librudp Overview

PlayStation®Vita Programmer Tool Runtime Library Release 3.50

Document serial number: 000004892117

Table of Contents

1 Library Overview

Characteristics

librudp is a library that supports reliable data transmission (RUDP) on the UDP (User Datagram Protocol; RFC 768). Compared to TCP over UDPP2P, which is the protocol supported by libnet, RUDP provides more of the features required in online games and attains better transmission efficiency through its use of smaller packet headers.

librudp is a library located in user space. It is designed mainly for use with the UDPP2P protocol, but it can also be used for other transmission methods, such as an original NAT traversal system on UDP, or communication with the game server by using UDP.

librudp also has the following characteristics.

- Delivery Critical (DC) and Order Critical (OC) options are provided for specifying the quality level of data transmission.
- Urgent messages can be given priority with the Latency Critical (LC) flag.
- The overhead due to headers added in the base layer (UDP, IP, etc.) can be significantly reduced because short packets with the same destination in a session are automatically sent together.
- Multiple channels (contexts) can be used on a single UDPP2P socket.
- Congestion control equivalent to that provided in TCP is available. Furthermore, there are features equivalent to Fast Retransmission and Selective ACK, allowing stable transmission efficiency even in a relatively loss-prone network.
- Statistics provide a variety of in-depth information.
- The APIs are flexible and support various programming styles.

Embedding into a Program

Include rudp.h in the source program.

Load also the PRX module in the program as follows.

```
if ( sceSysmoduleLoadModule(SCE_SYSMODULE_RUDP) != SCE_OK ) {
    // Error handling
}
```

Upon building the program, link libSceRudp_stub.a or libSceRudp_stub_weak.a.

Sample Programs

The following sample programs are available for librudp.

sample_code/network/api_librudp/simple/callback/callback.c

This sample program shows how to use the librudp context event handler. Two RUDP contexts (sender and receiver contexts) are created, and 100 messages of 4 bytes each are sent from the sender to the receiver. The internal network I/O thread is not used, so `sceRudpProcessEvents()` is called periodically. This is a typical example of single thread programming.

sample_code/network/api_librudp/simple/polling/polling.c

This sample program shows how to use the librudp polling feature. Like the callback.c sample program, two RUDP contexts (sender and receiver contexts) are created, and 100 messages of 4 bytes each are sent from the sender to the receiver. Since the internal network I/O thread is used, `sceRudpProcessEvents()` is not called. Also, since the polling feature is used, the context event handler is not used.

sample_code/network/api_librudp/simple/blocking/blocking.c

This sample program shows how to use the librudp blocking mode. Like the callback.c and polling.c programs, two RUDP contexts (sender and receiver contexts) are created, and 100 messages of 4 bytes each are sent from the sender to the receiver. Since the internal network I/O thread is used (like polling.c), `sceRudpProcessEvents()` is not used. An application thread is created per context for processing transmissions in parallel with the blocking functions. The context event handler is not used.

2 Using the Library

Basic Procedure

(1) Initialize librudp

To use librudp, first load the PRX module by calling `sceSysmoduleLoadModule()` with `SCE_SYSMODULE_RUDP` specified as the argument. Then call `sceRudpInit()` to initialize the library. Allocate the memory area to be used internally by librudp in advance by the application and specify the pointer and size of this area as arguments of `sceRudpInit()`. `sceRudpInit()` will return `SCE_RUDP_SUCCESS` (0) when initialization succeeds. Since the necessary memory varies significantly depending on usage, it is necessary to find out memory used space at the stage of development. Check the maximum value of allocated memory size (*`memPeak`*) using `sceRudpGetStatus()`.

Then call `sceRudpSetEventHandler()` to register a callback function (common event handler) for receiving librudp event notifications.

As an additional option, it is possible to call `sceRudpEnableInternalIOThread()` and create a thread inside librudp that handles librudp internal events and network I/O (libnet) events. This thread is called the internal network I/O thread. If this internal network I/O thread is created, it will not be necessary for the application to call `sceRudpProcessEvents()` in Step (5).

(2) Create an RUDP Context

Call `sceRudpCreateContext()` to create an RUDP context and register a callback function (context event handler) for receiving notification of events regarding the context. The RUDP context is an `int`-type ID that is the equivalent of a socket (file descriptor) in TCP/IP. It is used for specifying the particular connection on which to execute various operations.

When initialization with `sceRudpCreateContext()` succeeds, the context ID will be stored to the variable pointed to by the argument *`ctxId`*, and return `SCE_RUDP_SUCCESS` (0).

Note
It is not necessary to register a context callback function if the librudp polling function `sceRudpPollWait()` will be used (this function is equivalent to `socketselect()` in libnet), or if communications will be processed in blocking mode (in other words, if `sceRudpInitiate()`, `sceRudpActivate()`, `sceRudpRead()`, `sceRudpWrite()`, and `sceRudpFlush()` will be called as blocking functions). For details, refer to the sections "Polling" and "Blocking Mode", both of which can be found later in this document.

(3) Bind the Context

Use a libnet API to create a UDPP2P socket. Then call `sceRudpBind()` to associate the context ID created in the previous step with the UDPP2P socket, librudp virtual port, and a multiplexing mode. `sceRudpBind()` will return `SCE_RUDP_SUCCESS` (0) when this bind operation is successful.

Information of multiplexing modes can be found in the section "Multiplexing Mode".

(4) Start a Connection

Call `sceRudpInitiate()`, specifying the context ID created above and the socket address of a connection peer. This is the active-open method, in which librudp issues a connection request to a peer.

It is also possible to wait for a connection request from a peer whose socket address is known by calling `sceRudpActivate()` instead of `sceRudpInitiate()`. This is the passive-open method. There is also the simultaneous-open method, in which `sceRudpInitiate()` is issued simultaneously by both parties.

(5) Handle Communication Events

After starting a connection, call `sceRudpProcessEvents()` periodically so that the common event handler and context event handler are called as necessary to handle internal events and network communication, including the connection with the peer. The recommended interval is 16msec (60Hz).

If an internal network I/O thread was created in Step (1) by calling `sceRudpEnableInternalIOThread()`, this thread will be responsible for handling internal events and network I/O, and so it will not be necessary for the application to call `sceRudpProcessEvents()`.

(6) Wait for Connection to Complete

When a connection is successfully established, the event `SCE_RUDP_CONTEXT_EVENT_ESTABLISHED` will be notified for the context ID.

If the connection fails, the event `SCE_RUDP_CONTEXT_EVENT_CLOSED` will be notified, along with an error code. The application must handle this error.

(7) Transmit Data

Data transmissions are executed with `sceRudpWrite()` and `sceRudpRead()`. By calling `sceRudpGetSizeWritable()` before sending data, it is possible to check whether there is space in the send buffer inside librudp. Similarly, `sceRudpGetSizeReadable()` can be used to check whether readable data exists in the librudp receive buffer. `sceRudpGetNumberOfPacketsToRead()` can be used to check the number of readable datagrams in the librudp receive buffer before getting their size.

Details of transmission procedures can be found in the section "Network I/O".

(8) End Communications

After all data transmissions are finished, call `sceRudpTerminate()` with the context ID specified. This invalidates the context ID, and librudp will start terminating the connection.

If the connection is closed by the peer, the event `SCE_RUDP_CONTEXT_EVENT_CLOSED` will be notified for the context ID. When this happens, call `sceRudpTerminate()` to destroy the context ID.

(9) Terminate librudp

When librudp is no longer necessary, call `sceRudpEnd()` to carry out termination processing and free all the resources that were allocated inside the library. The internal network I/O thread, if used, will also be terminated at this time. The memory area provided to librudp by `sceRudpInit()` can safely be collected back after this.

Finally, unload the PRX module by calling `sceSysmoduleUnloadModule()` with `SCE_SYSMODULE_RUDP` specified as the argument.

List of Functions

Initialization, Configuration, Termination

Function	Description
sceRudpInit()	Initializes librudp
sceRudpEnd()	Terminates librudp
sceRudpEnableInternalIOThread()	Starts up the internal network I/O thread
sceRudpSetEventHandler()	Registers a common event handler
sceRudpSetMaxSegmentSize()	Sets the maximum segment size (MSS)
sceRudpGetMaxSegmentSize()	Gets the maximum segment size (MSS)

Context and Option Configuration

Function	Description
sceRudpCreateContext()	Creates a context
sceRudpSetOption()	Sets context options
sceRudpGetOption()	Gets context options

States and Statistics

Function	Description
sceRudpGetContextStatus()	Gets the status and statistics of context
sceRudpGetStatus()	Gets status and statistics of librudp
sceRudpGetLocalInfo()	Gets local information of context
sceRudpGetRemoteInfo()	Gets remote (peer) information of context

Connection

Function	Description
sceRudpBind()	Binds a context
sceRudpInitiate()	Starts connection to peer
sceRudpActivate()	Starts waiting for connection from peer
sceRudpTerminate()	Ends connection and destroys context ID

Data Transmission

Function	Description
sceRudpRead()	Reads receive data
sceRudpWrite()	Writes send data
sceRudpGetSizeReadable()	Gets size of readable data
sceRudpGetNumberOfPacketsToRead()	Gets the number of packets with readable data
sceRudpGetSizeWritable()	Gets size of writable data
sceRudpFlush()	Checks if sent data arrived at peer's

Polling

Function	Description
sceRudpPollCreate()	Creates a polling ID
sceRudpPollDestroy()	Destroys a polling ID
sceRudpPollControl()	Modifies polling ID settings
sceRudpPollWait()	Waits for events by polling
sceRudpPollCancel()	Cancels waiting for events

Event Handling and Network I/O

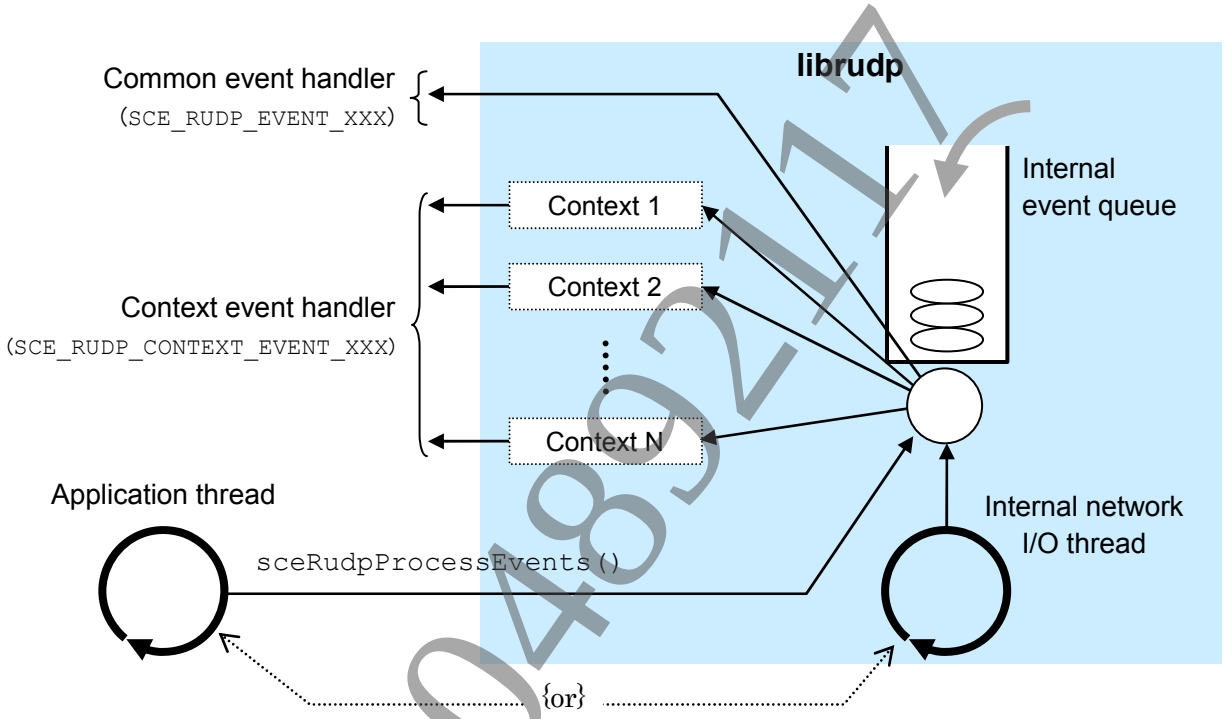
Function	Description
sceRudpNetReceived()	Inputs received UDP data
sceRudpProcessEvents()	Processes library events

3 Library Operation

Processing of librudp Internal Events

The internal events of librudp are first stored to the internal event queue, and then processed when the application calls `sceRudpProcessEvents()`. In other words, all the callbacks notified by librudp are called on the application thread on which `sceRudpProcessEvents()` was called.

Figure 1 Processing of Internal Events



If an internal network I/O thread was created with `sceRudpEnableInternalIOThread()`, all events are handled by the internal thread (shown on the righthand side in Figure 1). Therefore, all the callbacks notified by librudp are called on this internal thread. Remember this point if user variables need to be accessed in the callback.

Context State Transitions

In librudp, the "context" is the unit for managing connections. A context is the equivalent of a socket (file descriptor) in TCP/IP. Each context is given an `int`-type ID, and this ID is used when notifying and obtaining the context state and for performing other operations.

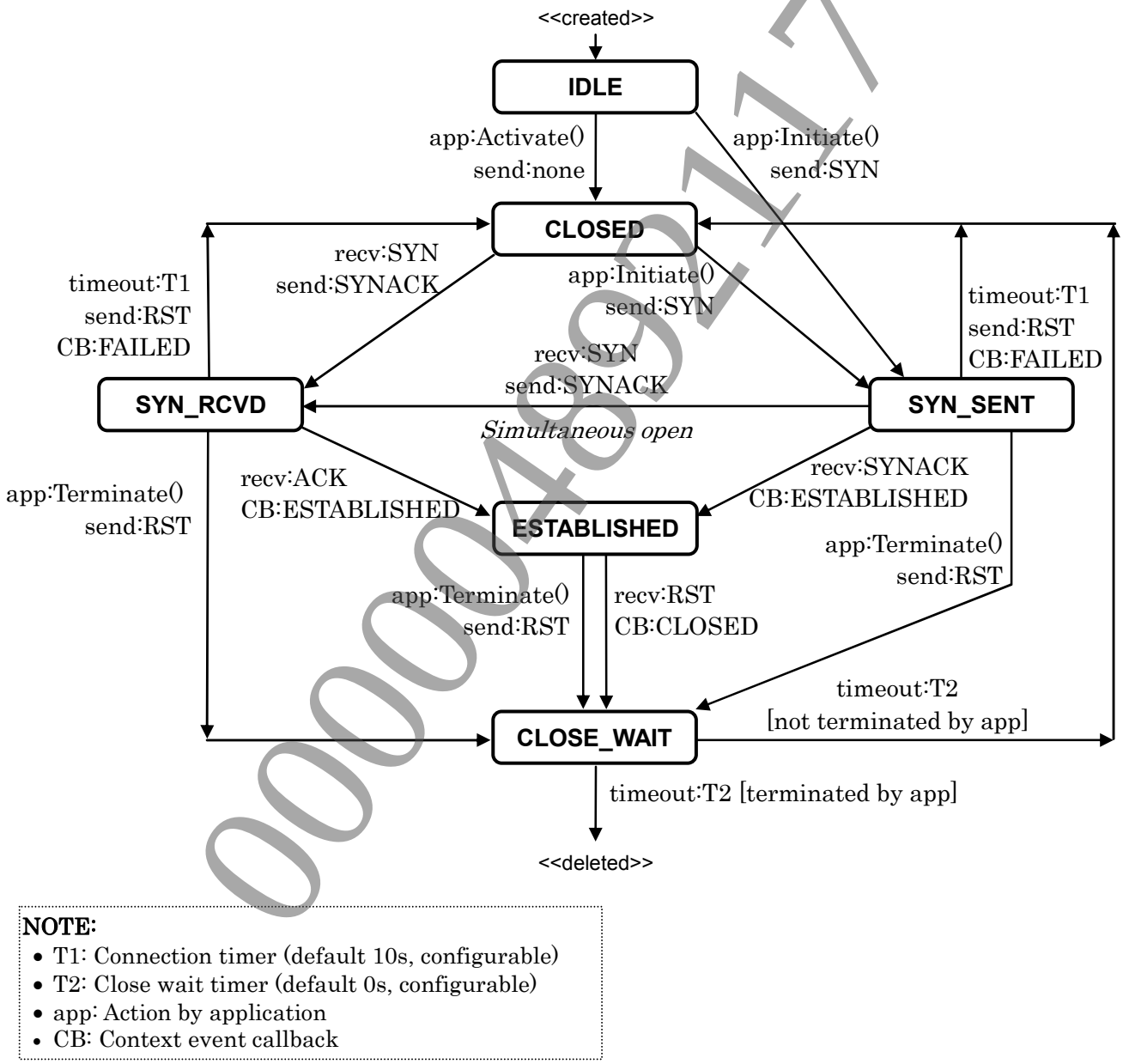
There are 6 possible states for a context, as shown in Table 1.

> **Note**
> Connections are processed according to protocol specifications specific to librudp. Messages equivalent to SYN, SYNACK, and ACK in TCP are defined, and a connection is established with a three-way handshake. The message RST (reset) is used to terminate the communication with the peer.

Table 1 Context States

State	Description
IDLE	Initial state immediately after a context is created
CLOSED	Context is active, but is not yet connected
SYN_SENT	`sceRudpInitiate()` has been called, and a connection request (SYN) has just been sent (or is being sent)
SYN_RCVD	A connection request (SYN) from a peer has just been received
ESTABLISHED	Connection is established
CLOSE_WAIT	Connection is being terminated

Figure 2 Context State Transitions



NOTE:
- T1: Connection timer (default 10s, configurable)
- T2: Close wait timer (default 0s, configurable)
- app: Action by application
- CB: Context event callback

The context state can be obtained by calling `sceRudpGetContextStatus()`. However, if the context state transitions to CLOSE_WAIT after `sceRudpTerminate()` is called, the application cannot obtain any more information (beyond the CLOSE_WAIT state) because the context ID is no longer valid.

Connection Timeout

If a certain interval elapses after the first connection request message (SYN) is sent with `sceRudpInitiate()` and there is no response from the peer, librudp notifies the context event handler of the `SCE_RUDP_CONTEXT_EVENT_CLOSED` event and aborts the connection processing. Until the connection timeout, librudp keeps resending the connection request message (SYN) to the peer. The interval for resends starts at 1 second, then is lengthened to 2, 4, 8, and 16 seconds. After this point, the message is resent every 16 seconds.

The connection timeout value can be modified by setting `SCE_RUDP_OPTION_CONNECTION_TIMEOUT` with `sceRudpSetOption()`. The default timeout value is 60000 [msec], or 60 seconds.

Close-Wait Timeout

If `sceRudpTerminate()` is called for a context that has an established connection and sends the peer a termination request message (RST), or the context receives an RST, the context transitions to CLOSE_WAIT state. The context stays in this state until a close-wait timeout occurs. If data is received in this CLOSE_WAIT state, an RST is sent again to notify the peer that the context is waiting to be terminated.

The close-wait timeout value can be modified by setting `SCE_RUDP_OPTION_CLOSE_WAIT_TIMEOUT` with `sceRudpSetOption()`. The default timeout value is 0 [msec].

Transport Type

librudp supports two types of transports with respect to handling user data boundaries, DGRAM and STREAM. The transport type can be specified per context by setting `SCE_RUDP_OPTION_STREAM` with `sceRudpSetOption()`. The default transport type is DGRAM.

DGRAM

DGRAM is like UDP in that the boundaries of the data sent by the application are maintained in the data received by the peer. This means that the application does not need to frame the data. Because in general "messages" of known formats and finite lengths are used in online games and normal applications, the DGRAM type will be used in most cases.

When sending a message longer than the maximum payload size, librudp must break up the data internally before sending it, and librudp on the receiver end must put it back together. For this reason, slightly largely delays than those caused by the STREAM type may occur.

> **Note**
> In this document, the units of DGRAM-type data written/read by the application may be referred to as "messages" because of the data boundaries.

STREAM

STREAM is like TCP, in that the boundaries of the data sent by the application are not maintained, and transports are executed in octet streams. Because there is no framing executed in librudp internally, the buffer usage is efficient, and a relatively high throughput can be expected. However, in most cases, framing will be necessary by the application.

Unless there are special circumstances, the DGRAM type is recommended for transports. Special circumstances that may warrant the use of the STREAM type instead include running an existing TCP application with librudp (using librudp for TCP emulation), or relaying data of unknown data structures (media gateways, proxies, or VPN and other tunneling).

Transport Quality Level

In librudp, it is possible to specify differing transport quality levels per context by using the DC (Delivery Critical) and OC (Order Critical) options. These options are specified in `sceRudpSetOption()` with `SCE_RUDP_OPTION_DELIVERY_CRITICAL` and `SCE_RUDP_OPTION_ORDER_CRITICAL`, respectively. The default for both options is 1 (enabled).

> **Note**
> If STREAM is set for the transport type, the settings for DC and OC are ignored. The operation will always be the equivalent of when the settings are DC=1 and OC=1.

DC Option

When DC=1, it is guaranteed that the sent data will arrive at the receiver's. If a packet loss occurs on the network, the packets are automatically resent.

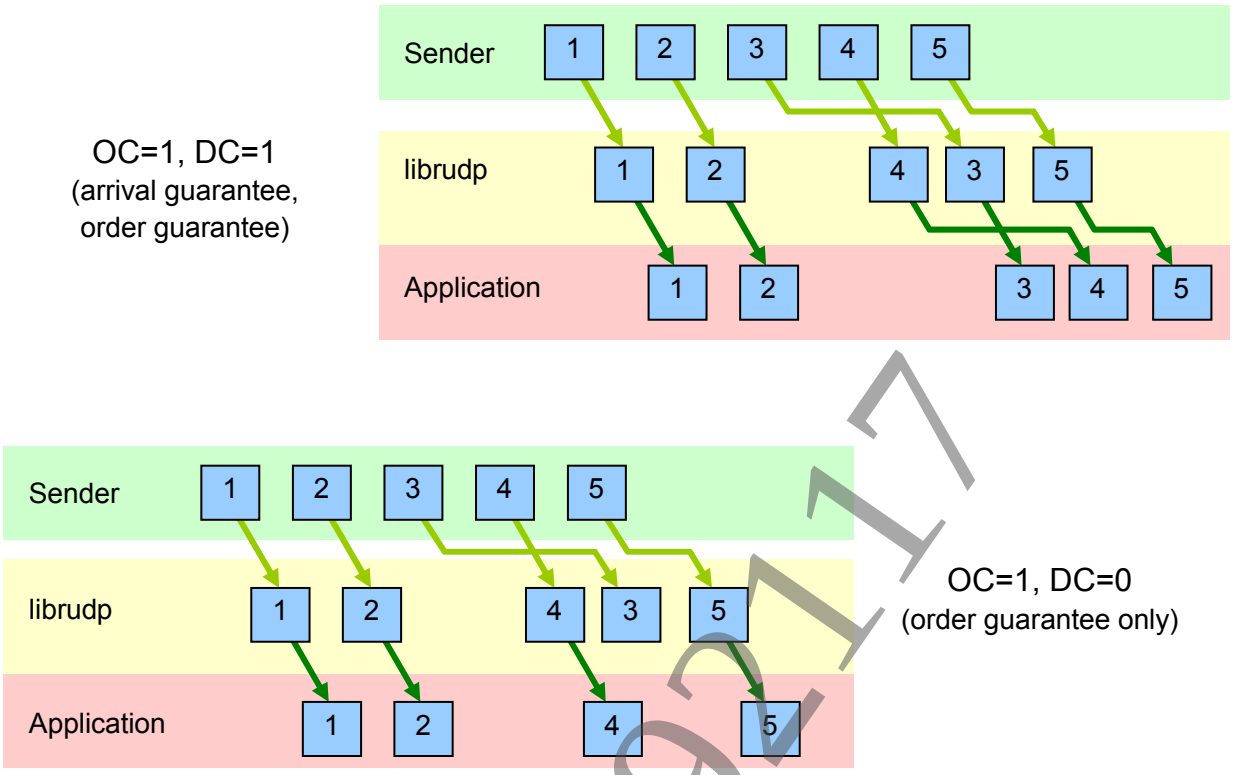
Using this option requires a relatively large bandwidth because ACK messages are returned to notify the safe arrival of packets, and transmission delays may occur due to resends. However, specifying DC=1 also means that RFC 2581 based congestion control (equivalent to that provided in TCP) is carried out.

When DC=0, there is no guarantee that all data will arrive. However, there are no confirmation ACK messages, so the bandwidth usage is efficient, and there are no delays due to resends. The disadvantage is that there is no congestion control.

OC Option

When OC=1, it is guaranteed that the sent data will arrive in the same order that it was sent. If the order of packets was changed on the network, librudp will hold on to the "later" packets and not pass them to the application in an attempt to preserve the original order. This means that when DC=1, a missing packet will hold up the subsequent packets and librudp will wait for the late packet before passing all of the packets to the application in order. When DC=0, librudp will ignore any missing packets and continue to pass the packets to the application. If the late packet arrives eventually, it will be destroyed.

Figure 3 Order Guarantee Using the OC Option



When OC=0, the order of the data arrival does not depend on the order the packets were sent. However, if there is data remaining in the receive buffer, the data is passed to the application as much as possible in the order it was sent.

DC/OC Option Combinations

The DC and OC options are orthogonal (in other words, the settings do not affect one another). Thus, four levels of transport quality are available from the combination of the two options. The following table describes each quality level and the typical use cases.

Table 2 Transport Quality Levels and Use Cases

DC	OC	Use Cases
1	1	Important messages at the start of a game, order-dependent object fields, image files
1	0	This is basically the same as when DC=1 and OC=1. It can also be used to prioritize the transmission of an urgent message to a peer by the using the LC (Latency Critical) flag
0	1	Data concerning positions, speeds, angles, which are frequently updated and sent. Data that must be up-to-date (requiring realtime significance)
0	0	Equivalent to UDP. This setting can be used for order-independent and loss-tolerant data

Multiplexing Mode

Multiplexing mode in librudp refers to the multiplexing/demultiplexing of multiple virtual ports associated with a single UDPP2P socket. In the current version, only "P2P mode" is supported. Always specify `SCE_RUDP_MUXMODE_P2P` to the argument *`muxMode`* in `sceRudpBind()`.

P2P Mode

In a peer-to-peer connection environment, communication is enabled by the use of multiple virtual ports on one UDPP2P socket.

Figure 4 P2P Mode

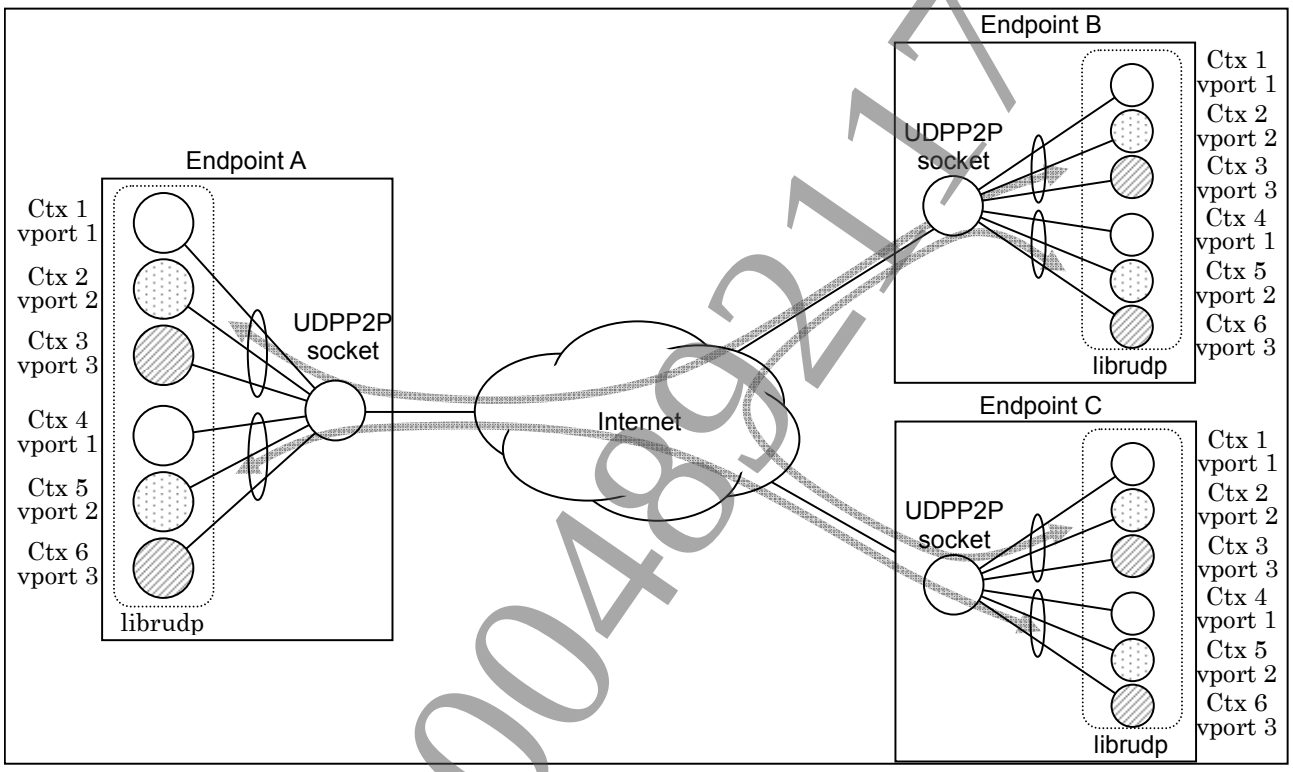


Figure 4 is an example of full-mesh connections between Endpoints A, B, and C by using P2P mode. By using multiple virtual ports (vports 1, 2, 3) at a single peer endpoint, the communication channels can be multiplexed.

Message Aggregation

When numerous messages of relatively short lengths are sent, the network bandwidth efficiency can drop precipitously because base-layer headers (in UDPP2P, UDP, IP, etc.) are added to each message. For this reason, librudp provides a message aggregation feature, in which short packets with the same destination (socket address) are sent as much as possible as one UDP packet, regardless of the context to which a packet belongs.

Message aggregation is realized by stopping the transmission for a certain interval when a short packet becomes sendable. This aggregation timeout value can be set in `sceRudpSetOption()` with `SCE_RUDP_OPTION_AGGREGATION_TIMEOUT`. The default is 30 [msec]. When this value is increased, there is a higher possibility of messages being aggregated; however, the delay will lengthen accordingly. In a typical online game, data for sending is created every rendering frame, so it is reasonable to set a timeout of the equivalent of 1 to 2 frame intervals.

Using the aggregation feature can adversely affect the bandwidth efficiency when relatively large files need to be sent as quickly as possible, because there will be delays in ACK messages. In such cases, the maximum throughput can be attained by specifying 0 for the aggregation timeout on the receiver, or by setting the `SCE_RUDP_OPTION_NODELAY` option to the receiver context.

LC (Latency Critical) Message Flag

Sometimes it is necessary to send a message to a peer as quickly as possible. In such cases, set `SCE_RUDP_MSG_LATENCY_CRITICAL` to the argument *`flags`* in `sceRudpWrite()`. This cancels the message aggregation feature and allows data to be sent immediately. However, when DC=1, the data may not be sent immediately due to network congestion or a full receive buffer.

When OC=0, data sent with the LC flag is sent even earlier than the data in the send buffer. On the receiver side too, data with the LC flag is passed to the application before the data that is already in the receive buffer.

> **Note**
> Using the LC flag allows the speedy transmission of urgent messages to peers, but causes the bandwidth efficiency to deteriorate because the aggregation feature is temporarily disabled. Do not use the LC flag frequently.

Network I/O

The I/O procedure for a UDPP2P socket depends on whether `sceRudpEnableInternalIOThread()` was called to create an internal network I/O thread, or the application will carry out the I/O by itself. The separate procedures are described below.

I/O on the Internal Network I/O Thread

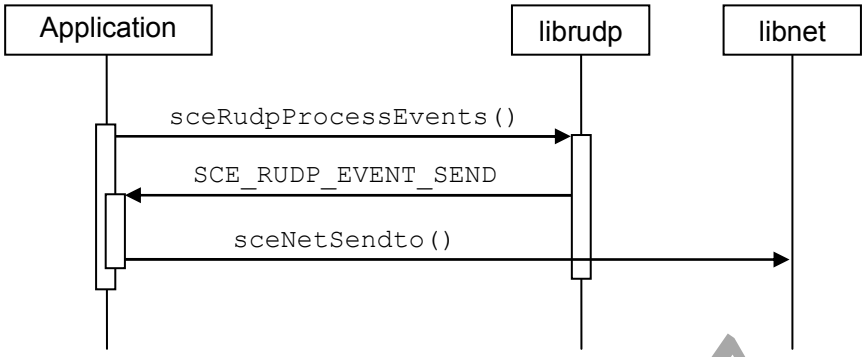
When the internal network I/O thread is used, data transmissions can be executed just by calling `sceRudpRead()` and `sceRudpWrite()` in the application. The application is not required to execute any of the event handling described in this section.

Network I/O by the Application

Sending Data on a UDPP2P Socket

Writes are executed to a UDPP2P socket when the event `SCE_RUDP_EVENT_SEND` is notified to the common event handler. All the information necessary for calling the libnet API `sceNetSendto()` is passed to the arguments of the common event handler, so use this information to handle the transmission inside the common event handler.

Figure 5 UDPP2P Socket Transmissions: Sending Data



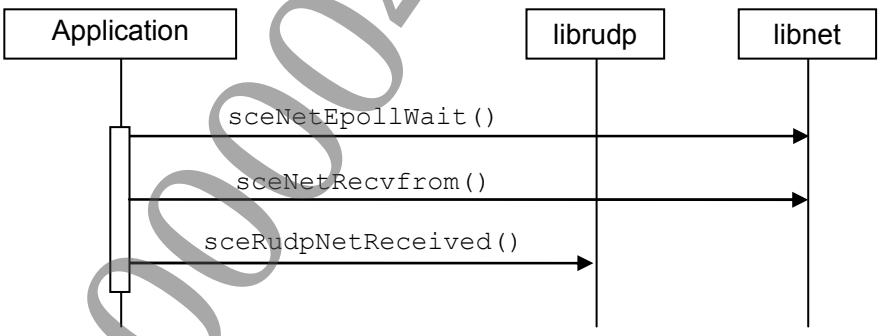
The common event handler must return the `sceNetSendto()` call results to librudp. If the return value of `sceNetSendto()` is 0 or over, indicating normal termination, return the value as is. If `sceNetSendto()` returns a negative value, check the value of `sce_net_errno`. If the error value is `SCE_NET_EAGAIN` or `SCE_NET_EWOULDBLOCK`, return `SCE_RUDP_ERROR_WOULDBLOCK`. For all other errors, return `SCE_RUDP_ERROR_INVALID_SOCKET`.

When the common event handler returns `SCE_RUDP_ERROR_INVALID_SOCKET`, librudp notifies the error to all the contexts associated with the socket. When `SCE_RUDP_ERROR_WOULDBLOCK` is returned, librudp simply ignores the error.

Receiving Data on a UDPP2P Socket

Reads from a UDPP2P socket are executed as part of the application's interval processing, or by calling the libnet API `sceNetRecvfrom()` on the network I/O thread of the application. When data is received, set the necessary arguments and call `sceRudpNetReceived()` to pass the data to librudp.

Figure 6 UDPP2P Socket Transmissions: Receiving Data



Freeing a UDPP2P Socket

After a context is closed with `sceRudpTerminate()`, the UDPP2P socket is still used internally by librudp (for sending RST messages, etc.) until the context moves beyond the CLOSED_WAIT state.

It is also possible for the UDPP2P socket to be in use by another context. Freeing a socket immediately after closing a context can lead to a peer closing the connection on a timeout because the peer is not notified of the context closing.

To avoid this situation, free the UDPP2P socket only after the event `SCE_RUDP_EVENT_SOCKET_RELEASED` is notified to the common event handler by librudp. librudp manages the UDPP2P sockets bound with `sceRudpBind()` and will notify this event when it detects that

a UDPP2P socket is no longer used by any of the contexts. When this event notification is received, a socket can be safely freed.

Blocking Mode

The default is for a context to operate in nonblocking mode, which means that functions like `sceRudpRead()` and `sceRudpWrite()` are executed as nonblocking functions. The blocking mode can be specified by setting the value of `SCE_RUDP_OPTION_NONBLOCK` option to 0 with `sceRudpSetOption()`.

Blocking Functions

The applicable functions behave as follows in blocking mode.

Table 3 Behavior of Blocking Functions in Blocking Mode

Blocking Function	Behavior in Blocking Mode
`sceRudpInitiate()`	This function will be blocking until the connection processing completes. If the connection is successful, 0 will return. If it fails, one of the defined error codes will return. The events that are notified in nonblocking mode (ESTABLISHED and CLOSED) will not be notified.
`sceRudpActivate()`	This function will be blocking until the connection request from a peer is processed. If the connection is successful, 0 will return. If it fails, one of the defined error codes will return. The event notified in nonblocking mode (ESTABLISHED) will not be notified.
`sceRudpRead()`	**DGRAM type:** This function will be blocking until the next message (to receive) is completely written to the buffer passed by the application. **STREAM type:** This function will be blocking until 1 byte or more of the data to receive is written to the buffer passed by the application. **For both transport types:** When the read is successful, the data length read will return. If it fails, one of the defined error codes will return. The event notified in nonblocking mode (READABLE) will not be notified.
`sceRudpWrite()`	**DGRAM type:** This function will be blocking until the message that was requested to be sent is written completely to the send buffer. **STREAM type:** This function will be blocking until 1 byte or more of the data requested to be sent is written to the send buffer. **For both transport types:** When the write is successful, the data length written will return. If it fails, one of the defined error codes will return. The event notified in nonblocking mode (WRITABLE) will not be notified.
`sceRudpFlush()`	When DC=1, this function will be blocking until it is confirmed that the data written until immediately before has been written to the peer's receive buffer. When DC=0, this function will return when the data is sent. The event notified in nonblocking mode (FLUSHED) will not be notified.

Timeouts for Blocking Functions

Timeouts can be set for all blocking functions by using `sceRudpSetOption()`.

Table 4 Blocking Timeouts

Blocking Function	Option	Default Value [msec]
`sceRudpInitiate()`	`SCE_RUDP_OPTION_CONNECTION_TIMEOUT`	60000 (60 seconds)
`sceRudpActivate()`	`SCE_RUDP_OPTION_CONNECTION_TIMEOUT`	60000 (60 seconds)
`sceRudpRead()`	`SCE_RUDP_OPTION_READ_TIMEOUT`	0 (no timeout)
`sceRudpWrite()`	`SCE_RUDP_OPTION_WRITE_TIMEOUT`	0 (no timeout)
`sceRudpFlush()`	`SCE_RUDP_OPTION_FLUSH_TIMEOUT`	0 (no timeout)

When a timeout occurs with `sceRudpInitiate()` or `sceRudpActivate()`, the error code `SCE_RUDP_ERROR_CONN_TIMEOUT` will return. When any other blocking function times out, `SCE_RUDP_ERROR_WOULDBLOCK` will return.

SCE_RUDP_MSG_DONTWAIT Flag

When this flag is set to the argument *`flags`* in `sceRudpRead()` or `sceRudpWrite()` in blocking mode, the functions will return immediately without blocking. If at this time readable data does not exist, or there is no space in the send buffer, the error `SCE_RUDP_ERROR_WOULDBLOCK` will return, as when a blocking timeout occurs.

Cancelling Blocking

A function can be released from its blocking state when nonblocking mode is specified for the context ID on another thread, or when `sceRudpTerminate()` is called. A function released from its blocking state will return `SCE_RUDP_ERROR_CANCELLED`.

Polling

librudp has a polling feature similar to epoll that monitors asynchronous context events. When `sceRudpProcessEvents()` is called on a different thread, or when the internal network I/O thread is used, synchronization with the application thread is necessary because callbacks are called asynchronously. Using the librudp polling feature greatly simplifies the required programming as it will handle this synchronization for asynchronous callbacks.

The procedure for the polling feature is as follows.

(1) Create a Polling ID

Call `sceRudpPollCreate()` and obtain a polling ID. To the argument *`size`*, specify the maximum number of contexts expected to be monitored by the polling ID. This size does not indicate the limit; it is merely used to indicate the size of the internal memory to allocate in the initial state. If the number of contexts to monitor increases, the internal memory is automatically increased. The actual limit on the number of contexts is 65536.

(2) Add a Context and Specify the Events to be Monitored

Call `sceRudpPollControl()`, specifying the new polling ID, `SCE_RUDP_POLL_OP_ADD`, the ID of the context to be monitored, and the event flags (`SCE_RUDP_POLL_EV_XXX`) to watch for.

(3) Wait for Events

Call `sceRudpPollWait()` to wait for events. To the arguments *`events`* and *`eventLen`*, specify a pointer to a `SceRudpPollEvent` type array and the size of this array. The array can be any size; set the size to the maximum number of events to process at one time.

Information of events will be stored to this array in the order that they occur, starting from the beginning of the array. It is possible to set the maximum wait time to the argument *`timeout`*.

When one or more events are detected, the number of detected events is returned by `sceRudpPollWait()`. The application should use this number to scan the `SceRudpPollEvent` array and perform event handling. If a timeout occurs during a `sceRudpPollWait()` call, 0 will return.

It is possible to cancel this blocking wait for events by calling `sceRudpPollCancel()` on another thread. If `sceRudpPollWait()` has already been called, it will return immediately with `SCE_RUDP_ERROR_CANCELLED` as the return value. If there are no threads waiting with `sceRudpPollWait()`, the next call of `sceRudpPollWait()` will return immediately with `SCE_RUDP_ERROR_CANCELLED`. This cancellation feature can be useful when it is necessary to execute a non-network operation immediately on a thread that is currently executing a polling.

(4) Modify the Events to be Monitored

Call `sceRudpPollControl()`, specifying the polling ID, `SCE_RUDP_POLL_OP_MODIFY`, the context ID, and the new event flags.

(5) Delete a Context (End Monitoring)

Call `sceRudpPollControl()`, specifying the polling ID, `SCE_RUDP_POLL_OP_REMOVE`, and the context ID. Specify 0 to *`events`*, which will not be used.

(6) Destroy a Polling ID

Call `sceRudpPollDestroy()`, specifying the polling ID to destroy.

Polling Event Flags

The following four types of polling event flags are defined. Set the polling event flags according to the events to be monitored.

Table 5 Polling Event Flags

Event Flag	Corresponding Event
`SCE_RUDP_POLL_EV_READ`	- Receive data became readable - Communication was terminated (*1)
`SCE_RUDP_POLL_EV_WRITE`	- (When processing a connection) The connection succeeded or failed (*2) - (After a connection is established) Data can be sent
`SCE_RUDP_POLL_EV_FLUSH`	- Send data was flushed
`SCE_RUDP_POLL_EV_ERROR`	- Context is already invalid (*3) - A memory error occurred (*4) - A socket error occurred while sending data (*4)

(*1) This event will be detected regardless of the state of the receive buffer when there is a disconnect request from a peer. If there is readable data in the receive buffer at this time, the data can be read. If there is no more data to be read in CLOSE_WAIT state, the error `SCE_RUDP_ERROR_END_OF_DATA` will be returned by `sceRudpGetSizeReadable()` or `sceRudpRead()`.

(*2) This event will be detected when connection processing completes, regardless of whether the connection is successful.
The success of a connection can be determined by obtaining the context's error information (by specifying `SCE_RUDP_OPTION_LAST_ERROR` and calling `sceRudpGetOption()`), or by calling `sceRudpGetContextStatus()` and checking whether the context state is ESTABLISHED.

(*3) This error will be detected if `sceRudpTerminate()` was called without calling `sceRudpPollControl()` to delete the context ID being monitored.

(*4) When this event is detected, obtain the error information of a context by calling `sceRudpGetOption()` and specifying `SCE_RUDP_OPTION_LAST_ERROR`.

> **Note**
> In polling, `SCE_RUDP_POLL_EV_ERROR` is a fatal error. If the context is not deleted after this error is detected (by calling `sceRudpPollControl()` with `SCE_RUDP_POLL_OP_REMOVE` specified), the context will automatically be disregarded as a polling target when `sceRudpPollWait()` is next called.

Lag Switch Cheat Detection

When the transport type is DGRAM and DC=1, it can be determined whether the data read with `sceRudpRead()` was resent. The number of times the data was resent and the time it took in msec (from the first attempt to the last resend) can be obtained. This information can be obtained by passing a pointer to a `SceRudpReadInfo` object (for storing the information) to the argument *`info`* when calling `sceRudpRead()`.

> **Note**
> If the sent message was divided into segments larger than the maximum payload size, only the information of the last segment will be obtained.

This information can be used as part of the process of determining whether there was cheating with a lag switch. However, resends can also occur due to valid reasons such as congestion and other network problems, so the application must use other statistical information when making a judgment.

Keep-Alive

librudp provides a keep-alive feature as supported by the widely used TCP protocol. This particular feature of librudp does not have to be used if the signaling feature of the NP Matching 2 library is used for NAT traversal, since keep-alive is performed by signaling. However it can be useful if the NAT traversal system (or peer address exchange) is an application original.

In the default settings, the keep-alive feature of librudp is set to disabled. It is enabled when `SCE_RUDP_OPTION_KEEP_ALIVE_INTERVAL` is specified in `sceRudpSetOption()` with a non-zero value (in milliseconds). If there are no send/receive transmissions during this specified interval, a keep-alive packet will be sent.

If the keep-alive timeout elapses without a response to the keep-alive packet, the connection is seen to be closed, and either `SCE_RUDP_CONTEXT_EVENT_CLOSED` will be notified to the application, or an error (a negative value) will return to `sceRudpRead()`. The keep-alive timeout can be set in `SCE_RUDP_OPTION_KEEP_ALIVE_TIMEOUT` with `sceRudpSetOption()`. The default timeout is 20 seconds.

Connection Sequences

UDPP2P Connection Sequence

Figure 7 and Figure 8 show the connection and termination sequences between Endpoint A and Endpoint B, when UDPP2P is used as the base layer.

Figure 7 UDPP2P Connection Sequence

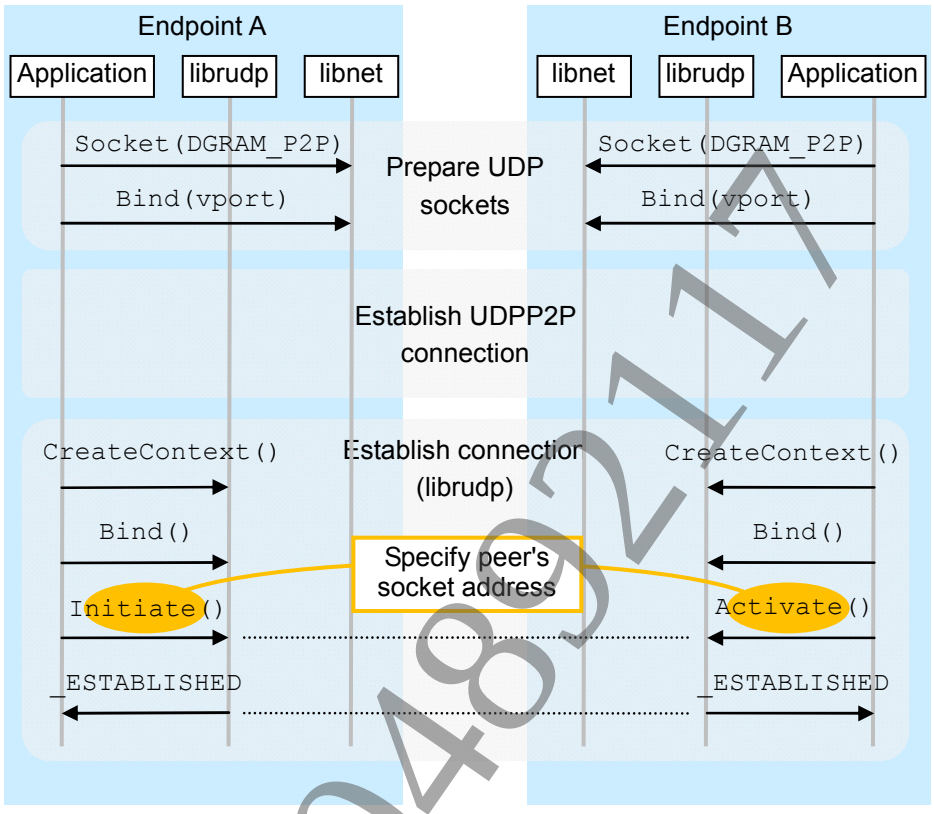
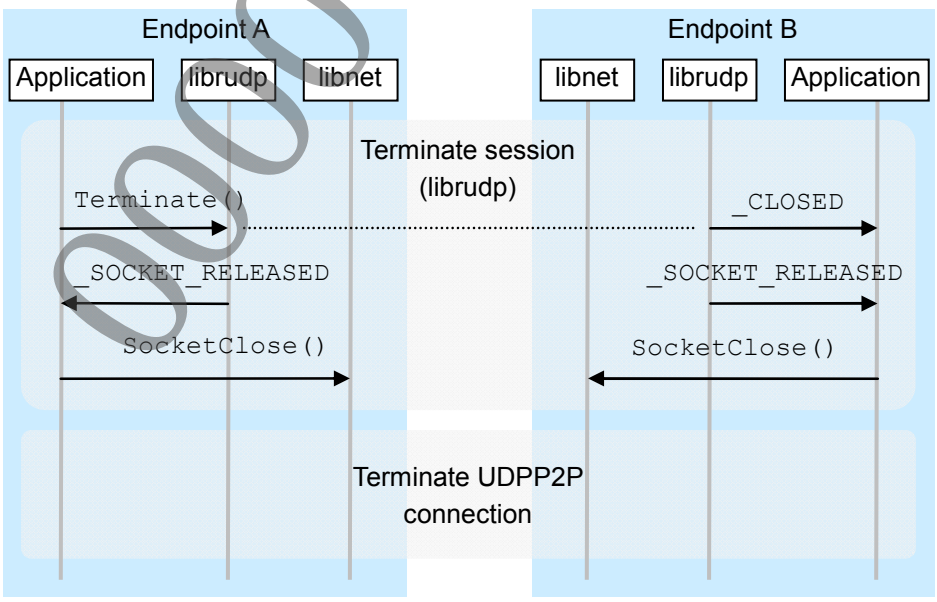


Figure 8 UDPP2P Termination Sequence



4 Notes

Conditions for READABLE and WRITABLE Events

When a context is in nonblocking mode and polling is not used, the events `SCE_RUDP_CONTEXT_EVENT_READABLE` and `SCE_RUDP_CONTEXT_EVENT_WRITABLE` are notified to the context event handler to indicate the states of the send/receive buffer. The conditions under which these events are notified are as follows.

SCE_RUDP_CONTEXT_EVENT_READABLE

- Readable data exists in the receive buffer, and `sceRudpRead()` has been called at least once since the last READABLE event

SCE_RUDP_CONTEXT_EVENT_WRITABLE

- Not all of the requested size was written to the send buffer in the last write request, and there is space of at least one payload size (`SCE_RUDP_OPTION_MAX_PAYLOAD`) in the send buffer

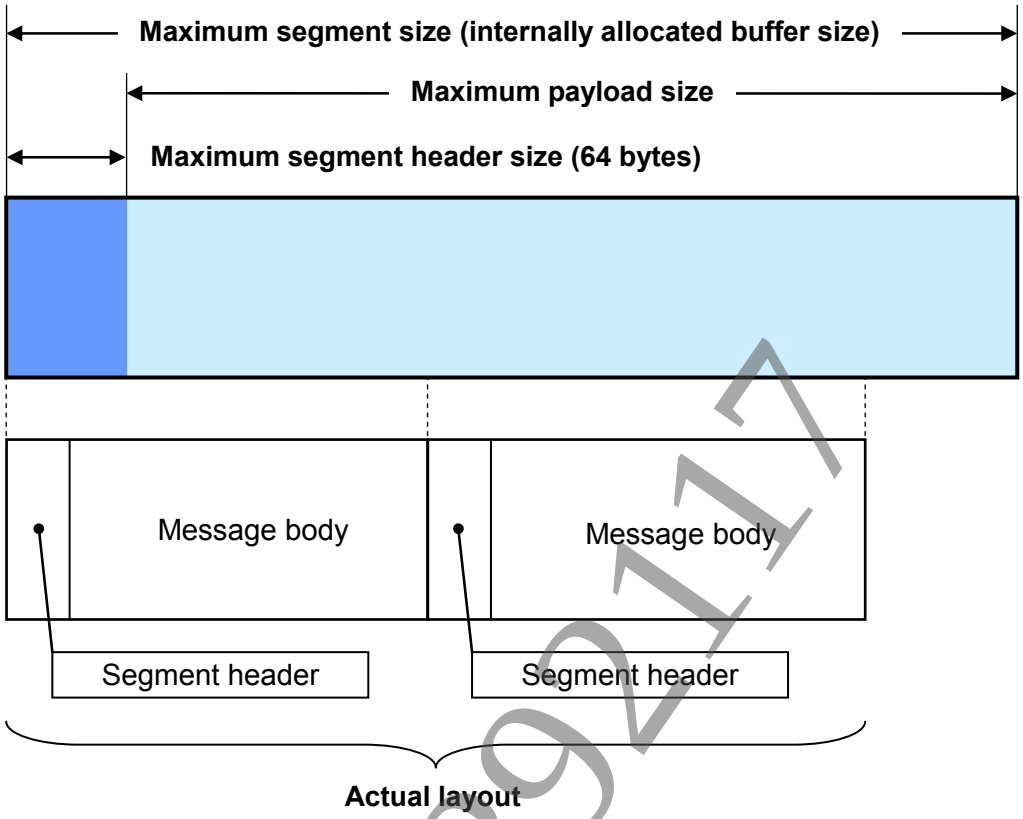
The READABLE event will never occur more than the number of times `sceRudpRead()` was called.

Even when the WRITABLE event returns, note that an attempt to write data exceeding the length of `SCE_RUDP_OPTION_MAX_PAYLOAD` will not necessarily succeed.

Maximum Payload Size and Maximum Segment Size

A different maximum payload size can be set per context with `SCE_RUDP_OPTION_MAX_PAYLOAD`. However, the maximum segment size is the same for all contexts and is always the payload size of the base layer (UDPP2P). Since the maximum segment size includes the header size (maximum 64 bytes) of the RUDP protocol, the maximum payload size of a context must be set to a value that is at least 64 bytes smaller than the maximum segment size. This is because it must be guaranteed that the maximum payload size always fits within the maximum segment size regardless of the header size. Segment header size varies according to communication conditions; however, it is usually 8 to 10 bytes.

Figure 9 Maximum Payload Size and Maximum Segment Size



Maximum Size of Sendable Messages

In general, the value set to `SCE_RUDP_OPTION_MAX_PAYLOAD` is the maximum size of messages that can be sent. With the STREAM type, and when both DC and OC are enabled with the DGRAM type, a message longer than the specified payload length can be sent by calling `sceRudpWrite()` just once, but the message is broken up into multiple packets of the maximum segment size or smaller. (The maximum segment size can be set with `sceRudpSetMaxSegmentSize()`.) With the DGRAM type, this can lead to a performance hit because a relatively large area of the receive buffer is used by the receiver for merging the packets. For this reason, it is recommended for data to be sent in a size equal to or smaller than the maximum payload size so that it will not be divided into packets. There are no performance hits with the STREAM type, because the boundaries between the units of sent data are not maintained.

When using other modes of DGRAM type, a message longer than the specified payload length can not be sent.

Improving the Usage Efficiency of the Send/Receive Buffer

In order to keep the frequency of allocations from heap memory to a minimum, the send/receive buffer inside librudp is allocated in units of the maximum payload size (`SCE_RUDP_OPTION_MAX_PAYLOAD`) plus 64 bytes (the maximum header length in RUDP). For this reason, the usage efficiency of the send/receive buffer will improve if the maximum size of the messages sent/received by the application is set to `SCE_RUDP_OPTION_MAX_PAYLOAD`.

Limitation of UDP Sockets and Multiplexing Mode

Contexts that share a single UDPP2P socket must have the same multiplexing mode specified. If a context specifies the same UDPP2P socket as a previous context but a different multiplexing mode, the attempt to bind the context with `sceRudpBind()` will fail and return `SCE_RUDP_ERROR_INVALID_MUXMODE`.