Programming Overlay Networks with Overlay Sockets[*]

Jörg Liebeherr, Jianping Wang and Guimin Zhang

Department of Computer Science, University of Virginia, Charlottesville, VA 22904, USA
{jorg, jwang, gz8d}@cs.virginia.edu

Abstract. The emergence of application-layer overlay networks has inspired the development of new network services and applications. Research on overlay networks has focused on the design of protocols to maintain and forward data in an overlay network, however, less attention has been given to the software development process of building application programs in such an environment. Clearly, the complexity of overlay network protocols calls for suitable application programming interfaces (APIs) and abstractions that do not require detailed knowledge of the overlay protocol, and, thereby, simplify the task of the application programmer. In this paper, we present the concept of an *overlay socket* as a new programming abstraction that serves as the end point of communication in an overlay network. The overlay socket provides a socket-based API that is independent of the chosen overlay topology, and can be configured to work for different overlay topologies. The overlay socket can support application data transfer over TCP, UDP, or other transport protocols. This paper describes the design of the overlay socket and discusses API and configuration options.

Key words: Overlay Networks, Application-layer Multicast, Overlay Network Programming.

1 Introduction

Application-layer overlay networks [5, 9, 13, 17] provide flexible platforms for developing new network services [1, 10, 11, 14, 18–20] without requiring changes to the network-layer infrastructure. Members of an overlay network, which can be hosts, routers, servers, or applications, organize themselves to form a logical network topology, and communicate only with their respective neighbors in the overlay topology. A member of an overlay network sends and receives application data, and also forwards data intended for other members.

This paper addresses application development in 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

[*] This work is supported in part by the National Science Foundation through grant ANI-0085955.

network. The diversity and complexity of building and maintaining overlay networks make it impractical to assume that application developers can be concerned with the complexity of managing the participation of an application in a specific overlay network topology.

We present a software module, called *overlay socket*, that intends to simplify the task of overlay network programming. The design of the overlay socket pursues the following set of objectives: First, the application programming interface (API) of the overlay socket does not require that an application programmer has knowledge of the overlay network topology. Second, the overlay socket is designed to accommodate different overlay network topologies. Switching to different overlay network topologies is done by modifying parameters in a configuration file. Third, the overlay socket, which operates at the application-layer, can accommodate different types of transport layer protocols. This is accomplished by using *network adapters* that interface to the underlying transport layer network and perform encapsulation and de-encapsulation of messages exchanged by the overlay socket. Currently available network adapters are TCP, UDP, and UDP multicast. Fourth, the overlay socket provides mechanisms for bootstrapping new overlay networks.

In this paper, we provide an overview of the overlay socket design and discuss overlay network programming with the overlay socket. The overlay socket has been implemented in Java as part of the HyperCast 2.0 software distribution [12]. The software has been used for various overlay applications, and has been tested in both local-area as well as wide-area settings. The HyperCast 2.0 software implements the overlay topologies described in [15] and [16]. This paper highlights important issues of the overlay socket, additional information can be found in the design documentation available from [12].

Several studies before us have addressed overlay network programming issues. Even early overlay network proposals, such as Yoid [9], Scribe [4], and Scattercast [6], have presented APIs that aspire to achieve independence of the API from the overlay network topology used. Particularly, Yoid and Scattercast use a socket-like API, however, these APIs do not address issues that arise when the same API is used by different overlay network topologies. Several works on application-layer multicast overlays integrate the application program with the software responsible for maintaining the overlay network, without explicitly providing general-purpose APIs. These include Narada [5], Overcast [13], ALMI [17], and NICE [2]. A recent study [8] has proposed a common API for the class of so-called *structured overlays*, which includes Chord [19], CAN [18], and Bayeux [20], and other overlays that were originally motivated by distributed hash tables. Our work has a different emphasis than [8], since we assume a scenario where an application programmer must work with several, possibly fundamentally different, overlay network topologies and different transmission modes (UDP, TCP), and, therefore, needs mechanisms that make it easy to change the configuration of the underlying overlay network.

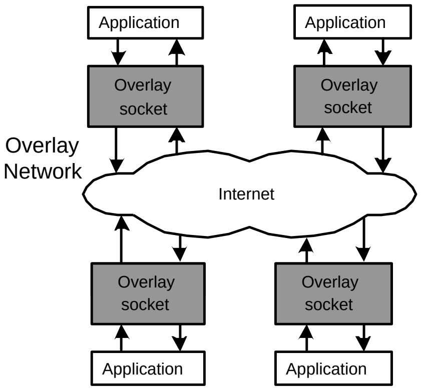

Fig. 1. The overlay network is a collection of overlay sockets.

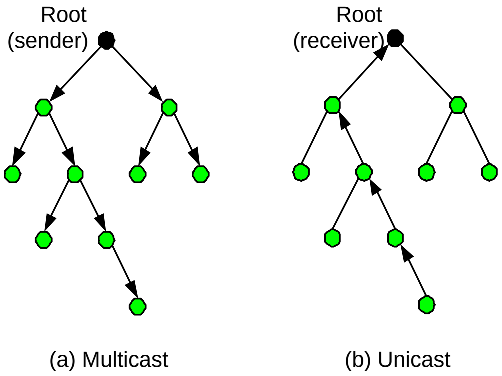

Fig. 2. Data forwarding in overlay networks.

The rest of the paper is organized as following. In Section 2 we introduce concepts, abstractions, and terminology needed for the discussion of the overlay socket. In Section 3 we present the design of the overlay socket, and discuss its components. In Section 4 we show how to write programs using the overlay socket. We present brief conclusions in Section 5.

2 Basic Concepts

An *overlay socket* is an endpoint for communication in an overlay network, and an overlay network is seen as a collection of overlay sockets that self-organize using an overlay protocol (see Figure 1). An overlay socket offers to an application programmer a Berkeley socket-style API [3] for sending and receiving data over an overlay network. Each overlay socket executes an *overlay protocol* that is responsible for maintaining the membership of the socket in the overlay network topology.

Each overlay socket has a *logical address* and a *physical address* in the overlay network. The logical address is dependent on the type of overlay protocol used. In the overlay protocols currently implemented in HyperCast 2.0, the logical addresses are 32-bit integers or (x,y) coordinates, where x and y are positive 32-bit positive integers. The physical address is a transport layer address where overlay sockets receive messages from the overlay network. On the Internet, the physical address is an IP address and a TCP or UDP port number. Application programs that use overlay sockets only work with logical addresses, and do not see physical addresses of overlay nodes.

When an overlay socket is created, the socket is configured with a set of configuration parameters, called *attributes*. The application program can obtain the attributes from a configuration file or it downloads the attributes from a server. The configuration file specifies the type of overlay protocol and the type of transport protocol to be used, but also more detailed information such as the size of internal buffers, and the value of protocol-specific timers. The most important attribute is the *overlay identifier* (overlay ID) which is used as a global identifier for an overlay network and which can be used as a key to access the other attributes of the overlay network. Each new overlay ID corresponds to the creation of a new overlay network.

Overlay sockets exchange two types of messages, *protocol messages* and *application messages*. Protocol messages are the messages of the overlay protocol that maintain the overlay topology. Application messages contain application-data that is encapsulated in an overlay message header. An application message uses logical addresses in the header to identify source and, for unicast, the destination of the message. If an overlay socket receives an application message from one of its neighbors in the overlay network, it determines if the message must be forwarded to other overlay sockets, and if the message needs to be passed to the local application. The transmission modes currently supported by the overlay sockets are unicast, and multicast. In multicast, all members in the overlay network are receivers. In both unicast and multicast, the common abstraction for data forwarding is that of passing data in spanning trees that are embedded in the overlay topology. For example, a multicast message is transmitted downstream a spanning tree that has the sender of the multicast message as the root (see Figure 2(a)). When an overlay socket receives a multicast message, it forwards the message to all of its downstream neighbors (children) in the tree, and passes the message to the local application program. A unicast message is transmitted upstream a tree with the receiver of the message as the root (see Figure 2(b)). An overlay socket that receives a unicast message forwards the message to the upstream neighbor (parent) in the tree that has the destination as the root.

An overlay socket makes forwarding decisions locally using only the logical addresses of its neighbors and the logical address of the root of the tree. Hence, there is a requirement that each overlay socket can locally compute its parent and its children in a tree with respect to a root node. This requirement is satisfied by many overlay network topologies, including [15, 16, 18–20].

3 The Components of an Overlay Socket

An overlay socket consists of a collection of components that are configured when the overlay socket is created, using the supplied set of attributes. These components include the overlay protocol, which helps to build and maintain the overlay network topology, a component that processes application data, and interfaces to a transport-layer network. The main components of an overlay socket, as illustrated in Figure 3, are as follows:

- The *overlay node* implements an overlay protocol that establishes and maintains the overlay network topology. The overlay node sends and receives overlay protocol messages, and maintains a set of timers. The overlay node is the only component of an overlay socket that is aware of the overlay topology. In the HyperCast 2.0

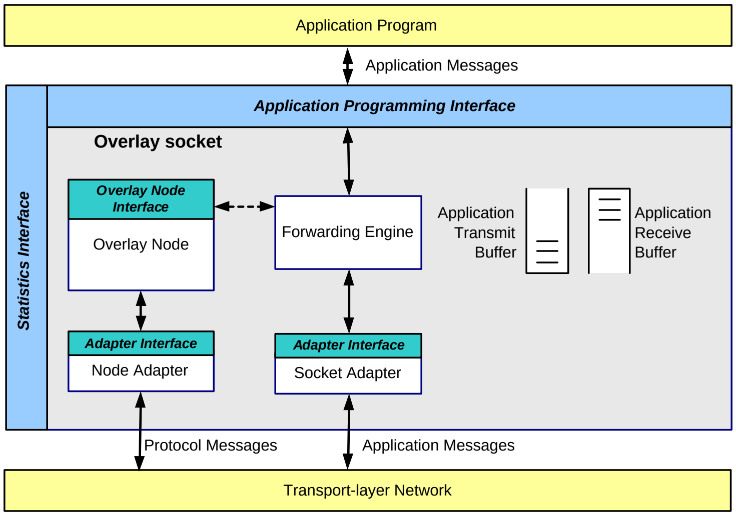

Fig. 3. Components of an overlay socket.

software, there are overlay nodes that build a logical hypercube [15] and a logical Delaunay triangulation [16].

- The *forwarding engine* performs the functions of an application-layer router, that sends, receives, and forwards formatted application-layer messages in the overlay network. The forwarding engine communicates with the overlay node to query next hop routing information for application messages. The forwarding decision is made using logical addresses of the overlay nodes.
- Each overlay socket has two network adapters that each provides an interface to transport-layer protocols, such as TCP or UDP. The *node adapter* serves as the interface for sending and receiving overlay protocol messages, and the *socket adapter* serves as the interface for application messages. Each adapter has a transport level address, which, in the case of the Internet, consists of an IP address and a UDP or TCP port number. Currently, there are three different types of adapters, for TCP, UDP, and UDP multicast. Using two adapters completely separates the handling of messages for maintaining the overlay protocol and the messages that transport application data.
- The *application receive buffer* and *application transmit buffer* can temporarily store messages that, respectively, have been received by the socket but not been delivered to the application, or that have been released by the application program, but not been transmitted by the socket. The application transmit buffer can play a role when messages cannot be transmitted due to rate control or congestion control constraints[1].

[1] The application transmit buffer is not implemented in the HyperCast 2.0 software.

– Each overlay socket has two external interfaces. The *application programming interface* (API) of the socket offers application programs the ability to join and leave existing overlays, to send data to other members of the overlay network, and receive data from the overlay network. The *statistics interface* of the overlay socket provides access to status information of components of the overlay socket, and is used for monitoring and management of an overlay socket. Note in Figure 3 that some components of the overlay socket also have interfaces, which are accessed by other components of the overlay socket.

The *overlay manager* is a component external to the overlay socket (and not shown in Figure 3). It is responsible for configuring an overlay socket when the socket is created. The overlay manager reads a configuration file that stores the attributes of an overlay socket, and, if it is specified in the configuration file, may access attributes from a server, and then initiates the instantiation of a new overlay socket.

4 Overlay Network Programming

An application developer does not need to be familiar with the details of the components of an overlay socket as described in the previous section. The developer is exposed only to the API of the overlay socket and to a file with configuration parameters. The configuration file is a text file which stores all attributes needed to configure an overlay socket. The configuration file is modified whenever a change is needed to the transport protocol, the overlay protocol, or some other parameters of the overlay socket. In the following, we summarize only the main features of the API, and we refer to [12] for detailed information on the overlay socket API.

4.1 Overlay Socket API

Since the overlay topology and the forwarding of application-layer data is transparent to the application program, the API for overlay network programming can be made simple. Applications need to be able to create a new overlay network, join and leave an existing overlay network, send data to and receive data from other members in the overlay.

The API of the overlay socket is message-based, and intentionally stays close to the familiar Berkeley socket API [3]. Since space considerations do not permit a description of the full API, we sketch the API with the help of a simplified example. Figure 4 shows the fragment of a Java program that uses an overlay socket. An application program configures and creates an overlay socket with the help of an overlay manager (*om*). The overlay manager reads configuration parameters for the overlay socket from a configuration file (*hypercast.prop*), which can look similarly as shown in Figure 5. The application program reads the overlay ID with command *om.getDefaultProperty("OverlayID")* from the file, and creates an configuration object (*config*) for an overlay socket with the

```
  // Generate the configuration object
OverlayManager om = new
OverlayManager("hypercast.prop");
String MyOverlay =
  om.getDefaultProperty("OverlayID");
OverlaySocketConfig config =
    new om.getOverlaySocketConfig(MyOverlay);
  // create an overlay socket
OL_Socket socket =
  config.createOverlaySocket(callback);
  // Join an overlay
socket.joinGroup();
  // Create a message
OL_Message msg = socket.createMessage(byte[]
data, int length);
  // Send the message to all members in overlay network
socket.sendToAll(msg);
  // Receive a message from the socket
OL_Message msg = socket.receive();
```

Fig. 4. Program with overlay sockets.

```
  # OVERLAY Server:
OverlayServer =
  # OVERLAY ID:
OverlayID = 1234
KeyAttributes= Socket,Node,SocketAdapter
  # SOCKET:
Socket = HCast2-0
HCAST2-0.TTL = 255
HCAST2-0.ReceiveBufferSize = 200
  # SOCKET ADAPTER:
SocketAdapter = TCP
SocketAdapter.TCP.MaximumPacketLength = 16384
  # NODE:
Node = DT2-0
DT2-0.SleepTime = 400
  # NODE ADAPTER:
NodeAdapter = NodeAdptUDPServer
NodeAdapter.UDP.MaximumPacketLength = 8192
NodeAdapter.UDPServer.UdpServer0 =
128.143.71.50:8081
```

Fig. 5. Configuration file (simplified).

given overlay ID. The configuration object also loads all configuration information from the configuration file, and then creates the overlay socket (*config.createOverlaySocket*). Once the overlay socket is created, the socket joins the overlay network (*socket.joinGroup*). When a socket wants to multicast a message, it instantiates a new message (*socket.createMessage*) and transmits the message using the *sendToAll* method. Other transmission options are *sendToParent*, *sendToChildren*, *sendToNeighbors*, and *sendToNode*, which, respectively, send a message to the upstream neighbor with respect to a given root (see Figure 2), to the downstream neighbors, to all neighbors, or to a particular node with a given logical address.

4.2 Overlay Network Properties Management

As seen, the properties of an overlay socket are configured by setting attributes in a configuration file. The overlay manager in an application process uses the attributes to create a new overlay socket. By modifying the attributes in the configuration file, an application programmer can configure the overlay protocol or transport protocol that is used by the overlay socket. Changes to the file must be done before the socket is created. Figure 5 shows a (simplified) example of a configuration file. Each line of the configuration file assigns a value to an attribute. The complete list of attributes and the range of values is documented in [12]. Without explaining all entries in Figure 5, the file sets, among others, the overlay ID to *'1234'*, selects version 2.0 of the DT protocol as overlay protocol (*'Node=DT2-0'*), and it sets the transport protocol of the socket adaptor to TCP (*'SocketAdapter=TCP'*).

Each overlay network is associated with a set of attributes that characterize the properties of the overlay sockets that participate in the overlay network. As mentioned earlier, the most important attribute is the overlay ID, which is used to identify an overlay network, and which can be used as a key to access all other attributes of an overlay network. The overlay ID should be a globally unique identifier.

A new overlay network is created by generating a new overlay ID and associating a set of attributes that specify the properties of the overlay sockets in the overlay network. To join an overlay network, an overlay socket must know the overlay ID and the set of attributes for this overlay ID. This information can be obtained from a configuration file, as shown in Figure 5.

All attributes have a name and a value, both of which are strings. For example, the overlay protocol of an overlay socket can be determined by an attribute with name *NODE*. If the attribute is set to *NODE=DT2-0*, then the overlay node in the overlay socket runs the DT (version 2) overlay protocol. The overlay socket distinguishes between two types of attributes: *key attributes* and *configurable attributes*. Key attributes are specific to an overlay network with a given overlay ID. Key attributes are selected when the overlay ID is created for an overlay network, and cannot be modified afterwards. Overlay sockets that participate in an overlay network must have identical key attributes, but can have different configurable attributes. The attributes *OverlayID* and *KeyAttributes* are key attributes by default in all overlay networks. Configurable attributes specify parameters of an overlay socket, which are not considered essential for establishing communication between overlay sockets in the same overlay network, and which are considered ‘tunable’.

5 Conclusions

We discussed the design of an *overlay socket* which attempts to simplify the task of overlay network programming. The overlay socket serves as an end point of communication in the overlay network. The overlay socket can be used for various overlay topologies and support different transport protocols. The overlay socket supports a simple API for joining and leaving an overlay network, and for sending and receiving data to and from other sockets in the overlay network. The main advantage of the overlay socket is that it is relatively easy to change the configuration of the overlay network.

An implementation of the overlay socket is distributed with the HyperCast2.0 software. The software has been extensively tested. A variety of different applications, such as distributed whiteboard and a video streaming application, have been developed with the overlay sockets.

Acknowledgement. In addition to the authors of this article the contributors include Bhupinder Sethi, Tyler Beam, Burton Filstrup, Mike Nahas, Dongwen Wang, Konrad Lorincz, Jean Ablutz, Haiyong Wang, Weisheng Si, Huafeng Lu, and Guangyu Dong.

References

1. D. G. Andersen, H. Balakrishnan, M. F. Kaashoek, and R. T., Morris. Resilient overlay networks. In *Proceedings of the 18th ACM Symposium on Operating Systems Principles*, pp.

131-145, Lake Luise, Canada, October 2001.

2. S. Banerjee, B. Bhattacharjee, and C. Kommareddy. Scalable Application Layer Multicast. In *Proceedings of ACM SIGCOMM*, pp. 205-220, Pittsburgh, PA, August 2002.
3. K. L. Calvert, M. J. Donhahoo. TCP/IP Sockets in Java: Practical Guide for Programmers. *Morgan Kaufman*, October 2001.
4. M. Castro, P. Druschel, A-M. Kermarrec and A. Rowstron. SCRIBE: A large-scale and decentralized application-level multicast infrastructure. *IEEE Journal on Selected Areas in Communications (JSAC)*, Vol. 20, No. 8, October 2002.
5. Y. Chu, S. G. Rao, and H. Zhang. A case for end system multicast. In *Proceedings of ACM SIGMETRICS*, pp. 1-12, Santa Clara, CA, June 2000.
6. Y. D. Chawathe. Scattercast: An Architecture for Internet Broadcast Distribution as an Infrastructure Service. *Ph.D. Thesis, University of California, Berkeley*, December 2000.
7. Y. Chu, S. G. Rao, S. Seshan and H. Zhang. Enabling Conferencing Applications on the Internet using an Overlay Multicast Architecture. In *Proceedings of ACM SIGCOMM*, pp. 55-67, San Diego, CA, August 2001.
8. F . Dabek, B. Zhao, P. Druschel, J. Kubiatowicz, and I. Stoica. Towards a Common API for Structured Peer-to-Peer Overlays. In *Proceedings of the 2nd International Workshop on Peer-to-Peer Systems (IPTPS '03)*, Berkeley, CA, February 2003.
9. P. Francis. Yoid: Extending the Internet multicast architecture, Unpublished paper, April 2000. Available at http://www.aciri.org/yoid/docs/index.html.
10. *The FreeNet Project*. http://freenetproject.org.
11. *The Gnutella Project*. http://www.gnutella.com.
12. *The HyperCast project*. http://www.cs.virginia.edu/~hypercast.
13. J. Jannotti, D. K. Gifford, K. L. Johnson, M. F. Kaashoek, and J. OToole. Overcast: Reliable multicasting with an overlay network. In *Proceedings of the Fourth Symposium on Operating Systems Design and Implementation*, pp. 197-212, San Diego, CA, October 2000.
14. *The JXTA Project*. http://www.jxta.org.
15. J. Liebeherr and T. K. Beam. HyperCast: A protocol for maintaining multicast group members in a logical hypercube topology. In *Proceedings of First International Workshop on Networked Group Communication (NGC 99)*, In Lecture Notes in Computer Science, Vol. 1736, pp. 72-89, Pisa, Italy, November 1999.
16. J. Liebeherr, M. Nahas, and W. Si. Application-layer multicasting with Delaunay triangulation overlays. *IEEE Journal on Selected Areas in Communications*, Vol. 20, No. 8, October 2002.
17. D. Pendarakis, S. Shi, D. Verma, and M. Waldvogel. ALMI: An application level multicast infrastructure. In *Proceedings of 3rd Usenix Symposium on Internet Technologies and Systems*, pp. 49-60, San Francisco, CA, March 2001.
18. S. Ratnasamy, P. Francis, M. Handley, R. Karp, and S. Shenker. A Scalable Content-Addressable Network. In *Proceedings of ACM SIGCOMM*, pp. 161-172, San Diego, CA, August 2001.
19. I. Stoica, R. Morris, D. Karger, F. Kaashoek, and H. Balakrishnan. Chord: A Scalable Peer-To-Peer Lookup Service for Internet Applications. In *Proceedings of ACM SIGCOMM*, pp. 149-160, San Diego, CA, August 2001.
20. S. Q. Zhuang, B. Y. Zhao, A. D. Joseph, R. H. Katz, and J. Kubiatowicz. Bayeux: An Architecture for Scalable and Fault-tolerant Wide-Area Data Dissemination. In *Proceedings of the Eleventh International Workshop on Network and Operating System Support for Digital Audio and Video, (NOSSDAV 2001)*, pp. 11-20, Port Jefferson, NY, January 2001.