Bell Labs/Columbia/UMass RTP Library* Internal Function Descriptions

Dan Rubenstein, Jonathan Lennox, Jonathan Rosenberg, and Henning Schulzrinne

Technical Report 99-76
Department of Computer Science
November, 1999

Comments related to this document or to the Bell Labs/Columbia/UMass RTP Library should be sent to rtplib@cs.columbia.edu

Abstract

This documentation describes the internal functions that are components of the Bell Labs/Columbia/UMass RTP Library.[1] The purpose of this document is to provide the experienced networking programmer who is familiar with the details of RTP/RTCP with a detailed understanding of the operation of the internal aspects of the Library. This document should not be necessary for implementing an application that uses the RTP/RTCP protocols. Use of the library for such a purpose is described in [1]. The Bell Labs/Columbia/UMass RTP Library attempts to conform to the protocol description discussed in [2]. The software is provided as-is: neither the authors nor Lucent Technologies make any guarantees as to its correctness. Furthermore, the reader is warned that modifying any part of the library code can result in a protocol that no longer satisfies the requirements of the RTP/RTCP protocol as specified in [2].

[1] In its current form, the document describes the library as it appeared in January, 1998. We point out that there have been significant modifications since then.

Contents

1 Introduction

The Bell Labs/Columbia/UMass RTP Library is provided as a tool to facilitate development of applications that implement RTP/RTCP protocols for delivering real-time data. It also provides a common implementation of the RTP/RTCP protocols. By using the library, various applications' protocols will conform to the same set of requirements. This will make it easier for different applications to share data over a network. Furthermore, the cost of updating an application to a newer version of RTP/RTCP is simplified. The application programmer simply needs to compile in a newer version of the library.

The library itself can be viewed in two levels. The top level is the interface that is provided to the application programmer. Through calls to this interface, the application programmer should be able to easily construct an application whose network transport protocol conforms to the RTP/RTCP specification in [2]. This interface is described in [1]. In this document, we describe how the internals of the interface operate (i.e. all the details that an application programmer shouldn't need to know).

2 Library Internals Overview

The Internals of the Library perform many functions that are required to support RTP/RTCP. This support can be broken down into two basic components: **Network Interface Operations** and **Membership Operations**. Network interface operationsinvolve sending, receiving, processing, and perhaps decoding of RTP/RTCP packets. Membership operationsinvolve keeping track of the members of the current session, and any statistics for these members that are relevant to the operation of RTP/RTCP.

3 rtp_mlist_internal.{h,c}

These files provide structures and functionality that perform generic list operations. They also provide structures and functionality for operations that are specific to RTP/RTCP, but which are for the mostpart strictly membership operations.

3.1 `#defines`

_RTP_DEBUG
: If this variable is `#defined`, then certain interal messages will be printed to `stdout`.

_RTP_WATCH_ALLOCATION
: If this variable is `#defined`, then dynamic memory allocations are tracked, messages to `stdout` will provide notification whenever a memory location is freed by the library that was not allocated by the library. Furthermore, functionality is provided that can be called to examine the status of memory at any time.

_RTP_SEMI_RANDOM
: If this variable is `#defined`, then the random number generator does not use a random seed, so that it generates random numbers in a predictable order, and makes it easier to test for things like SSRC collisions.

_RTP_NUM_SDES_TYPES
: This variable must be `#defined` to the number of SDES types that are supported by the library. The current default is 12.

MAXMALLOCS
: Defined for internal use only. Number of items that can be dynamically allocated during a session. Currently set to 10,000. Its use (and the imposed limit) is only in effect if `_RTP_WATCH_ALLOCATION` is `#defined`.

3.2 structures

rtp_sndr_local

Holds information about observations from a particular member M about a particular sender, S. For instance, it keeps track of the jitter and arrival times and sequence numbers for recently received packets from S as observed M, and communicated locally via RTCP reports. M can also be the local member, such that the information contained in the structure is what is observed of sender S from a local perspective.

member

Holds information about a particular member of the RTP/RTCP session. Sample members for which this structure is used include: the local member, any member that is sending RTP or RTCP packets, or any member that appears in the CSRC list of an RTCP packet.

struct link

An object used within a doubly linked list to connect the members of the list.

doubly_linked_list

A doubly linked list of members that are linked together via `struct link`s.

membership_list

The list of members for the current context. This is simply a doubly linked list with a few extra fields that allow it to group members of different status (`_RTP_MEMBER_CONFIRMED`, `_RTP_MEMBER_PENDING`, and `_RTP_MEMBER_EXPIRED`).

hash_table

Simply an array of several `doubly_linked_list`s, where a function maps each member to a particular list via a hash function.

3.3 global variables

long mallctr

Counts the number of bytes that were dynamically allocated made throughout the entire running of the executable. Only in use if `_RTP_WATCH_ALLOCATION` is `#defined`.

long mallcalls

Counts the number of dynamic memory allocations made throughout the entire running of the executable. Only in use if `_RTP_WATCH_ALLOCATION` is `#defined`.

long freecalls

Counts the number of times memory was freed. Only in use if `_RTP_WATCH_ALLOCATION` is `#defined`.

void* malladdr[MAXMALLOCS]

Stores the currently active memory allocations. When an allocation is no longer active (i.e. it has been `free()`d), its value is set to `NULL`.

3.4 functionality

void InitMallocs()

Initialization function for `malloc()` tracking. Should only be called when `_RTP_WATCH_ALLOCATION` is `#defined`.

void CheckMallocs()

Examines what has been `malloc()`d and `free()`d and reports any unfreed memory. Should be called right before termination of the executable to check for memory leaks. Should only be used when `_RTP_WATCH_ALLOCATION` is `#defined`.

void* my_malloc(size_t size)

This function is called throughout the library code in place of `malloc()`. If `_RTP_WATCH_ALLOCATION` is `#defined`, then the call performs a `malloc()` and also retains information about the memory allocation. Otherwise, it simply calls `malloc()`.

void* my_calloc(size_t num, size_t size)

Similar to `my_malloc()`, except replaces `calloc()`.

void my_free(void* tofree)

Similar to `my_malloc()`, except replaces `free()`.

void InitList(doubly_linked_list *l)

Initializes a list to be empty. Does not allocate memory for the list.

struct link *CreateLink(member *the_member)

Allocates memory for a link which can be used to insert `the_member` into some list. The function itself does not place the link in any lists.

int PullMember(doubly_linked_list *l, struct link *the_link)

Remove link `the_link` from the list `l`.

int UnsortedInsertMember (doubly_linked_list *l, struct link *the_link, struct link *prev_on_list)

Insert the link `the_link` into the list `l` immediately behind the link, `prev_on_list`. If `prev_on_list` is `NULL`, then insert the member into the front of the list.

int SortedInsertMember (_RTP_CONTEXT *the_context, doubly_linked_list *l, struct link *the_link, int (*sort_func)(_RTP_CONTEXT *the_context, struct link *compare_me, struct link *to_me), struct link *start_point)

Insert the link `the_link` into the sorted list, `l`, sorted by the member that each link points to, ordered by the sorting function `sort_func()`. The sorting should begin by comparing `the_link` to the link `to_me`. For the list to be sorted properly, the `SortedInsertMember()` function must always use the same sort function for a particular list. At this point in time, this function is not used within the library.

void MembershipListInit(membership_list *l)

Initialize the membership list. This entails initializing the `doubly_linked_list` structure as well as setting the number of confirmed members to 0, and setting the pointer to the oldest member that has not soft timed out yet to `NULL` (since there are no members at all).

int MembershipListPullMember(membership_list *l, strcut link *the_link)

Remove a member from the membership list, `l`. Involves taking the member out of the list and updating the confirmed member count (if necessary) and the oldest not-yet-soft pointer (if necessary).

void ChangeMemberStatus (_RTP_CONTEXT *the_context, member *m, memberstatus new_status)

Should be called whenever a member's status is changed. It not only updates the member's information appropriately, but also makes the necessary changes within the context `the_context`'s membership list.

void InitSSRCHashTable(_RTP_CONTEXT *the_context)

Initializes the member hash table that hashes on the SSRC.

void DeleteSSRCHashTable(_RTP_CONTEXT *the_context)

Destroys the member hash table that hashes on the SSRC.

int EnterSSRCHashTable(_RTP_CONTEXT *the_context, member *the_member)

Places a member into the hash table that hashes on the member's SSRC.

member *GetMemberFromSSRCHash(_RTP_CONTEXT *the_context, u_int32 ssrc)

Retrieves *a* member with SSRC `ssrc` (there might be more than one if a collision exists). If no such member exists, the function returns `NULL`.

int RemoveMemberFromSSRCHash (_RTP_CONTEXT *the_context, member *the_member)

Removes a particular member from the SSRC hash table. Returns `FALSE` if `the_member` is not currently in the hash table.

void InitUniqueIDHashTable(_RTP_CONTEXT *the_context)

Creates the hash table that hashes on the canonical identifier for each member.

void DeleteUniqueIDHashTable(_RTP_CONTEXT *the_context)

Deletes the hash table that hashes on the canonical identifier for each member.

int EnterUniqueIDHashTable(_RTP_CONTEXT *the_context, member *the_member)

Places the member `the_member` into the unique ID hash table.

member *GetMemberFromUniqueIDHash(_RTP_CONTEXT *the_context, person id)

Gets the member with the unique id `id`. Returns `NULL` if no such member exists.

int RemoveMemberFromUniqueIDHash (_RTP_CONTEXT *the_context, member *the_member)

Remove the member `the_member` from the hash table. Returns `FALSE` if no such member is present in the table.

void InitCNAMEHashTable(_RTP_CONTEXT *the_context)

Creates the hash table that hashes on the canonical name of a member.

void DeleteCNAMEHashTable(_RTP_CONTEXT *the_context)

Deletes the hash table that hashes on the canonical name of a member.

int HashOnName(char *the_name)

Converts a NULL-terminated string `the_name` into an integer that can then be fed to a hash function to produce a hash value.

int EnterCNAMEHashTable(_RTP_CONTEXT *the_context, member *the_member)

Places the member `the_member` into the CNAME hash table.

member *GetMemberFromCNAMEHash(_RTP_CONTEXT *the_context, char *cname)

Gets the member with the CNAME `cname`. Returns `NULL` if no such member exists.

int RemoveMemberFromCNAMEHash (_RTP_CONTEXT *the_context, member *the_member)

Remove the member `the_member` from the hash table. Returns `FALSE` if no such member is present in the table.

void Init_RR_Hash(member *the_member)

Creates the hash table used by the member `the_member` that hashes on a receiver report for a sender.

void Delete_RR_Hash(member *the_member)

Deletes the hash table used by the member `the_member` that hashes on a receiver report for a sender.

receiver_report *Update_RR (member *reporter, member *sender, rtcp_report_block *the_block)

Updates the report to `the_block` for the sender `sender` that was issued by the member `reporter`.

void Clear_Member_RRs(_RTP_CONTEXT *the_context, member *the_member)

Removes hash enttries for the member `the_member`.

static void Merge_RRs(receiver_report *prev_sr, receiver_report *new_sr)

Merges two receiver reports and stores the results in `prev_sr`.

void Merge_RR_Hashes(member* prev_member, member *new_member)

Merges the hash table from member `new_member` into the hash table for `prev_member`.

void panic(char *format, ...)

Called by **tclHash.c** whenever something goes wrong. Hopefully, it never gets called.

4 rtp_mlist.{h,c}

These files provide network interface operationsthat affect the membership lists.

4.1 functions

4.1.1 Reaction to Events

Library member list operations are performed whenever an event occurs. An event is defined as a (RTP or RTCP) packet, or a timeout (i.e. when the application makes a call to `RTPExecute()` (see [1]).

rtperror UpdateMemberInfoByRTCP (_RTP_CONTEXT *the_context, rtcp_packet *the_packet,
struct sockaddr *fromaddr,
int addrlen,
int part_in_compound_pkt)

This function is called for the member from which an RTCP packet was just received. The member is moved to status `RTP_MEMBER_CONFIRMED` if two RT(C)P packets have been received from it. All fields which are affected by information in RTCP compound packets are updated. SSRC collisions are detected and handled via calls to `HandleSSRCCollision()` if two members with the same SSRC but different CNAMEs are detected. Members are merged if a member obtains a CNAME and turns out to be identical to a previous member that was previously involved in an SSRC collision. If the RTCP packet includes a BYE packet, those members specified in the packet are removed. An APP packet updates member info. At the end of the function, a call to `UpdateMembershipLists()` is made to update the appropriate lists.

rtperror UpdateMemberInfoByRTP (_RTP_CONTEXT *the_context, rtp_packet *the_packet,
struct sockaddr *fromaddr,
int addrlen)

Called upon receipt (or sending) of an RTP packet. Updates the info of the sending member of the RTP packet. The status of the member is changed to `RTP_MEMBER_CONFIRMED` if two RTP packets have been received. Jitter and sequence number fields are updated as well.

void UpdateMembershipLists(_RTP_CONTEXT *the_context)

Called after an RTCP packet is received (i.e. called by `UpdateMemberInfoByRTCP()`) as well as when an RTCP packet is sent (see `SendRTCPPacket()` in [1]. Updates the status of all members on the membership list, calls the necessary callbacks when membership status changes, and purges those members from the list that no longer belong on it. It also updates the sender status appropriately for members.

4.1.2 Member Initialization / destruction functions

Functions that correspond to initializing members are called whenever the library believes that a new member has joined the session (i.e. a packet arrives from a previously unknown SSRC, or a collision of SSRCs has been detected). Functions that destroy members are called when a member hard times out, or when a collision is resolved (i.e. two members with different SSRCs in fact refer to the pre- and post-collission members).

member *EstablishNewMember (_RTP_CONTEXT *the_context, u_int32 ssrc,
void* user_data, struct sockaddr *fromaddr)

Constructs a new member with the specified SSRC. The member is given a unique ID (the CNAME is not yet known so it is assumed to be a new member). The initial status of the member is `RTP_MEMBER_PENDING`.

Calling this function also triggers callbacks `UpdateMemberCallBack()` and `ChangedMemberInfo-CallBack()` if they have been set to indicate a new member and announce the IP address of the member, respectively.

static person AssignID(_RTP_CONTEXT *the_context)
Allocates a unique ID each time the function is called. The returned number is simply incremented each time.

int RemoveMember(_RTP_CONTEXT *the_context, member *remove_me)
Removes a member from all lists and hash tables.

int DestroyMember(_RTP_CONTEXT *the_context, member *destroy_me)
Deallocates memory associated with a member. Removes it from any collisions that it is involved in as well.

4.1.3 Functions to update a member's status

The following functions update the status of a member, and the appropriate function is called in reaction to an event triggered by the member that is being updated.

int UpdateMemberTime (_RTP_CONTEXT *the_context, member *the_member, struct timeval newtime)
Updates the field that tracks the last time that the member received an RTP/RTCP packet. The function gets called whenever an RTP or RTCP packet is received from the member `the_member`.

int UpdateSenderTime(_RTP_CONTEXT *the_context, member *the_member, struct timeval newtime)
Updates the field that tracks the last time the member sent an RTP packet. If this member is a new sender, it initiates the RR Hash table and calls the `UpdateMemberCallBack()`. The function is called whenever an RTP packet is received from `the_member`.

static member* UpdateTimeOrCreateMember (_RTP_CONTEXT *the_context, u_int32 ssrc, struct sockaddr *fromaddr, int addrlen)
This function is called internally by functions that wish to receive a `member` structure that contains the given SSRC with the given address. If no such member exists, it is created. If there are multiple members with this SSRC (i.e. all members with the SSRC are involved in a collision), it attempts to retrieve the member with the same SSRC. If it finds a member whose address isn't known, it sets that member's address to the address specified (See the comments above the function for details) and calls the `ChangedMemberInfoCallBack()` function to notify about changes in address and port. Finally, it updates the time associated with the member to the current time.

4.1.4 Queries of a member's status

int SenderTimedOut (_RTP_CONTEXT *the_context, member *themember, struct timeval now)
Returns `TRUE` if the member has timed out as a sender.

int SoftTimedOut (_RTP_CONTEXT *the_context, member *themember, struct timeval now)
Returns `TRUE` if the member has soft timed out.

int HardTimedOut (_RTP_CONTEXT *the_context, member *themember, struct timeval now)
Returns `TRUE` if the member has hard timed out.

int FromDifferentSource (struct sockaddr *addr1, struct sockaddr *addr2, int complen)
Returns `TRUE` if the addresses do not match. If the address is of type `AF_INET`, then either the addresses must not match, or the ports must differ by a value greater than 1. If of another address type, then the addresses are considered different if the first `complen` bytes don't match exactly.

5 rtp_api_internal.{h,c}

These files provide the basic network interface operations, as well as the definition of the `_RTP_CONTEXT` structure, in which all information about the session is stored. What is in these files is meant to support what appears in `rtp_api.h` and `rtp_api.c`, but which can be kept hidden from the application programmer.

5.1 #defines

_RTP_INIT_CONTEXTS_AVAIL
How many contexts can be constructed before the `ContextList` structure must be expanded in size. The current value is 256.

_RTP_VERSION
The version of the RTP protocol. The current value is 2.

_RTP_CONTEXT_INC
The increment by which the `ContextList` is increased if an increase is necessary. The current value is 10.

_RTP_DEFAULT_TTL
The default TTL (time to live field) for a mulicast channel. The default value is 128. The TTL value cannot be set for unicast communication.

_RTP_DEFAULT_ENCRYPTION
The default form of encryption used. The default is currently `RTP_ENCRYPTION_NONE`.

_RTP_MAX_BYE_SIZE
Maximum storage allocated for the BYE reason. The current value is 1024.

_RTP_MAX_PKT_SIZE
The maximum size allowed for an RTP packet. The default is 10,000.

_RTP_MAX_PORT_STR_SIZE
The maximum length of a UDP port number when represented as a string. The default is 8, which is an overestimate.

_RTP_MAX_PKTS_IN_COMPOUND
The maximum number of RTCP packets that can appear in a compound packet. The default value is 200, which is an overestimate. The value is used to create a structure that maintains pointers to the start of each packet, which does not require much memory.

_RTP_MAX_PAYLOAD_TYPES
Number of payload types available. The default is 128.

_RTP_DEFAULT_BANDWIDTH
The default bandwidth that an RTP session is believed to use, given in Kb / sec. The default is 120.

_RTP_DEFAULT_RTCP_FRAC
The fraction of bandwidth that should be used by RTCP. The default is .05 (5%).

_RTP_DEFAULT_SENDER_BW_FRAC
The fraction of the RTCP bandwidth that is to be shared among active senders. The default is .25 (25%).

_RTP_MAX_UNIQUE_ID_REMAPS
The maximum number of ID remaps that are tracked within a context. An ID is remapped when a collision is resolved and it is realized that two member structures have been built for a single member (each with a unique ID). The member with the larger ID is merged into the other member, and is subsequently destroyed. Any further references to the larger ID will map to the smaller ID, as long as the information is maintained. If the number of remaps grows larger than `_RTP_MAX_UNIQUE_ID_REMAPS`, then earlier remaps are dropped from memory, and remapping for certain members will fail to take place. The default value is 2,000, which should be more than the number of remappings that take place in a session.

_RTP_HARD_MEMBER_TIMEOUT
The time until a hard timeout is a fixed multiple of the time it takes to soft timeout. `_RTP_HARD_MEMBER_TIMEOUT` equals this multiple. The default value is 3.

_RTP_SOFT_MEMBER_TIMEOUT
The time until a member has a "soft" timeout, or the time that a member who has not been "validated" (i.e. does not yet have a CNAME or is currently involved in a collision) will be terminated. The value in `_RTP_SOFT_MEMBER_TIMEOUT` should be multiplied by the time of the current RTCP receiver interval to determine the soft timeout time. The current defaul it 5.

_RTP_SENDER_TIMEOUT
The number of RTCP packets that appear in a row from a particular member (i.e. no RTP packets arrive during this interval) that cause the member to revert to non-sender status. The current default is 1.

_RTP_SENDER_MAX_TIMEOUT
The maximum time until a sender times out (in seconds). The current value is 3,600 (1 hour).

_RTCP_MIN_TIME
The minimum time (period in seconds) in which a member can send RTCP packets. The current default is 5.

_RTCP_SIZE_GAIN
RTCP packet size is computed using an exponentially decaying average. `_RTCP_SIZE_GAIN` is the fraction by which the most recent RTCP packet influences the average. The default value is 1/16.

_RTP_ADDRESS_NOT_YET_KNOWN
The default value for an address of an SSRC before the address can be determined. The current value is 30.

_RTP_DEFAULT_PORT
A default port number. The current value is 5,000.

_BIND_COUNTER
The maximum number of tries to allocate a dynamic RTP port. The current value is 20.

_UDP_PORT_BASE
Starting UDP port for dynamic ports. The current value is 49,152.

_UDP_PORT_RANGE
The range of UDP dynamic ports. The current value is 16382

_GETTIMEOFDAY_TO_NTP_OFFSET
The number of seconds between 1/1/1900 and 1/1/1970. The value is 2,208,988,800

RTP_OPAQUE_SEND_RTCP
The only opaque type currently used by the `RTPExecute()`. Its value is 1.

5.2 structures

struct little_endian_rtp_hdr
A header equivalent to the RTP header, except that its bit-fields are reversed so that they map to a big endian ordering on a little endian machine.

struct little_endian_rtcp_hdr
A header equivalent to the RTCP header, except that its bit-fields are reversed so that they map to a big endian ordering on a little endian machine.

address_holder_t
This is used to hold addresses to send to for RTP and RTCP packets. It is a linked list of addresses, ports, and TTLs.

_RTP_CONTEXT

This structure holds information that pertains to a context. It holds or points to all information relevant to the current session.

5.3 global variables

_RTP_CONTEXT **ContextList

An array of pointers to contexts. The contexts are constructed when they are needed (via a call to `RTPCreate()`). The context's `cid` is the index of in this array of the context.

long _RTP_context_above_used

Points to the smallest index that pertains is larger than the `cid`s of any active context.

long _RTP_contexts_in_use

Counts the number of contexts in use.

rtperror _RTP_cur_err

Keeps track of the most recent error that occured during a session.

char _RTP_err_msg[200]

Keeps track of the error message associated with the most recent error that occured during a session.

int _RTP_PAYLOAD_CLOCK_CONVERSIONS[_RTP_MAX_PAYLOAD_TYPES]

Holds clock conversion rates (NTP ticks in terms of an RTP tick) for various payload types.

5.4 functions

void InitRandom()

Initializes random number generators with a random seed. If `_RTP_SEMI_RANDOM` is `#defined`, then no initialization is performed. Otherwise, the `drand48()` generator is initialized via the time of day (μsec).

u_int32 random32(int type)

Returns a 32 bit random number. If `_RTP_SEMI_RANDOM` is `#defined`, it simply calls `rand()`. Otherwise, it calls `md_32()`,

static u_long md_32(char *string, int length)

The random number generator code presented in [2].

void SetDefaultPayloadRates(_RTP_CONTEXT *the_context)

Set the default payload rates. This currently sets the rates as is specified in **draft-ietf-avt-profile-new-01** from May 15, 1997.

rtperror ValidRTPContext(context cid, char *calling_func)

Validate that a putative RTP context actually exists. If not, set error message and return appropriately.

rtperror GetMemberForContext(context cid, person p, member **the_member_p, char *calling_func)

Retrieves the member for a putative person with a given context, or complains.

int IsMulticast(struct in_addr addr)

Determines if an IPv4 address is a multicast address.

struct timeval AddTimes(struct timeval *time1, struct timeval *time2)

Adds two timevals. Does not handle the overflow of `tv_sec` (year 2038 problem).

int TimeExpired(struct timeval *init_time, struct timeval *cur_time, struct timeval *interval)

Returns `TRUE` if `init_time` + `interval` $<=$ `cur_time`, i.e. if an expiration period has elapsed.

struct timeval ConvertDoubleToTime(double interval)

Convert a `double` value to a `struct timeval` structure. The value `interval` is in terms of seconds.

static ntp64 ConvertTimevalToNTP(struct timeval tv)

Converts a `struct timeval` structure to an NTP timestamp.

double RTPTimeDiff (_RTP_CONTEXT *the_context,
struct timeval *later_time,
struct timeval *earlier_time,
int8 payload_type)

Returns the time difference on an RTP scale between two NTP timestamps. The result is in milliseconds, and should be an integer.

static double InternalComputeRTCPSendDelay (_RTP_CONTEXT *the_context,
int sender)

Returns the current delay interval (in seconds) between RTCP packets for the session described by `the_context`. The `sender` boolean variable determines whether or not the results should be returned for a sender, whose rates are different from a non-sender.

double ComputeRTCPSendDelay(_RTP_CONTEXT *the_context)

Compute the local member's RTCP interval, dependent on whether or not the member is a sender.

double ComputeSenderRTCPSendDelay(_RTP_CONTEXT *the_context)

Compute the RTCP interval for any sender.

double ComputeReceiverRTCPSendDelay(_RTP_CONTEXT *the_context)

Compute the RTCP interval for any non-sender.

void ComputeBlockInfo (_RTP_CONTEXT *the_context, member *the_member,
rtcp_report_block *cur_block, u_int32 *expected_ptr)

Compute the values for SR or RR packets for `the_member`, based on statistics that have been observed during the session. This function is called by `BuildBlockInfo()` as well as the various API calls that enquire about member statistics. Those fields which are computed here are:

- cumulative number of packets lost that were sent by `the_member`.
- fraction of packets lost that were sent by `the_member`.
- highest sequence number received
- jitter
- lsr
- dlsr

static void BuildBlockInfo (_RTP_CONTEXT *the_context, u_int32 ssrc,
rtcp_report_block *cur_block)

Build a report block which appears in an SR or RR packet for the member with SSRC `ssrc`, based on statistics that are collected from local observation. If the member with SSRC `ssrc` does not exist, it is created. If a previous report block does not exist, it is created. Otherwise, the information is computed via a call to `ComputeBlockInfo()`. Also, update those fields that track the values since the last sent RTCP packet (since this is part of the process to build and send such a packet). Finally, the local information about this member is updated as if an RR or SR packet had arrived.

static int AddPad(char* cur, int cur_tot_len)

Adds padding to the end of a stream with total length `cur_tot_len` such that the length of the stream with the padding is $0 mod 8$. The ending of the stream with length `cur_tot_len` is at `cur` (before the padding is added). The padding is all 0's, except for the last byte, which is set to a value that indicates the number of bytes (including itself) that make up the padding.

char *Build_SR_Packet (_RTP_CONTEXT *the_context,
char *buffer, struct link **first_sender_link)

Builds an SR packet, or builds an additional SR packet. The packet is built at memory location `buffer`. If `first_sender_link` is set to `NULL`, then the initial SR packet is built. If the function returns with `first_sender_link` set to a non-`NULL` value, then an additional SR packet is required to describe all senders in the current session (SR packets can describe a limit of 31). The function returns a pointer to the end of the buffer of the current SR packet that was built, so that a function to build an additional RTCP packet within the compound packet can be given the starting location of the next packet.

char *Build_RR_Packet (_RTP_CONTEXT *the_context,
char *buffer, struct link **first_sender_link,
int build_empty)

Builds an RR packet, or builds an additional RR packet. The packet is built at memory location `buffer`. If `first_sender_link` is set to `NULL`, then the initial RR packet is built. If the function returns with `first_sender_link` set to a non-`NULL` value, then an additional RR packet is required to describe all senders in the current session (RR packets can describe a limit of 31). The function returns a pointer to the end of the buffer of the current RR packet that was built, so that a function to build an additional RTCP packet within the compound packet can be given the starting location of the next packet.

int PlaceSDESInfoForMember (member *the_member, char *buffer,
int init_buffer_offset)

Builds an SDES packet at location `buffer` for the member `the_member`. `buffer` points to the start of the SDES packet, and `init_buffer_offset` gives the offset from the point of `buffer` where the current info should be placed. If all of this member's SDES fields are `NULL`, then no information is placed into the buffer, and the value returned equals `init_buffer_offset`.

char *Build_SDES_Packet (_RTP_CONTEXT *the_context, char *buffer,
int lastpkt_and_encrypt)

Builds an SDES packet which contains information for the local member and each sender that has at least one non-`NULL` SDES field. The parameter `lastpkt_and_encrypt` should be set to `TRUE` if the SDES packet is the last packet in the compound RTCP packet and the packet is being encrypted. This way, padding will be added.

char *BuildByePacket (_RTP_CONTEXT *the_context, char *buffer,
int bye_for_csrcs, int lastpkt_and_encrypt, char *reason)

Builds a BYE packet. If `bye_for_csrcs` is `TRUE`, then the csrcs in the csrc list will be included the BYE. The parameter `lastpkt_and_encrypt` should be set to `TRUE` if the BYE packet is the last packet in the compound RTCP packet and the packet is being encrypted. This way, padding will be added. `reason` points to the buffer that contains the reason for leaving the session (should be terminated with a $\backslash 0$, or set to `NULL` if no reason is desired.)

int SendRTCPPacket(_RTP_CONTEXT *the_context, int special)

Builds and sends an RTCP packet, but only after performing reconsideration (if it is enabled) which confirms whether or not it is currently the time to send. Returns `TRUE` if the packet was sent, `FALSE` otherwise. The time for the next RTCP packet to be sent is also scheduled here (or the time for the current packet to be sent is rescheduled if reconsideration caused the packets sending to be delayed). The parameter `special` is used to perform partial encryption. If partial encryption is enabled in the context, the setting `special` = 0 sends the encrypted SDES packet, and setting `special` = 1 sends the non-encrypted portion of the compound RTCP packet. Thus, the function must be called twice when partial encryption is in use. When partial encryption is not in use, `special` should be set to 0.

rtperror RemapPerson(_RTP_CONTEXT *the_context, person *p)

Given a unique ID, `p`, returns the canonical identifier for that member. (Person IDs can be remapped after a collision resolution where two member structures that describe the same member (but have different canonical IDs) are mapped into a single structure).

int SplitAndHostOrderLengthCompoundRTCP (char *rtcppacket, char *indpkts[], int len)

This function is identical in all respects save one as the `RTPSplitCompoundRTCP()` function that is provided to the application programmer, and is described in [1]. The only difference is that the length field in each RTCP packet within the compound packet is converted into host byte order. This conversion should only be performed once per arriving compound packet, and is therefore called internally by the library upon packet arrival. The function is identical to `RTPSplitCompoundRTCP()` on Big Endian machines.

void FixRTPByteOrdering(char *the_packet, int pktlen, int is_nw_to_host)

Converts a network-byte-ordered RTP packet into a host-byte-ordered RTP packet when `is_nw_to_host` is set to `TRUE`. Otherwise, converts the RTP packet in the opposite direction.

void FixRTCPByteOrdering(rtcp_packet *the_packet, int is_nw_to_host)

Converts a network-byte-ordered RTCP packet into a host-byte-ordered RTCP packet when `is_nw_to_host` is set to `TRUE`. Otherwise, converts the RTCP packet in the opposite direction.

void Flip24(char *the_24bit_thing)

Flips a 24-bit value from host to network byte order, or vice versa. This is used for the `cum_packets_lost` field in the RTCP report block, which is a 24-bit quantity.

void ReverseRTPHeader(rtp_hdr_t *the_hdr)

Takes an RTP header in Big Endian format and converts it to Little Endian format.

void ReverseRTCPHeader(rtcp_overlay *the_hdr)

Takes an RTCP header in Big Endian format and converts it to Little Endian format.

void StraightenRTPHeader(rtp_hdr_t *the_hdr)

Takes an RTP header in Little Endian format and converts it to Big Endian format.

void StraightenRTCPHeader(rtcp_overlay *the_hdr)

Takes an RTCP header in Little Endian format and converts it to Big Endian format.

6 rtp_collision.{h,c}

These files provide support that is specific to detecting and resolving collisions between member identifiers (SSRCs) within a session.

6.1 functions

void ComputeNewSSRC(_RTP_CONTEXT *the_context)

This function is called by the local member to choose a new SSRC. The function attempts to retrieve a member with an identical SSRC, and if one is located, the process is repeated. This guarantees that the new SSRC is unique at the time of its creation. The local member's state is then updated to reflect that it is no longer colliding with any other members.

void HandleSSRCCollision (_RTP_CONTEXT *the_context, member *the_member, struct sockaddr *new_addr, char *new_cname)

This function is called after an SSRC collision has been detected. It constructs a new member (whose entry to the session has induced the collision), updates fields in all colliding members that indicate involvement in a collision. If the collision involves the local member, then the collision callback is called, and a BYE packet is sent.

static void MergeLocalInfo(rtp_sndr_local *prev_local, rtp_sndr_local *new_local)

This function takes two type `rtp_sndr_local` parameters, and merges their fields into the `prev_local` parameter. Fields that count events are added together, and fields that represent maximums and minimums take the max / min respectively of the respective field between the two parameters. This function is only called by `MergeMembers()` and has no scope outside the file **rtp_collision.c**.

member *MergeMembers (_RTP_CONTEXT *the_context, member *prev_member, member *new_member)

This function takes two separate copies of a member (due to an SSRC change after which certain events occurred with the new SSRC before it could be determined that this new SSRC resulted from a change) and merges them into a single member, and returning the merged member in `prev_member`.

7 rtp_encrypt.{h,c}

These files provide support for encryption of RTP/RTCP packets. Note that the encryption techniques aren't supplied themselves, but allows an application programmer to interface an encryption package with the library.

7.1 functions

rtperror DoEncryption (context cid, struct iovec *pktpart, int pktlen, int IsRTP)

This function takes an IO Vector that contains a packet and performs the necessary operations that result in sending an encrypted packet into the network. The function copies the IO Vector into a buffer [2], prepends the random 8 byte header, calls the encryption routines supplied by the application programmer, and sends the packet over the appropriate socket.

rtperror DoDecryption (context cid, char *decrypt_buff, long decrypt_bufflen)

This function takes an encrypted stream and applies the decryption function supplied by the application programmer, and strips off the encryption header.

int IsValidRTPPacket(_RTP_CONTEXT *the_context, rtp_packet *the_pkt)

Performs some checks that can often detect non-RTP or improperly decrypted RTP packets. The function currently examines the version byte in the packet and makes sure it matches `_RTP_VERSION`. It also makes sure that the packet isn't an RTCP packet. Currently, it skips payload checks and valid sequence number checks.

int IsValidRTCPPacket (_RTP_CONTEXT *the_context, char **pktpos, int num_parts, long totlen)

Performs some checks that can often detect non-RTCP or improperly decrypted RTCP packets. The function currently examines a compound RTCP packet, and looks at the version byte in the packet and makes sure it matches `_RTP_VERSION`. It also makes sure that the first packet, which should be an SR or RR packet, doesn't have its padding bit set. Finally, it ensures that each packet's length in the compound packet is set correctly.. Currently, it skips payload checks.

8 Other Files

Several files that are used by the library are not discussed in this document. They are either discussed elsewhere, or standard components of other software packages and are not discussed here. Specifically, the files are:

tclHash.{h,c}: These files were extracted from Tcl 8.0p2 to use Tcl's hash table functions, and are stripped down versions of **tcl.{h,c}** and **tclInt.{h,c}**.

config.h: Generated automatically by configure (part of make), setting `#define`s that are machine specific to the compiling platform.

global.h: A small set of `#define`s that are used if the compiler supports function argument prototyping.

md5.{h,c}: Part of the RSA Data Security package. These functions are used by the library to perform random number generation.

[2] This was under the assumption that the encryption package would not support encryption over `iovec` structures. Future revisions will assume that the encryption library can support encryption over such structures.

hpt.c: Provides a single function, `hpt(char *h, stuct sockaddr *sa, unsisgned char *ttl)`, which parses [host]/port[/ttl].

rtp_api.{h,c}: Provides an interface to the application programmer. Details of the code in these files are discussed in [1].

References

[1] On-line documentation of the Bell Labs/Columbia/UMass RTP Library, available at `http://www.cs.columbia.edu/~jdrosen/rtp_api.html`.

[2] H. Schulzrinne, S. Casner, R. Frederick, and V. Jacobson, *RTP: A Transport Protocol for Real-Time application*, Internet Draft draft-ietf-avt-rtp-new-00.ps, December 1997.