“Hybrid Transmit” – Engineering a Hybrid TCP Unicast and IP Multicast Content Delivery Platform

Bradford W. Galiette
CPSC 490a Senior Project
December 14, 2007

Advisor – Yang R. Yang
Department of Computer Science
Yale University

Abstract

This project suggests an approach for combining the “on-demand” functionality of TCP unicast services with the efficiencies and potential of readily-advancing IP multicast standards to reduce the amount of bandwidth required to transmit high-demand material to large numbers of clients. A protocol for this kind of architecture – dubbed “Hybrid Transmit” – is proposed, implemented in Java, and then subject to rigorous testing to determine both feasibility of the concept and to determine the extent of any reductions in transmission costs. The preliminary results of these tests indicate an effective solution to what is apt to be an increasing problem of network congestion resulting from sending multiple copies of identical, large-sized content simultaneously.

Background

Bandwidth-intensive on-line content has grown exponentially in recent years with the advent of popular media-sharing websites. The paradigm of broadcasting high-demand, professionally-created content (e.g., TV series, movies, etc.) over IP networks stands to markedly increase this already high degree of network utilization, potentially posing structural – and, indeed, economical – limits on content availability. While UDP
multicast applications have been sufficient for reducing bandwidth loads associated with live content (so-called “one-to-many” streams), the same cannot be directly applied to video-on-demand and equivalent applications where individual consumer start and watch times vary from client to client. In these cases, content is transmitted in so-called “one-to-one” fashion over TCP. Nonetheless, with the advent of professionally-created, high-demand, licensed content being made available for retrieval over IP networks, it is foreseeable that high-demand content could be transmitted more efficiently to clients by consolidating the number of times the content is transmitted.

Hybrid Transmit offers an application-layer solution for the issue to serve a wide variety of needs ranging from streaming applications to file transfer services. Three separate server applications allow clients to query content by way of conventional-appearing URLs and to retrieve data simultaneously through both unicast and multicast transmissions. A central directory server coordinates client requests for different kinds of material. The data sources themselves (which multiple clients presumably have an interest to download at any given time) are transmitted repeatedly over unique multicast groups. Individual clients pair this data with bytes from unicast transmissions of the same content to allow for true on-demand access that also provides significant bandwidth efficiencies from avoiding one-to-one disbursement of a large amount of the original source.

For the purpose of this project, we implement both a Java framework for the protocol (detailed below) as well as a series of small driver applications to simulate one client downloading a single source, multiple clients downloading a single source, and multiple clients downloading multiple sources. Testing is done primarily on local networks, with some efforts to simulate more distant client/server connections.

**Network Considerations**

Hybrid Transmit is deliberately designed to support a high level of distributed functionality for use in high-volume settings. To that end, the different components are
compartmentalized in such a way so that they can be spread over as few or as many servers and networks as possible.

The general case involves a scenario of several clients viewing the same piece of content (albeit at different start times) wherein all the distributable server components are hosted on different systems:

In short, the protocol is designed so that an address can be specified in the form of \texttt{http://server:port/file\_name.ext} and the application layer takes care of managing the transmission. Upon requesting an \texttt{htmp} URL, the client communicates to the directory server which provides information about the addresses and ports of both the unicast and multicast media servers. This is where the architecture has the opportunity to support extensive amounts of load distribution. Since any single user need only interface with the directory server a limited number of times (for the sake of finding media), the directory server has the ability to delegate users – based on random selection, user location, or some other determinant – to a unicast/multicast pair of servers that is ideal for their use.
In the above scenario, clients 1, 2, 4 and 5 have already communicated with the directory server to request the same piece of content and have used the server parameters returned to setup individual connections with the unicast server [TCP] and to subscribe to the multicast [UDP] server. While they all receive an identical stream of data over the multicast connection (which is routed by the network layer in the most efficient way possible), each unicast connection is distinct to reflect the fact that the different clients start downloading the data source at different times, and therefore need immediate access to the beginning of the source at different times. (Additionally, the unicast connection is used to help counter error correction and to avoid lost packets.)

Client 3 does not have connections to either the unicast or multicast servers since it is still in the process of submitting a request to the directory server. Once the client receives the addresses and ports for each of these remote hosts, it is up to the client to connect to each host and submit appropriate requests for the media it is attempting to obtain.

For the sake of this project, we assume that multicast transport service is available among all interested clients; in most network environments today, this is not always the case. However, because of the nature in which Hybrid Transmit combines a standard TCP transmission with inbound multicast packets, clients should not experience any service unavailability if they are unable to join the multicast server’s broadcast (granted, the efficiencies afforded by this architecture will not be realized in those cases).

**Protocol**

The design for Hybrid Transmit it segmented into two distinct components – the directory service and the actual transmission interface. Each of these derives substantial implementation methodology from well-known preexisting standards in such a way to provide a lightweight protocol that is also easily portable and easily implementable.
The directory server’s protocol is very similar to that of POP3 in that it involves single-line textual commands, provides machine-and-human readable error messages, and in other respects manages message communication in a lot of the same ways (e.g., all lines from the server are terminated by CRLF, all server replies are prefixed by either “+OK” or “-ERR” to signal success and failure, respective, etc.). The repertoire of commands in the directory server consists of:

- Login (separate commands for username and password) (**USER** and **PASS**)
- Listing of all available sources (**SHOW ALL**)
- Listing of details for an individual data source (**SHOW INFO**)
- Access credential verification (**VALIDATE**)

The login functionality is very straightforward and entails the user specifying a username and password at anytime during the session (as there is no login state). The reason for this is that while there may be some data sources limited to certain users, by and large most data will be public. Therefore, an anonymous login is assumed unless the user indicates otherwise.

The validation mechanism is for use by the unicast media server to confirm a user’s ability to download a certain source. In this manner, the server passes a user’s username (as provided directly to the unicast server) along with the original request URL to validate a user’s ability to access a certain piece of content to the directory server, which responds with a simple yes or no indication. Note that while there is no approach for prohibiting an individual host’s access to a multicast stream (it is architecturally impossible to do so), there are ways at the network level to set routing limits on which networks may join a multicast group. We further assume that the limited likelihood for a user to accurately guess multicast stream address/port parameters (which are only known to the directory server prior to dissemination to clients) is low and that any particularly sensitive data may be transmitted by layering
Hybrid Transmit over SSL or some other encryption method. However, we do not attempt such an implementation in the course of this project.

The **SHOW ALL** command displays a simple line-by-line listing of available content which consists of tab-delimited information pertaining to file names, MIME types, and total content sizes; the listing is terminated by a line consisting solely of a period and CRLF (much like the **TOP** command in POP3). This kind of data return is similar to that of **SHOW INFO**, which takes a third argument (the URL-encoded file name) and returns, in tab-delimited fashion, the content’s filename, MIME type, unicast server address and port, multicast address and port, as well as the source’s size (in bytes) and, if available, duration (in seconds). This command in particular is essential for the rest of Hybrid Transmit’s function since it is the only point at which the client is informed of the network resources from which it may obtain the data it has requested.

Once the communication is complete with the directory server, the client establishes connectivity to the actual transmission interface, beginning the flow of data. The typical packet used by both unicast and multicast streams for transmitting content data has effectively the same simple structure:

```
0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20
+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+
|    OFFSET    | E| LEN |             SOURCE DATA              /
+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+
```

In short, the protocol transmits bytes directly from the source generally unchanged. The only additions to the data are three “header” fields for aiding in the transmission and reassembly of bytes at the client. The first field is a five-byte header identifying the position in the file from which the bytes included in the packet originated. Since this allows for $2^{10}$ values to be represented, the protocol can support file sizes of up to 1 terabyte, essentially imposing no architectural limits to the kind of data it can transmit. The second header is a command error flag, primarily used during initial connection setup (to be discussed later). Lastly, the third header indicates the amount
of source bytes included in the packet and is sufficiently large (two bytes in size itself) to support up to 65 kilobytes to be transmitted in a single packet (which should be more than adequate for networks of any throughput).

It is worth noting that no where does this packet include an identifier for a source file, nor does it include any checksum or similar error detection data. However, neither of these fields are needed given the fundamentals of the architecture. Since only one file will be transmitted at a time over a single unicast connection, it is both the client’s and the server’s responsibility for keeping track of the file currently being transmitted; likewise, a given multicast address and port can inherently transmit only a single stream at a given time and since we assume that each transmission is allocated a unique address/port pair, there is no need to verify that what should be getting transmitted over a multicast connection is, in fact, what is being sent. With regard to error detection, there is a high probability that the multicast stream will cause some clients to not receive some select packets for a variety of reasons. We do not need to worry about this, however, since any bytes that are expected via multicast but not received will ultimately be queried via the unicast transmission which, as it is built upon TCP, inherently does not need extra error correction faculties. Thus the size of the packet can be limited to promote protocol efficiency without compromising the quality of the transmission.

While packets will flow automatically from the multicast stream, they must be requested explicitly by the client from the unicast host. The reason for this is two-fold: first, it provides a way for the client to exercise control over the stream of data without having to abruptly close a connection (and, hence, potentially cause the server to make unnecessary transmissions); second, it allows the client to submit urgent requests for bytes exempted from the multicast stream. The unicast client therefore has its own packet for communicating with the unicast server which takes the following form:
Notice that there is a single-byte prefix to this packet; this identifies the operation that the client has chosen to invoke. This function field can take any of the following values:

- 0 – Keep-alive (empty) packet
- 1 – Login
- 2 – Select File
- 3 – Request Bytes
- 255 – Logout

Both the keep-alive and logout packets have zero length (and, hence, no command-specific arguments that follow). However, they each elicit a “blank” reply from the unicast server which consists of zero values for each field except for the command error flag, which is set to two to indicate a non-error, special circumstance reply (see prior page). Further note that the logout command terminates the unicast TCP connection.

As with the directory server, the login command simply provides a manner through which the client can submit a user name and password; it does not, however, reply with any information about the validity of the credentials the user attempted to use even though the client is sent the kind of “blank” reply mentioned in the last paragraph. For passing user name and password arguments, the data component of the client packet takes the following form:

```
  0  1  2  3  4  5  6  7  8  9 10 11 12 13 14 15
+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+
| LEN |    USER NAME    | LEN |     PASSWORD    |
+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+
```

Note that there is no set length on either the user name or password fields, and these can be as long or as short as the client specifies (granted, up to the maximum-acceptable size of the embodying unicast client packet). Both the user name and
password are prefixed by a two byte length field which defines the size of the fields that follow. This same kind of structure is duplicated when the client specifies the URL of the Hybrid Transmit resource they originally requested through a select file packet:

```
  0  1  2  3  4  5  6  7  8  9 10 11 12 13
+------------------------------------------+
  | LEN |              URL                  |
+------------------------------------------+
```

The most “interesting” packet that the client can (and does!) send, however, is the byte request packet. This packet is sent almost immediately each and every time the connection between the client and the unicast server idles to either request the next stream of new bytes or to obtain bytes that were expected, but not received, from the multicast stream. Since the latter case presents the possibility that we may need to request multiple “blocks” of bytes, we define an extensible packet body that allows for as many or as few byte fragments to be requested as possible:

```
  0  1  2  3  4  5  6  7  8  9 10 11 12 13 14 15 16 17 18
+------------------------------------------+------------------------------------------+
  | REQ |    OFFSET    |  LEN   |    OFFSET    |  LEN   |  ...   | /
+------------------------------------------+------------------------------------------+
```

This packet format allows us to specify the number of requested packet segments (in the 16 bits available in \texttt{REQ}), which is then followed by that number of pairs of both offset and length values. These pairs specify the positions in the actual file at which the transmission should begin, followed by a value that indicates the amount that should be read starting from that position. Typically, these packets are submitted with only a single offset request to reflect the fact that, by and large, these packets are used to receive new data via TCP.
Application Framework

Much of the design for Hybrid Transmit focuses upon segmenting the different components of the project so that they can all operate independently but fit together to comprise the overall system. The core framework has been written in Java 2 Standard Edition with limited use of APIs beyond the `java.io`, `java.net`, and `java.util` libraries. All associated java source/class files have been packaged into `edu.yale.cs.hybridtransmit`. A brief description of all key framework Java source files follows:

- **Client.java** – handles all aspects of session construction and retrieval of source data; spawns `MultiastClient` in a separate thread which obtains the multicast stream of the data source and relays the received data back to `Client` by way of a synchronized method; this code is also responsible for identifying “persistent data holes” (cases where bytes that should have been received do not arrive) and submitting requests to recover from them
- **ClientPacket.java** – composes and deconstructs byte request packets as well as packets for simple commands (i.e., login/logout, select file, etc.)
- **DirectoryServer.java** – implements the listener for the directory server
- **DirectoryServerProtocol.java** – analyzes and acts on commands passed to a thread of the directory server
- **DirectoryServerThread.java** – offers an individually-threaded client-facing implementation of the directory server
- **HybridFile.java** – allocates and manages temporary file resources for Hybrid Transmit clients in order to allow them to compile disparate amounts of unicast and multicast data into accurate representations of downloaded material
- **HybridUser.java** – provides a framework representation of username and password combinations
- **HybridUtility.java** – serves utility functions to the library for simple byte conversions and other operations
• **MulticastClient.java** – individually-threaded client process that joins the multicast stream of a requested data source and relays the received data back to Client by way of a synchronized method

• **MulticastServer.java** – allocates individual multicast server threads and handles the construction/demolition of the multicast transmissions

• **MulticastServerThread.java** – iterates repeatedly over the balance of a specified data file gradually transmitting all content via multicast by embedding the data in Hybrid Transmit packets

• **ServerPacket.java** – composes and deconstructs client-bound packets filled with data from the requested files; error flags – as applicable – are set in these packets as well

• **SourceInfo.java** – provides a framework representation of data source parameters (i.e., server addresses/ports, file size, etc.)

• **UnicastServer.java** – implements the listener for the unicast server

• **UnicastServerThread.java** – offers an individually-threaded client-facing implementation of the unicast server

In addition to the actual framework, three server “drivers” – **MulticastServerDriver.java**, **UnicastServerDriver.java** and **DirectoryServer.java** – were written, in addition to a client driver (**ClientDriver.java**) to use this framework for the purpose of testing. These files tend to be modified in the course of completing different kinds of tests. The backend used to setup configurations is combination of both text files and hard-coded data in each of these driver source files.

### Design Challenges

Some difficulties were encountered finding networks within Yale that support multicast routing. While others among Yale’s peer universities appear to have this service available (per information obtained from these schools’ IT departments), Yale’s
network prevents this kind of transmission. For the sake of initial testing, therefore, multicast routing was enabled over “closed-circuit” networks on the same physical segment (i.e., connecting multiple computers to a hub and initiating a download of the same piece of content on each of them). While this method should provide some indication of how efficiently Hybrid Transmit reduces bandwidth costs, a more sophisticated setup will need to be arranged for testing a “typical” setup of multiple smaller networks separated from each other by several hops.

Beyond this, the infrastructure for the Hybrid Transmit protocol is very similar to a conventional client/server framework, and therefore required no unanticipated modifications to overcome unexpected design problems. That being said, the process of coordinating inbound data from two sources (one of which – the multicast stream – does not guarantee in-order receipt of bytes, or receipt of a certain string of bytes from the data source at all). To get around this problem, a number of possible solutions were considered, including mapping each expected byte to a single bit in a second temporary file and reviewing this “bit map” periodically to ensure that bytes were being received. Ultimately, the current implementation of a more intelligent priority queue algorithm to submit second requests for data that should have already arrived provides a solution that allows for timely and accurate receipt of remote data.

**Test Results**

All core components have been developed and tested individually and initial testing shows that the framework and corresponding protocol work as desired. Although there is no indication of any significant problems with the currently-written parts of the project, additional testing will be needed to determine the performance metrics of the project at large.

**Future Development**

While much of the core engineering for this project has been completed, there are a number of opportunities to add additional functionality. Some of these are as
follows:

- **Peer to Peer Source Sharing** – While much of the current implementation of Hybrid Transmit rests on the assumption that data is centrally located, it is foreseeable that mechanisms could be engineered to allow individual client machines to serve as unicast or multicast servers themselves in coordination with the original directory server. In this manner, the directory server would maintain a database of previously-downloaded data among clients who have opted to serve as mini-servers themselves when surplus bandwidth is available to them. This could further reduce the load placed on a “global” network by enabling a client to obtain the unicast portion of a data stream from a machine more local to the user. As part of this kind of implementation, however, the directory server and sending client in tandem would need to be able to manage and avoid circumstances that might result in unreasonable network or internal resource utilization.

- **Dynamic Client Routing** – In environments where multiple unicast or multicast servers are available, the directory server has the ability to load balance client requests based on either server utilization or “optimal” network routes (which can imply finding either the shortest path or the path with the lowest round trip times). This approach can provide additional efficiencies beyond those already afforded by the current protocol.

- **Multicast Optimization** – The multicast server broadcasts the same piece of content in loops without regard for the times at which clients begin downloading the content. It is foreseeable that the amount of data transmitted over TCP unicast could be further minimized by periodically determining the point in the download of an “average” user and resetting the multicast stream to begin broadcasting at that point.

- **Low-Priority Client Service** – In some cases (e.g., download of a video to be viewed later, download of a file wherein it does not matter if the first bits received are from the beginning or from the middle-point, etc.), a client may
require extremely limited use of its “on-demand” unicast connection. A variation of the original client framework to provide this kind of “low-priority” service could add additional efficiencies beyond those already considered in the development of this protocol.

- Data Source Synchronization – We currently assume that the “as-is” setup of our unicast and multicast servers is such that they both have access to identical sets of the content they’re serving; of course, this is not necessarily going to be true in a “real world” environment. To that end, the directory server could also be given functionality to communicate with all unicast and multicast servers in its realm to make sure that they maintain synchronized copies of the data they are supposed to have.

- Hybrid Transmit Over SSL – For transmission of either sensitive material or of content with managed rights, SSL over Hybrid Transmit would be a useful extension of the current protocol.

Additionally (as noted in the prior section) load testing still needs to be conducted and associated metrics need to be derived. Much of the work on this project over the next several months will consist of this analysis as well as the development of some of the additional features noted above.

**Conclusion**

Hybrid Transmit has thus far demonstrated its ability to serve as a functional means for transmitting material simultaneously over both unicast and multicast connections with the intention of reducing the amount of bandwidth required when many clients attempt to retrieve the same piece of content at a given time. While the extent to which Hybrid Transmit is successful in its goal is not yet known, additional testing in the coming months will provide some indication of the protocol’s efficiency metrics. This work will also complement the development of new features and the optimization of core functions (which are mostly already developed) to deliver a
comprehensive, reliable solution to the original specification that is backed by demonstrated results.

References


