A Kernel-Level RTP for Efficient Support of Multimedia Service on Embedded Systems

Dong Guk Sun and Sung Jo Kim

Chung-Ang University, 221 HukSuk-Dong,
DongJak-Ku, Seoul, Korea, 156-756
{dgsun, sjkim}@konan.cse.cau.ac.kr

Abstract. Since the RTP is suitable for real-time data transmission in multimedia services like VoD, AoD, and VoIP, it has been adopted as a real-time transport protocol by RTSP, H.323, and SIP. Even though the RTP protocol stack for embedded systems has been in great need for efficient support of multimedia services, such a stack has not been developed yet. In this paper, we explain *embeddedRTP* which supports the RTP protocol stack at the kernel level so that it is suitable for embedded systems. Since *embeddedRTP* is designed to reside in the UDP module, existing applications which rely on TCP/IP services can be processed the same as before, while applications which rely on the RTP protocol stack can request RTP services through *embeddedRTP*'s API. Our performance test shows that packet-processing speed of *embeddedRTP* is about 7.8 times faster than that of UCL RTP for multimedia streaming services on PDA in spite that its object code size is reduced about by 58% with respect to UCL RTP's.

1 Introduction

Multimedia services on embedded systems can be classified into on-demand multimedia services like VoD and AoD, and Internet telephone service like video conference system and VoIP. H.323[1], SIP[2] and RTSP[3] are representative protocols that support these services. In fact, these protocols utilize RTP[4] for multimedia data transmission. Therefore, embedded systems must support RTP in order to provide multimedia services.

Most recent researches related to RTP have focused on implementation of RTP library for their own application. Typical implementations include RADVISION company's RTP/RTCP protocol stack toolkit[5], Bell Lab's RTPlib[6], common multimedia library[7] of UCL(University College London) and vovida.org's VOCAL(the Vovida Open Communication Application Library)[8]. These libraries can be used along with traditional operating systems such as LINUX or UNIX. Among these implementations, RADVISION Company's RTP/RTCP toolkit[5] is a general library that can be used in an embedded system as well as a large-scale server system. This RTP/RTCP toolkit, however, is not suitable for embedded system; this is because it did not consider typical characteristics of embedded system such as memory shortage. No RTP for embedded systems has been developed so far.

In this paper, we design and implement RTP at the kernel level(called as **embeddedRTP**) to support smooth multimedia service in embedded systems. The primary goals of *embeddedRTP* are to guarantee the small code size, fast packet-processing and high resource utilization. Furthermore, *embeddedRTP* can resolve resource-wasting problem caused by RTP that was implemented redundantly by applications.

This paper is organized as follow. In Section 2, we discuss current RTP's problems and propose a solution to them. Then, Section 3 explains design and implementation of *embeddedRTP*. Section 4 presents performance evaluation of *embeddedRTP* and finally Section 5 concludes our work and discusses future work.

2 Current RTP's Problem

Because RTP is not a mandatory protocol to use Internet, network modules of embedded system do not necessarily include RTP. Therefore, each application requiring RTP in an embedded system should contain its own RTP as shown in Fig.1. In this case, if embedded system must support various kinds of multimedia services such as H.323[1], SIP[2] and RTSP[3], these applications must implement RTP redundantly. Therefore, this method is inappropriate to embedded systems, which do not have sufficient amount of memory.

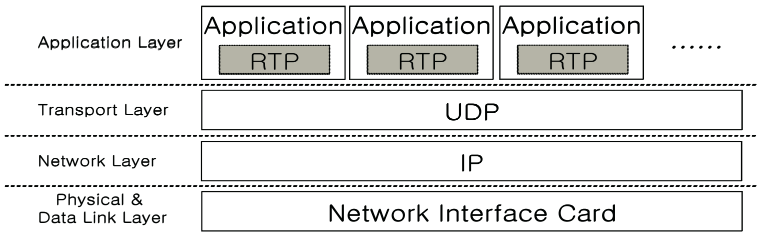

Fig. 1. Protocol stack in case that RTP is included in each application

RTP can be also offered as a library for application layer as shown in Fig.2. This method can resolve RTP's redundancy problem. However, when data is transmitted from UDP to RTP and subsequently from RTP to an application, overhead due to memory copy and context switching may occur. Since various applications must share one library, library compatibility problem may also occur.

If we implement RTP at the kernel level as shown in Fig.3, the protocol redundancy and context switching problems can be resolved. Moreover, we can reduce the code size of applications. To check if a received packet is in fact a RTP packet, port numbers of all UDP packets should be examined. This checking

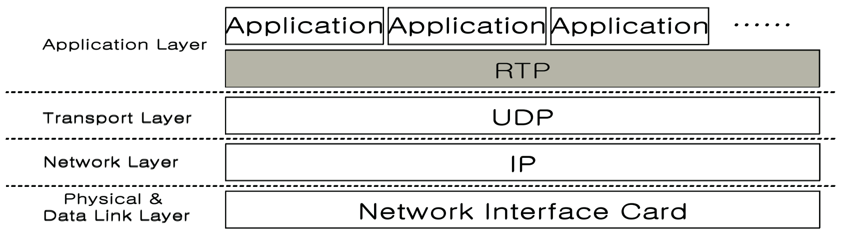

Fig. 2. Protocol stack in case that RTP is supported by library

may cause overhead to other application layer protocols that use UDP. However, such application layer protocols as TFTP, DHCP and ECHO are not used in multimedia communication and do not cause much traffics. Moreover, these do not require high performance. Consequently, we can expect that overhead caused by RTP checking does not have serious effect on performance of these application layer protocols.

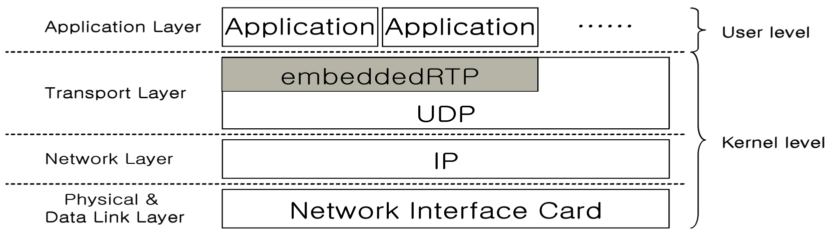

Fig. 3. Protocol stack in case that *embeddedRTP* is implemented at the kernel level

3 Design and Implementation of *embeddedRTP*

3.1 Overall Structure

For efficient memory usage, fast packet processing and removal of redundancy problem, *embeddedRTP* is designed to reside in the UDP module as shown in Fig.3. Fig.4 shows overall structure of *embeddedRTP*.

EmbeddedRTP is composed of API, session checking module, RTP packet-reception module, RTP packet-processing module, RTCP packet-reception module, RTCP packet-transmission module and RTCP packet-processing module. When an application uses TCP or UDP, BSD socket layer can be utilized as before. On the other hand, applications requiring RTP can utilize *embeddedRTP*'s API for RTP services.

3.2 Communication Mechanism

Since RTP is implemented at the kernel level in *embeddedRTP*, communication channel between RTP module of application layer and UDP module of the

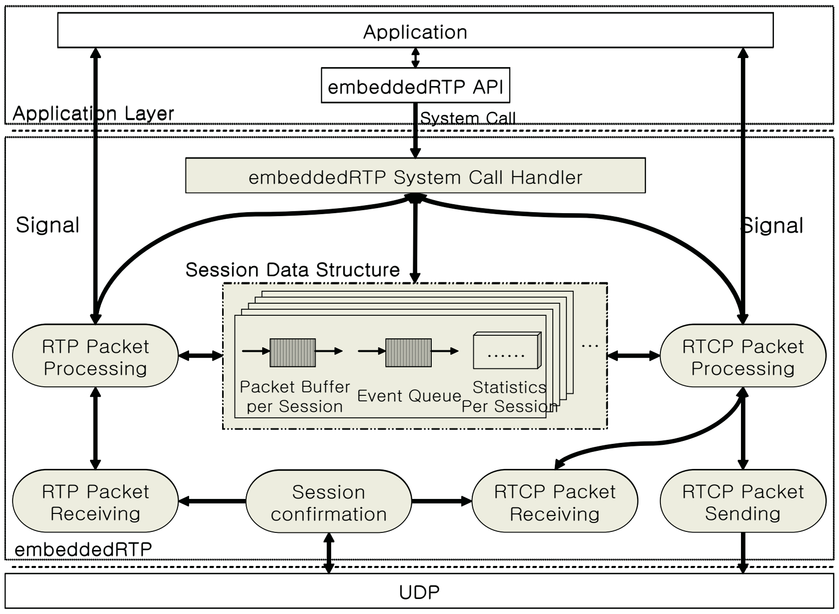

Fig. 4. Overall Structure of *embeddedRTP*

kernel should be changed to communication channel between *embeddedRTP* and UDP module of the kernel. For communication between application and *embeddedRTP*, special mechanisms supported by OS should be provided.

Application Calls *EmbeddedRTP*: In this case, we use system call as a communication channel. System call is the one and only method by which application can use kernel modules in LINUX system. However, the current LINUX kernel does not include RTP-related system calls. Therefore, we register new system calls at the kernel so that application can call *embeddedRTP*.

***EmbeddedRTP* Calls Applications:** In this case, we use signal and event queue as a communication channel. Signal is a message transmission mechanism that is used generally in LINUX, but has following problems. First, since we can only use predefined signals, the signal mechanism cannot be used in data transmission. Second, since the types of signals that can be transmitted are not diverse, this mechanism's usage should be restricted. Finally, there is no mechanism to store signals. Therefore, if another signal arrives before the current signal is handled, the current one will be overridden. Nonetheless, it is enough for notifying the occurrences of events. The signal overriding problem can be resolved by event queue. When an event occurs, it is stored in the event queue and a signal is sent to an application to notify occurrence of an event. Event queue used in *embeddedRTP* manages FIFO(First-In First-Out) queue which is the simplest form of queue. And each queue entry stores the multimedia session ID and event arisen.

3.3 Session Management

Session management is important in *embeddedRTP* because it should be used by all applications which need RTP services. When RTP is implemented at application layer, it is enough for applications themselves to manage their session. However, in *embeddedRTP*, all multimedia sessions must be managed by the kernel.

As shown in Fig.5, multimedia session is managed by one-dimensional array combined with circular queue. Each element of the array contains session information, which includes transmission related data such as the server address and port number, event queue and packet buffer related data structures, and statistical information. In this mechanism, each session can be identified by the index of array; so session information can be accessed quickly. Moreover, *front* and *rear* pointers can prevent the queue from overflowing. However, since there is trade-off between the number of multimedia session and memory usage, it is difficult to decide the proper queue size.

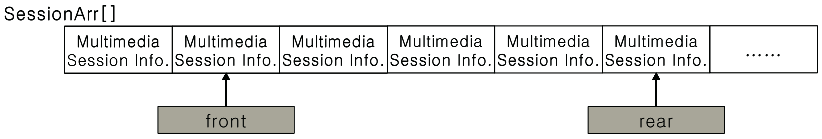

Fig. 5. Data Structure for Session Management

3.4 Session Checking Module

When an RTP packet is received, UDP module does all processing related to UDP packets such as checksum. After that, UDP module stores the packet in socket buffer. At this time, this module checks whether there is a multimedia session associated with the received packet. If a multimedia session exists, RTP/RTCP packet-reception module is called according to the packet type.

3.5 RTP Packet-Reception Module

RTP packet-reception module reads in a packet from socket buffer and sets packet header structure. After that, it checks the version field and length of packets to examine that packets are valid. If valid packets are received, it calls RTP packet-processing module. Received RTP packet is managed by *rtp_packet* structure. This structure consists of *rtp_packet_data* structure that composes packet buffer and *rtp_packet_header* that stores header information.

3.6 RTP Packet-Processing Module

RTP packet-processing module inserts a packet into packet buffer, and updates statistical information. After that, this module informs application that the RTP packet be received.

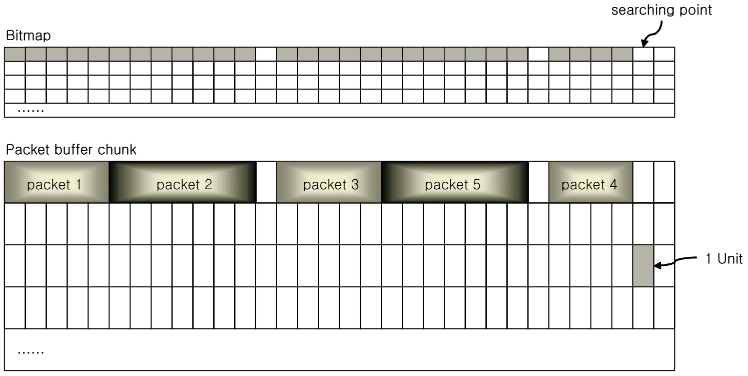

Fig. 6. Packet Buffer

Packet Buffer: Packet buffer stores received RTP packets. As shown in Fig.6, packet buffer is consisted of packet buffer chunk and bitmap. The former consists of units with fixed size, while the latter is a bit-stream that shows whether each unit of packet buffer chunk is available or not.

When a RTP packet is received, RTP packet-processing module checks if it can be stored contiguously from a unit in packet buffer chunk corresponding to searching point[1] of bitmap. If so, it is stored in packet buffer chunk; otherwise, searching starts from the next available searching point repeatedly until enough units can be found. Once it has been stored in packet buffer chunk, bitmap corresponding to the allocated units are set.

This mechanism utilizes memory more efficiently as the size of unit becomes smaller. On the other hand, as it becomes smaller, the size of bitmap becomes larger, resulting in longer searching time for bitmap. Since there is trade-off between memory efficiency and searching time, the unit size should be determined appropriately according to the resources of target systems and the characteristics of applications.

Packet Ordering: Since RTP uses UDP as a transmission protocol, RTP cannot guarantee that packets are received sequentially(See Fig.6). Therefore, in order to guarantee timeliness of multimedia data, *embeddedRTP* must maintain the order of packets that a server transmits. We use circular queue using doubly linked list to maintain the order of packets. Each node of circular queue stores information on a packet stored in packet buffer chunk. *Head* and *tail* pointers of circular queue indicate the first and last packet in the ordered list, respectively.

Each received packet will not be inserted into packet buffer from the unit which is adjacent to the unit pointed by *tail* pointer, unconditionally. Instead, it is inserted in a suitable place of the queue according to its sequence number. If the sequence number is the same as a stored packet, it is dropped because it

[1] The location of bitmap corresponding to a unit of packet buffer chunk from which to search for enough unit(s) to store a packet.

is duplicated one. Consequently, the application can access packets sequentially from the head pointer.

3.7 RTCP

RTCP packet-reception module acts similar to RTP packet-reception module. It reads in a packet from socket buffer and checks the validity of packet. After that, it calls RTCP packet-processing module.

RTCP packet-processing module checks if received packets are in a multimedia session using the SSRC(Synchronization Source) field of RTCP header. If so, it calls relevant processing routine according to the RTCP message type. After RTCP messages are processed, it stores reception event into event queue and sends a signal to the application to inform that RTCP message be received.

When an RTCP packet needs to be transmitted, RTCP packet-transmission module creates and transmits the packet with calculated statistical information. This packet includes various RTCP messages such as RR(Receiver Report), SDES(Source Description) and BYE. Upon transmitting the packet, it resets statistical information and terminates its execution.

4 Performance Evaluations

To evaluate performance of *embeddedRTP*, we measure the packet-processing time, memory requirement, the code size, and session checking overhead. We then compare them with those of UCL RTP library, which has been used in MPEG4IP project[9].

4.1 Packet-Processing Time

Packet-processing time is one of the most important factors to evaluate the performance of protocol stack. Packet-processing time is defined as time between the moment when RTP packets are confirmed in multimedia session and the moment when the application consumes all the packets, after RTP packet-processing has been done in UDP module. While tens of RTP packets are transmitted per second, RTCP packets are transmitted about every 5 seconds. Therefore, we excluded RTCP packet-processing time in our measurement. Table 1 shows the measured packet-processing time.

Table 1. RTP Packet-Processing Time(ms/packet)

	EmbeddedRTP	UCL RTP
Packet-Processing Time	0.289	2.253

As shown in Table 1, *embeddedRTP* is about 7.8 times faster than UCL RTP library. The biggest cause of this difference is that *embeddedRTP* uses statically allocated memory, while UCL RTP library allocates packet storage data structure dynamically. Another cause is the time when packet is stored. While *embeddedRTP* stores packets at the kernel level, UCL RTP library stores packets at the application layer. Therefore, UCL RTP library can be preempted to other process by kernel scheduler while received RTP packets are being processed. On the other hand, *embeddedRTP* is not preempted because RTP packets are processed in the LINUX kernel.

4.2 Memory Requirement

To compare the memory requirement, four sample animations which have different bit and frame rates are used in measurement. We measure the average payload size of sample animation, the average number of packets being buffered and the number of units that are used in *embeddedRTP*. Since the average payload size of animations used in the measurement tends to be multiple of a roughly 200 bytes, we adopt 200 bytes as the size of unit. The media buffering time when to start animation playback greatly affect the amount of memory to be used. While buffering time is about 5 - 10 seconds for playing sample animations, in general, we allow only 2 seconds as buffering time to reduce memory requirement without affecting animation playback.

Table 2. Memory requirement of UCL RTP and *embeddedRTP*

Sample	Payload (bytes)	Buffered Packets	Units	Memory Requirement(bytes)		Bit Rates	Frame Rates
				UCL RTP	*EmbeddedRTP*		
1	798	27	4.683	40,087	25,031	125	15
2	849	57	4.935	85,580	56,315	277	30
3	389	87	2.778	131,180	47,469	101	30
4	280	58	2.099	86,594	24,231	68	30

As shown in Table 2, *embeddedRTP*'s memory requirement is about 28% - 65% than UCL RTP's. Also, the amount of memory required by sample animations varies according to their characteristics. Sample 2 and 4 have the similar number of buffered packets, but their payload sizes differ as much as 570 bytes. Since UCL RTP library uses the equal sized data structure regardless of the payload size, memory requirement are similar. On the other hand, in *embeddedRTP*, sample 2 and 4 require about 65% and about 28% of memory required by UCL RTP library, respectively.

The payload size of sample 1 is greater than that of sample 4, while the average number of sample 1's buffered packets is much less than sample 4's. Considering these two samples, *embeddedRTP*'s memory requirements are similar in both samples. However, in UCL RTP library, memory requirements increase proportionally by the average number of buffered packets since the library allocates pre-determined amount of memory per packet. Since sample 3 has smaller payload size comparing with other samples, but has more average number of buffered packets, sample 3 shows the largest difference in memory requirement between *embeddedRTP* and UCL RTP.

4.3 Code Size

Because embedded systems have much smaller physical memory than desktop PC or server system, the code sizes of software is very important in embedded systems. Table 3 shows the code size of *embeddedRTP* and that of UCL RTP library. The former is determined by the total size of *embeddedRTP* modules that are implemented at the kernel level and *embeddedRTP*'s API. The latter is determined only by modules relevant to packet reception. As shown in Table 3, *embeddedRTP*'s code size is reduced to 42% of UCL RTP's.

Table 3. Code Size(bytes)

EmbeddedRTP		UCL RTP Library
EmbeddedRTP Modules	*EmbeddedRTP*'s API	
25,080	804	61,132
25,884		

4.4 Session Checking Overhead

In *embeddedRTP*, all packets received by network system should be checked whether they are in fact RTP packets. Therefore, we measure session checking overhead for RTP packets which are not in a multimedia session and other UDP packets which are not a RTP or RTCP packet. In the worst case, it takes 0.014 ms per packet. This time is relatively short comparing with packet-processing time of *embeddedRTP*(0.289 ms). Note that such application protocols that use UDP but are not be used in multimedia services as TFTP or DHCP, neither generate much traffics nor require high performance. Consequently, the overhead caused by session checking does not have serious effects on performance of these application layer protocols.

5 Conclusions and Future Work

In this paper, we explained *embeddedRTP* which supports the RTP protocol stack at the kernel level so that it is suitable for embedded systems. Since *embeddedRTP* is designed to reside in the UDP module, existing applications which rely on TCP/IP services can be processed the same as before, while applications which rely on the RTP protocol stack can request RTP services through *embeddedRTP*'s API. *EmbeddedRTP* stores received RTP packets into per session packet buffer, using the packet's port number and multimedia session information. Communications between applications and *embeddedRTP* is performed through system calls and signal mechanisms. Additionally, *embeddedRTP*'s API makes it possible to develop applications more conveniently. Our performance test shows that packet-processing speed of *embeddedRTP* is about 7.8 times faster than that of UCL RTP for multimedia streaming services on PDA in spite that its object code size is reduced about by 58% with respect to UCL RTP's.

To improve the result of this paper, the followings should be investigated further. Static packet buffer used in *emdbeddedRTP* can reduce packet-processing time but restricts the extensibility. In order to overcome this shortcoming, extensible packet buffer using overflow buffer has to be implemented. In addition, research on how to determine the size of unit dynamically is needed to obtain optimal performance. Furthermore, we need to investigate a mechanism to reduce the number of memory copies between protocols stack and the applications.

References

1. ITU-T Recommendation H.323: Packet based multimedia communications systems. Feb. (1998).
2. J.Rosenberg, et al. : SIP: Session Initiation Protocol , RFC 3261, Jun. (2002).
3. H.Schulzrinne, et al.: Real Time Streaming Protocol (RTSP)., RFC 2326, Apr. (1998).
4. H.Schulzrinne, et al.: RTP: A Transport Protocol for Real-Time Applications. , RFC 1889, Jan. (1996).
5. RADVISION : RTP/RTCP Toolkit., `http://www.radvision.com/TBU/Products/RTP-RTCP+Toolkit/default.htm`.
6. Lucent Labs : Lucent Technologies Software distribution. , `http://www.bell-labs.com/topic/swdist`.
7. University College London, "UCL Common Multimedia Library", `http://www-mice.cs.ucl.ac.uk/multimedia/software/common/index.html`.
8. VOVIDA.org : Vovida.org., `http: // www.vovida.org/protocols/downloads/rtp` .
9. MPEG4IP : MPEG4IP - Open Streaming Video and Audio., `http://www.mpeg4ip.net`