An Architecture for Seamless Access to Multicast Content

Pieter Liefooghe, Marnix Goossens
Vrije Universiteit Brussel, INFO/TW Tele.Com
{Pieter,Marnix}@info.vub.ac.be

Abstract

In this paper we describe an architecture, which allows transparent access to the multicast infrastructure even when not directly connected to it. In that context, we introduce the concepts dynamic tunnel server location and Rate-Based path characterization. We propose changes to UMTP in order to allow dynamic tunnel server location through firewalls. We present dynamic congestion discovery with an associated tunnel hand-over mechanism. The features of a session directory application, which integrates the dynamic tunnel mechanism, is shown. The concept of "channels" for session announcements is introduced. And finally we present a SAP/SDP proxy location and query mechanism.

1. Introduction.

IP multicast [1] is an efficient method to do one-to-many and many-to-many communication. It is efficient in the sense that in order to distribute data to multiple receivers we only have to send one copy of the data; this compared to the unicast case where we have to send a copy for each individual receiver.

A quite extensive protocol set (IGMP, DVMRP, PIM-SM, PIM-DM, MSDP, MBGP, BGMP) has been developed in the IETF, to build up an efficient distribution tree between the sender and the receivers. The majority of the router vendors are gradually supporting these protocols. So we could conclude that by now a large part of the Internet would be multicast enabled. But currently we see that the deployment is not going as fast as one could normally expect.

The major reason for this is that we are here in some sort of two-fold deadlock situation with problems situated at three parties:

The first party involved are the ISPs. As described by Diot C. et al. [2], ISPs are reluctant to deploy IP multicast because it relies on a protocol architecture that requires more setup and administration than the unicast architecture. This makes that in order for a multicast solution to be cost efficient one will need potentially a large number of multicast capable customers.

The second party are the customers. Only a minority of users knows about multicast and the potential benefits this technology could bring them. From queries in popular software archives (e.g. winfiles.com) one could falsely conclude that there is almost no software available that supports multicast. Although quite popular applications like RealPlayer, Quicktime Player, FreeAmp etc. do support multicast. Since it is for consumers not all that clear what multicast is about and what they can do with it, they won't be asking their ISPs to get multicast.

The content providers are the last group involved. Although the use of multicast could result in a huge bandwidth saving, content providers are not eager to invest in this new technology, because there are almost no "consumers" for this type of content.

From the above it is clear that in order for IP multicast to become ubiquitous we need to try to provide as many end-users with a multicast "connection".

The work described in this paper situates in breaking this deadlock situation. A mechanism, which allows an Internet user with enough access bandwidth to access multicast content even when not connected to a multicast enabled ISP is described. In this manner we increase the multicast-audience and hence make it more attractive for content providers to deploy this bandwidth-saving technology. At the same time, when we reach the "critical-mass" of multicast "interested" end-users, ISPs will be more willing to deploy multicast.

The basic idea behind our solution is that it is possible to encapsulate (tunnel) the multicast packets in unicast packets between the portion where we do have multicast and the end-user.

We want this tunneling to be fully transparent for the user. This means that the user simply has to choose a multicast session and that the system should dynamically locate the best tunnel end-point and tunnel the multicast data between the multicast-enabled part and the client. Further, in the case of congestion on the path between the tunnel server and the end-user a session hand-over can be performed. As a result of this research, a prototype was created which supports the above-mentioned techniques

0-7695-0912-6/00 $10.00 © 2000 IEEE

and additional features to enhance the multicast experience for the end-user.

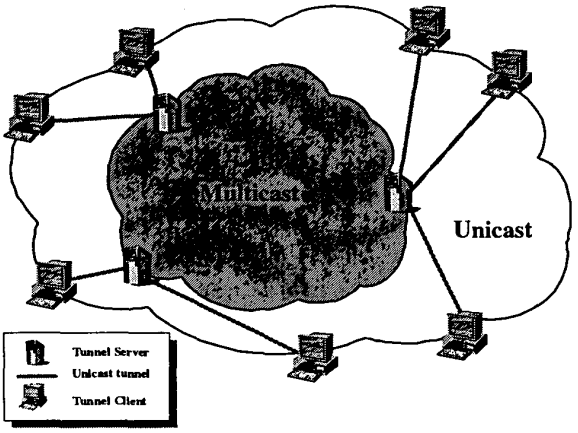

Figure 1: Tunneling of IP multicast packets.

2. Application level tunneling.

This research started by evaluating the possible mechanisms to tunnel multicast data between the multicast-enabled part of the Internet and an end-user. It was found that some work had already been performed by Peter Parnes with mTunnel [3] and by Ross Finlayson with the UDP Multicast Tunneling Protocol (UMTP) [4].

Both solutions are based on the concept of application level tunneling. In this approach tunneled data is sent over a UDP unicast "connection" between the two tunnel endpoints. The client uses some sort of control channel to indicate to the server which multicast address and port number to join. All data received on this multicast channel is then encapsulated in UDP datagram packets and sent to the client. Clients will decapsulate and multicast the received packets locally. The same mechanism is used in the other direction, which makes bi-directional communication possible.

The client applications, however, must satisfy the following conditions:

- Their multicast packets must use UDP.
- The tunnel endpoint must have a way of knowing each (group, port) that the application uses.
- The application must not rely upon the source address of its incoming multicast packets. In particular, the application must not use source addresses to identify the original data source(s).

Most multicast-based applications - especially those based on RTP [5] satisfy these requirements.

This application level tunneling has the big advantage that no special privileges are needed to run the application and that there is no need for kernel-level access as compared to kernel-level tunneling solutions like implementing a full-blown routing protocol like DVMRP [6].

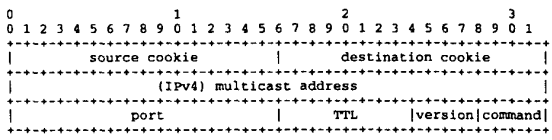

Figure 2: UMTP Packet Trailer.

The two application level tunneling solutions use trailers rather than headers. This has the big advantage that, since the encapsulated data, will often be an RTP packet, by retaining this data in its usual position; IP/UDP/RTP header compression [7] - if present - will work properly. A secondary reason is that this order may allow application level tunneling implementations written in some type-safe programming languages - such as Java - to reduce the amount of byte copying that they would otherwise have to perform.

The difference between both solutions is that UMTP uses the multicast address and port numbers in each packet to uniquely identify a tunneled multicast channel, compared to the more efficient method of using a flow identifier in mTunnel.

Both use a different approach to setup the tunneling of a given session:

With mTunnel one has to use a web browser to connect to the local tunnel endpoint and use this to indicate which session to tunnel. The user has to manually launch the tools or use a session directory tool like Sdr [8]. We also need to use the web browser to explicitly terminate the tunneling of a session when we no longer want to attend a session.

UMTP on the other hand is integrated in a session directory tool called Multikit, which makes it possible to simply double click a session announcement to establish the tunneling and to launch the associated tools. The tunneling of a session needs to be terminated explicitly.

Both solutions have as drawback, that the user needs to know in advance the tunnel server and that an explicit tunnel termination is required.

After evaluation of the different approaches it was recognized that UMTP was suited for our purpose especially by the fact that this protocol is fully documented in [4] compared to only a high-level description of the mTunnel protocol in [3].

3. Dynamic tunnel server location.

It is clear that if we want to have a user-friendly application, we cannot expect the user to know which tunnel server to use. Further, if we would statically configure a list of servers it would not be feasible for an average end-user to decide which tunnel server would be the best. Hence we need to devise a dynamic and fully transparent mechanism to locate the "best" server for a given client. Here we chose to implement a separate protocol rather than to augment the UMTP protocol with some extra negotiation component.

We introduced the concept of Tunnel Regions in order to enhance the scalability of the tunnel server location. As Tunnel Regions we defined: local, country and regional. With local we indicate tunnel servers located in the corporate/ISP network; with country we indicate all tunnel servers located in a given country and with regional we indicate the tunnel servers located in the neighboring countries. E.g. all tunnel servers in Europe for a client located in the UK.

The idea is that a client will try to look first for tunnel servers in its own network and expand to the country if none is found. If there none is found, one in the region will be used.

The country and the (sub)network(s) to which a server belongs are indicated at installation of the server. Once a server starts, it registers this information in a central tunnel database. At server shutdown, a deregistration is performed. This database hence contains all available servers and an indication for what network(s) and for which country they are willing to act as tunnel server. The region is inferred from the country. A client is also configured with the country of the end-user. This information, combined with the IP address of the client is then used to query this central database using the following criterion:

First a match between the client's IP address and the registered (sub)networks of the tunnel servers is tried, using longest prefix matching. If no match is found, a match at the level of the country is attempted; if no match is found a query at the level of the region is performed. A "default" tunnel server is returned if no match is found.

In our prototype we chose to use HTTP GET interactions [9] as the basis for the queries/registrations in the tunnel database because of its very simple nature. We are currently investigating how we can replicate the registration information among different tunnel database servers as to come to a more reliable architecture.

When we have located multiple potential tunnel servers we still need to "measure" which server is best suited for handling the client. If the query only resulted in one server, we simply set up a UMTP tunnel with this particular server.

In the case of multiple endpoints, the following sequence of operation is carried out: first a temporary UMTP tunnel is set up with the tunnel server that was used in a previous run, if this server is member of the list of potential servers or with a randomly chosen server if the previously used server is not present in the list. Through this tunnel it is already possible to join some multicast sessions. While the tunneling of these sessions is happening, we try to figure out which server is best suited for acting as final tunnel server.

3.1. Path characterization.

In one of our initial prototypes we used a "fixed" multicast address to communicate from the client, through the temporary tunnel, with the different servers in the "nearest" region. Through this "channel" a tunnel request message with suitable TTL was sent which triggered the tunnel servers to start to perform a characterization of the path between the server and the client. The problem with this solution was that there is a high probability, that multiple servers will perform their measurements at the same time. This had as effect that the measurements of the servers were negatively influenced by each other. We can minimize this effect by starting the measurement not immediately at receipt of the tunnel request message but only after some random time. But even with this approach we see that there is a negative influence even when we take quite a large value for the random timer interval. This is mainly due to the fact that the characterization of the path is taking some time to finish.

This is why we chose to let the client do all the characterizations. The client will set up "dummy" UMTP tunnels towards all the servers one after the other. Through such a tunnel a special "test channel" is joined. The tunnel server will act differently for this test channel compared to "normal" channels in the sense that all traffic sent towards this special channel is looped back towards the client. For the characterization, we start sending, on this test channel, UDP packets with a size of 512 bytes (+/- average size of packets in a session).

It is clear that when we want to do a measurement, we want this measurement to be the least intrusive as possible. Since our measurement is based on the sending of UDP packets and that we know that UDP traffic is very unfriendly to TCP traffic we need to make our measurement "TCP-Friendly". We decided to add a "Rate-Based Flow Control" mechanism based on the work described in [10, 11]. In this mechanism, the rate at which packets are sent is controlled by the degree of packet loss and by the round trip time. In order to make the path

characterization rate-based we added a header with several fields to carry the sequence number and the acknowledgements. We also added redundant information to the ACK stream in order to specify the last hole in the delivered sequence number space and to accommodate for single ACK losses, as described by Rejaie et al. in [11]. Which results in the following header fields:

1. Sequence Number: sequence number of the data packet on which this ACK is piggybacked or the sequence number of the previously sent packet when no data is present.

2. Current ACK: the sequence number of the packet that we just received and that we are acknowledging.

3. Missing ACK: The sequence number of the last packet before the Current ACK that was still missing, or 0 if no packet was missing.

4. Last ACK: The sequence number of the last packet before Missing ACK that was received, or 0 if Current ACK was the first packet.

The Inter-Packet-Gap (IPG) i.e. the time between two consecutive messages is evaluated every SRTT. SRTT is the smoothed version of the measured round trip time. The formulas used to come to a TCP-friendly characterization are given below and are based on the Jacobson/Karel's algorithm:

$$\text{Difference} = \text{MeasuredRTT} - \text{SRTT}_i$$
$$\text{SRTT}_{i+1} = \text{SRTT}_i + (d * \text{Difference})$$
$$\text{Deviation} = \text{Deviation} + d\,(|\text{Difference}| - \text{Deviation}))$$

Where $d = 1/8$.

We know that there are packets lost when explicitly indicated by the Missing ACK field or implicitly after:

$$\text{Timeout} = \text{SRTT}_i + 4 * \text{Deviation}$$

The control of the IPG is based on the Additive Increase and Multiplicative Decrease (AIMD) algorithm: In the absence of packet loss, we decrease IPG according to:

$$\text{IPG}_{i+1} = (\text{IPG}_i * \text{SRTT}_i)/(\text{IPG}_i + \text{SRTT}_i)$$

In case of packet loss we double the Inter-Packet-Gap:

$$\text{IPG}_{i+1} = 2 * \text{IPG}_i$$

We continue to send packets during 3 seconds or until we have sent a total of 50 packets. The obtained parameters are then combined into a metric using the following formula:

$$M = 50*(1 - Loss/10)+40*(Bandwidth - Throughput)/(Bandwidth) + 10*(SRTT/2000\ ms.)^1$$

Where Loss = average packet loss (%); Bandwidth = Upstream Bandwidth of the client's Internet access; Throughput = Throughput expressed in Bytes Per Second; SRTT = the value of SRTT at the end of the measurement, expressed in milliseconds. The result of this calculation is a number between 0 and 100. The lower the metric the better the path. From the formula of the metric we see that the principal parameter is packet loss. When we encounter during our measurement packet losses in excess of 10% we simply stop the measurement, this because at such high loss rates the tunneled media (especially Audio and Video) becomes unusable. The next important parameter is the throughput; ideally it should reach the maximal bandwidth of the access device. Finally as the least important factor we use the smoothed RTT, because it is felt that although delay is important - especially in an interactive session - we see that most multicast applications available now are made for one-way communications i.e. streaming and are hence less affected by high delay paths.

The client will know through this mechanism which server has the best metric. If the server with the best metric is different from the temporary tunnel server a tunnel hand-over procedure is started.

If two or more offers have an identical metric, one at random is selected. Once the client has chosen the appropriate server, a UMTP tunnel is set up with this server.

From the moment we start to receive data on a joined channel we stop the tunneling of that channel on the temporary server. We continue to do this until we no longer have any "temporary" sessions open. Finally we close the UMTP session with the temporary server. From this point on we tunnel from the new tunnel server.

4. Dynamic tunneling and firewalls.

Since all our communication is based on the UMTP protocol and in view of the fact that the UMTP protocol only uses a fixed UDP port it is easy to configure a firewall for traversal of tunneled multicast. But it has as big disadvantage that the dynamic tunneling is made impossible because in the firewall one has to define a fixed mapping between the tuple (firewall IP address and port) and a tuple (UMTP Server IP address and port).

By use of a UMTP Proxy module in the firewall or as process running on a machine in the "Demilitarized Zone", one could "restore" the dynamic capabilities. For

[1] Loss is limited to 10% and SRTT is limited to 2000ms.

this we need to add proxy functionality to the UMTP protocol. This can be achieved by defining a new command called PROXY, which still uses the same trailer layout (see Figure 2) as all the other UMTP packets. But we change the interpretation of the field "(IPv4) multicast address", which will now contain the IP address of the tunnel server and the "Port" field will contain the port of the tunnel server to contact. All other fields remain unused. This proxy message will be sent as the first message in the UMTP communication between the client and the firewall/proxy. From the moment the proxy received this message it knows that it needs to forward all messages to the tunnel server defined in the PROXY command. This forwarding continues until the proxy encounters the TEAR_DOWN message.

5. Dynamic congestion discovery and tunnel hand-over.

It is very well possible that the path between the UMTP tunnel server and the client gets congested. It would be nice that in such a situation a hand-over to a new and "better" server could be made. Before this is possible a mechanism to detect the degree of congestion should be available. A possible way to achieve this is by modifying the UMTP protocol in such a way that some sequence number is incremented for each new packet tunneled. The RTP sequence number can't be used for this purpose because there could already be missing packets before they were tunneled. When the percentage of missing sequence numbers (and hence packets) exceeds a certain level (e.g. 5%) a new tunnel server discovery round could be started in order to locate a better server. This item requires further research.

6. New generation session directory.

Initially we developed the dynamic tunneling protocol in order to incorporate it with new applications. But it was recognized that there are many applications currently available which would not be able to benefit from the transparent tunneling capabilities of our solution.

This is why we decided to implement our own session directory application and to integrate the dynamic tunneling in it. This application listens on a "reserved" multicast address for announcements by the different content providers. Such an announcement is a combination of Session Announcement Protocol (SAP) [12] packets with Session Description Protocol (SDP) [13] packets as its content. The SAP protocol is quite simple in the sense that it is only used to "carry" SDP packets and that it provides a mechanism for encrypting the SDP packet and/or an authentication mechanism.

The SDP protocol on the other hand will carry detailed information about the session. It contains for example the title of the session and a short description, some contact information and scheduling information.

Further, for each media (Audio, Video, Whiteboard etc.) present in a session we find a listing of multicast addresses used and their associated port numbers. This media specific information is then used to launch the applications capable of processing these media types and to establish the appropriate tunneling.

6.1. The channel concept.

While working on this session directory we identified that a flat session listing as used by many session directories, was unrealistic, especially when the number of sessions becomes large. So we implemented the directory concept as first introduced by Ross Finlayson and described in [14]. But we also introduced the concept of Channel. With this it is possible for a content originator to announce all its sessions under a given "Channel Label". The difference with a directory is that only the creator of the channel is "allowed" to announce sessions within this channel. For this, we propose a new media type "channel" and an attribute "cert".

m=channel <port> SAP SDP
a=cert:<url>

<url>, is the URL pointing to the location where the x509 or PGP certificate of the channel originator can be downloaded.

The session announcement of the channel and all session announcements within this channel will be "signed" at the SAP level with the private key of the channel originator. Only announcements with a valid signature and corresponding with the root channel certificate will be listed in the channel.

6.2. Universal tool launcher.

During the development cycle it was also recognized that if we wanted to promote the use of multicast it would be necessary to be able to launch as many multicast capable applications as possible. These multicast capable applications can be divided in to two categories.

The first category contains those applications where all the information about the media/session to join, is transferred to the application via command line options. The mapping from the media parameters in the SDP announcement to a command line is defined through the

use of plug-in files. In these small files we define for each type of media what the formats are that a given tool supports as well as the different command line options that need to be used for a given set of entries in the SDP announcement. Although the concept of plug-ins for use in session directory applications is not new, we enhanced it with new features like descriptions of the tools, human readable names for the different media formats etc in order to make it more user friendly.

The other mechanism for launching tools consists of using an SDP file. We enhanced the session directory tool so that (if indicated in the plug-in of the tool) all parameters of a session are stored in an SDP file and that the tool is launched with the name of this SDP file as command line parameter.

Whenever for a media type no plug-in is found a query is performed in a central database in order to locate and download an appropriate plug-in. A query is also performed in case an application defined in a plug-in is not found on the client's machine. In case a reference to the tool can be found in the central database, a web browser is launched pointing to the site where the tool can be downloaded.

7. SAP/SDP proxy.

The session directory tool continuously listens for announcements made by potential content originators. This "announce-listen" mechanism has some major drawbacks:

As described in [12], SAP announcements should be made with an:

interval =max(300s ; (8*no_of_ads*ad_size)/limit)

From this formula we can determine that the minimal value for the interval equals to 5 minutes (300s).

This means that it takes at least 5 minutes to build up a full list of session announcements. But it becomes even worse in the case of packet loss. It is quite common to see it take 10 or more minutes for a whole list to be built.

This also has a negative effect on the multicast address allocation called Informed Partitioned Random Multicast Address Allocation (IPRMA) as analyzed in [16]. The IPRMA mechanism allows for a session directory to locally generate a multicast address with minimal chance that this multicast address will conflict with another multicast address already in use. This scheme depends on the allocator knowing a large portion of the addresses already in use. If this is not the case the chance for an address clash increases.

This problem can be lightened partially by implementing a caching mechanism in the client. But to solve this issue to the maximum extent, we chose to implement a separate SAP/SDP proxy. This is a daemon process, which actively listens on all announcements being made (including those in directories and channels). Now whenever a session directory tool starts it uses an SAP/SDP Proxy discovery mechanism, based on expanding disk search [16] in order to locate an appropriate SAP/SDP Proxy for a given scope. We start by sending query messages with a TTL of 1 and gradually increase the TTL until we receive an offer or reach 63. We limit the scope to 63 because we feel that, when working within the global scope, it would be a waste of network resources to query at the global scale. Seen the fact that we gradually increase the TTL, we are resilient to packet losses. In the query message we provide the TTL value with which the packet is sent. A SAP/SDP Proxy, who receives this query, will send back an offer on the same multicast address as the one on which the query was received. We use in our prototype a negative offset of 3 from the upper bound multicast address of the administrative or global scope.

This offer message indicates how many session announcements within the queried scope have a TTL larger or equal to the value of the TTL field in the query message. It also contains the TTL value itself. Other SAP/SDP Proxies will also send their offers unless their "calculated" number of sessions is smaller or equal to the ones already seen from other proxies. The session directory tool will potentially receive multiple offers and will take the best offer (the one with the highest number of sessions and lowest TTL). If there are two with an equivalent number of sessions the one that was received first is chosen.

Once the session directory tool has decided about which SAP/SDP Proxy to use, a HTTP GET command is used to query for sessions with a TTL higher or equal to the TTL value present in the best offer.

Rather than to specify a path based on directories and filenames, we specify it based on the multicast address of the scope/directory/channel we want to query.

GET /224.2.127.254/224.2.152.92/?TTL=12 ...

The query shown in this example would query for all session announcements in the directory/channel 224.2.152.92, which is announced in the 224.2.127.254 scope. We provide the TTL value that was returned in the offer i.e. 12 as argument to the query. We define a new content-type application/sapstream for the "virtual" file that will be transferred as a result of a successful query. This virtual file contains the SAP packets delimited by a 4-byte header containing the length (in network order) of the appended SAP packet. The client recursively queries all subdirectories and/or channels.

We extended this architecture further in order to make it possible to use the SAP/SDP Proxy as a remote session announcer. For this we chose to use the HTTP PUT command. This command provides the path on which the announcement(s) should be made. The announcements themselves are sent just after the PUT command's header as an application/sapstream file. Deletion of an entry is achieved via a PUT command with the original SAP message in the SAP stream, but where the message type bit is turned on (i.e. delete). Modification of an entry is done by a PUT command where we provide an updated SAP message in the SAP stream. The URI for this command is formatted as indicated below.

URI=<path>/<orig_addr.>?id=< session id>

Where <path> corresponds to the location, <orig_addr> to the originator address and <session_id> to the session id from the SDP message.

The PUT command always contains login and password information. This authentication information is necessary in order to prevent others from modifying or deleting an entry.

8. Conclusions and further work.

We described a network architecture, which allows transparent access to the multicast infrastructure even when not directly connected to it. We introduced the new concepts: dynamic tunnel location and Rate-Based path characterization. The changes that are needed to UMTP in order to allow dynamic tunnel server location through corporate firewalls were identified. The potential benefit of dynamic congestion discovery and its related tunnel hand-over mechanism was shown. A discussion was made about a new generation of session directory tool, that allows seamless access to various multicast sessions using a wide variety of multicast applications and that integrates the dynamic tunnel mechanism. The channel concept for sessions was introduced and we presented how a SAP/SDP proxy can be located and how it can be queried to return session announcements in specific scopes, directories or channels. Finally we identified and described the methods to use the SAP/SDP proxy as a remote session announcer. We are currently working to integrate all these concepts and prototypes into: a user-friendly session directory, a multi platform UMTP tunnel server and a SAP/SDP Proxy. We plan to investigate the possibilities to apply rate-based flow control to the tunnels and we also plan to develop a tool, which would allow an ISP to see how many of their customers are tunneling and what amount of bandwidth they are using.

9. References.

[1] Deering, S., "Host Extensions for IP Multicasting.", Request for Comments 1112, August 1989.

[2] Diot C. et al., "Deployment Issues for the IP Multicast Service and Architecture", IEEE Network, January 2000.

[3] Parnes P. et al., "mTunnel: a multicast tunneling system with a user based Quality-of-Service model", IDMS, September 1997.

[4] Finlayson R., "UDP Multicast Tunneling Protocol", draft-finlayson-umtp-04.txt, June 1999.

[5] Schulzrinne H. et al., "RTP: A Transport Protocol for Real-Time Applications", Request for Comments 1889, January 1996.

[6] Waitzman D. et al., "Distance Vector Multicast Routing Protocol", Request for Comments 1075, November 1988.

[7] Casner S. et al., "Compressing IP/UDP/RTP Headers for Low-Speed Serial Links", Request for comments 2508, February 1999.

[8] Handley M. et al., "sdr – A Multicast Session Directory", Unix manual page, University College London.

[9] Berners-Lee et al., "Hypertext Transfer Protocol -- HTTP/1.0.", Request for Comments 1945, May 1996.

[10] Mahdavi J. et al., "TCP-friendly unicast rate-based flow control". Note sent to end2end-interest mailing list, January 1997.

[11] Rejaie R. et al. "RAP: An end-to-end rate-based congestion control mechanism for realtime streams in the Internet", Proc. IEEE Infocom, March 1999.

[12] Handley M. et al. "Session Announcement Protocol", draft-ietf-mmusic-sap-v2-06.txt, March 2000.

[13] Handley M. et al., "SDP: Session Description Protocol", Request for comment 2327, April 1998.

[14] Finlayson R., "The directory SDP media type", draft-ietf-mmusic-sdp-directory-type-00.txt, March 2000.

[15] Handley M., "Session Directories and Scalable Internet Multicast Allocation", Proceedings of ACM SIGCOMM, September 1998.

[16] Swan A. et al., "Layered Transmission and Caching for the Multicast Session Directory Service", ACM Multimedia, September 1998.