Available online at www.elixirpublishers.com (Elixir International Journal)

Network Engineering

Elixir Network Engg. 42 (2012) 6499-6502

ISSN: 2229-712X

Transport Layer and UDP

P.Suresh
Department of Computer Science, Salem Sowdeswari College, Salem, Tamilnadu, India.

ARTICLE INFO

Article history:
Received: 20 November 2011;
Received in revised form:
19 January 2012;
Accepted: 30 January 2012;

Keywords
Transport Layer,
User Datagram Protocol (UDP),
Remote Procedure Call.

ABSTRACT
The User Datagram Protocol (UDP) is one of the core members of the Internet Protocol Suite, the set of network protocols used for the Internet. With UDP, computer applications can send messages, in this case referred to as datagrams, to other hosts on an Internet Protocol (IP) network without requiring prior communications to set up special transmission channels or data paths. UDP uses a simple transmission model without implicit handshaking dialogues for providing reliability, ordering, or data integrity. Thus, UDP provides an unreliable service and datagrams may arrive out of order, appear duplicated, or go missing without notice. UDP assumes that error checking and correction is either not necessary or performed in the application, avoiding the overhead of such processing at the network interface level. Time-sensitive applications often use UDP because dropping packets is preferable to waiting for delayed packets, which may not be an option in a real-time system. The Internet has two main protocols in the transport layer, a connectionless protocol and a connection-oriented one. The connectionless protocol is UDP. The connection-oriented protocol is TCP. Because UDP is basically just IP with a short header added.

Introduction

The Internet protocol suite supports a connectionless transport protocol, UDP (User Datagram Protocol). UDP provides a way for applications to send encapsulated IP datagrams and send them without having to establish a connection. UDP is described in RFC 768. UDP transmits segments consisting of an 8-byte header followed by the payload. The header is shown in Fig. 1. The two ports serve to identify the end points within the source and destination machines. When a UDP packet arrives, its payload is handed to the process attached to the destination port. This attachment occurs when BIND primitive or something similar is used, in TCP (the binding process is the same for UDP). In fact, the main value of having UDP over just using raw IP is the addition of the source and destination ports. Without the port fields, the transport layer would not know what to do with the packet. With them, it delivers segments correctly.

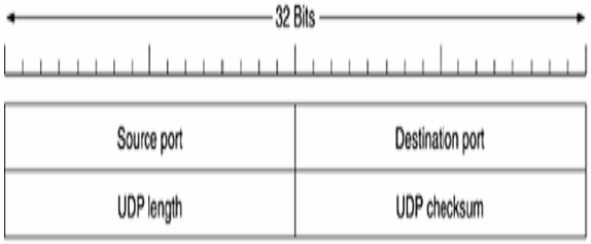

Figure 1. The UDP header

The source port is primarily needed when a reply must be sent back to the source. By copying the source port field from the incoming segment into the destination port field of the outgoing segment, the process sending the reply can specify which process on the sending machine is to get it.

The UDP length field includes the 8-byte header and the data. The UDP checksum is optional and stored as 0 if not computed (a true computed 0 is stored as all 1s). Turning it off is foolish unless the quality of the data does not matter (e.g., digitized speech).

It is probably worth mentioning explicitly some of the things that UDP does not do. It does not do flow control, error control, or retransmission upon receipt of a bad segment. All of that is up to the user processes. What it does do is provide an interface to the IP protocol with the added feature of demultiplexing multiple processes using the ports. That is all it does. For applications that need to have precise control over the packet flow, error control, or timing, UDP provides just what the doctor ordered.

One area where UDP is especially useful is in client-server situations. Often, the client sends a short request to the server and expects a short reply back. If either the request or reply is lost, the client can just time out and try again. Not only is the code simple, but fewer messages are required (one in each direction) than with a protocol requiring an initial setup.

Remote Procedure Call

In a certain sense, sending a message to a remote host and getting a reply back is a lot like making a function call in a programming language. In both cases you start with one or more parameters and you get back a result. This observation has led people to try to arrange request-reply interactions on networks to be cast in the form of procedure calls. Such an arrangement makes network applications much easier to program and more familiar to deal with. For example, just imagine a procedure named get_IP_address (host_name) that works by sending a UDP packet to a DNS server and waiting for the reply, timing out and trying again if one is not forthcoming quickly enough. In this way, all the details of networking can be hidden from the programmer.

In a nutshell, what Birrell and Nelson suggested was allowing programs to call procedures located on remote hosts.

Tele:
E-mail addresses: sur_bhoo71@rediffmail.com

When a process on machine 1 calls a procedure on machine 2, the calling process on 1 is suspended and execution of the called procedure takes place on 2. Information can be transported from the caller to the callee in the parameters and can come back in the procedure result. No message passing is visible to the programmer. This technique is known as RPC (Remote Procedure Call) and has become the basis for many networking applications. Traditionally, the calling procedure is known as the client and the called procedure is known as the server.

The idea behind RPC is to make a remote procedure call look as much as possible like a local one. In the simplest form, to call a remote procedure, the client program must be bound with a small library procedure, called the client stub, that represents the server procedure in the client's address space. Similarly, the server is bound with a procedure called the server stub. These procedures hide the fact that the procedure call from the client to the server is not local.

The actual steps in making an RPC are shown in Fig. 2. Step 1 is the client calling the client stub. This call is a local procedure call, with the parameters pushed onto the stack in the normal way. Step 2 is the client stub packing the parameters into a message and making a system call to send the message. Packing the parameters is called marshaling. Step 3 is the kernel sending the message from the client machine to the server machine. Step 4 is the kernel passing the incoming packet to the server stub. Finally, step 5 is the server stub calling the server procedure with the unmarshaled parameters. The reply traces the same path in the other direction.

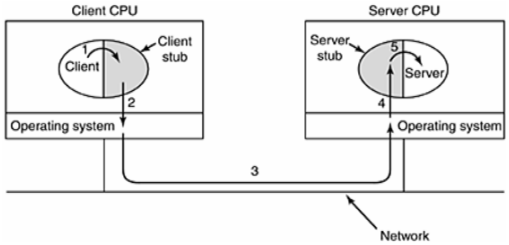

Figure 2. Steps in making a remote procedure call. The stubs are shaded

The key item to note here is that the client procedure, written by the user, just makes a normal (i.e., local) procedure call to the client stub, which has the same name as the server procedure. Since the client procedure and client stub are in the same address space, the parameters are passed in the usual way. Similarly, the server procedure is called by a procedure in its address space with the parameters it expects. To the server procedure, nothing is unusual. In this way, instead of I/O being done on sockets, network communication is done by faking a normal procedure call.

A big one is the use of pointer parameters. Normally, passing a pointer to a procedure is not a problem. The called procedure can use the pointer in the same way the caller can because both procedures live in the same virtual address space. With RPC, passing pointers is impossible because the client and server are in different address spaces.

In some cases, tricks can be used to make it possible to pass pointers. Suppose that the first parameter is a pointer to an integer, k. The client stub can marshal k and send it along to the server. The server stub then creates a pointer to k and passes it to the server procedure, just as it expects. When the server procedure returns control to the server stub, the latter sends k back to the client where the new k is copied over the old one, just in case the server changed it. In effect, the standard calling sequence of call-by-reference has been replaced by copy restore. Unfortunately, this trick does not always work, for example, if the pointer points to a graph or other complex data structure. For this reason, some restrictions must be placed on parameters to procedures called remotely.

The RPC need not use UDP packets, but RPC and UDP are a good fit and UDP is commonly used for RPC. However, when the parameters or results may be larger than the maximum UDP packet or when the operation requested is not idempotent (i.e., cannot be repeated safely, such as when incrementing a counter), it may be necessary to set up a TCP connection and send the request over it rather than use UDP.

The Real-Time Transport Protocol

Client-server RPC is one area in which UDP is widely used. Another one is real-time multimedia applications. In particular, as Internet radio, Internet telephony, music-on-demand, videoconferencing, video-on-demand, and other multimedia applications became more commonplace, people discovered that each application was reinventing more or less the same real-time transport protocol. It gradually became clear that having a generic real-time transport protocol for multiple applications would be a good idea. Thus was RTP (Real-time Transport Protocol) born.

The position of RTP in the protocol stack is somewhat strange. It was decided to put RTP in user space and have it (normally) run over UDP. It operates as follows. The multimedia application consists of multiple audio, video, text, and possibly other streams. These are fed into the RTP library, which is in user space along with the application. This library then multiplexes the streams and encodes them in RTP packets, which it then stuffs into a socket. At the other end of the socket (in the operating system kernel), UDP packets are generated and embedded in IP packets. If the computer is on an Ethernet, the IP packets are then put in Ethernet frames for transmission. The protocol stack for this situation is shown in Fig. 3(a). The packet nesting is shown in Fig. 3(b).

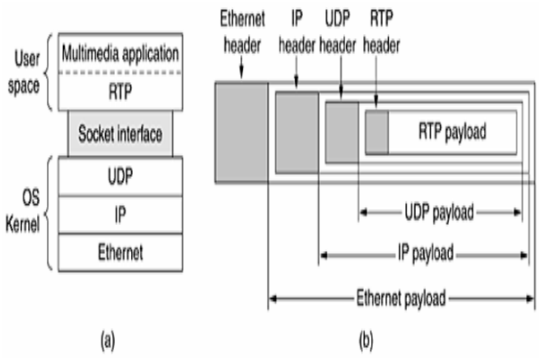

Figure 3. (a) The position of RTP in the protocol stack. 3. (b) Packet nesting

As a consequence of this design, it is a little hard to say which layer RTP is in. Since it runs in user space and is linked to the application program, it certainly looks like an application protocol. On the other hand, it is a generic, application-independent protocol that just provides transport facilities, so it also looks like a transport protocol. Probably the best description is that it is a transport protocol that is implemented in the application layer.

The basic function of RTP is to multiplex several real-time data streams onto a single stream of UDP packets. The UDP stream can be sent to a single destination (unicasting) or to multiple destinations (multicasting). Because RTP just uses

normal UDP, its packets are not treated specially by the routers unless some normal IP quality-of-service features are enabled. In particular, there are no special guarantees about delivery, jitter, etc.

Each packet sent in an RTP stream is given a number one higher than its predecessor. This numbering allows the destination to determine if any packets are missing. If a packet is missing, the best action for the destination to take is to approximate the missing value by interpolation. Retransmission is not a practical option since the retransmitted packet would probably arrive too late to be useful. As a consequence, RTP has no flow control, no error control, no acknowledgements, and no mechanism to request retransmissions.

Each RTP payload may contain multiple samples, and they may be coded any way that the application wants. To allow for interworking, RTP defines several profiles (e.g., a single audio stream), and for each profile, multiple encoding formats may be allowed. For example, a single audio stream may be encoded as 8-bit PCM samples at 8 kHz, delta encoding, predictive encoding, GSM encoding, MP3, and so on. RTP provides a header field in which the source can specify the encoding but is otherwise not involved in how encoding is done.

Another facility many real-time applications need is timestamping. The idea here is to allow the source to associate a timestamp with the first sample in each packet. The timestamps are relative to the start of the stream, so only the differences between timestamps are significant. The absolute values have no meaning. This mechanism allows the destination to do a small amount of buffering and play each sample the right number of milliseconds after the start of the stream, independently of when the packet containing the sample arrived. Not only does time stamping reduce the effects of jitter, but it also allows multiple streams to be synchronized with each other. For example, a digital television program might have a video stream and two audio streams. The two audio streams could be for stereo broadcasts or for handling films with an original language soundtrack and a soundtrack dubbed into the local language, giving the viewer a choice. Each stream comes from a different physical device, but if they are timestamped from a single counter, they can be played back synchronously, even if the streams are transmitted somewhat erratically.

The RTP header is illustrated in Fig. 4. It consists of three 32-bit words and potentially some extensions. The first word contains the Version field, which is already at 2. Let us hope this version is very close to the ultimate version since there is only one code point left (although 3 could be defined as meaning that the real version was in an extension word). 32 bits.

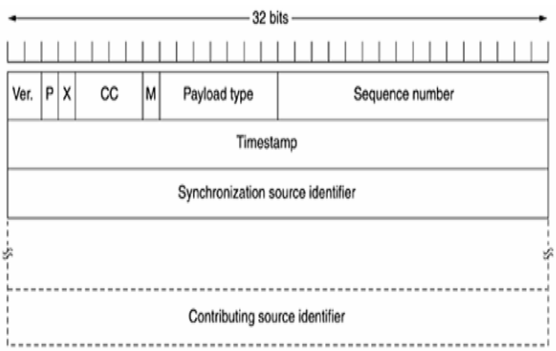

Figure 4. The RTP header

The P bit indicates that the packet has been padded to a multiple of 4 bytes. The last padding byte tells how many bytes were added. The X bit indicates that an extension header is present. The format and meaning of the extension header are not defined. The only thing that is defined is that the first word of the extension gives the length. This is an escape hatch for any unforeseen requirements.

The CC field tells how many contributing sources are present, from 0 to 15. The M bit is an application-specific marker bit. It can be used to mark the start of a video frame, the start of a word in an audio channel, or something else that the application understands. The Payload type field tells which encoding algorithm has been used. Since every packet carries this field, the encoding can change during transmission. The Sequence number is just a counter that is incremented on each RTP packet sent. It is used to detect lost packets.

The timestamp is produced by the stream's source to note when the first sample in the packet was made. This value can help reduce jitter at the receiver by decoupling the playback from the packet arrival time. The Synchronization source identifier tells which stream the packet belongs to. It is the method used to multiplex and demultiplex multiple data streams onto a single stream of UDP packets. Finally, the Contributing source identifiers, if any, are used when mixers are present in the studio. In that case, the mixer is the synchronizing source, and the streams being mixed are listed here.

Realtime Transport Control Protocol

RTP has a little sister protocol (little sibling protocol?) called RTCP (Realtime Transport Control Protocol). It handles feedback, synchronization, and the user interface but does not transport any data. The first function can be used to provide feedback on delay, jitter, bandwidth, congestion, and other network properties to the sources. This information can be used by the encoding process to increase the data rate (and give better quality) when the network is functioning well and to cut back the data rate when there is trouble in the network. By providing continuous feedback, the encoding algorithms can be continuously adapted to provide the best quality possible under the current circumstances. For example, if the bandwidth increases or decreases during the transmission, the encoding may switch from MP3 to 8-bit PCM to delta encoding as required. The Payload type field is used to tell the destination what encoding algorithm is used for the current packet, making it possible to vary it on demand.

RTCP also handles interstream synchronization. The problem is that different streams may use different clocks, with different granularities and different drift rates. RTCP can be used to keep them in sync. Finally, RTCP provides a way for naming the various sources (e.g., in ASCII text). This information can be displayed on the receiver's screen to indicate who is talking at the moment.

Conclusion

UDP can provide many advantages over TCP communication, depending on the application in question. The decision regarding whether to select UDP or TCP as your communication protocol comes down to speed versus reliability. With Lab Windows/CVI, you have built-in support for the most commonly used network protocols, such as TCP, Network Variables, and now UDP, which gives you the flexibility to choose which protocol is right for your application.

Reference

1. J.Postel, "User Datagram Protocol,", August 1980.

2. R. Braden, D. Borman, C. Partridge, "Computing The Internet Checksum,", September 1988.
3. J. Mahdavi and S. Floyd, "The TCP-Friendly Website," http://www.psc.edu/networking/tcp_friendly.html
4. *Andrew S. Tanenbaum "Computer Networks" Fourth Edition, August 19, 2002.*
5. A. Okmianski "Transmission of Syslog Messages over UDP", March 2009
6. J.Postel., "Internet Protocol", STD 5, September 1981.