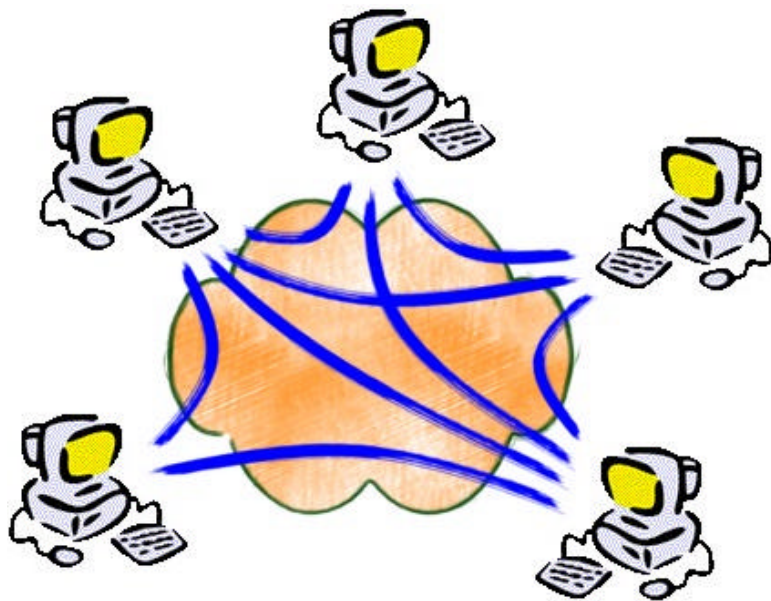


# HyperCast 2.0

## Notes on Implementation

Description of classes



HyperCast Team  
Department of Computer Science  
University of Virginia

Email: [hypercast@cs.virginia.edu](mailto:hypercast@cs.virginia.edu)  
Web: <http://www.cs.virginia.edu/~hypercast>

January 2002

---

|   |           |
|---|-----------|
| <b>1. HYPERCAST 2.0 OVERVIEW.....</b>                     | <b>2</b>  |
| 1.1. COMPONENTS .....                                     | 3         |
| 1.2. INTERFACES.....                                      | 4         |
| <b>2. OVERLAY SOCKET DESIGN AND IMPLEMENTATION .....</b>  | <b>5</b>  |
| 2.1. OVERVIEW.....  | 5         |
| 2.2. OL_SOCKET DESIGN .....                               | 6         |
| 2.3. UDP OVERLAY SOCKET (OL_SOCKET_CL) .....              | 9         |
| 2.4. TCP OVERLAY SOCKET (OL_SOCKET_CO) .....              | 10        |
| 2.5. HYPERCUBE UDP OVERLAY SOCKET (OL_SOCKET_CL_HC) ..... | 11        |
| 2.6. HYPERCUBE TCP OVERLAY SOCKET .....                   | 12        |
| 2.7. DELAUNAY TRIANGULATION UDP OVERLAY SOCKET .....      | 13        |
| 2.8. DELAUNAY TRIANGULATION TCP OVERLAY SOCKET .....      | 14        |
| <b>3. ADAPTERS .....</b>                                  | <b>15</b> |
| 3.1. ADAPTER DESIGN HIERARCHY .....                       | 15        |
| 3.2. TCP_UNICASTADAPTER DESIGN .....                      | 16        |
| 3.3. UDP_MULTICAST AND UNICAST ADAPTER DESIGN .....       | 19        |
| <b>4. OVERLAY NODE DESIGN .....</b>                       | <b>22</b> |
| 4.1. NODE INTERFACE (I_NODE) IMPLEMENTATION .....         | 23        |
| 4.2. PROTOCOL NODE SPECIFIC METHODS .....                 | 23        |
| 4.3. ADAPTERCALLBACK INTERFACE IMPLEMENTATION .....       | 24        |
| <b>5. FORWARDING ENGINE.....</b>                          | <b>25</b> |
| <b>6. STATISTICS INTERFACE.....</b>                       | <b>28</b> |
| 6.1. OVERVIEW.....  | 28        |
| 6.2. NAMES OF STATISTICS.....                             | 28        |
| 6.3. XML SCHEMA .....                                     | 29        |
| 6.4. IMPLEMENTATION OF I_STATS INTERFACE .....            | 30        |
| 6.5. IMPLEMENTATION OF STATSEXCEPTION.....                | 30        |
| <b>7. OVERLAY MANAGEMENT.....</b>                         | <b>32</b> |
| 7.1. OVERVIEW.....  | 32        |
| 7.2. STRUCTURE.....                                       | 32        |
| 7.3. OBJECTS.....   | 32        |

---

## 1. Hypercast 2.0 Overview

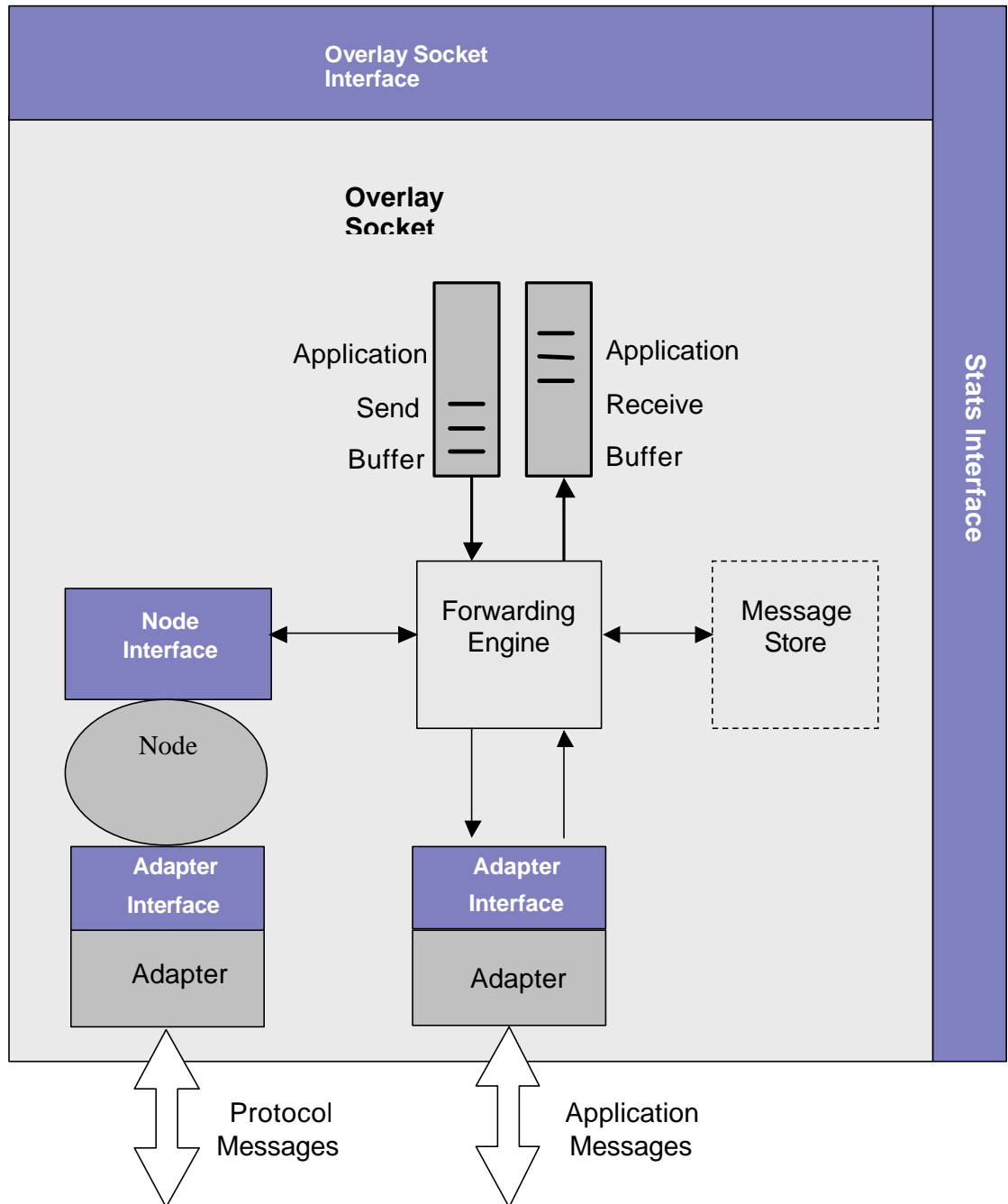


Figure 1. Components of an Overlay Socket in Hypercast 2.0.

The Hypercast 2.0 overlay socket provides a socket style API for applications to transmit messages through different overlay networks. This document contains brief notes, which describe the Java implementation of the HyperCast Software. For a language independent description of HyperCast 2.0, we refer to the design document.

---

## 1.1. Components

An overlay socket consists of the following components. Note that not all of them are object.

### Overlay Socket

An overlay socket in HyperCast 2.0 is implemented by the abstract class `OL_Socket`. It contains a configuration object, a protocol node, a socket adapter, and a receive buffer. The configuration object provides the configuration parameters, the protocol node implements the overlay protocol, the socket adaptor sends and receives application messages, and the receive buffer stores received application messages.

### Protocol Node

A protocol node is an object node that constructs and maintains the overlay topology. A protocol nodes executes the finite state machine for a specific overlay protocol. Currently, there are two protocol node classes: `HC_Node` runs the hypercube protocol, and `DT_Node` runs the DT protocol. Each protocol node contains a configuration object and an adaptor. The adaptor sends and receives protocol messages.

### Adapter

An adapter provides an interface to send and receive messages over a network, and maintains timers. There are three different adapters:

- o **UDP\_UnicastAdapter** provides unicast transmission over UDP and timer functions.
- o **UDP\_MulticastAdapter**: provides multicast transmission over UDP and timer functions.
- o **TCP\_UnicastAdapter** provides unicast transmission over TCP and timer functions.

Each overlay socket has two adapters. One adapter is used for protocol message transmissions and is part of the protocol node. The other is used for application message transmissions and is part of the overlay socket.

### ReceiveBuffer

The `ReceiveBuffer` stores received application messages. The `ReceiveBuffer` is an object from class `MessageBuffer`. The application program reads received messages from this buffer. If the application has supplied a callback function, received messages are **not** put into the `ReceiveBuffer`. Instead, the callback function is executed when the overlay socket has received and processed the message. Each `ReceiveBuffer` contains a configuration object.

### SendBuffer

The `SendBuffer`, an object of class `MessageBuffer`, stores application messages that are transmitted by the application program, but are not transmitted due to rate or congestion control restrictions.

*The `SendBuffer` is not implemented in Hypercast 2.0!*

### ForwardingEngine

The `ForwardingEngine` performs forwarding functions of application messages in the overlay network. The `ForwardingEngine` forwards and delivers messages using the adapter. The `ForwardingEngine` is implemented as a set of methods of the overlay socket and is not a separate object.

---

---

## MessageStore

The MessageStore provides a set of services, such as synchronization, streaming, and reliable delivery, on top of the application message delivery service of the overlay socket.

*The Message Store is not implemented in Hypercast 2.0.*

## 1.2. Interfaces

### Overlay Socket Interface (I\_OverlaySocket)

The I\_OverlaySocket interface provides a socket like API for message transmission on the overlay topology. It is implemented by the OL\_Socket class.

### Node Interface (I\_Node)

The I\_Node interface provides an API for accessing information on a protocol node and its neighborhood. The interface is implemented by HC\_Node and DT\_Node. The I\_Node interface is accessed by the overlay socket.

### Adapter Interfaces

There are two different adapter interfaces:

#### I\_UnicastAdapter

Provides timers and a network interface. It is implemented by both UDP\_UnicastAdapter and TCP\_UnicastAdapter.

#### I\_AdapterCallback

Provides function callbacks for an adapter. It is implemented by the OL\_Socket and protocol node.

### Statistics Interface (I\_Stats)

The I\_Stats interface provides an API for setting and accessing statistics information of an object that implements this interface. It is implemented by the Node class, Adapter classes, MessageBuffer classes, and OL\_Socket class.

---

## 2. Overlay Socket Design and Implementation

### 2.1. Overview

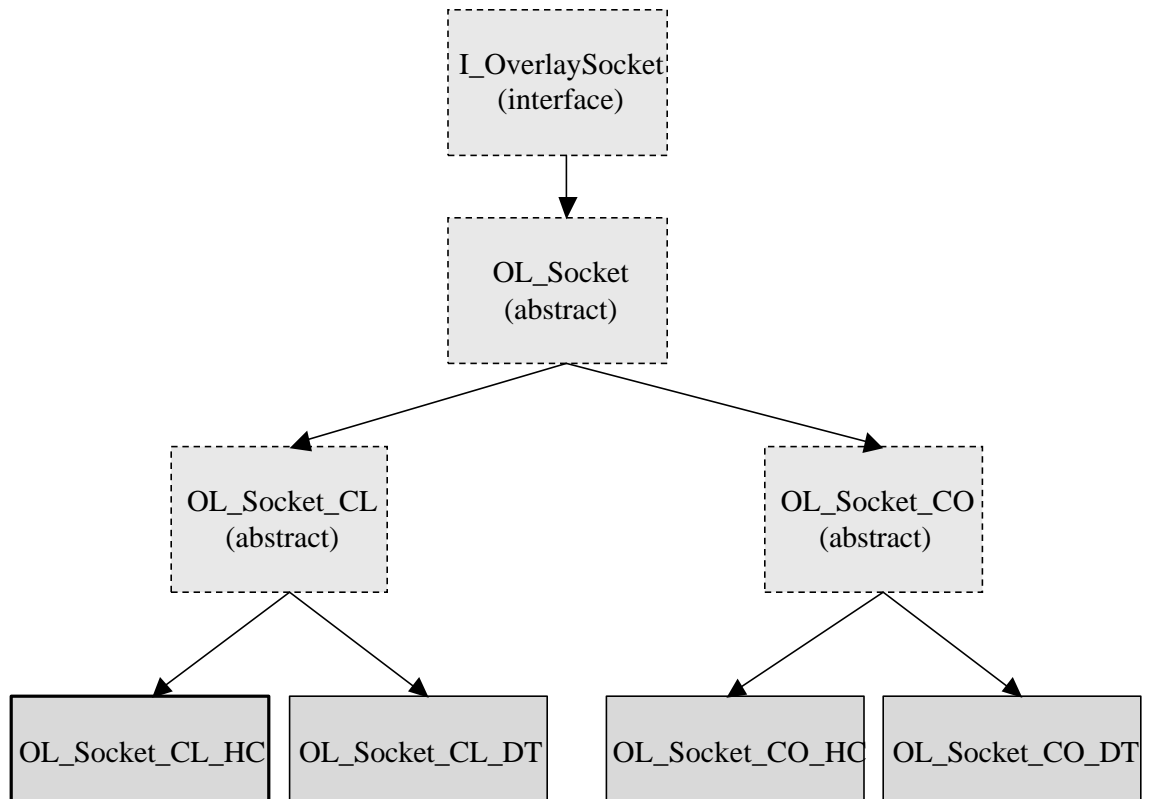


Figure 2. Overlay Socket Class Hierarchy.

The following list provides an overview of the class hierarchy for the overlay socket classes in HyperCast 2.0. The hierarchy is illustrated in Figure 2.

- I\_OverlaySocket** An interface that defines a socket style API between application and the overlay topology.
- OL\_Socket** An abstract class that implements the I\_OverlaySocket interface.
- OL\_Socket\_CL** An abstract class that extends OL\_Socket and adds a UDP transmission service.
- OL\_Socket\_CO** An abstract class that extends OL\_Socket and adds a TCP transmission service.
- OL\_Socket\_CL\_HC** A derived class that extends OL\_Socket\_CL and runs on the Hypercube protocol. (HC2-0)
- OL\_Socket\_CO\_HC** A derived class that extends OL\_Socket\_CO and runs on the Hypercube protocol. (HC2-0)
- OL\_Socket\_CL\_DT** A derived class that extends OL\_Socket\_CL and runs on the DT protocol. (DT2-0)
- OL\_Socket\_CO\_DT** A derived class that extends OL\_Socket\_CO and runs on the DT protocol. (DT2-0)

## 2.2. OL\_Socket Design

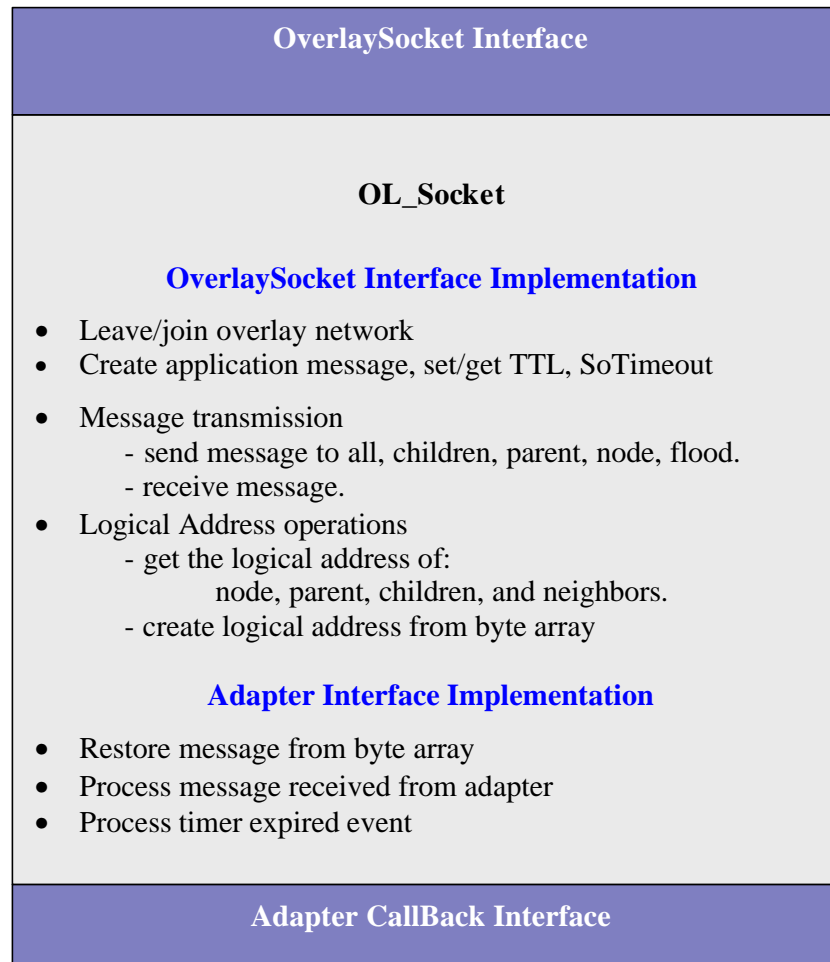


Figure 3. OL\_Socket structure and functions

The OL\_Socket (Overlay Socket) class is an abstract class that provides a socket style API for message transmission in an overlay topology. OL\_Socket does not specify the overlay protocol that maintains the topology structure or the type of adapter that is used to send and receive messages over a network. OL\_Socket is made concrete by classes OL\_Socket\_CL\_HC, OL\_Socket\_CL\_DT.

OL\_Socket contains an implementation of the OverlaySocket Interface and the AdapterCallBack Implementation.

### 2.2.1 OverlaySocket Interface (I\_OverlaySocket) Implementation

This part contains the API between the application program and the overlay network. It provides all the operations that an application needed to make use of the overlay topology.

#### Overlay Operations

|                   |   |
|-------------------|---|
| void joinGroup()  | Starts an attempt to join an overlay network (implemented by OL_Node) |
| void leaveGroup() | Leaves a multicast group (implemented by OL_Node)                     |

**Application message (OL Message) Operations**

OL\_Message createMessage(byte[] payload, int length)  
Creates an overlay message

void setTTL(int ttl)  
Sets TTL for application message

int getTTL()  
Gets TTL form application message

void setSoTimeout(int timeout)  
Sets socket timeout for application message

int getSoTimeout()  
Gets socket timeout for application message

**Application message (OL Message) Transmission Operations**

- **Send an overlay message from this socket**

void sendToAll(I\_OverlayMessage m)  
Sends (multicasts) an application message to all overlay sockets in the overlay network

void sendToChildren(I\_OverlayMessage m, I\_LogicalAddress root)  
Sends an application message to children with respect to an embedded tree with given root

void sendToAllNeighbors(I\_OverlayMessage m)  
Sends an application message to all neighbors

void sendToParent(I\_OverlayMessage m, I\_LogicalAddress root)  
Sends an application message to parent node with respect to an embedded tree with given root

void sendToNode(I\_OverlayMessage m, I\_LogicalAddress destla)  
Sends an application message to a specified node with a given logical address

void sendFlood(I\_OverlayMessage m)  
Sends an application message using “flooding”, i.e., the message is forwarded to all neighbors with exception of the node from which the message was received

- **Receive application message from this socket:**

I\_OverlayMessage receive()  
Receives an application message from the socket

- **Logical Address operations**

I\_LogicalAddress createLogicalAddress(byte[] laddr, int offset)  
Creates a logical address for the overlay node in this socket

I\_LogicalAddress getLogicalAddress()  
Retrieves the logical address of the node in this socket

I\_LogicalAddress getParent(I\_LogicalAddress root)  
Given the root of an embedded tree, returns the logical address of the node’s parent

I\_LogicalAddress getChildren(I\_LogicalAddress root)  
Given the root of an embedded tree, returns the logical addresses of the node’s children

I\_LogicalAddress getNeighbors()

---

Retrieves the logical addresses of all the neighbors

- **Other operation**

Byte[] getUniqueIdentifier()

Retrieves a byte array which is a unique identifier for this socket. This byte array may be used for example, as a Message Identifier, to uniquely represent an overlay message

### **2.2.2 Adapter Callback Interface (I\_AdapterCallback) Implementation**

The Adapter Callback interface, is invisible to application programs, and defines the operations executed by the OL\_Socket when data arrives from the socket adapter. When the socket adapter (under OL\_Socket, TCP or UDP) receives data from another socket, the data cannot be interpreted by the socket adaptor. It is the OL\_Socket's responsibility to interpret the content and process the arriving data. Thus, when the socket adaptor of overlay socket A receives data, it calls methods of overlay socket X, which interpret the arrived data. These methods are implementations of the I\_AdapterCallback interface.

I\_Message restoreMessage(byte[] data, int start[], int end)

Restores a message from a byte array

void messageArrivedFromAdapter(OL\_Message msg)

Processes an incoming message

void timerExpired(int timerID)

Not supported by OL\_Socket

---

### 2.3. UDP Overlay Socket (OL\_Socket\_CL)

The OL\_Socket\_CL class constructs overlay socket which performs data transmissions over UDP ports. OL\_Socket\_CL extends OL\_Socket, by adding a socket adapter that sends and receives UDP datagrams. The OL\_Socket\_CL performs the following functions:

1. Reserves two successive port numbers, one for the node adaptor (which is an UDP adapter), the other for the socket adapter, which is also an DP adapter, and
2. Creates an UDP adapter for the overlay socket.

In this class, the protocol node uses the UDP adapter for protocol message transmission, and the overlay socket uses another UDP adapter for application message transmission.

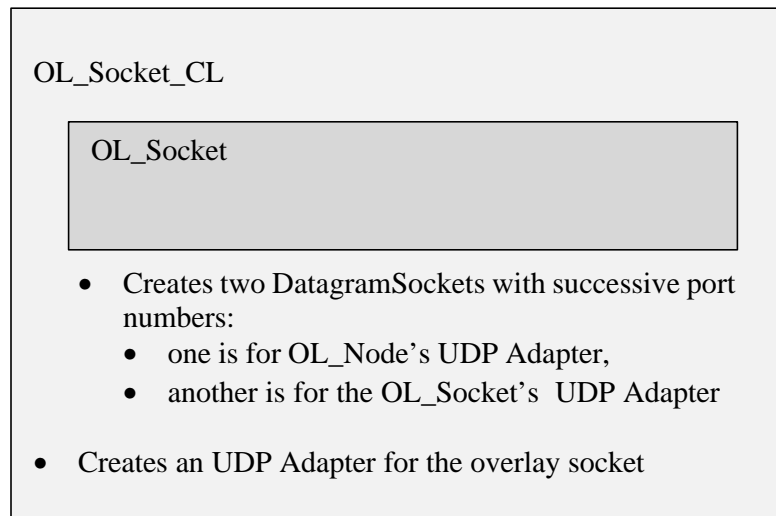


Figure 4. OL\_Socket\_CL Structure and Functions.

## 2.4. TCP Overlay Socket (OL\_Socket\_CO)

The OL\_Socket\_CO class constructs overlay socket which performs data transmissions over TCP connections. OL\_Socket\_CO extends the OL\_Socket class, and adds a socket adapter which handles TCP traffic. It includes:

- Reserves two successive port numbers, one is for overlay node's UDP adapter, another for the overlay socket's TCP adapter.
- Creates a TCP adapter for the overlay socket.

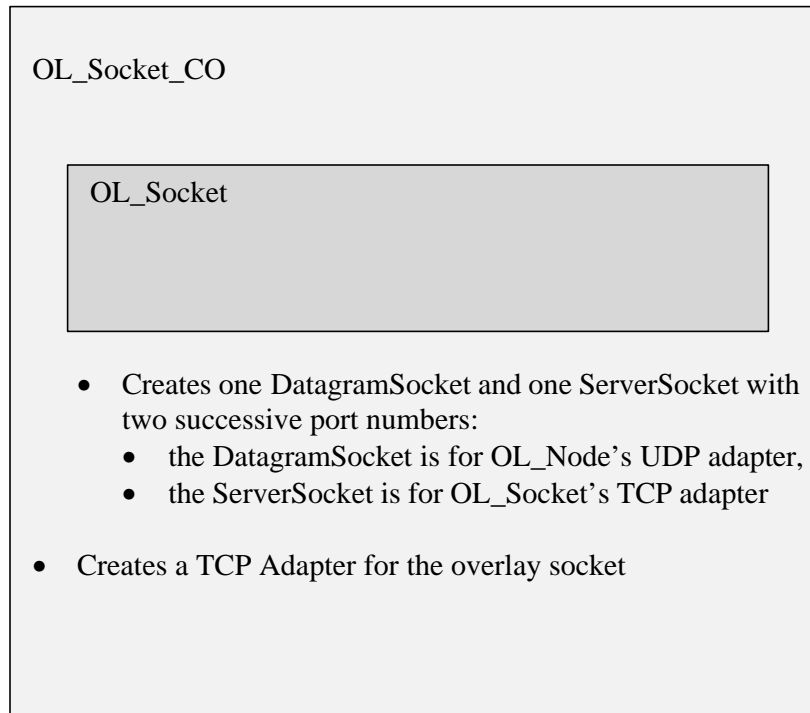


Figure 5. OL\_Socket\_CO Structure and Functions.

## 2.5. Hypercube UDP Overlay Socket (OL\_Socket\_CL\_HC)

The class `OL_Socket_CL_HC` is a Hypercube overlay socket with data transmission over UDP. It extends `OL_Socket_CL` and adds Hypercube specific functions. This includes the creation of a UDP Unicast adapter for the protocol node (recall that the HC protocol transmits some protocol messages using IP multicast), a hypercube protocol node, and some functions specific to the HC protocol.

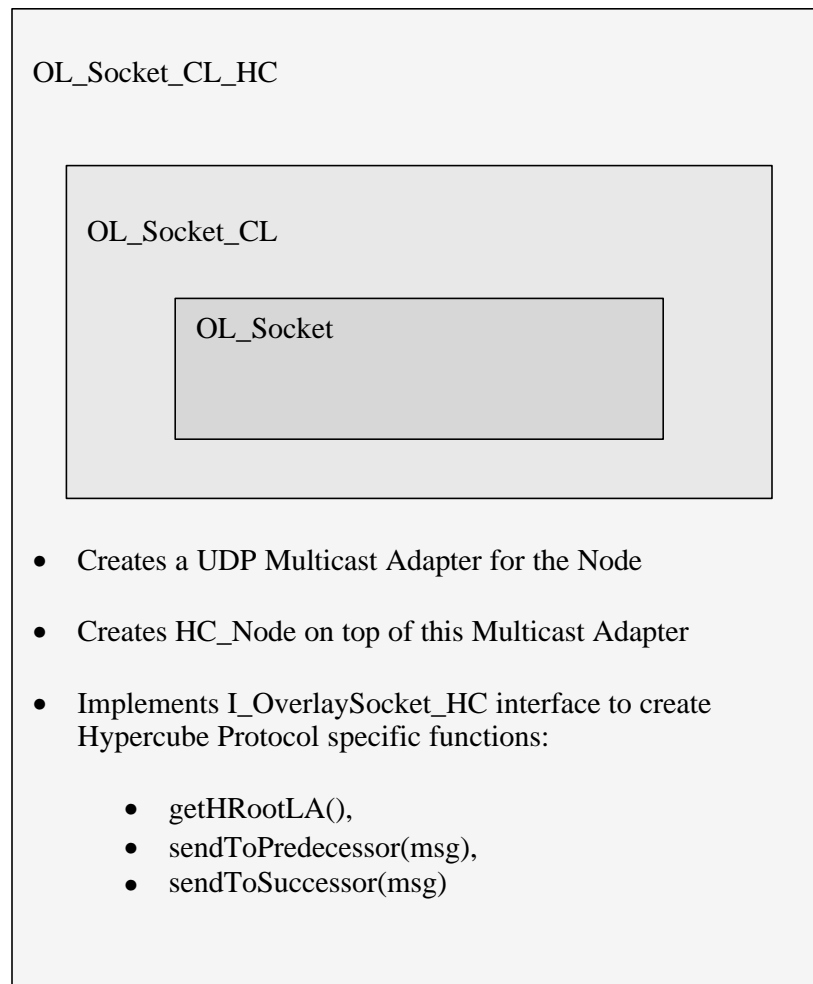


Figure 6. `OL_Socket_CL_HC` Structure and Functions

## 2.6. Hypercube TCP Overlay Socket

The class `OL_Socket_CO_HC` is a Hypercube overlay socket with data transmission over TCP connection. The class extends `OL_Socket_CO`, and adds Hypercube specified functions. This includes the creation of an UDP Multicast adapter for the protocol node, a hypercube protocol node, and functions specific to the HC protocol.

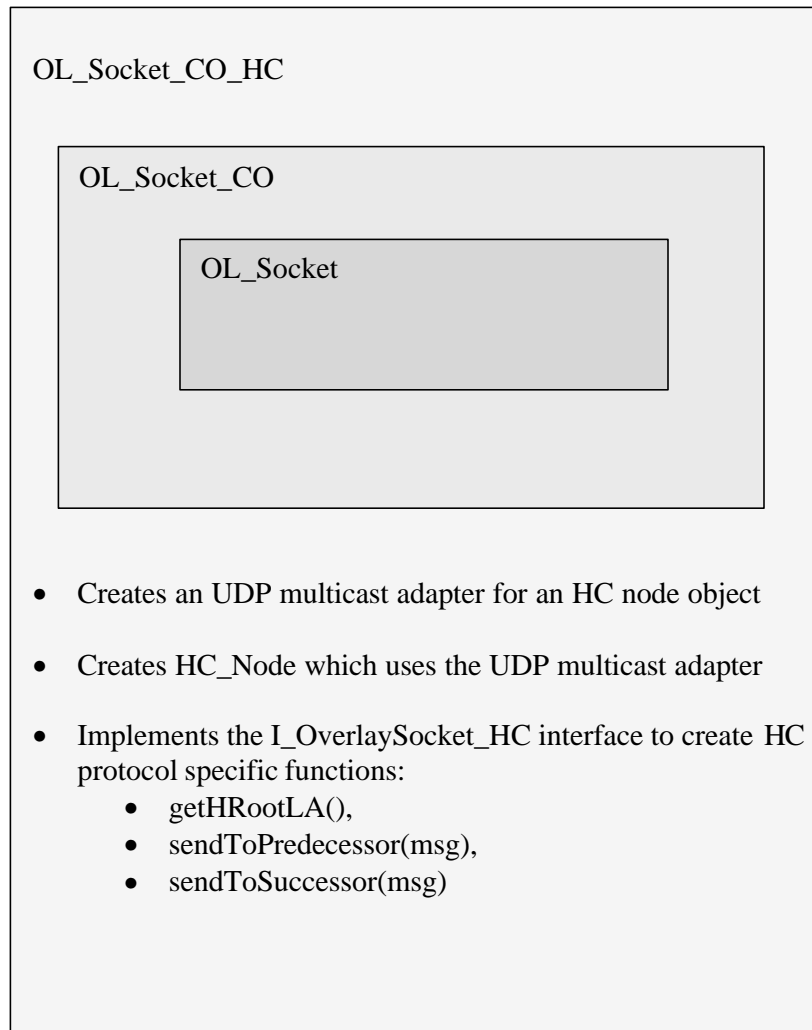


Figure 7. `OL_Socket_CO_HC` Structure and Functions.

## 2.7. Delaunay Triangulation UDP Overlay Socket (OL\_Socket\_CL\_DT)

The class OL\_Socket\_CL\_DT is a Delaunay Triangulation overlay socket with data transmission over UDP datagrams. The class extends OL\_Socket\_CL, and adds DT specific functions. This includes the creation of an UDP Multicast adapter for the node, a DT protocol node, and some functions specific to the DT protocol.

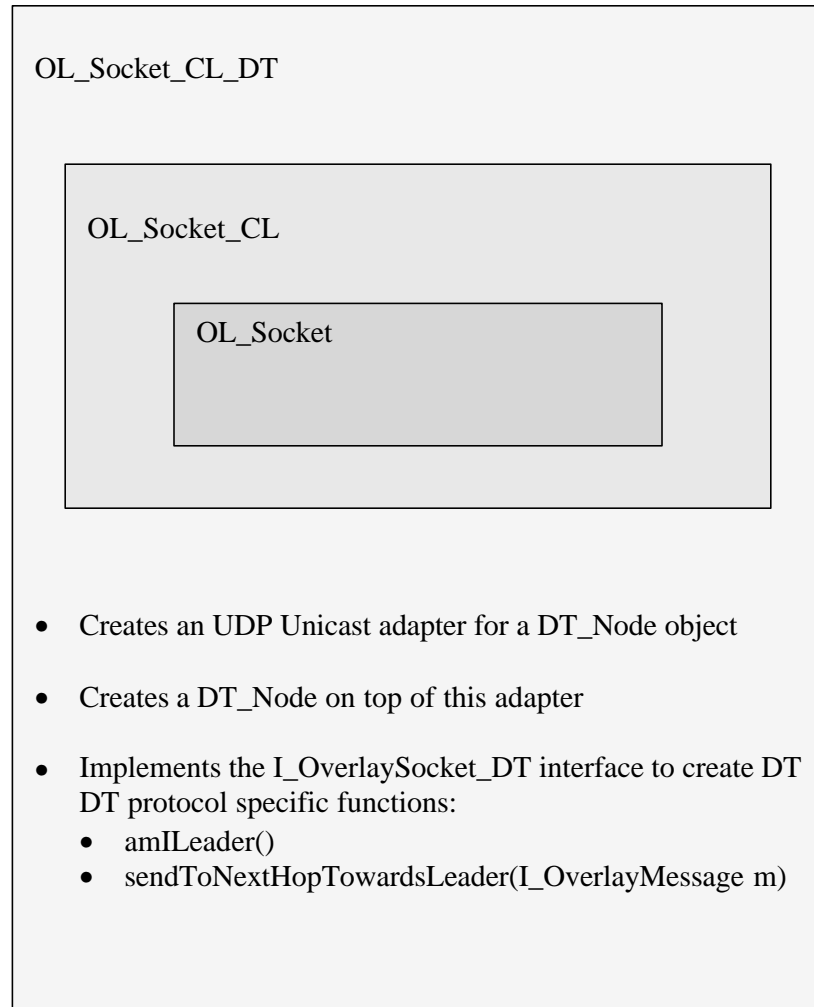


Figure 8. OL\_Socket\_CL\_DT Structure and Functions

## 2.8. Delaunay Triangulation TCP Overlay Socket (OL\_Socket\_CO\_DT)

The class `OL_Socket_CO_DT` is a DT overlay socket with data transmission over TCP connection. The class extends `OL_Socket_CO`, and adds DT specific functions. This includes the creation of an UDP Unicast adapter for the DT protocol node, a DT protocol node, and some functions specific to the DT protocol.

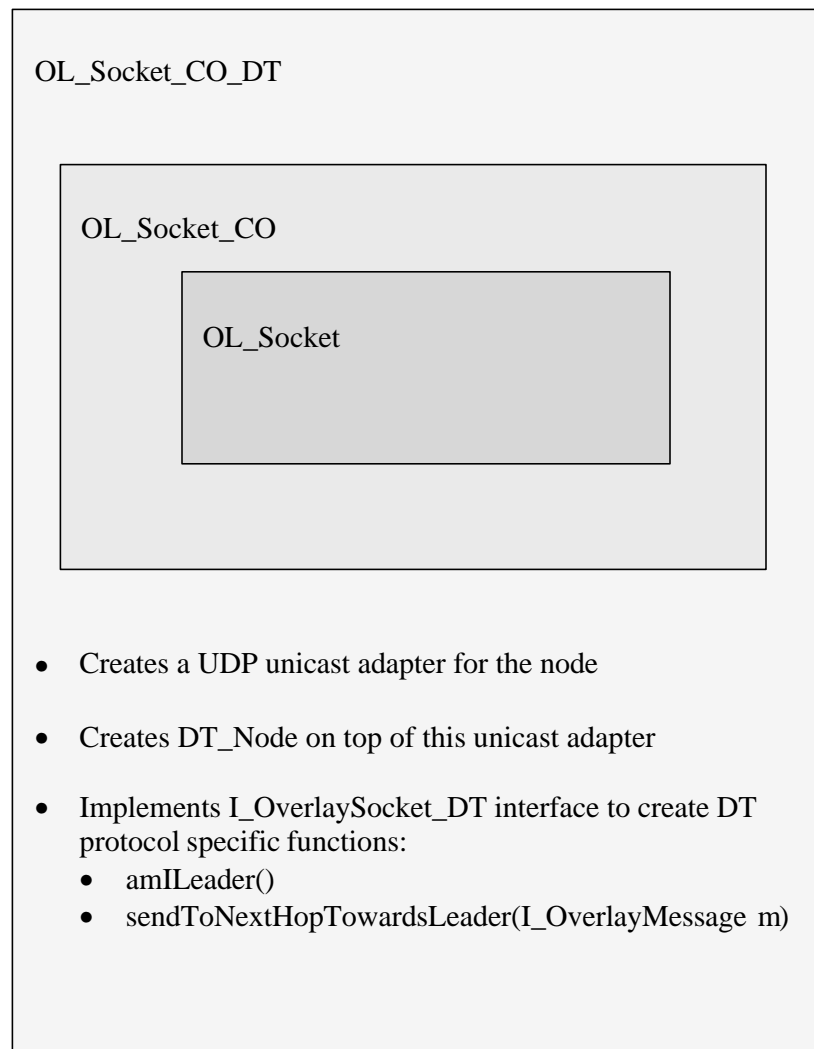


Figure 9. `OL_Socket_CO_DT` Structure and Functions.

## 3. Adapters

### 3.1. Adapter Design Hierarchy

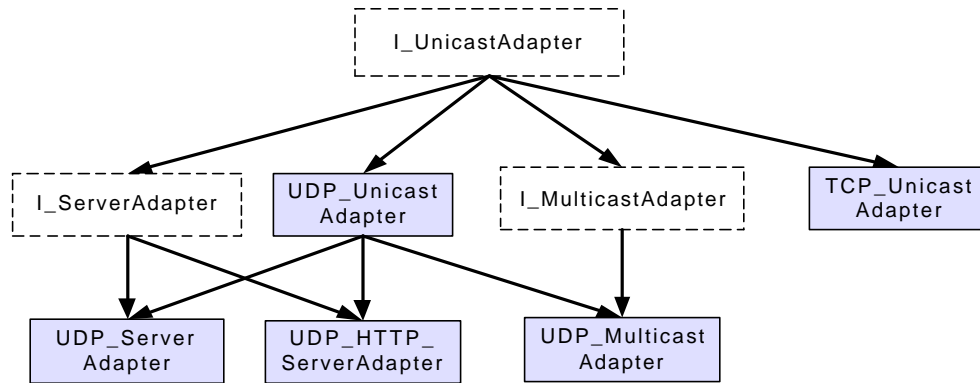


Figure 10. Adapter design hierarchy.

The adapters are concerned with the delivery of messages over UDP unicast, UDP multicast or TCP ports. Since the adapters entirely hide all Internet specific functions to the overlay socket, most code of the overlay socket is not specific to Internet protocols. Generally, the adapters contain all of the threads of the overlay socket.

The following list provides an overview of the class hierarchy for the adapter classes in HyperCast 2.0. The hierarchy is illustrated in Figure 10.

|                               |  |
|-------------------------------|--|
| <b>I_UnicastAdapter</b>       | An interface that defines a unicast adapter interface  |
| <b>I_MulticastAdapter</b>     | An interface that defines a multicast adapter interface, which extends I_UnicastAdapter                                |
| <b>I_Server</b>               | An interface that defines a server adapter interface, which extends I_UnicastAdapter                                   |
| <b>UDP_UnicastAdapter</b>     | A derived class that implements I_UnicastAdapter and runs over UDP connection  |
| <b>UDP_MulticastAdapter</b>   | A derived class that implements I_MulticastAdapter and uses UDP for data transfer                                      |
| <b>TCP_UnicastAdapter</b>     | A derived class that implements I_UnicastAdapter and uses TCP connection for data transfer                             |
| <b>UDP_ServerAdapter</b>      | A derived class that implements I_ServerAdapter, extends UDP_UnicastAdapter and uses UDP to handle messages to server  |
| <b>UDP_HTTP_ServerAdapter</b> | A derived class that implements I_ServerAdapter, extends UDP_UnicastAdapter and uses HTTP to handle messages to server |

In Hypercast2.0, the `UDP_MulticastAdapter` provides the note adapter for the `HC_Node`, and `UDP_UnicastAdapter` provides the protocol adapter for the `DT_Node`. The socket adapter can be either an `UDP_UnicastAdapter` or a `TCP_UnicastAdapter`.

## 3.2. TCP\_UnicastAdapter Design

### 3.2.1 Overview

If the socket adapter of an `OL_Socket` is a `TCP_UnicastAdapter` case, the `OL_Socket` sends and receives application data through TCP connections between neighbors. One TCP connection is used as a unidirectional link between two neighbors in the overlay network. So, up to two TCP connections may exist between two neighbors. TCP connections are created when necessary, that is, upon demand, and they are shut down when they are not used for `TIMEOUT_TIME` milliseconds. A `TCP_UnicastAdapter` adapter as an elaborate deadlock detection mechanism. When the thread assigned to receive from a TCP connection has not read data for `DROP_TIME` seconds, a message is dropped.

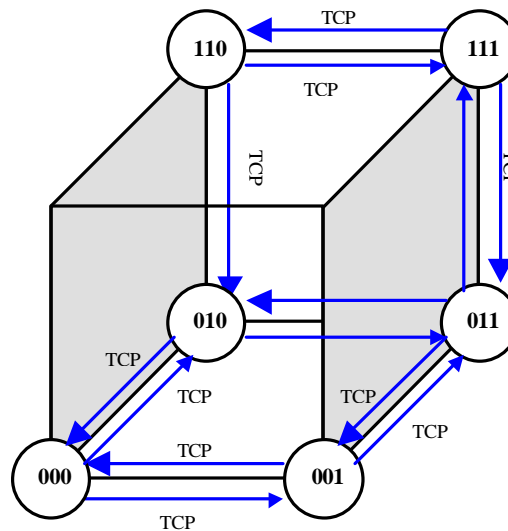


Figure 11. TCP connections between neighbors.

The structure of the `TCP_UnicastAdapter` is shown in Figure 12. The `TCP_UnicastAdapter` contains a TCP Server thread, a `DeadlockMonitor` thread, a Dropper thread, a Timer thread, and up to  $N$  Receiver threads, where  $N$  is the number of neighbors in the overlay network.

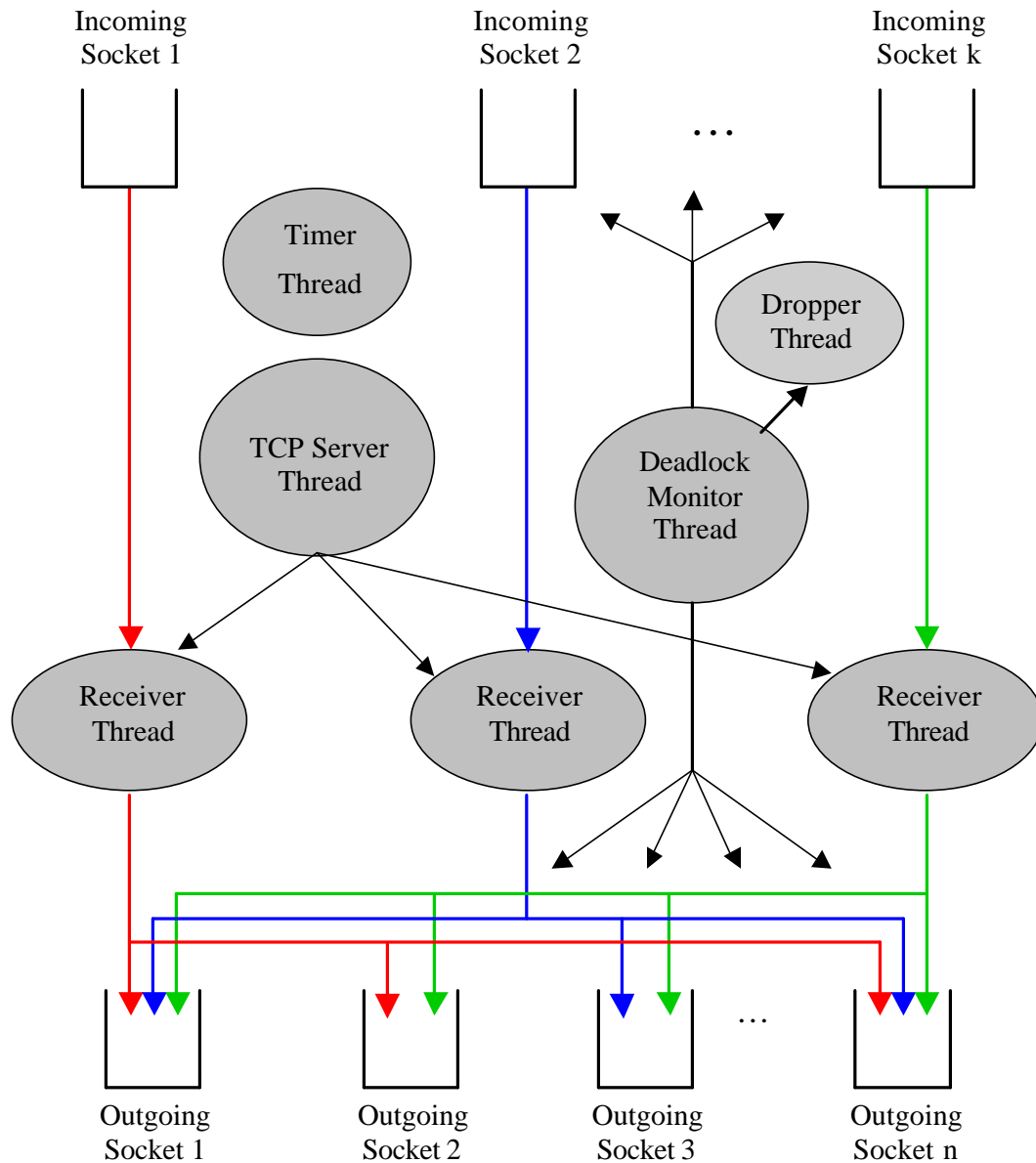


Figure 12. TCP\_UnicastAdapter structure.

### 3.2.2 TCP Server Thread

The TCP Server thread is used to accept TCP connections. For each connection, it creates an IncomingSocket object and a Receiver thread. As in the OL\_SocketAdapter, the Server thread listens to a port number, which is well known by all the nodes neighbors. This port is allocated when the OL\_Socket is created; The reserved port number is one greater than the OL\_NodeAdapter port number. TCP connection is built on demand. For example, if a node sends a message to it's three children, it will open three different TCP connections, save these connections in a hashtable, and use these connections until they are closed because of a timeout or some other reasons. If there are two neighbors sending messages to a node, the node accepts the two connections

and creates two Receiver threads. So, there are up to two TCP connections between each pair of neighbors in the overlay network.

### **3.2.3 DeadlockMonitor Thread**

The DeadlockMonitor thread is used to detect deadlocks and timeouts. It periodically checks all sockets including incoming and outgoing sockets. If an incoming socket is not read for more than a certain time, it will create a dropper thread to ensure progress. If an outgoing socket is idle for more than a certain time, the socket is closed automatically. In this way, TCP deadlock is broken by dropping messages, and unused TCP connections are closed by timing out.

### **3.2.4 Timer Thread**

Timer thread is used to perform some timer functions.

### **3.2.5 Receiver Thread**

The Receiver thread receives byte arrays from an IncomingSocket object, and passes them to the I\_AdapterCallBack of an overlay socket. In this way, the received stream is passed to an object from class OL\_Socket. The overlay socket creates an OL\_Message from the stream, (possibly) forwards the message to its neighbors by writing the message to the outgoing sockets which connect to neighbors, stores the message in the OL\_ReceiverBuffer, or passes the message to the application's CallBack function. The number of Receiver threads can be as large as the number of neighbors. Messages received from different IncomingSockets by the Receiver thread may be forwarded to different OutgoingSockets.

### **3.2.6 Dropper thread**

The DeadlockMonitor thread checks all sockets periodically. If an incoming socket is not read for more than a certain time, it will create a dropper thread to ensure progress. In the Dropper thread, a single message is read from the associated IncomingSocket Object and then dropped. Some TCP deadlocks can be gracefully broken in this way.

---

### 3.3. UDP\_Multicast and Unicast Adapter Design

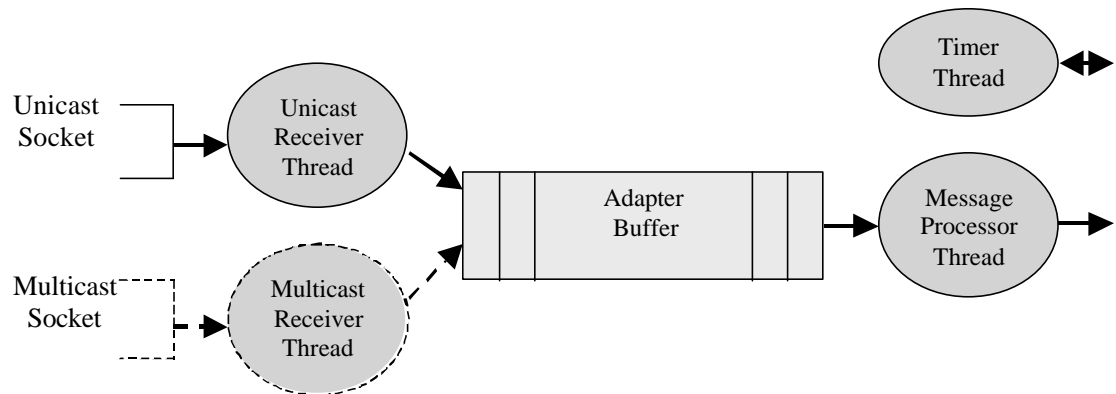


Figure 13. UDP\_Unicast and Multicast Adapter Structure

#### 3.3.1 Overview

UDP\_Multicast and UDP Unicast Adapters provide message transmission over UDP socket, and perform functions, such as, message buffering, message processing, and timer functions. As shown in Figure 13, an UDP\_UnicastAdapter contains three threads and a message buffer: Unicast receiver thread, Message processing thread, and a Timer thread. UDP\_MulticastAdapter added in a Multicast receiver thread. The message buffer stores the received messages from both Unicast and Multicast incoming socket.

#### 3.3.2 UnicastReceiver Thread

The UnicastReceiver thread receives DatagramPackets from an incoming DatagramSocket, calls the restoreMessage method of the AdapterCallBack to create a message from the received byte array, and stores the returned message in the AdapterBuffer.

```
I_Message restoreMessage(byte[] data, int start[], int end)
```

#### 3.3.3 MulticastReceiver Thread

The MulticastReceiver thread receives DatagramPackets from an incoming MulticastSocket, calls the restoreMessage method of the AdapterCallBack to create a message from the byte array, and stores the returned message in the AdapterBuffer.

#### 3.3.4 Message Processor Thread

This thread reads messages from the AdapterBuffer, processes the messages by calling the processMessage method in the AdapterCallBack, which is implemented either by the protocol node or the overlay socket to perform message processing.

```
void processMessage(I_Message msg)
```

#### 3.3.5 Timer Thread

The Timer thread provides a timer mechanism for the process that implements the AdapterCallBack interface. Whenever a time event expires, it calls the timerExpired method.

For example, HC\_Node implements timerExpired by changing the node's current state and sending messages to all neighbors.

```
void timerExpired(int timeID)
```

The Message Processor thread and the Timer thread together define the behavior of a protocol node which implements this AdapterCallBack interface.

### 3.2.6 Adapter Buffer

The Adapter Buffer is a FIFO buffer, which stores messages coming from the Multicast Receiver thread and the Unicast Receiver thread. If the buffer is full, an incoming message will be dropped. Since the overlay protocol node changes its state based on the messages read from the AdapterBuffer, messages in this buffer should be processed as fast as possible, otherwise the protocol behavior can be erratic due out-dated messages.

### 3.3.7 Design Issues of UDP\_Multicast and Unicast Adapters

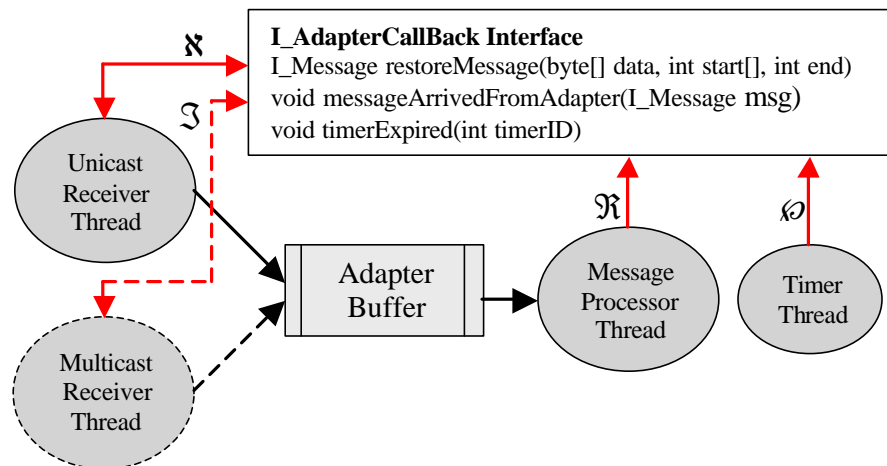


Figure 14. Threads behaviors in UDP\_Unicast and Multicast Adapter

An important design decision is, where to perform the parsing of the received DatagramPackets, which must be converted from a byte array to an OL Message. Note in Figure 14, that both the UnicastReceiver thread and the MulticastReceiver thread receive DatagramPackets from the incoming sockets. The conversion can be done either by the UnicastReceiver and MulticastReceiver thread, or by the MessageProcessor thread after the Adapter Buffer. To balance the load between these three threads, we implement the first solution. In this way, the MessageProcessor thread can process the message quicker. Note that the MessageProcessor thread runs the method of the OL\_Node and, possibly, must trigger the transmission of a protocol message. The procedures are shown in Figure 14:

1. When the UnicastReceiver thread receives a DatagramPacket, it first calls the `restoreMessage()` function in the AdapterCallBack interface to convert the datagram to the `I_Message` format, then stores the message in AdapterBuffer.
2. When the MulticastReceiver thread receives a DatagramPacket, it does the same steps.
3. The MessageProcessor thread reads a message out of the AdapterBuffer, and calls `messageArrivedFromAdapter()` function in the Adapter\_CallBack interface for processing.

4. The Timer thread reads a time event from a table. When the time expires, it calls the `timerExpired()` method in the `AdapterCallBack`.

Summarizing, the `UDP_Unicast` and `Multicast` Adapters have been designed in a general way. They are used for `DatagramPacket` transmission (sending and receiving), storing, and processing, but are not protocol or socket overlay specific, as they do not interpret or process the received `DatagramPacket`. All protocol specific functions are implemented by the `AdapterCallBack` interface. For example, in the `OL_Socket` design, both the protocol node and `OL_Socket` have adapters. Thus, both classes implement the `AdapterCallBack` interface.

---

## 4. Overlay Node Design

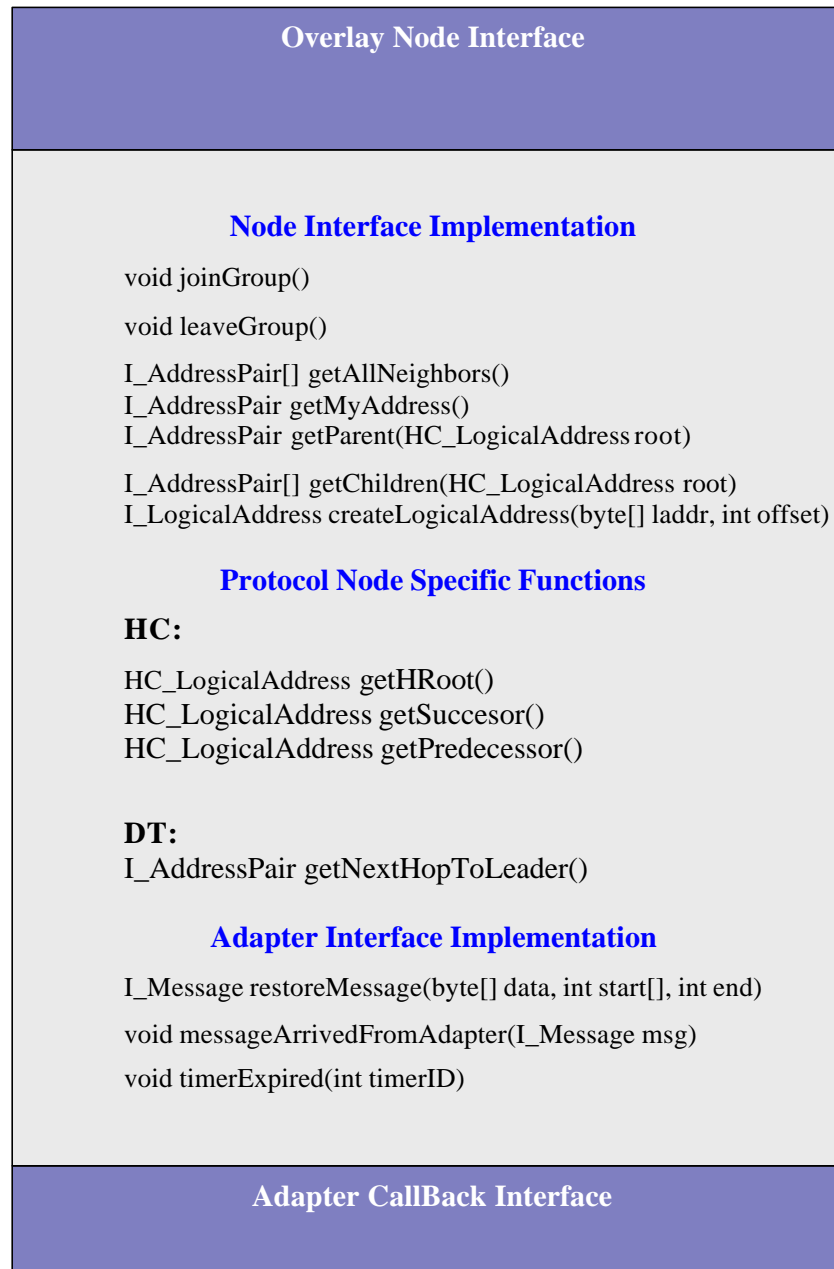


Figure 15. Overlay Node Structure

The overlay node (HC\_Node or DT\_Node) is the core of the overlay network protocol. The overlay node uses an adapter to exchange messages with neighbors in the overlay network to maintain the overlay topology, using HC or DT protocol messages. The overlay node defines the node's behavior. An overlay node provides two interface implementations for the overlay socket and the protocol adapter.

The overlay node has three main parts:

- (1) Node Interface Implementation
- (2) Hypercube or DT protocol specific functions
- (3) Adapter Interface Implementation

These parts will be discussed in the following sections.

#### 4.1. Node Interface (I\_Node) Implementation

Each overlay node has methods for joining and leaving an overlay, and for accessing the data about the node and its neighbors. Note that there is some similarity between methods of the I\_OverlaySocket interface and the I\_Node interface. In fact, when the method “joinGroup” of the overlay socket interface is called, it executes the “joinGroup” method of the overlay node interface.

##### Overlay Operations

void joinGroup()  
Defines the operations to join an overlay network

void leaveGroup()  
Defines the operations to leave an overlay network

##### Logical Address operations

I\_LogicalAddress createLogicalAddress(byte[] laddr, int offset)  
Creates a logical address from a byte array

I\_AddressPair getMyAddress()  
Returns the address pair of this node

I\_AddressPair getParent(HC\_LogicalAddress root)  
Returns the address pair of the parents of this node with respect to an embedded tree with given root

I\_AddressPair[] getChildren(HC\_LogicalAddress root)  
Returns the address pair of the children of this node with respect to an embedded tree with given root

I\_AddressPair[] getAllNeighbors()  
Returns a pair of addresses, i.e., PA/LA (= Physical address/Logical address), for all neighbors

#### 4.2. Protocol Node Specific Methods

The following are methods specific to the overlay protocol. The methods are concerned with access to overlay-specific logical addresses.

##### 4.2.1 Hypercube Node Specific Functions (HC\_Node)

HC\_LogicalAddress getHRoot()  
Returns the logical address of HRoot

HC\_LogicalAddress getSuccessor()  
Returns the logical address of the successor node, with respect to the Gray ordering of the logical addresses.

HC\_LogicalAddress getPredecessor()  
Returns the logical address of the predecessor node

#### 4.2.2 DT Node Specific Functions (DT\_Node)

I\_AddressPair getNextHopToLeader()

Retrieves the logical and physical address pair of the next hop to the leader with respect to compass routing

#### 4.3. AdapterCallback Interface (I\_AdapterCallback) Implementation

The role of the AdapterCallback was explained in Section 3.3.7. The methods of this interface are called by the node adaptors, when a message arrives, when a message has to be converted from a byte array to a protocol specific format, and when a timer expires.

I\_Message restoreMessage(byte[] data, int start[], int end)

Restores an OL\_Message (in a protocol specific format) from a byte array

void messageArrivedFromAdapter(I\_Message msg)

Defines the behavior of the node protocol when a message arrives

void timerExpired(int timerID)

Defines the behavior of the node when a timer expires

---

## 5. Forwarding Engine

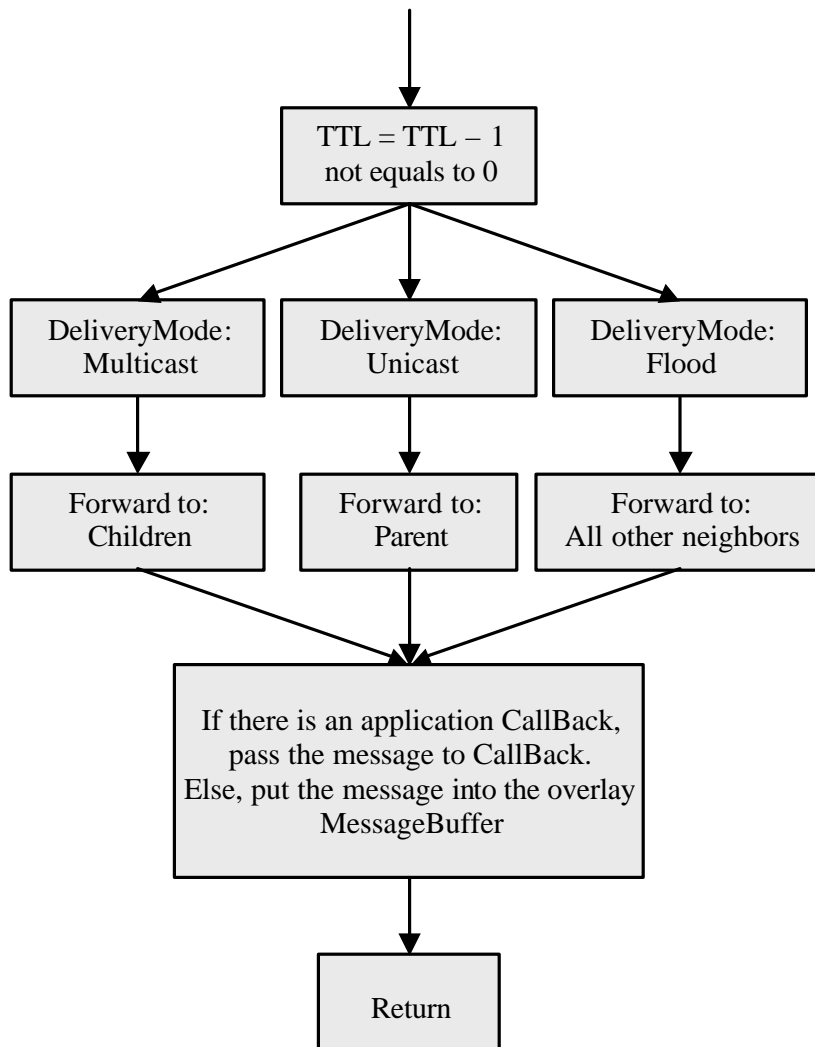


Figure 16. Forwarding Engine Structure.

The Forwarding Engine is a set of methods in the overlay socket which has two main functions: (1) Place incoming messages into the ReceiverBuffer or, if a Callback is provided, pass the message to the application provided Callback for processing; and (2) Determine if the message must be forwarded to other overlay sockets, and forward the message to the neighbors in the overlay network.

The forwarding decisions of the Forwarding Engine are entirely based on the logical addresses of a node. The physical address, generally an IP address and a UDP or TCP port are used to transmit the message to the neighbor. Note that the logical and physical addresses of a node are kept by the overlay node (DT\_Node or HC\_Node). This table is updated whenever the overlay topology changes and the information in the table is used for message routing and transmission.

The Forwarding Engine inspects fields in the header of an overlay message to determine if and how the message is to be forwarded. The following fields are inspected:

- Delivery Mode (Dmd), which is unicast, multicast, or flood,
- Source Logical address (Src LA), and
- Destination Logical address (Dest LA, optional for multicast).

The forwarding functions are explained in Figures 16 and 17. A unicast message is forwarded to the parent in an embedded tree that has Dest LA as root, a multicast message is forwarded to all children nodes in an embedded tree that has Src LA as the root of the tree, and a flood message is forwarded to all neighbors except the node from which the message arrived. Refer to the design document for a discussion how packets are forwarded in the overlay networks along trees which are embedded in the overlay network.

---

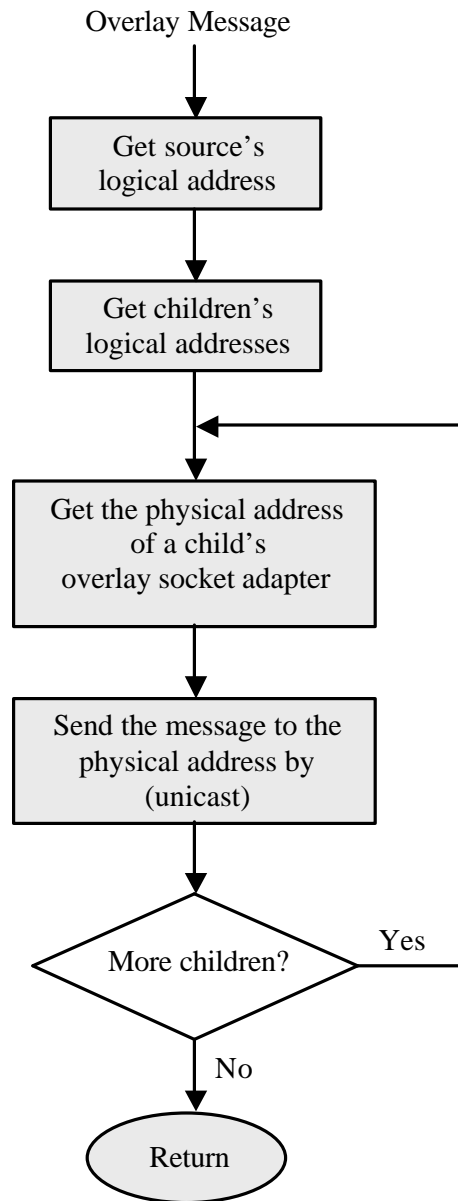


Figure 17. Forwarding to Children nodes of Multicast messages.

## 6. Statistics Interface

### 6.1. Overview

The statistics interface (I\_Stats) provides access to control and monitoring information of an object that implements this interface. The I\_Stats interface is implemented by all main components of an overlay socket. Currently, the following classes implement I\_Stats:

- overlay node,
- receive buffer,
- adapters,
- overlay socket configuration,
- send buffer, message store (in the future).

The design of the control and monitoring architecture of HyperCast 2.0 is described in the design document.

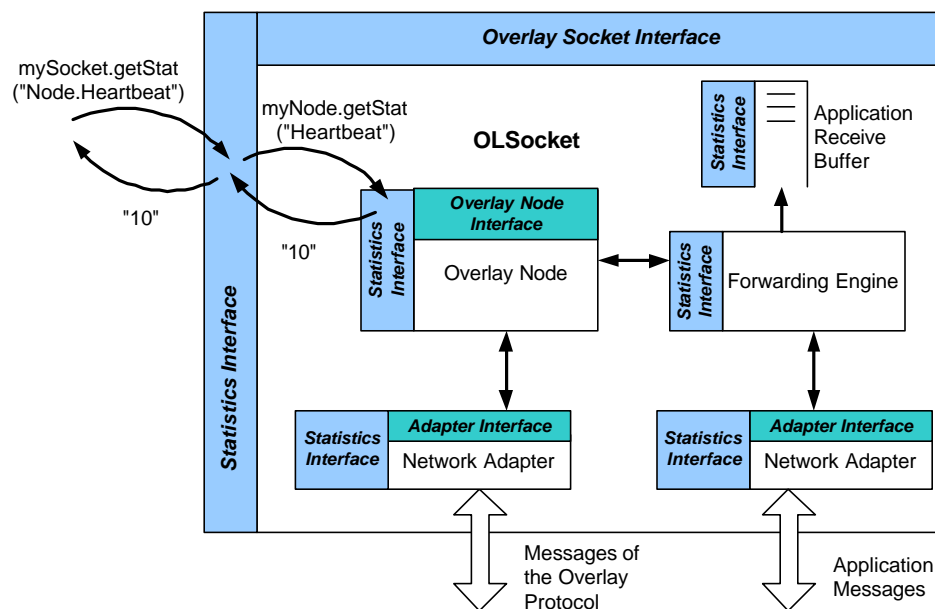


Figure 18. Statistics Interfaces.

### 6.2. Names of Statistics

All statistics have a name and a value, both of which are strings. The naming scheme for statistics is hierarchical. For example, (“mySocket.Node.Heartbeat”, “20”) indicates that 20 is the value of Heartbeat in the overlay node component of an overlay socket. In Figure 18, we illustrate how the hierarchical organization is exploited when accessing statistics.

- A call `mySocket.getStat("Node.Heartbeat")` to the overlay socket requests the value of the statistics "Heartbeat" at in the overlay node component of an overlay socket with name "mySocket".
- The `getStat` implementation of the overlay socket executes the requests by calling a method `getStat("myNode.Heartbeat")`, where `myNode` is the name of the overlay object in `mySocket`. This method returns the value, say "10", to the `getStat` method of `mySocket`. Upon receiving the value, the `mySocket.getStat` returns the value "10" to the calling method.

Figure 19 shows the hierarchical structure of statistics names that are used in the overlay socket. The set of all statistics of the overlay socket is given in Appendix I of the HyperCast 2.0 design document.

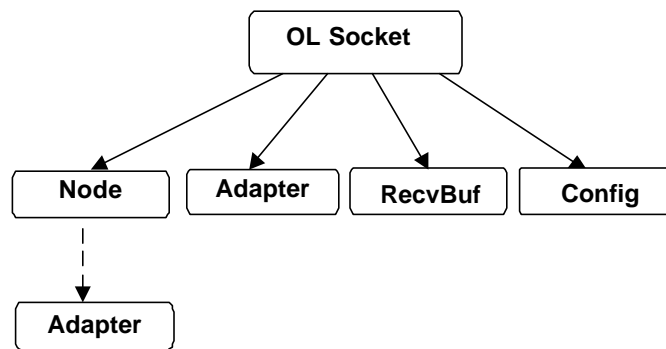


Figure 19. The OL Socket Naming Space

### 6.3. XML Schema

Each object, which implements the `I_Stats` interface maintains a description of all its statistics, and their access form, in the form of an XML schema. The XML schema can be retrieved with the method `GetSchema` of the `I_Stats` interface. The XML schema describes the statistics supported by this object and all its sub-components. The schema contains a description of the statistics' names, the types of the statistics' values and the list of statistics that can be modified. The description is a partial XML schema which assumes that the namespace "xsd" is bound to <http://www.w3.org/2000/10/XMLSchema> and these definitions are in force:

```

<xsd:simpleType name="IPv4AndPort">
  <xsd:restriction base="xsd:string">
    <xsd:pattern value="\d+\.\d+\.\d+\.\d+\/\d+" />
  </xsd:restriction>
</xsd:simpleType>

<xsd:complexType name="Empty">
</xsd:complexType>

<xsd:attributeType name="arrayIndex" use="required"
  type="nonnegativeInteger">
</xsd:attributeType>
  
```

## 6.4. Implementation of I\_Stats interface

Given the name of a statistic, the value of the statistics can be obtained and (possibly) modified through the I\_Stats interface. The I\_Stats interface supports three methods:

- `getStats()` for accessing statistics,
- `setStats()` for modifying the value of a statistics, and
- `getSchema()` to obtain an XML scheme which describes the statistics supported by the component and its subcomponent.

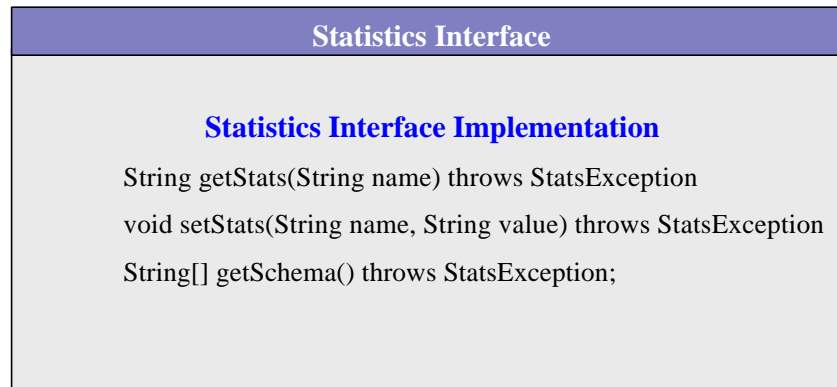


Figure 20. Implementation of Statistics Interface.

`String getStats(String sname) throws StatsException;`  
 Gets the statistics value of a statistics with name "sname".

`void setStats(String sname, String svalue) throws StatsException;`  
 Sets statistic's (named "sname") value to "svalue".

`String[] getSchema() throws StatsException;`  
 Gets statistics description in the form of an XML Schema from an object which implements this interface. The returned array of strings contains 3 strings. The first string states the arguments which can be passed to `getStats`. The second string specifies the arguments types of values that can be passed to `setStats`. The third string specifies the names and values returned by `getStats`.

## 6.5. Implementation of StatsException

The `StatsException` object handles exceptions raised when accessing or setting statistics values. The exception is constructed with the type of the exception and the name of the statistics to be accessed. The `StatsException` object gives the description of the exception based on the type of the exception. Table 1 lists the exception types and their related descriptions.

Table1: Types and description of the exceptions when accessing statistics.

| Type of Exception (integer) | Description of the Exception  |
|-----------------------------|---|
| 1                           | Setting the value of a statistics which is not managed by the object;   |
| 2                           | Setting the value of a statistics which is read only;   |
| 3                           | Schema violation;   |
| 4                           | The value assigned to a statistics is of an inappropriate type;   |
| 5                           | The value assigned to a statistics is inappropriate for some other reasons (e.g. out of range, file doesn't exist, etc.); |
| All the other integers      | Other internal errors;  |

String getMessage()

Returns the description of the exception occurred.

## 7. Overlay Management

### 7.1. Overview

The overlay management is implemented outside the OL\_Socket. An overlay network is identified by a global unique identifier, called Overlay ID, and each overlay network is associated with a set of attributes. In our current implementation, attributes of an overlay network can be obtained either by

- reading the attributes from a configuration file, or
- by querying the attributes from an overlay server.

The obtained attributes are stored in a configuration object, which is then used to create an overlay socket belonging to the overlay network.

### 7.2. Structure

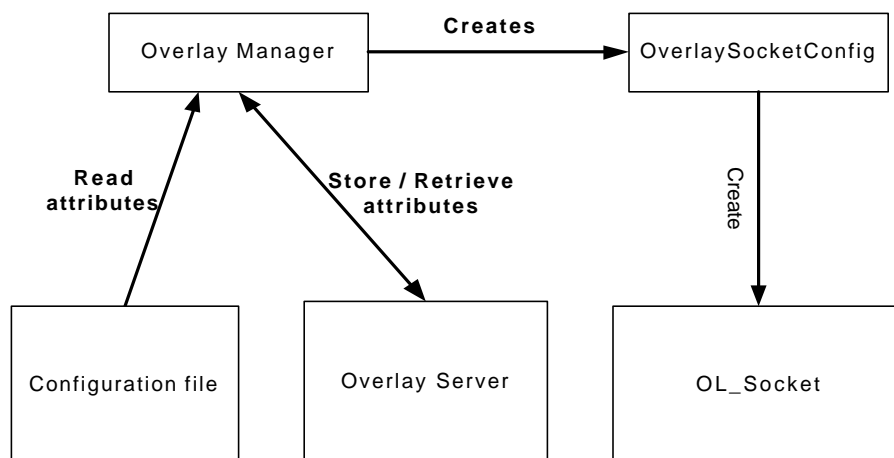


Figure 21. Structure of Overlay Management

### 7.3. Objects

#### 7.3.1 Overlay Manager

The OverlayManager provides an interface between an application program and the management of overlay networks. The OverlayManager reads the configuration file and decides based on the value of the attribute “OverlayManager” whether to obtain the overlay attributes from a file or from an overlay server (A file is used if OverlayManager= “”, and a server is used if OverlayManager = HTTP). The description of the overlay socket API has a detailed discussion about using the OverlayManager.

The main methods of the OverlayManager include:

**(1) Create Overlay Operation:**

OverlaySocketConfig createOverlay(String overlayID);

Creates a new OverlaySocketConfig object based on the overlay ID. If the overlay ID does not exist, a new overlay network is created

boolean doesOverlayExist(String overlayID);

Returns true if the overlay already exists

### **(2) Create OverlaySocketConfig Object Operation:**

OverlaySocketConfig getOverlayConfig();

Returns attributes of the existing overlay

### **(3) Overlay Attributes Operations:**

String getDefaultProperty();

Returns the default attributes

String getKeyAttributes();

Returns the major attributes

Properties OverrideDefault();

Overwrites the default attributes with the attributes obtained from the server

void setDefaultProperty();

Sets default attributes.

## 7.3.2 Overlay Server

The overlay server is a simple web server that handles CGI requests. These requests include operations to

- create an overlay network;
- test if an overlay network exists, and
- obtain the attributes of an existing overlay network.

The overlay server has two threads. One thread is used to process all connections from HTTP clients and the other thread is used to process all CGI requests.

Overlay attributes are stored in a hash table indexed with the overlay ID.

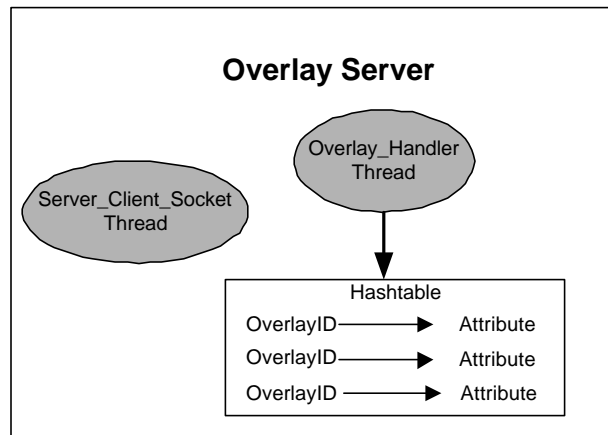


Figure 22. Structure of the Overlay Server.

---

The Hash table stores the overlay ID and the corresponding attributes. The Overlay\_Handler thread handles overlay functions including those to create overlays and test if overlays exist. The Server\_Client\_Socket thread handles HHTP requests.

### 7.3.3 Configuration object (OverlaySocketConfig)

The OverlaySocketConfig is class that stores the Overlay ID and the attributes related to the overlay. The OverlaySocketConfig class is used to create new OL\_sockets with the stored attributes.

The OverlaySocketConfig can generate a so-called “overlay hash” for the Overlay ID. The protocols of the HC and DT protocols use this hash in their protocol messages. The hash gives a protocol means to decide whether a received message belongs to a network with a given overlay ID.

I\_OverlaySocket createOverlaySocket(I\_Callback callback)

Returns a new OL\_Socket based on key attributes of an overlay.

int generateOverlayHash()

Generates the hash code based on the OverlayID.

---