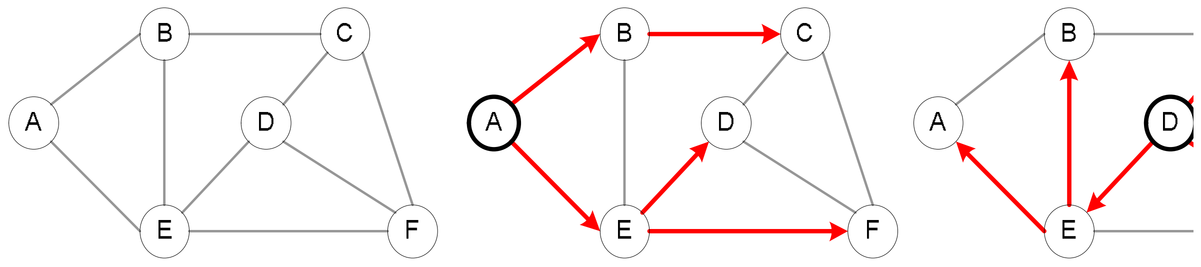

A
B
C
D
E
F
(a) Overlay network topology.
(b) Embedded tree with A as root.
(c) Embedded tree

Figure 2. Overlay network topology with embedded trees.

When A performs the broadcast operation *sendToAll*, the message is forwarded as shown in Figure 2(a). A sends the message its downstream neighbors B and E. B, in turn, forwards the message to C, and E forwards the message to nodes D and F. When D is the sender, the message is forwarded as shown in Figure 2(c).

When C sends a unicast message to A with *sendTo(Message, A)*, the message is passed to node B, since B is the parent of C with respect to root A. B, in turn, forwards the message to A. Once the message arrives at A, the message is passed to the application program.

HyperCast has a second broadcast operation with name *sendFlood*. When an application program sends a message with *sendFlood*, the overlay socket forwards the message to all its neighbors in the overlay topology. All neighbors that receive the message forward the message to all of their neighbors, with exception of the neighbor from which the message was received, and each application passes the message to the application program. Nodes forward data until the *HopLimit* of the message expires and the message is dropped. This type of forwarding is commonly referred to *flooding*. Flooding is a delivery method that can quickly reach a large number of nodes. The drawback of flooding is that each application receives a large number of duplicates of the same message. The resource usage of a flooding operation can be forbidding and can easily cause congestion. However, the *sendFlood* operation is useful when an application wants to send a message to all overlay sockets that are close in the overlay topology, e.g., less than three hops away. This is done by setting the hop limit of overlay messages as discussed below. For example, by setting the hop limit to two, a flood message will reach all applications in the overlay network that are no more than two hops away. In Figure 2, when A floods a message, it forwards the message to B and E. Node B forwards the message to its neighbors C and E. Already, both B and E have received two copies of the message. Then, B and E will forward the duplicates to their neighbors. Note that flooding messages leads to an explosion of duplicates of the same message. The creation of duplicates stops when the hop limit field of the transmitted message is decremented to zero.

In addition to unicast and broadcast, there are methods that send a message to an immediate neighbor in the overlay network. These methods are *sendToAllNeighbors*, *sendToParent*, and *sendToChildren*. In Figure 2, when A sends a message with *sendToAllNeighbors*, the message is forwarded to B and E. B and E pass the message to the application, but do not forward the message any further. When F issues a *sendtoParent(Message, createLogicalAddress('A'))*, the message is forwarded to node E. Upon receiving the message, E passes the message to the application. When E sends a message with *sendtoParent* and root set to A, the message is forwarded to A. When E sends a message with sendToChildren*(Message, createLogicalAddress('A')),* the message is forwarded to D and F. When D and F receive the message, they invoke the receive operations of the application programs, but do not forward the message. When D sends invokes *sendToChildren(Message, A),* the message is silently discarded, since D has no children in the embedded tree with A as root.

We next provide the specifications of the methods for sending overlay messages.

`I_OverlayMessage createMessage(byte [] payload)`

This method creates an overlay message and assigns the payload. *I_OverlayMessage* is an interface that refers to all types of overlay messages.

`void sendTo (I_OverlayMessage msg, I_LogicalAddress dest)`

Sends a unicast message to the overlay socket with the specified logical address. When an application invokes this method, the overlay socket forwards the message to its parent in the embedded tree that is rooted at the destination address. When the parent receives the message, it forwards the message to its own parent, and so forth. When the message arrives at the destination, the message is passed to the application.

void sendToAll (I_OverlayMessage msg)

Broadcasts an overlay message to all applications in the overlay network, including the application that generates the message. The message is forwarded along a rooted tree that is embedded in the overlay network, where the source of the message is the root. Each socket that receives a broadcast message passes the message to the application, either by writing the message into the application buffer or by issuing a *ReceiveCallback.*

void sendFlood (I_OverlayMessage msg)

Floods a message in the overlay network. The mechanics of flooding, and its pitfalls, are explained above. A message is forwarded until the HopLimit of the message expires and the message is dropped. Application programs should use this method with care, e.g., by setting the HopLimit field to a small value.

void setHopLimit (int value)

Sets the default for the hop limit for outgoing messages. The hop limit is the maximum number of overlay sockets traversed before a message is dropped. When an overlay socket sends an overlay message, the hop limit is written in the header of the message. The value of the hop limit field in the header is decremented by one at each overlay socket that receives the message. When the hop limit field reaches zero, the message is dropped. The default value for the hop limit is determined from an attribute in the configuration file, and is modified with setHopLimit. The method I_OverlayMessage interface has a method that permits to change the HopLimit field for individual messages without changing the default value. This is done with. The hop limit is set to a value between 1 and 65535.

short getHopLimit ()

Returns the current default limit for overlay messages sent out by this overlay socket.

void sendToAllNeighbors (I_OverlayMessage msg)

Sends a message to all neighbors of the overlay socket in the overlay network topology. A neighbor that receives the message passes the message to the application program, but does not forward the message to other overlay sockets.

void sendToParent (I_OverlayMessage msg, I_LogicalAddress root)

Sends a message to the upstream neighbor (parent) in the embedded tree that has the given logical address as the root. When the parent receives the message it passes the message to the application. The parent does not forward the message. If the root address is the address of the sending socket, the message is discarded.

void sendToChildren (I_OverlayMessage msg, I_LogicalAddress root)

Sends a message to the downstream neighbors (children) in the embedded tree that has the given logical address as the root. If a node does not have children, the message is discarded. Upon receiving the message the children pass the message to the application, but do not forward the message to their neighbors.

Table 1. Representation of logical addresses as a String.

Overlay protocols	Logical address	
	Internal representation	Representation as a text string
HC protocol	One integer, representing a binary bit string	One number, that represents the index of the Gray code encoding of the corresponding bit string. If the integer represents a number *N*, the logical address generated by *N* is *N ^ (N >>> 1).* Examples: "0", "999"
DT protocol	Two integers, representing a point in a plane.	Two nonnegative numbers separated by a comma. Examples: "10,397", "0,0"
SPT protocol	One integer, representing a number.	One nonnegative number. Examples: "35984", "12", "0"
None protocol	????	??
Pastry	???	???

The following methods of the *I_Overlaysocket* interface operate on logical addresses and provide information about logical addresses of neighbors of the overlay socket to the application program.

I_LogicalAddress createLogicalAddress (String laStr)

Creates a properly formatted logical address from a text string. The format of the text string for a logical address is specific to the overlay network topology. Table 1 summarizes the formats of the string that is expected by different overlay protocols for the creation of a logical address.

String toString (I_LogicalAddress ladr)

Converts a logical address into the string representation. This method is used when logical addresses need to be represented as a string.
(Should it be required that createLogicalAddress (toString (ladr)) and toString.createLogicalAddress ("ladjfla") return the original argument?)

I_LogicalAddress getLogicalAddress()

Returns the logical address of the local overlay socket.

setLogicalAddress(I_LogicalAddress ladr)

Sets the logical address of the overlay socket to the given value. In some overlay topologies, the logical address of a socket is determined by the overlay protocol. In these cases, modifying the logical address by the application program may interfere with the operation of the overlay protocol, and influence the stability of the overlay topology stability.

`I_LogicalAddress getParent(I_LogicalAddress root)`

Returns the logical address of the parent, i.e., the immediately upstream neighbor, in the embedded tree that has the provided address as root.

`I_LogicalAddress[] getChildren(I_LogicalAddress root)`

Returns the logical addresses of the children of the overlay socket, i.e., the immediately downstream neighbors, in the embedded tree that has the provided address as root.

`I_LogicalAddress[] getNeighbors()`

Returns the logical addresses of all neighbors of this overlay socket in the overlay network topology.

2.3. MANIPULATING OVERLAY MESSAGES

In the message-based API, the transmission of a message by an overlay socket occurs in three steps. First the overlay socket creates an overlay message. In the second step, it determines the content of the message. In the last step, the message is transmitted. This section discusses the available operations for setting and reading the contents of a message.

An overlay message consists of a header and payload. The header contains control information needed to deliver the message to the desired destination(s). The payload contains the application data. Most fields in the header are entered by the overlay socket, without requiring action by the application program. Other fields, however, must be set by the application program.

When an application receives an overlay message it obtains the entire message, that is, both header and payload. Usually, the application program is only interested in extracting the payload of the message. For example, in the HelloWorld program, the payload was extracted as follows:

```
byte[] data = msg.getPayload ();
```

In addition to the payload, an application program can access all fields of the header of an overlay message. The same is true for messages that are transmitted. When an overlay socket creates a message, e.g.., with

```
I_OverlayMessage msg =  MySocket.createMessage (byte[] payload);
```

the overlay socket creates a complete overlay message with header and payload. Before (or even after) transmitting the message, the application program can manipulate the content of the header fields. All methods to access or modify the header or payload of an overlay message are available through the I_OverlayMessage interface. When a message is transmitted, e.g., using the SendTo or SendToAll methods, the overlay socket may overwrite header fields that have been set by the application program. The last point is illustrated in the following program fragment:

```
I_OverlayMessage msg =  MySocket.createMessage(MyString.getBytes());
byte dmode = msg.getDeliveryMode();
```

```
        HelloWorld.printmode(msg.getDeliveryMode());
        MySocket.sendToAll(msg);
        HelloWorld.printmode(msg.getDeliveryMode());
```

The program invokes a method with *printmode*, that is defined as follows:

```
public static void printmode (byte mode) {
        switch (mode) {
        case 1:
                System.out.println("Delivery mode is broadcast.");
                break;
        case 3:
                System.out.println("Delivery mode is unicast.");
                break;
        default:
                System.out.println("Neither unicast nor broadcast.");
        }

}
```

The *printmode* method analyzes and displays the delivery mode of an overlay message. The *printmode* method is called once after an overlay message has been created and once again after the message is transmitted with *sendToAll*. In both cases, the method is invoked with parameter *msg.getDeliveryMode*, which retrieves the value of the delivery mode field from an overlay message header. The delivery field mode of an overlay message is one byte long and is set to broadcast (0x01), flood (x02), or unicast (0x03). When *printmode* is invoked for the first time, the message has been created, but the delivery mode of the overlay message has not been set. Therefore, the output of the first invocation is:

```
Neither unicast nor broadcast.
```

When the message has been transmitted as a broadcast message with the sendToAll method, the overlay socket sets the delivery mode of the message to broadcast. Consequently, the second call to *printmode* results in the output:

```
Delivery mode is broadcast.
```

In the next example an application program retrieves information from the header of a received message:

```
msg = MySocket.receive();
String SenderSrc = msg.getSourceLogicalAddress().toString();
byte[] data = msg.getPayload ();
System.out.println("Received <"+ new  String(data) + ">from logical address: " +
SenderSrc +" . ");
```

This program rads a message from the application buffer and extracts the payload from the message. The method *msg.getSourceLogicalAddress* retrieves the logical address of the source of the message. The *I_OverlayMessage* interface provides methods to read or modify most of the fields in the header of an overlay message. A list of available methods is given in Table 2. Some methods require a good understanding of the message format of an overlay message.

In addition to retrieving information from the header of an overlay message, there are corresponding methods for modifying the contents of a message. These methods are listed in Table 3, with the values permitted in the field. When a message is transmitted the overlay socket overwrites some of the fields in the header. If an application programs modifies header fields of an overlay message before transmitting the message, the overlay socket may overwrite the values.

Table 2. Methods from I_OverlayMessage for accessing fields of an overlay message.

Method	Description of returned value
byte getVersion()	Version of the HyperCast software.
byte getDeliveryMode()	Delivery mode of the message.
short getHopLimit()	Current value of the hop limit field.
byte getTrafficClass()	Traffic class field is used for service differentiation of traffic classes.
short getFlowLabel()	Identifies a flow and is intended for service differentiation.
byte[] getPayload()	Payload of the message.
I_LogicalAddress getSourceAddress()	Logical address of the source of the message.
I_LogicalAddress getDestinationAddress()	Logical address of destination of the message.
I_LogicalAddress getPreviousHopAddress()	Logical address of the neighbor from which the messages was received.
Vector getExtension ()	Returns all header extensions of the overlay message.

Table 2. Methods from I_OverlayMessage to modify fields of an overlay message

Method	Permitted values:	Actions taken by the overlay socket:
setVersion(byte)	0 – 15	Sets value to 3.
setDeliveryMode (byte)	0x1,0x2,0x3	Sets the delivery mode. The overlay socket sets and possibly overwrites the delivery mode when the message is transmitted.
setHopLimit(short)	1– 65535	Sets the value according to the hop limit defined for the overlay socket.
setTrafficClass(byte)	0 – 255	HyperCast 3.0 ignores this value.
setFlowLabel(int)	0 – 65535	HyperCast 3.0 ignores this value.
setPayload(byte[])	Byte array	The overlay socket does not inspect or modify the payload.

setSourceAddress (I_LogicalAddress)	Valid logical address	The value is reset by the source overlay socket.
setPreviousHophopAddress (I_LogicalAddress)	Valid logical address	The value is reset by the overlay socket.
setDestinationAddress (I_LogicalAddress)	Valid logical address	The value is reset by the overlay socket for unicast messages. Otherwise, the value is ignored.

2.4. THE HYPERCAST APPLICATION

The *HyperCastAppl* class offers a very simple API for programming with overlay sockets. The constructor of the class creates an overlay socket and automatically starts the process of joining an overlay network. An application program that extends the *HyperCastAppl* class The only needs to issue commands to send and receive messages. Application programs that want to use this API need to extend the *HyperCastAppl* class. This simplified interface hides some of the interactions of an application program with an overlay sockets at the cost of reduced flexibility. For example, *HyperCastAppl* only creates a single overlay socket and is therefore not suitable for application programs that require more than one overlay sockets. Also, the overlay socket created for *HyperCastAppl* objects must use a callback method for processing incoming messages. Having said this, the *HyperCastAppl* class may be sufficient for most applications The *HyperCastAppl* supports all available configuration options and all extensions to the API, such as security, enhanced services, and others.

The HelloWorld program written as a *HyperCastAppl* can look like this:

```
import hypercast.*;

public class HelloWorldApp extends HyperCastAppl  {
         public HelloWorldApp (String cfile) {
        super(cfile);
  }

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

   public static void main ()  {
        String MyString = new String("Hello World");
        HelloWorldApp hw =  new HelloWorldApp ("hcast.xml");
        hw.WaitUntilNODE_HASBECOMESTABLE ();
        while (true) {
                I_OverlayMessage msg = hw.createMessage(MyString.getBytes());
                hw.sendToAll (msg);
                try {
                        Thread.sleep (2000);
                }
                catch (InterruptedException e) { }
        }
   }
}
```

The simplicity of the program is apparent. The constructor of the class configures and creates the overlay socket from the configuration file that is passed as a parameter and starts the process of joining the overlay network. The method *WaitUntilNODE_HASBECOMESTABLE* pauses the application program until the overlay protocol has reached a stable state. The remainder of the program is similar to the previous versions of the HelloWorld program. The methods available to the application program include all methods available in the message-based API, as specified by the *I_OverlaySocket* interface. Particularly, the application can leave and rejoin the overlay network.

The HyperCast application includes a notification handler which permits application programs to wait for events or to execute a callback function when an event occurs. We refer to Chapter **[API-Advanced]** for a discussion of the event notification system in HyperCast. There is a pre-defined set of events. For each event that is defined in HyperCast, an application can supply a callback function or it block until the event occurs. In the above example, the method *WaitUntilNODE_HASBECOMESTABLE* blocks the application until the event *NODE_HASBECOMESTABLE* occurs. This method is defined for many but not for all overlay protocols.

The HelloWorld application can control how HyperCast runtime exceptions are handled. This is done with the following methods:

`void setFatalAction (int action)`
`void setWarningAction (int action)`

Sets the action to be taken when one of the exceptions *HyperCastFatalRunTimeException* or *HyperCastWarningRunTimeException* is raised. The actions can be set to one of the following four actions: the exception is logged (*EXCEPTION_LOG*), the exception is thrown (*EXCEPTION_ THROW*), the exception is logged and thrown (*EXCEPTION_LOG_AND_ THROW*), or the exception is ignored (*EXCEPTION_IGNORE*). By default, the *HyperCastAppl* class ignores exceptions.

`int getFatalAction()`
`int getWarningAction()`

Queries the setting of the actions taken on an exception of type *HyperCastFatalRunTimeException* or *HyperCastWarningRunTimeException*. The returned value is one of *EXCEPTION_LOG, EXCEPTION_THROW, EXCEPTION_LOG_AND_ THROW.*

2.5. CONFIGURING OVERLAY SOCKETS

Each overlay socket is associated with a set of configuration parameters, called attributes, that specify the properties of the overlay socket. The attributes include the type of overlay protocol, the underlay network addresses, security options, and many other parameters. Every overlay socket that participates in an overlay network maintains a configuration file that contains the attributes as an XML document. When an overlay socket is created the contents of the configuration file provides the information to configure and initialize the overlay socket. Examples of configuration files are available from [**Hyperlink**]. In the *HelloWorld* programs in this chapter, overlay sockets were configured and created with the following two operations:

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

The first line declares the file *hypercast.xml* as the configuration file of the overlay socket. The configuration information is stored in a configuration object of type *HyperCastConfig*. The configuration file is not used for any purpose other than the creation of a configuration object. If the configuration file is properly configured, the overlay socket created with this configuration object can participate in an overlay network or create a new overlay network.

In the following we discuss how to use a configuration file, and how to edit the most important attributes. It is important to realize that the configuration file is managed by the user of an application program, and not by the application programmer. Changes to a configuration file generally do not require changing an application program. If an application program wants to join an existing overlay network, it must obtain a suitable configuration file. This configuration file is either well-known, or it is made available through out-of-band communications, e.g., by email or a web server. In Chapter **[API-Advanced]** we will see that a configuration file can specify the location of a server from which an application program can download configuration information

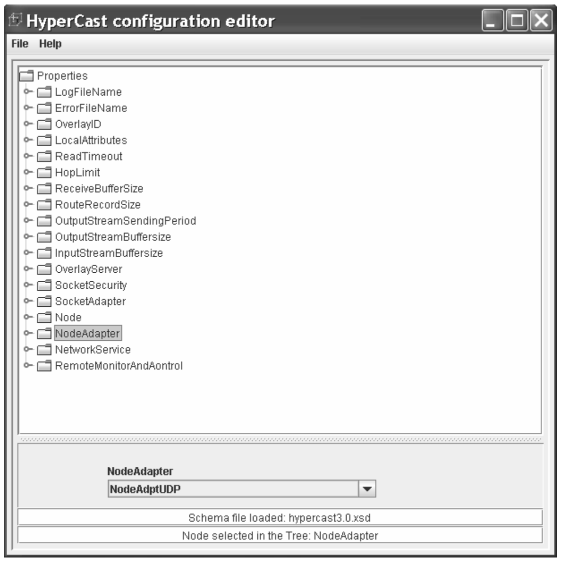
The configuration of overlay sockets that join the same overlay network must be compatible in the sense that the configured overlay sockets must be able to exchange overlay protocol messages for maintaining an overlay network topology and be able to exchange application data. In practice this means that configuration files of overlay sockets in an overlay network are very similar or even identical. Some configuration parameters, however, must be locally assigned. For example, the port numbers that may necessary to configure the underlay protocol may need to be locally configured. The set of attributes that must be locally configured are listed in an attribute with name *LocalAttributes*. If a user has obtained a configuration file, the user must modify the attributes specified in the local attribute list. For example, suppose the attribute is

```
<LocalAttributes>
 /Attributes/Node/DT/DTBuddyList/BuddyList, /Attributes/NodeAdapter/ PhyAddr
</LocalAttributes >
```

Here, the attributes that describe the *BuddyList* of the DT protocol, and the physical address in the node adapter are determined from the configuration file.

The easiest way to create a new configuration file is to start with an existing file and modify attributes. HyperCast provides a lot of support for writing usable configuration files. One level of support is provided by the XML schema that distributed with each version of the HyperCast software.[1] The XML schema file describes the format of all configuration parameters and specifies default values for all attributes. An XML configuration file document that is valid with respect to the schema file is likely to result in a usable configuration file.

Another level of support for writing configuration files is a standalone configuration tool, the *HyperCast configuration editor*, for creating and editing configuration files. The user interface of the configuration editor is shown in Figure 3. The configuration editor takes the XML schema file included in the HyperCast JAR file and enforces that all entered configuration parameters are valid with respect to that file. In addition, the tool enforces dependencies between attributes not checked by the XML schema **(Check!!If so, what is checked?)**. With the configuration editor, a user can open and edit existing configuration files, and can create a new template configuration file from the default values in the XML schema file.



NodeAdapter
NodeAdptUDP

Schema file loaded: hypercast3.0.xsd

Node selected in the Tree: NodeAdapter

Figure 3. HyperCast configuration editor.

A user who creates a new configuration for overlay sockets must select (1) the name of the overlay network that is joined by the overlay socket, and (2) the type of overlay protocol, (3) the protocols and addresses for the underlay network. Next we will discuss how to make these selections. There are many other configuration options, such as security settings, that are not explained in this chapter.

Selecting a name for an overlay network. The *overlay identifier* is the attribute that identifies an overlay network. All overlay sockets that join the same overlay network must have the same identical overlay identifier. A new overlay network is created by associating a new overlay identifier with a configuration file. An overlay identifier is a simple text string, and HyperCast does not assume or enforce a particular format. The identifier should be selected so that it is unique in the scope in which it is used. A simple method to create a new overlay network is to take an existing configuration file, change the overlay identifie, and create an overlay socket from the modified configuration file. To create an overlay with identifier "MyNewOverlayNetwork", the configuration file needs to have an entry that reads:

[1] The schema file is an *.xsd file that is included in the JAR archive of the HyperCast software.

```
<OverlayID>MyNewOverlayNetwork</OverlayID>
```

OverlayID is the tag of the overlay identifier. If an overlay configuration file does not contain an overlay identifier, then the configuration object creates an identifier, by concatenating a local address (e.g., an IP address) and a timestamp.

Selecting the overlay protocol. The overlay protocol maintains the participation of the overlay socket in the overlay network. The selection of the overlay protocol determines the type of overlay topology. The overlay protocol is selected by setting the attribute *Node* in the configuration file. For example, the spanning tree (SPT) protocol is selected by setting

```
<Node>SPT</Node>
```

The list of all available protocols is summarized in Table 2. Hyperast Version 3.0 has overlay protocols that construct a Delaunay triangulation overlay topology (DTserver, DTbuddylist, DTbroadcast), a hypercube topology (HC), and a spanning tree topology (SPT). The protocol for Pastry distributed hash table **[CitePastry]** which uses the code of the *FreePastry* implementation from **[CiteFreePastry]** is available as an extension, The protocol with name None permits the establishment of point-to-point connections with the HyperCast software.

Each overlay protocols must provide mechanisms by which new overlay sockets that are not connected to the overlay network can establish communications to overlay sockets already in the overlay. These mechanisms, referred to as *rendezvous methods,* are needed when new overlay sockets join an overlay, and when an overlay socket becomes disconnected from the overlay network. One can think of three rendezvous methods in an overlay network:

1. *Broadcast:* New or disconnected overlay sockets use a broadcast operation to announce themselves to members of the overlay network. This rendezvous method requires that the underlay network has an operation, where a single transmit operation can reach all overlay sockets in the overlay network. If the underlay network is the Internet, the availability of IP multicast permits a broadcast rendezvous.
2. *Buddy List:* Each overlay socket maintains a so-called buddy list, which consists of overlay sockets that are likely to be in the overlay network. New or disconnected overlay sockets contact overlay sockets in the buddy list to get (re-) connected to the overlay network.
3. *Rendezvous Server:* All overlay sockets use a well-known server to establish communication between members and non-members of an overlay network.

Table 3. Overlay protocols in HyperCast 3.0.

Attribute	**Overlay Topology**	**Rendezvous methods**
DTserver	Delaunay Triangulation	Rendezvous server, buddy list, broadcast
DTbuddylist	Delaunay Triangulation	Buddy list
DTbroadcast	Delaunay Triangulation	Broadcast
HC	Hypercube	Broadcast
SPT	Spanning Tree	Broadcast
Pastry	Pastry Distributed Hash	????

	Table	
None	Point-to-point	No rendezvous is needed

Table 1 includes a list of the rendezvous methods available for the overlay protocols of HyperCast. Each rendezvous method requires the specification of attributes in the configuration file. For example, in the Delaunay triangulation (DT) protocol, when a server-based rendezvous is selected (*DTserver*), the configuration file must have an attribute *DTServer* that includes the address of the well-known rendezvous server. The attribute can be specified as follows:

```
<DTServer>
   …
   <Rendezvous>
   <UnderlayAddress>
        <INETV4AndOnePort>192.168.123.176:8081</INETV4AndOnePort>
      </UnderlayAddress>
   </Rendezvous>
</DTServer>
```

The address of the rendezvous server is an address of the underlay network. In HyperCast, the DT server is a separate application that is started with the command:

```
java -classpath hypercast3.xxxx.jar hypercast.DT.DT_Server
```

In this case, the underlay network is the Internet and the address of the rendezvous server consists of an IP address and a port number. When the rendezvous is done with a buddy list (*DTbuddylist*), there is an attribute *Buddylist* which includes the addresses of the buddies. The specification of the attribute can be as follows:

```
<DTBuddyList>
   …
   <Buddy>
      <UnderlayAddress>
      <INETV4AndOnePort>192.168.123.176:8081</INETV4AndOnePort>
      </UnderlayAddress>
   </Buddy>
   <Buddy>
      <UnderlayAddress>
      <INETV4AndOnePort>192.168.123.176:8081</INETV4AndOnePort>
      </UnderlayAddress>
   </Buddy>
</DTBuddyList>
```

Here, the attributes specify that the buddy list has two entries, with addresses 192.168.123.176:8081 and 192.168.123.176:8081.

Selecting the Interface to the underlay network. Adapters provide the interfaces of the overlay socket to the underlay network. Each overlay socket has two adapters: a socket adapter handles application data and a node adapter handles overlay protocol messages. The selection of the adapters determines the protocols for exchanging messages between neighboring overlay sockets over the underlay network.

The following are the adapter types available in HyperCast 3.0. All available adapters are for underlay networks that use IPv4.

Table 4. Adapters available in HyperCast version 3.0.

Adapter types and names	Description
TCP adapter (NodeAdptTCP, ScktAdptTCP)	Data is exchanged over TCP connections. The adapter establishes TCP connections to TCP servers of remote adapters. Each adapter contains a TCP server that accepts connection requests from remote adapters.
UDP Adapter (NodeAdptUDP, ScktAdptUDP)	The adapter sends and receives data through a single UDP unicast port.
UDP Multicast Adapter (NodeAdptUDPMulticast, ScktAdptUDPMulticast)	The adapter has one UDP unicast port and one UDP multicast port. All adapters in the overlay network use the same UDP Multicast port.
TCP/UDP Adapter (NodeAdptTCPandUDPMulticast, ScktAdptTCPandUDPMulticast)	This adapter is a combination of the TCP adapter and the UDP multicast adapter.
SSL Adapter (NodeAdptSSL, ScktAdptSSL)	The adapter establishes SSL (Secure Socket Layer) tunnels to SSL servers of remote adapters. Each adapter contains a SSL server that accepts SSL tunnel requests from remote adapters. The security settings are TLS_RSA_WITH_AES_128_CBC_SHA

Node adapter and the socket adapters in the same overlay socket can be of a different type. On the other hand, all overlay sockets in an overlay network must to use the same type of node adapter and the same type of socket adapter. If the underlay network is the Internet, it is often desirable to select UDP for the node adapter since UDP incurs a low overhead for sporadic protocol messages. For overlay messages, it is often preferred to use the TCP protocol since TCP ensures reliable delivery between neighbors in the underlay network. For streaming applications, it may be desirable to send overlay messages as UDP datagrams.

The physical address of an overlay socket is determined in attributes of the node adapter. The physical address for an overlay in an IPv4 network specifies the IP address and the port numbers of node adapter and the socket adapter.

```
<PhysAddr>
      <INETv4TwoPorts>192.168.71.23:4444:2313</INETv4TwoPorts>
</PhysAddr>
```

The first port number configures the node adapter. The address of the node adapter is 192.168.71.23:4444, and the address of the socket adapter is192.168.71.23:2313. If no or an invalid IP address is provided, the IP address is selected by the system. If the port numbers are empty or set to 0, then the smallest available port number is selected. If an adapter is of type UDP Multicast, the adapter also specifies a multicast IP address and a port number.

```
<BroadcastAddr>
      <INETv4OnePort>224.228.19.78:4444</INETv4OnePort>
</BroadcastAddr>
```