Network Working Group
Internet-Draft
Expire in six months

Ross Finlayson
LIVE.COM
2003.11.21

The UDP Multicast Tunneling Protocol

<draft-finlayson-umtp-09.txt>

1. Status of this Memo

2. Abstract

Many Internet hosts - such as PCs - while capable of running multicast applications, cannot access the MBone (or other wide-area multicast network) because the router(s) that connect them to the Internet do not yet support IP multicast routing. The ''UDP Multicast Tunneling Protocol'' (UMTP) enables such a host to establish an 'ad hoc' connection to the MBone by tunneling multicast UDP datagrams inside unicast UDP datagrams. By using UDP, this tunneling can be implemented as a 'user level' application, without requiring changes to the host's operating system.

3. Revision History

November 2003: draft-finlayson-umtp-09.txt
Bumped the file's version number, to get the I-D editor to accept it this time.

October 2003: draft-finlayson-umtp-08.txt
Updated the references to the RTP RFC,

and the "Source-Specific Multicast" I-D.

September 2002: draft-finlayson-umtp-07.txt
Made it clear that UMTP does not include a tunnel endpoint location protocol.
Expanded the "Security Considerations" section to better explain the 'cookies' mechanism.
Updated the reference to the "Source-Specific Multicast" I-D.

March 2001: draft-finlayson-umtp-06.txt
Removed the word "encapsulation", because it was not strictly accurate, and was confusing some people into thinking that the original packet's UDP/IP headers were being carried within a DATA packet. (Instead, only the orignal UDP payload is carried, plus the "tunneling trailer" that carries the important information from the original headers.)
Cleaned up and shortened the abstract, to fit in one paragraph.

February 2001: draft-finlayson-umtp-05.txt
Corrected the version numbers in this revision history.

December 2000: draft-finlayson-umtp-04.txt
Added an example to illustrate the operation of the protocol.
Added a description of the extended trailer (and commands) that are used to tunnel Source-Specific Multicast ("SSM")
Corrected a URL

September 1998: draft-finlayson-umtp-03.txt
Added descriptions of two new commands: JOIN_RTP_GROUP and LEAVE_RTP_GROUP, for optimizing the control of RTP/RTCP sessions.

February 1998: draft-finlayson-umtp-02.txt
Rearranged the fields in the "tunneling trailer" to allow for possible variable-sized trailers in the future.

December 1997: draft-finlayson-umtp-01.txt
Changed the "tunneling header" into a "tunneling trailer".
Added 'cookie' fields as nonces to prevent source address spoofing.

December 1997: draft-finlayson-umtp-01.txt
Original draft.

4. Overview

UMTP operates using (unicast) UDP datagrams between pairs of nodes - each pair forming the endpoints of a "tunnel". Each datagram is either a "command" (e.g., instructing the destination endpoint to join or leave a group address & port); or "data": an enclosed multicast UDP datagram

payload, along with the destination (group, port) tuple.

For each (group, port) being tunneled, a tunnel endpoint can act either as a "master" or a "slave". A tunnel master (for a particular (group, port)) periodically sends a JOIN_GROUP command to the remote endpoint (a slave), instructing it that this (group, port) is still of interest, and should be tunneled (or continue to be tunneled). A slave will stop tunneling a (group, port) if it either (i) receives a LEAVE_GROUP command from the master, or (ii) has not received any JOIN_GROUP commands for some period of time (currently, 60 seconds). Typically, a host that is trying to access the MBone (e.g., a PC) will be a master, and its remote endpoint (a host already on the MBone) will be a slave. (It is also possible, however, for both endpoints of a tunnel to be masters.)

Whenever a tunnel endpoint - whether a master or a slave - receives a multicast UDP datagram addressed to a (group, port) that is currently being tunneled, it resends its payload (along with a "tunneling trailer") as a unicast datagram addressed to the other end of the tunnel. Conversely, whenever a tunnel endpoint receives, over the tunnel, a tunneled unicast datagram for a (group, port) of interest, it resends its payload as a multicast datagram (with a TTL that was specified as a parameter in the "tunneling trailer").

A node (typically a slave server) can be the endpoint of several different UMTP tunnels - i.e., each with a different endpoint master. Although, superficially, such a system appears similar to a multicast<->unicast reflector, it differs in two ways:
(i) The tunneling is application-independent, and handles any (UDP) multicast packets.
(ii) After traversing a tunnel, a tunneled payload is delivered to the endpoint's application(s) via multicast, not unicast. This allows regular multicast-based applications to make use of a UMTP tunnels (subject to some restrictions described below).

UMTP does not define the means by which each node chooses a remote node to act as a tunnel endpoint. Tunnels could be set up 'by hand', or else a separate (unspecified) tunnel endpoint location protocol might be used.

5. Restrictions

UMTP allows a multicast-capable - but otherwise non-MBone-connected - host to run multicast-based applications. Such applications, however, must satisfy the following conditions:

1/ Their multicast packets must all be UDP - not 'raw' IP
2/ The UMTP implementation (i.e., a tunnel endpoint) must have a way of knowing each (group, port) that the application uses.
3/ The application must not rely upon the source address of its incoming multicast packets being different for different original data sources. In particular, the application must not use source addresses to identify these original data sources.

Most multicast-based applications - especially those based on RTP [2] - satisfy these requirements. If the multicast-based applications are launched from a separate 'session directory' application, then the UMTP implementation may be built into the session directory. For some multicast applications, however, the (group, port) is not specified in advance, but instead is determined by the application itself - e.g., by querying a separate 'licensing' server. Depending on the host operating system, a separate UMTP implementation might not be able to independently determine this (group, port). In this case, UMTP could not be used, unless it were incorporated into the application itself.

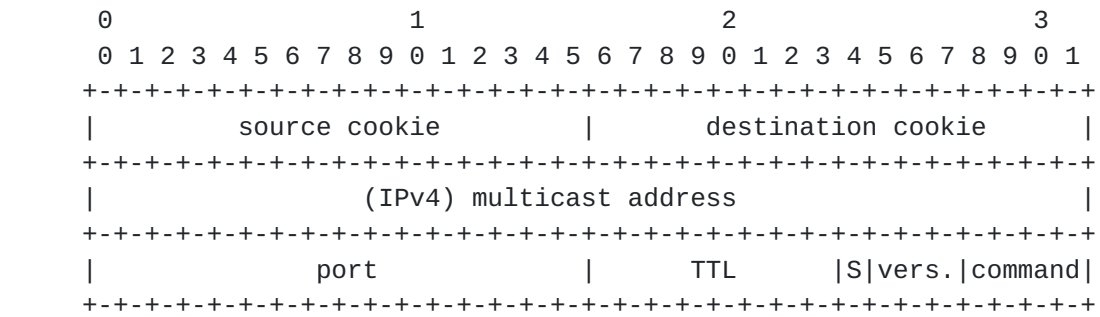
Note that for Source-Specific Multicast ("SSM") sessions [6], condition 3/ above will be violated: SSM uses a source address (as well as a regular group address) to identify a SSM "channel", but the act of tunneling a multicast packet (using UMTP) changes its source address. Therefore, any SSM-receiving application that uses UMTP to deliver its incoming multicast packets must be made aware of this change of source address.

These application restrictions reinforce the point that UMTP should be used *only* if full multicast routing cannot be provided instead.

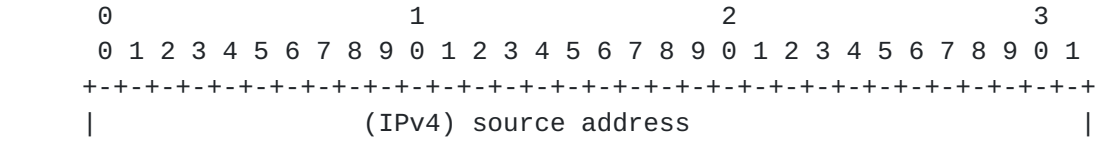
6. Packet Format, and Command Codes

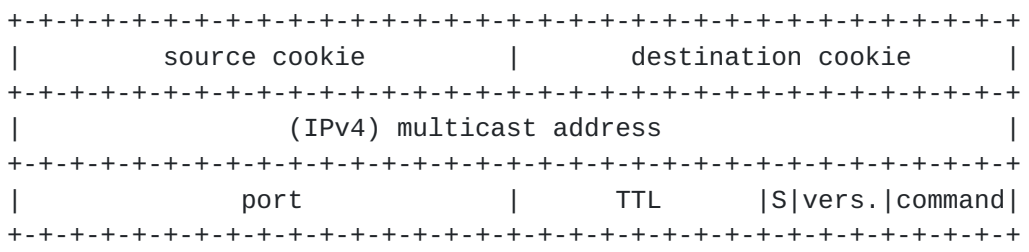
The payload of each UMTP datagram - i.e., excluding the outer UDP header - consists of a 12 or 16-octet 'trailer' descriptor. For commands other than "DATA", this descriptor makes up the entire UMTP payload. In the case of a "DATA" command, however, this descriptor is also preceded by the data that is being tunneled (i.e., the original UDP payload). The format of the 'trailer' descriptor is as follows:

12-octet trailer:
==========



16-octet trailer:
==========





Note that UMTP is defined only for IPv4. (In IPv6, native multicast routing should be ubiquitous.)

The "S" bit (the high-order bit of the last octet of the trailer) indicates the size of the trailer. S==0 denotes a 12-octet trailer; S==1 denotes a 16-octet trailer.

The 16-octet trailer is identical to the 12-octet trailer, except for the addition of an IP source address. This extended trailer is used to tunnel (and control the tunneling of) Source-Specific Multicast sessions.

"source cookie" and "destination cookie" are used to protect against IP source address spoofing, as described below.

"vers" - the protocol version - is currently zero.

The following "command"s are currently defined:

0	(unused)
1	DATA
2	JOIN_GROUP
3	LEAVE_GROUP
4	TEAR_DOWN
5	PROBE
6	PROBE_ACK
7	PROBE_NACK
8	JOIN_RTP_GROUP
9	LEAVE_RTP_GROUP
10-15	(unused)

Notes:

- The primary reason for having the tunneled payload data appear *before* the descriptor is that the payload data will often be a RTP packet [2], and by retaining this data in its usual position, IP/UDP/RTP header compression [3] - if present - will work properly. A secondary reason is that this order may allow UMTP 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. Note that UMTP implementations must allow for the possibility of the 'trailer' descriptor not being aligned on a 4-byte boundary.
- The S/vers/command fields always occupy the last byte of the data payload. This allows a receiving implementation to easily determine

the size of the trailer, and allows for the possibility of having other kinds of tunneling trailers (e.g., for trailer compression) in future versions of this protocol.

- The DATA, JOIN_GROUP, LEAVE_GROUP, JOIN_GROUP_RTP, and LEAVE_GROUP_RTP commands can be used with either a 12-octet trailer, or (if the "S" bit is set to 1) with a 16-octet trailer that adds a source address (specifying a SSM session). In the Protocol Operation described in the next section, replace "(group, port)" with "(group, source, port)" when dealing with a SSM session.

7. Protocol Operation

For the purposes of describing the protocol, a tunnel endpoint has the following state:

- a set E of allowable remote endpoints (each defined by a unicast address & port). Each such endpoint, e, is also tagged with:
    - localCookie_e: a hard-to-guess (e.g., random) 16-bit value
    - remoteCookie_e: a 16-bit value (initially, some arbitrary value)
- a set G of active (group, port) tuples. Each such tuple, g, is also tagged with:
    - a flag F_g with one of two values: {"master", "slave"}
    - a subset E_g of E: the tunnels that are members of g
    - a default TTL T_g (the TTL to use if one is not otherwise known)

Also, for each tunnel endpoint, the following events of note may occur:

1/ The local node requests the joining of a (group, port) g, with default TTL t
2/ The local node requests the leaving of a (group, port) g
3/ The local node sends a multicast packet to a (group, port) g, with TTL t
4/ A (non-local) multicast packet arrives for a (group, port) g, with source address s
5/ A 'JOIN_GROUP timeout' occurs on a (group, port) g
6/ A tunneled UMTP packet arrives, with source address s, and containing 'cookies' (srcCookie, dstCookie)

Each of these events is handled as follows:

```
1/ The local node requests the joining of a (group, port) g,
    with default TTL t
        add g to G; set F_g to "master"; set T_g to t
        locally join the multicast group g
        for each tunnel endpoint e in E_g,
                send, to e, at 15 second intervals, a command
                        JOIN_GROUP(group, port, t,
                                   srcCookie<-localCookie_e,
                                   dstCookie<-remoteCookie_e)

2/ The local node requests the leaving of a (group, port) g
        ignore this if g is not a member of G; otherwise:
        if F_g is "master"
```

```
                for each tunnel endpoint e in E_g,
                        cancel the ongoing periodic JOIN_GROUP commands
                        send, to e, a command
                                LEAVE_GROUP(group, port,
                                            srcCookie<-localCookie_e,
                                            dstCookie<-remoteCookie_e)
                                (The TTL field in the packet is unused)
                                (optional: Repeat this send several times)
                locally leave the multicast group g
                remove g from G

3/ The local node sends a multicast packet to a (group, port) g, with TTL t
        ignore this if g is not a member of G (or if t==0); otherwise:
        for each tunnel endpoint e in E_g,
                send, to e, a command
                        DATA(group, port, t-1,
                             srcCookie<-localCookie_e,
                             dstCookie<-remoteCookie_e)
                        with the original packet's data prepended

4/ A (non-local) multicast packet arrives for a (group, port) g,
    with source address s
        IMPORTANT (see below): If s is one of the endpoints e in E:
                send, to e, a command
                        TEAR_DOWN(srcCookie<-localCookie_e,
                                  dstCookie<-remoteCookie_e)
                        (the address, port, TTL fields are unused)
                remove e from E
        otherwise:
        ignore this if g is not a member of G; otherwise:
        if the TTL t of the incoming packet is not known:
                set t to T_g
        for each tunnel endpoint e in E_g,
                (if t>0) send, to e, a command
                        DATA(group, port, t-1,
                             srcCookie<-localCookie_e,
                             dstCookie<-remoteCookie_e)
                        with the original packet's data prepended

5/ A 'JOIN_GROUP timeout' occurs on a (group, port) g
        ignore this if g is not a member of G; otherwise:
        if F_g is "slave"
                remove g from G
                locally leave the multicast group g

6/ A tunneled UMTP packet arrives, with source address s,
    and containing 'cookies' (srcCookie, dstCookie)
        If s is not one of the endpoints e in E, ignore this packet
                *unless* the "command" is PROBE, in which case:
                        replace the packet's "command" field with PROBE_NACK,
                        swap the packet's source and destination cookie fields
```

```
                send the resulting packet back to s
                continue normal processing
If dstCookie does *not* equal localCookie_s [*]
        send, to s, a command
                PROBE_ACK(srcCookie<-localCookie_s,
                          dstCookie<-the srcCookie from the packet)
                (the address, port, TTL fields are ignored; they
                 should be given the same values as those in the
                 incoming packet)
        continue normal processing
        [*] (Note: An implementation may omit this check if it is sure
             that source addresses incoming packets cannot be spoofed.)
Set remoteCookie_s <- srcCookie
Process the packet, based on the "command" field:
DATA(group, port, t)
        set g to (group, port)
                (Note: It is not required that g be a member of G.)
        multicast the enclosed (prepended) data to g, with TTL t
                (optional: Instead limit the TTL; see below)
        (if t>0) for each tunnel endpoint e in E_g, EXCEPT for s
                send, to e, a command
                        DATA(group, port, t-1,
                             srcCookie<-localCookie_e,
                             dstCookie<-remoteCookie_e)
                        with the original packet's data prepended
JOIN_GROUP(group, port, t)
        set g to (group, port)
        ignore this if g is not a multicast address; otherwise:
        if g is not already a member of G:
                add g to G; set F_g to "slave"; set T_g to t
                locally join the multicast group g
        if F_g is "slave":
                set a 'JOIN_GROUP timeout' to occur in 60 seconds time
                        (replacing any existing such timeout)
LEAVE_GROUP(group, port)
        set g to (group, port)
        ignore this if g is not a member of G; otherwise:
        if F_g is "slave"
                locally leave the multicast group g
                remove g from G
TEAR_DOWN
        remove e from E
PROBE
        send, to s, a command
                PROBE_ACK(srcCookie<-localCookie_s,
                          dstCookie<-remoteCookie_s)
                (the address, port, TTL fields are ignored; they
                 should be given the same values as those in the
                 incoming packet)
PROBE_ACK, PROBE_NACK
        Ignore this packet (unless we have recently sent a PROBE)
```

Note: The PROBE command is one that a node can (optionally) use to determine whether a prospective endpoint exists, and if so, whether it would accept us as an endpoint in turn. A PROBE can also be used, by a master, to obtain the correct "remoteCookie" - via the subsequent PROBE_ACK - prior to sending its first JOIN_GROUP.

8. Illustration of Protocol Operation

In this simple example, consider a tunnel between a single pair of nodes, A and B, and suppose that node A (the master) wishes that data for a particular (group, port) "g" be relayed across the tunnel, with default TTL "t".

Initial state:

```
A (master)                          B (slave)
==========                          =========
E == {B}                            E == {A}
localCookie_B: 914                  localCookie_A: 1442
     (initially chosen at random)        (initially chosen at random)
remoteCookie_B: 2207 (ditto)        remoteCookie_A: 4913 (ditto)
G == {}                             G == {}
```

A begins by sending a PROBE command across the tunnel to B. This is optional, but it allows each endpoint to immediately 'synchronize' their cookies.

```
A --> PROBE(914,2207) -->B
```

B receives this command, and notices that the source (A) is one of his legitimate endpoints. However, he also notices that the command's destination cookie (2207) does *not* equal his local cookie (1442). Therefore, he sends back a PROBE_ACK, containing his correct cookie (and the source cookie from the original PROBE):

```
A<-- PROBE_ACK(1442,914) <--B
```

A receives this command, notices that the source (B) is one of his legitimate endpoints. He also notices that the command's destination cookie (914) equals his own local cookie. This authenticates the remote endpoint B (showing that it hasn't been spoofed), so A then sets his remoteCookie_B to 1442 (the local cookie from the incoming command).

Current state:

```
A (master)                          B (slave)
==========                          =========
E == {B}                            E == {A}
localCookie_B: 914                  localCookie_A: 1442
```

```
remoteCookie_B: 1442                remoteCookie_A: 4913
G == {}                             G == {}
```

Note that B hasn't yet authenticated the remote endpoint A. This happens after the next packet:

Next, A joins the multicast (group, port), and tells B to relay this:

```
       A--> JOIN_GROUP(group, port, t, 914, 1442) -->B
```

(A repeats this command every 15 seconds.)

B receives each such command, authenticates its endpoint and cookies, sets his remoteCookie_A to 914, and handles the JOIN_GROUP command by

- locally joining the group (if it hasn't already done so)
- setting F_g to "slave"
- setting a "JOIN_GROUP timeout" to occur in 60 seconds (if no more JOIN_GROUP commands are received within this time)

Current state:

```
A (master)                          B (slave)
==========                          =========
E == {B}                            E == {A}
localCookie_B: 914                  localCookie_A: 1442
remoteCookie_B: 1442                remoteCookie_A: 914
G == {g}                            G == {g}
F_g == "master"                     F_g == "slave"
E_g == {B}                          E_g == {A}
T_g == t                            T_g == t
```

Suppose now that B receives a local multicast packet addressed to "g". He then sends a DATA command across the tunnel to A:

```
    A<-- DATA(group, port, t-1, 914, 1442) <--B
         (with the original data prepended)
```

(If B knew the TTL of the incoming packet, he'd use this instead of t.)

A then receives this command, authenticates its endpoint and cookies, and re-multicasts the enclosed payload data to (group, port).

(The same steps happen, in the reverse direction, if A receives a local multicast packet addressed to "g".)

9. Relaying RTP/RTCP Sessions

RTP/RTCP sessions [2] use two ports: an even numbered port for RTP, and an odd-numbered port (the next highest) for RTCP. These sessions could be relayed using two separate JOIN_GROUP commands - one for each port.

As an alternative, the single command JOIN_RTP_GROUP can be used. This command works exactly like JOIN_GROUP, except that it implicitly specifies a pair of ports: the port in the command, and that port +1. Similarly, the command LEAVE_RTP_GROUP can be used to stop relaying a RTP/RTCP session, as an alternative to using two separate LEAVE_GROUPs.

A UMTP master should use JOIN_RTP_GROUP and LEAVE_RTP_GROUP only for RTP/RTCP sessions - not for other kinds of sessions that also happen to use port pairs. A UMTP implementation may handle these commands especially, based upon the knowledge that they represent RTP/RTCP sessions. For example, an implementation might wish to perform RTP/RTCP-specific monitoring or statistics gathering, or to check the RTP SSRC ("synchronization source") field in each incoming multicast packet for possible collisions (i.e., in case two separate multicast sources happen to be using the same SSRC, but have not yet detected and corrected this themselves) [4].

10. Loop Detection and Avoidance

A data loop may occur if the two endpoints of a UMTP tunnel are connected by multicast, or via another UMTP tunnel elsewhere. Each UMTP implementation must take steps to prevent a loop from occurring:
- When multicasting a payload extracted from a DATA packet, a UMTP implementation should choose a TTL that's no larger than necessary. It must also ensure that if this packet is then re-received (via loop-back), it is not resent back over the same tunnel.
- If a UMTP implementation receives a multicast packet whose source address is also the endpoint of a tunnel, it must immediately shut down this tunnel (& send a TEAR_DOWN command to the endpoint)
- If a UMTP implementation is running on a node for which no more than one tunnel is expected (e.g., the node is a non-MBone-connected PC), then this implementation should attempt to ensure that no more than one tunnel can be started on this note. (For example, the implementation could use operating system-level locking to prevent more than one copy of itself from running simultaneously.)
- If loops in the tunneling topology remain possible, then each end of the tunnel should periodically send a short 'status' packet - containing its unicast address - to a common multicast address, and also listen on this address, checking the contents of each received status packet. Should these contents be the same as its original status packet, it must immediately shut down all of its tunnels.
(Note: These are the same loop detection techniques used by "mTunnel" [5] - a similar multicast tunneling system, developed independently.)

11. Security Considerations

Each UMTP implementation should specify, in advance, its set of allowable endpoints (E), and thus should not permit arbitrary nodes to form tunnels (unless some additional authentication mechanism is used for this purpose).

Tunnels are authenticated by IP source address. However, the 'cookie' mechanism protects against source address spoofing, thereby preventing a malicious third party from - for example - joining an unwanted multicast group. To enhance this protection, an implementation may also choose to occasionally change its 'localCookies' while it is running. (This should be done immediately prior to sending a packet across the tunnel, so that the remote endpoint can learn about the new cookie immediately.)

Note, however, that the 'cookie' mechanism protects against source address spoofing only if packets cannot be inspected in transit. Thus, UMTP should be used with caution on insecure networks (such as unencrypted wireless LANs).

12. References

[1] LIVE.COM,
The "liveGate" multicast tunneling server
http://www.live.com/liveGate/
(This application is an implementation of UMTP.)

[2] Schulzrinne, H., Casner, S., Frederick, R., and Jacobson, V.,
"RTP: A Transport Protocol for Real-Time Applications", RFC 3550,
July, 2003.

[3] Jacobson, V., Casner, S.,
"Compressing IP/UDP/RTP Headers for Low-Speed Serial Links"
RFC 2508, February, 1999.

[4] Casner, S.,
Personal communication, December, 1997.

[5] Parnes, P.,
"mTunnel"
http://www.cdt.luth.se/~peppar/progs/mTunnel/

[6] Holbrook, H., Cain, B.,
"Source-Specific Multicast for IP"
Work-in-Progress, Internet-Draft "draft-ietf-ssm-arch-03.txt"
May, 2003.

13. Author's Address

Ross Finlayson,
Live Networks, Inc. (LIVE.COM)
email: finlayson(at)live.com
WWW: http://www.live.com/

</x-flowed>