

The Overlay Socket API

1. OVERVIEW

The HyperCast software provides an Application Programming Interface (API) for building applications that use overlay sockets. The overlay socket API hides most of the characteristics of the underlying overlay network. As a consequence, the API is fairly independent of the type and the configuration of the overlay network.

The overlay socket API is specific to the Java programming language. The API of the overlay socket is message-based, and is not very different from, Java UDP sockets. However, since the data transport between neighbors in the overlay networks can be done via TCP, the reliability of overlay sockets can be better than that of UDP sockets. (Better reliability semantics are achieved with the the HyperCast message store.)

Due to the similarity to UDP sockets, transcoding network applications from Java UDP sockets to the overlay sockets of HyperCast is generally straightforward.

This chapter provides an overview of the API of overlay sockets.

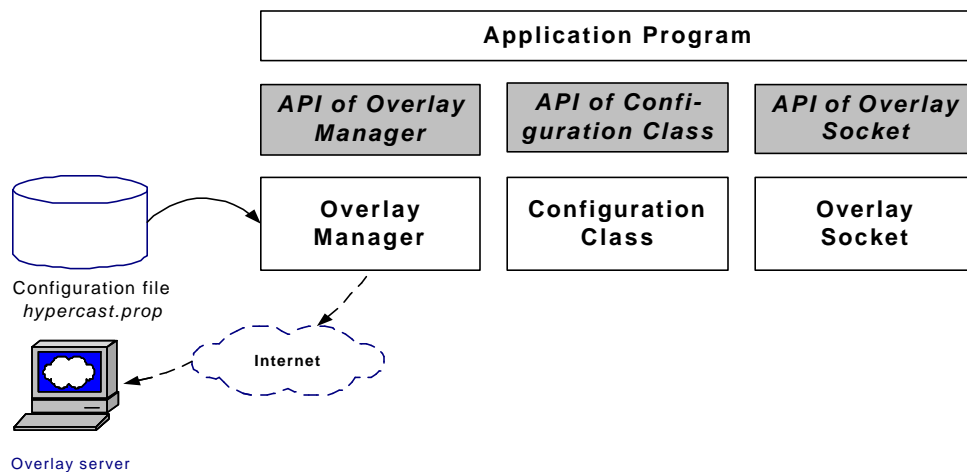


Figure 1. Relationship of application program with the APIs of the HyperCast.

As illustrated in Figure 1, applications interact with three HyperCast objects: an overlay manager, a configuration object, and an overlay socket. The overlay manager reads the configuration file. Dependent on the value of the attributes in the configuration file, the overlay manager may contact an overlay server to determine if a group exists or to download attributes of an overlay network. The overlay manager creates a configuration object that contains configuration information from the configuration file (and possibly from the overlay server). The application uses the configuration object to create a new overlay socket. The application interacts with the overlay manager and the configuration object only during the process of creating an overlay socket. Once an overlay socket is created, the application interacts generally only with the overlay socket API.

2. “HELLOWORLD” EXAMPLE

Here we present the Java code of a simple application, called *HelloWorld*, that uses the overlay socket of HyperCast. The *HelloWorld* application creates an overlay socket that joins an overlay network. After joining the overlay, the applications multicasts the string “Hello World” in a message to the overlay network. All applications that receive the message display the content of the message on the screen.

The main steps of the application are:

1. The application reads the attributes of the overlay socket from the configuration file *hypercast.prop*.
2. If the overlay configuration file contains no overlay ID, or if the file contains an overlay ID, but no overlay network has been created with this overlay ID, a new overlay ID is created.
3. The application then attempts to join the overlay network. It waits for a few seconds, to give the overlay network time to stabilize. Then, the application creates a message, which contains the string “*Hello World*” as its payload, and multicasts the message to all members of the overlay network.
4. After sending the message, the application waits for messages received from other application. If a message is received, the application prints the payload, the string “*Hello World*”, to the screen. The application also prints the logical address of the sender.

A skeleton of the program is shown in Table 1.

In Lines 1–3, an overlay manager is created. It reads the configuration file “*hypercast.prop*.” The file contains information on the overlay ID, the address of the overlay server (if it exists), the type of overlay to be used (e.g., hypercube, Delaunay triangulations), whether messages are transmitted over UDP or TCP, etc.

In Lines 4–9, the program tests if the overlay network exists and generates a configuration object. The configuration object contains all information necessary to create an overlay socket. In Line 5, the overlay ID is read from the configuration file. If no overlay ID is provided in the file, or if the overlay ID is provided, but the overlay network does not exist, the overlay manager creates a new overlay network (Line 7), possibly also a new overlay ID. The configuration object is created by `createOverlay` if a new overlay network is created, and by `getOverlaySocketConfig` if the overlay network already exists.

The overlay socket is created by the configuration object (Lines 10–11). In Lines 12–13, the socket joins the overlay network. Note that there is no notification when the process of joining an overlay network is complete. The reason that the API does not provide such a notification is that a node may not be able to decide locally if the process of joining the overlay network has been completed. In the example program, the program waits for a few seconds, to give the overlay socket enough time to join the overlay network.

In Lines 16–17 the node creates a message to be transmitted and then multicasts the message to all overlay sockets in the overlay network. The methods `createMessage` and `sendToAll` are structured similarly to the constructor and `send` method of the Java Multicast socket class. The `sendToAll` transmits the message to all neighbors of the overlay socket in the overlay network that are child nodes in a spanning tree with the

sender as the root. If the socket issues the `sendToAll` before it is fully integrated in the overlay networks, then the transmission does not reach all nodes in the overlay network.

Lines 20–31 describe an infinite loop where the overlay socket receives an overlay message (Line 23), extracts the payload (Line 25), converts the payload to a string (Line 27), and prints the string to the screen (Line 30). We note that the receive method is blocking.

Lines 32–33 show how a socket leaves an overlay. Due the infinite loop, these lines are never executed in the example program.

The following remarks emphasize aspects of the HyperCast overlay socket API.

Remarks:

- As already indicated, the API of the overlay socket bears similarity of the Java Multicast socket. This similarity is intentional.
 - The program in Table 1 is not specific to the overlay topology, the transport protocol (UDP or TCP) used to transfer overlay messages, if there is an overlay server, etc. This information is contained in the configuration file.
 - The service provided by the overlay socket above is a connectionless best-effort datagram service. If TCP is used for data transport between neighbors in the overlay network, then the reliability of transmission is high. However, there is no guarantee that all members of the overlay network receive the message. (Improved reliability semantics are achieved with the functions provided by the HyperCast message store.)
 - Instead of the synchronous receive operation in the example, the HyperCast overlay can also provide an asynchronous receive operation via upcalls. To use the upcall option, an object must be passed as a parameter when creating the overlay socket. This object must be from a class that implements the `I_Callback` interface. All classes that implement the `I_Callback` interface have a method called `MessageArrived`. This method is invoked when a message has been received by the overlay socket. In Table 2, we show a version of the *HelloWorld* program, which uses callbacks. A complete program is shown in a separate file.
-

```
1. //Generate the configuration object
2. OverlayManager om = new OverlayManager("hypercast.prop");
3. OverlaySocketConfig config = null;

4. //Test if overlay network exists and build a configuration
5. object
6. String overlayID = om.getDefaultProperty("OverlayID");
7. if (overlayID == null || overlayID.equals("") ||
8.     !om.doesOverlayExist(overlayID))
9.     config = om.createOverlay(overlayID);
10. if (config == null)
11.     config = om.getOverlaySocketConfig(overlayID);

12. //Create an overlay socket
13. OL_Socket socket = config.createOverlaySocket(null);

14. //Join an overlay
15. socket.joinGroup();

16. //Wait for some time
17. Thread.sleep(2000);

18. //Create an OL_Message with "Hello World" as payload
19. OL_Message msg = socket.createMessage(new String("Hello
20. World").getBytes(), new
21. String("HelloWorld").getBytes().length);

22. //Send the message to all members in overlay network
23. socket.sendToAll(msg);

24. //Infinite loop to receive messages
25. While(true) {
26.     //Receive a message from the socket
27.     OL_Message msg = socket.receive();

28.     //Extract the payload
29.     byte[] data = msg.getPayload();

30.     //Recover the "Hello World" message
31.     String helloworld = new String(data);

32.     //Print out the "Hello World" message together with
33.     the //sender's logical address
34.     System.out.println(msg.getSrcLogicalAddress().toString
35.     ( ) + " says " + helloworld + " ! " );
36. }

37. // Leave the overlay network
38. socket.leaveGroup();
```

Table 1 . Skeleton of the Hello World application.

```

1.  public class HelloWorld implements I_Callback {
2.      public void messageArrived(I_OverlayMessage msg) {
3.          // Add lines 24-30 from Table 1
4.      }
5.
6.      public static void main() {
7.          // Add Lines 1-9 from Table 1
8.          // Create a new Hello World object
9.          HelloWorld hw = new HelloWorld();
10.         // Create overlay socket and pass object hw for
11.         // callback
12.         OL_Socket socket = ConfObj.createOverlaySocket(hw);
13.         // Add Lines 12-19 from Table 1
14.         // Wait for 5 minutes for messages to arrive
15.         Thread.sleep(30000);
16.         // Leave the overlay network
17.         socket.leaveGroup();
18.     }
19. }

```

Table 2 . Skeleton of the Hello World application (with Callback).

3. OVERLAY MANAGER

The overlay manager of the HyperCast software provides an interface for the management of overlay networks and their attributes. The application programmer uses the overlay manager to determine and set the properties of sockets that connect to the overlay network, to determine if an overlay network already exists, and to create a new overlay network.

The use of the methods of the OverlayManager API is summarized below. Details can be obtained from the Javadoc documentation.

```
OverlayManager om = OverlayManager ("hypercast.prop")
```

The constructor reads properties from the configuration file.
Several constructors are available.

```
OverlaySocketConfig oc = om.createOverlay ("overlayname")
```

Creates a new overlay network with overlay ID "overlayname".
The overlay manager will contact the overlay server (if one is provided in the configuration file) and create an entry for the new

overlay network at the server. This method returns the overlay attributes in an `OverlaySocketConfig` object.

```
Boolean test = om.doesOverlayExist ("overlayname")
    Returns True if the overlay exists at the overlay server, and
    False otherwise. This method is meaningful only if an overlay
    server is used.
```

```
String prop = om.getDefaultProperty ("propertyname")
    This method is useful to query the value of specific properties.
    Both the parameter and the value returned are string.
```

```
OverlaySocketConfig conf =
    om.getOverlaySocketConfig ("overlayname")
    Get the overlay attributes of the overlay with specified overlay
    ID and return the overlay attributes in an OverlaySocketConfig
    object.
```

```
Properties prop =
    overrideDefaults (java.util.Properties defaults,
    java.lang.String overrides)
    Overwrite the default attributes with the attributes obtained from
    the server.
```

```
void
    setDefaultProperty (java.lang.String name,
    java.lang.String value)
    Set the default property with the property name and its value.
```

4. CONFIGURATION OBJECT

The configuration object `OverlaySocketConfig` stores the configuration for an overlay socket, and maintains the error and log files. The configuration object creates the overlay socket. The object is usually created by the overlay manager via the `OverlayManager.createOverlay(String overlayID)` method, as follows:

```
OverlaySocketConfig co= om.createOverlay(overlayID)
```

Note that the configuration object has a statistics (`I_Stats`) interface. Therefore, the configuration object can be queried by monitoring and control components of HyperCast.

```
I_OverlaySocket createOverlaySocket(I_Callback callback)
    This method creates an overlay socket. The type of overlay
    socket created depends on the value of the properties "Node" and
    "SocketAdapter". The callback provides an object that
    implements the I_Callback interface. Whenever a message is
    received, the method callback.messageArrived is invoked.
    If no callback is provided, i.e., socket =
    co.createOverlaySocket (null), then messages are
    received with the blocking receive method of the overlay
    socket.
```

Note:

The configuration object can also create overlay sockets which transmit data over a Java UDP multicast socket this is done

```
java.net.MulticastSocket createJavaMulticastSocket()
```

This creates a new object that implements the **Java.net.MulticastSocket**, which sends UDP multicast messages using an overlay socket.

5. OVERLAY SOCKET

The API of an overlay socket offers applications the ability to:

- Join and leave existing overlay networks;
- Send data to all or a subset of the members of the overlay network;
- Receive data from the overlay.

In HyperCast 2.0, four different types of overlay sockets are supported. The type of socket that is created depends on the values of attributes “Node” and “Socket_Adapter” in the configuration file. The types of sockets are:

1. **OL_Socket_CO_HC**: This overlay socket runs the hypercube overlay protocol. The overlay protocol uses UDP multicast and UDP unicast. Transmission of application messages between members of the overlay network is done with TCP.
2. **OL_Socket_CL_HC**: The same as (1), but application messages are transmitted with UDP.
3. **OL_Socket_CO_DT**: This overlay socket runs the DT overlay protocol. The overlay protocol uses UDP unicast. The protocol requires to run an application that acts as the DT server.¹ Transmission of application messages between members of the overlay network is done with TCP.
4. **OL_Socket_CO_DT**: The same as (1), but application messages are transmitted with UDP.

Note: When writing a program using the released jar files, the following package needs to be imported in your program:

```
import edu.virginia.edu.cs.hypercast.*;
```

While each type of overlay socket can provide its own application programming interface, all socket types support the API of the `I_OverlaySocket` class. Therefore, as long as application programs restrict themselves to the methods of `I_OverlaySocket`, application programs can use different overlay sockets.

The following table lists all the API methods provided by the `I_OverlaySocket`. The calls to the API can be grouped into the following categories.

(a) Overlay Participation

```
socket.joinGroup()      Join an existing overlay network.
```

¹ We refer to the User Manual for details.

`socket.leaveGroup()` Leave an existing overlay network.

(b) Sending and Receiving Messages

`OL_Message msg = socket.createMessage(payload, length)`
Create an overlay message with specified payload and the length of the payload. The payload is a byte array.

`msg = socket.receive()`
Receives a overlay application message from this socket. This call will block until there is a message returns.

`socket.setTTL(ttlvalue)`
Set the default hop limit for overlay message sent out by this socket.

`Socket.SendToAll (msg)`
Send the overlay message to all nodes in the overlay group. Messages are set to multicast delivery mode.

`Socket.SendToNode (msg, DestLA)`
Send the overlay message to a socket with the specified logical address. This message is sent in unicast delivery mode.

`Socket.SendFlood (msg, DestLA)`
Send the overlay message to all neighbors with exception of the neighbor from which it received the message; Messages are set in flood delivery mode.

`Socket.SendToAllNeighbors(msg)`
Send the message to all neighbors in the overlay network. Messages are set to unicast delivery mode.

`Socket.SendToParent(msg, LARoot)`
Send message to the parent in the spanning tree with respect to the root with logical address `LARoot`. This is a unicast message.

`Socket.SendToChildren (msg, LARoot)`
Send message to each of the children in the spanning tree with respect to the root with logical address `LARoot`. This is a set of unicast messages.

`Socket.setSoTimeout(int timeout) throws SocketException`
Enable/disable `SO_TIMEOUT` with the specified timeout, in milliseconds. With this option set to a non-zero timeout, a call to `receive()` for this overlay socket will block for only this amount of time.

(c) Information

`byte[] getUniqueIdentifier()`
Returns a byte array containing a unique identifier of this socket.

`Socket.createLogicalAddress(laddr, int offset)`
Create the logical address from a byte array.

`socket.getLogicalAddress()`
Get the logical address of the overlay socket.

`LogicalAddress LA = socket.getParent(LARoot)`
Get the logical address of the parent in the spanning tree with respect to the root with logical address `LARoot`.

LogicalAddress[] children = children.getChildren(LAroot)
Get the logical addresses of the children in the spanning tree with respect to the root with logical address LAroot.

LogicalAddress[] neighbors = socket.getNeighbors()
Get the logical addresses of all neighbors in the overlay network.

socket.getTTL() Get the default hop limit for overlay messages sent out by this overlay socket.

int timeout = socket.getSoTimeout()
Retrieve the value of SO_TIMEOUT. A return value of 0 implies that the option is disabled, i.e., there is no limit on the blocking time.
