Assignment 3: Reliable UDP

Peter Sjödin

Overview

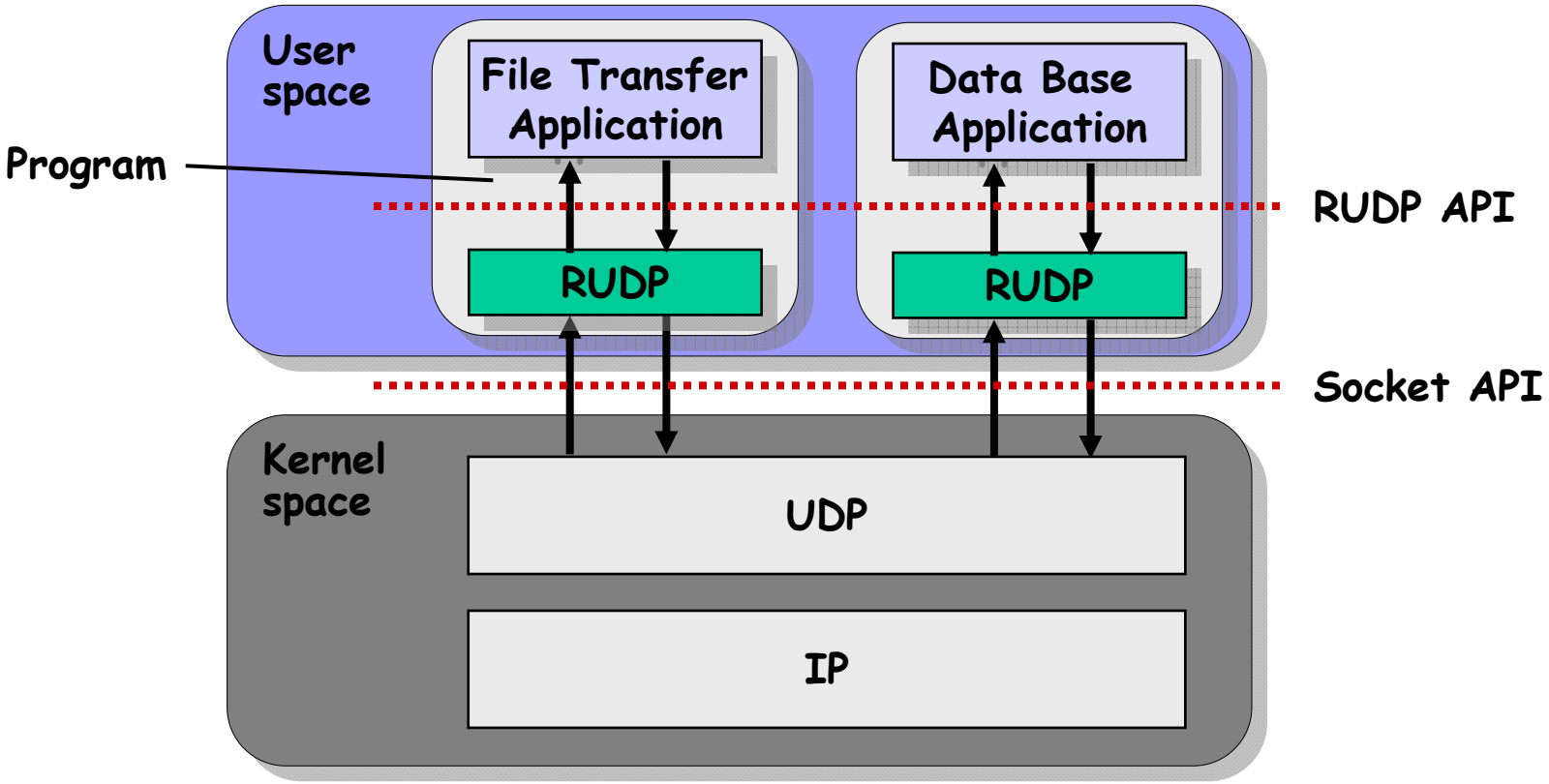
- Design and implement a sliding window protocol for reliable transmission of UDP packets
- Learn about
  - Socket programming
  - Sliding window flow and error control
  - Event-driven programming
  - Protocol state management
- Recommended language is C

RUDP

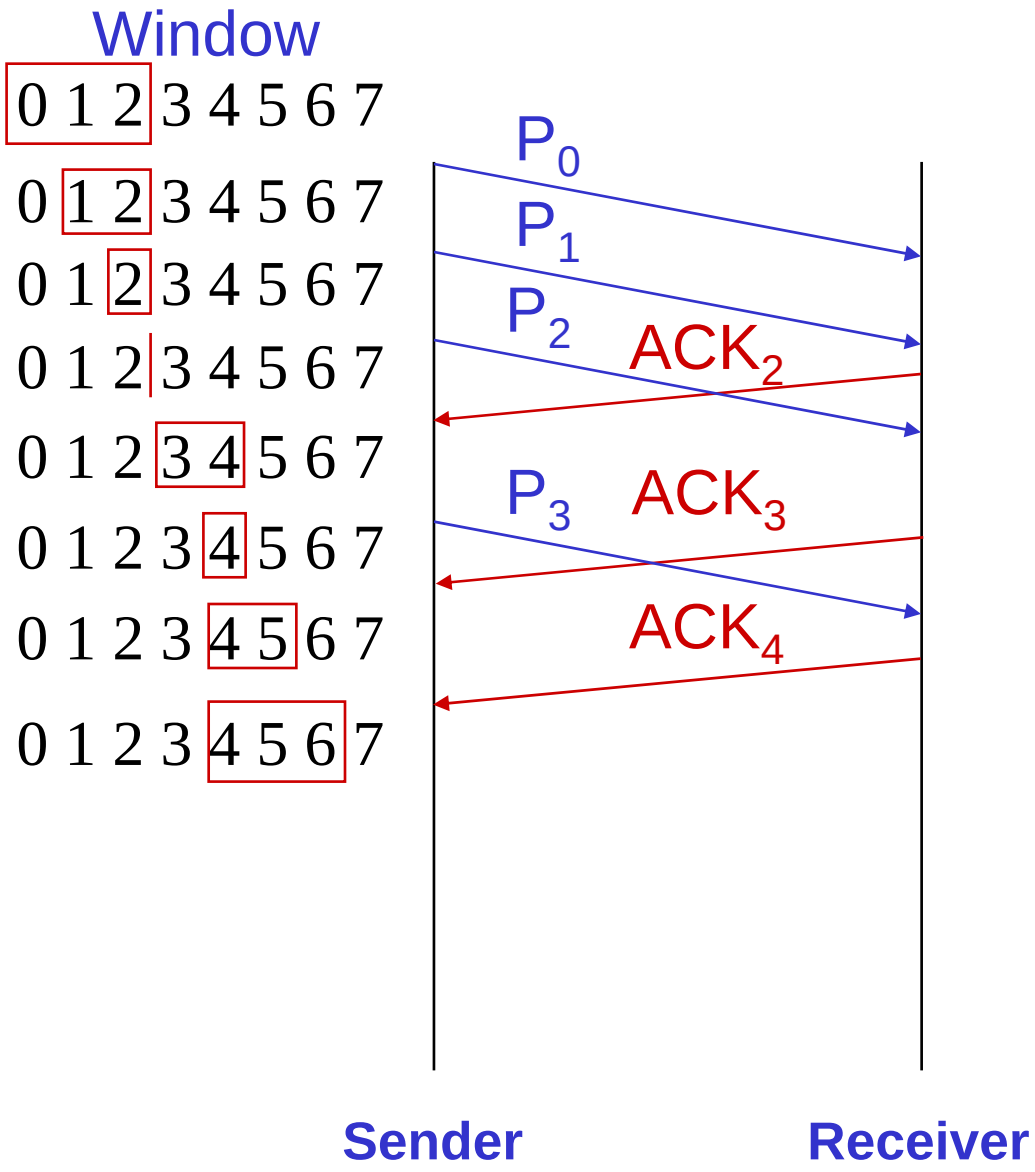
- Reliable UDP
- Sliding window flow control
- Detection and ARQ-based retransmission of lost packets
- Detection and (potentially) reordering of packets that arrive out of order
- Motivation
  - A reliable protocol for sending datagrams

RUDP Library

- RUDP runs in user space, in the same process as the application
  - Implemented as a library, with a well defined well-defined API (Application Programming Interface)
  - Set of function declarations, defined by us
  - Your job is to write those functions



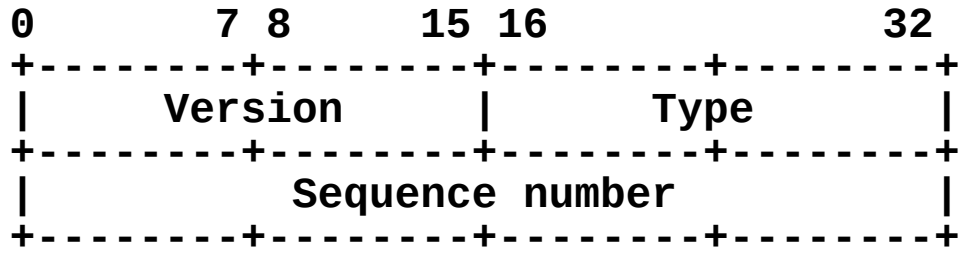
Sliding Window (Window Size 3)



Go-Back-N Sliding Window Protocol

- Understand the details of a basic, sliding window protocol
- Keep it simple
- An ACK is an ACK (and not a NACK)
  - The receiver sends an ACK if (and only if) it receives the next packet in sequence
    - You cannot use an ACK to tell the sender that a packet has been lost, for instance
  - The sender increases the window in accordance with the ACK
- Retransmissions are triggered by timeouts
  - And nothing else
    - In particular, receiving an ACK with unexpected sequence number **does not** trigger a retransmission
      - Why?

RUDP Header



- Version: version of the RUDP protocol
  - We use version 1!
- Type: packet type
  - SYN, FIN, DATA, or ACK
- How to use sequence numbers:
  - SYN packets: random
  - DATA packets: increases by one for each packet sent
  - ACK packets: sequence number of next expected DATA packet
  - FIN packets: sequence number of last DATA packet plus one

Asynchronous I/O Model

- An event scheduler is provided (in C)
- Schedules events based on
  - Input available on file descriptors
    - Sockets, files, etc
  - Timer-outs
- Internally, the event scheduler uses the **select()** system call
  - Blocking I/O with multiple descriptors
- Do not attempt to use threads or similar!
  - Although it may seem tempting at first…

Event Handler API

```
int event_fd(int fd,
             int (*callback)(int fd, void *arg),
             void *callback_arg, char *idstr);

int event_fd_delete(int (*callback)(int fd, void *arg),
                    void *callback_arg);
```

```
int event_timeout(struct timeval timer,
                  int (*callback)(int f, void *arg),
                  void *callback_arg, char *idstr);

int event_timeout_delete(int (*callback)(int fd, void *arg),
                         void *callback_arg);
```

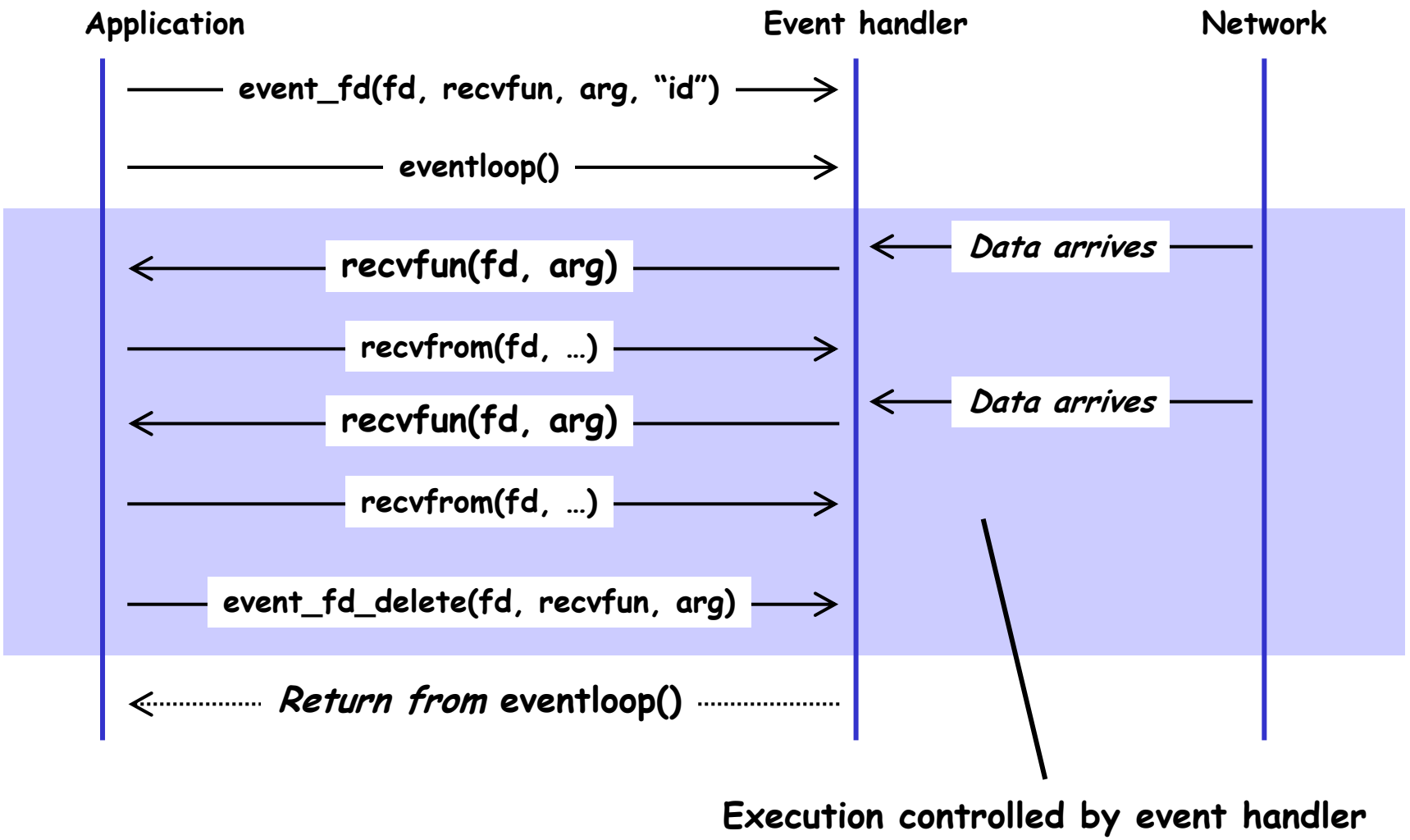
- Registration and deletion of event handlers
- Two kinds of events
  - Input events on file descriptors ("fd")
  - Timeouts
- An event handler is identified internally by
  - Handler function ("callback" parameter)
  - Parameter that will be passed to handler function ("callback_arg" parameter)

Activating the Event Handler

```
void eventloop();
```

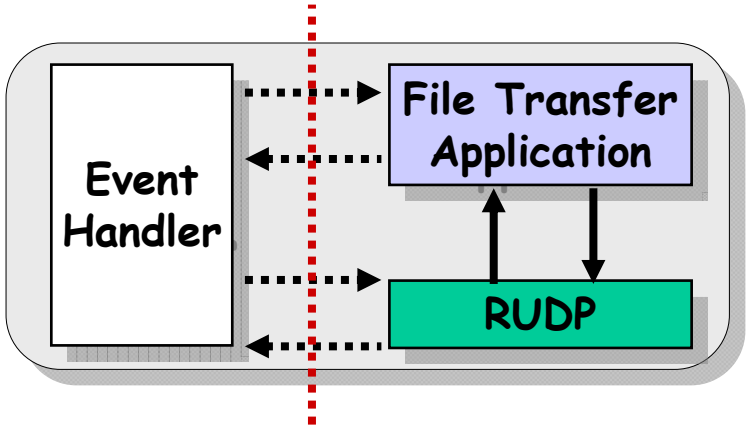
- By calling "eventloop", control is transfered to the event handler
- Event handler will return when either
  - There are no event handlers
    - All event handlers have been deleted
      - **event_fd_delete(), event_timeout_delete()**
  - An event handler returns a negative value

Asynchronous I/O With Event Handler



Event Handler is a Scheduler

- Controls the execution of the application program
- Implemented as a library
  - Just as RUDP!
- Both the application protocol and RUDP need to interact with event handler



RUDP API

```
rudp_socket_t rudp_socket(int port);

int rudp_close(rudp_socket_t socket);

int rudp_sendto(rudp_socket_t socket, void* data, int len,
                struct sockaddr_in* to);

int rudp_recvfrom_handler(rudp_socket_t socket,
                          int (*handler)(rudp_socket_t,
                                         struct sockaddr_in *from,
                                         char *data, int len));

int rudp_event_handler(rudp_socket_t socket,
                       int (*handler)(rudp_socket_t, rudp_event_t,
                       struct sockaddr_in *remote));
```

- Your code should conform to this API
- Function declarations provided by us

rudp_socket()

```
rudp_socket_t rudp_socket(int port);
```

- Create a RUDP socket
  - "port" is local port number; zero means a random port number is chosen
- Returns a RUDP socket handle (of type "rudp_socket_t")

rudp_close()

```
int rudp_close(rudp_socket_t socket);
```

- Close a RUDP socket
- Close implies a controlled shutdown of the socket!
  - Any pending data will be transmitted as a result
- Returns zero if OK, otherwise -1

rudp_sendto()

```
int rudp_sendto(rudp_socket_t socket, void* data, int len,
                struct sockaddr_in* to);
```

- Send a packet ("data" and "len")
- The destination address is given as an INET socket address ("struct sockaddr_in")
  - IP address ("sin_addr") and port number ("sin_port")
- Returns zero if OK, otherwise -1

rudp_recvfrom_handler()

```
int rudp_recvfrom_handler(rudp_socket_t socket,
                          int (*handler)(rudp_socket_t rsocket,
                                         char *data, int len,
                                         struct sockaddr_in *from));
```

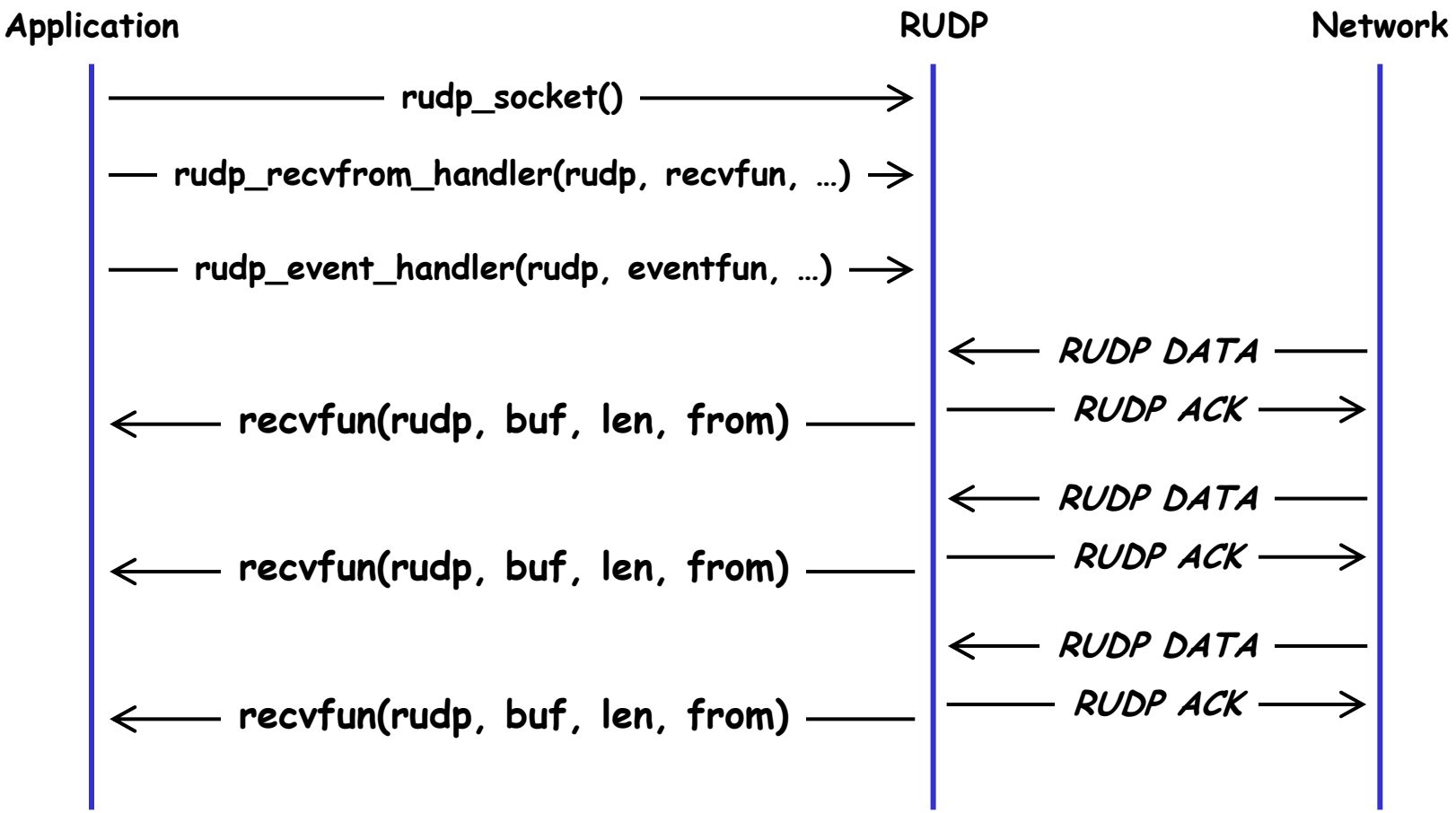
- Register a handler function ("handler") for receiving packets on a socket
- Call to rudp_recvfrom_handler() will return immediately
  - Zero if OK, -1 if error
- Handler will be called when packets arrive
  - With packet ("data" and "len") and address of sender ("from")

rudp_event_handler()

```
int rudp_event_handler(rudp_socket_t socket,
                       int (*handler)(rudp_socket_t rsocket,
                                      rudp_event_t event,
                                      struct sockaddr_in *remote));
```

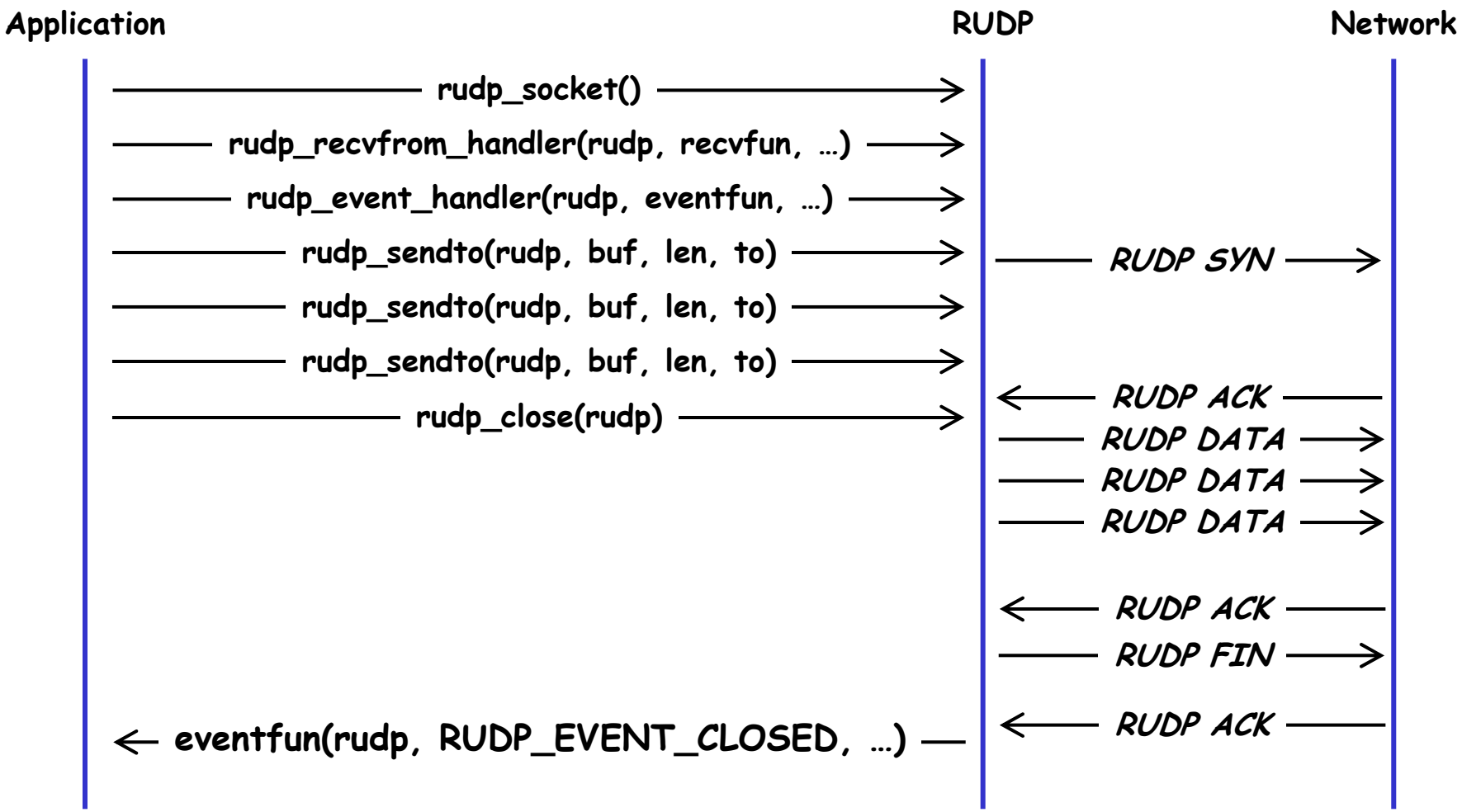
- Register handler function "handler" for other events
- Call to rudp_event_handler() will return immediately
  - Zero if OK, -1 if error
- Handler will be called when events happen
  - With event code ("event") and address of peer ("remote")
- Two events defined
  - **RUDP_EVENT_TIMEOUT** signals that a timeout occured in communication
    - During setup ("SYN"), data transfer, or shutdown ("FIN")
  - **RUDP_EVENT_CLOSED** signals that socket has been closed in a controlled manner
    - All packets have been transmitted and/or acknowledged

RUDP Receiver Side



In our sample application, the receiver never closes the socket (*why?*)

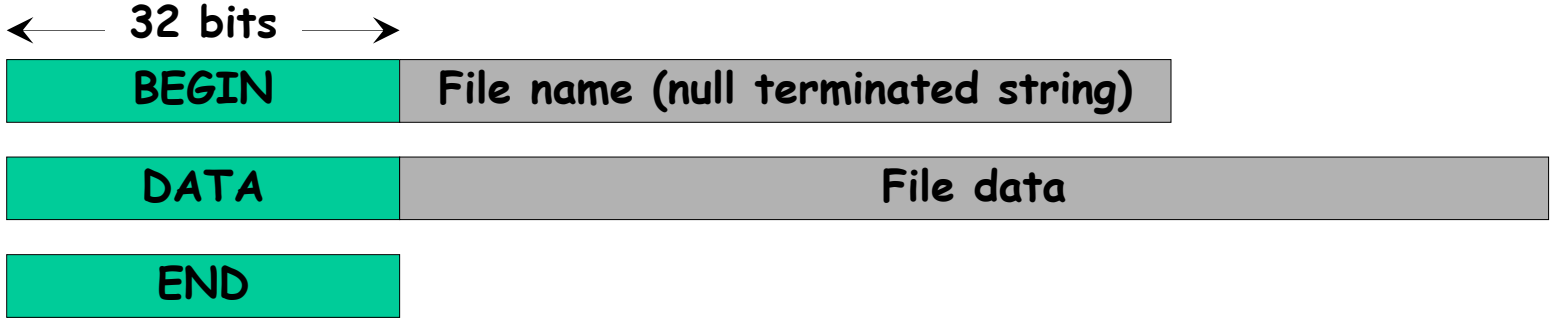
RUDP Sender Side



Protocol Control Blocks

- An application can open multiple RUDP sockets
- Each RUDP socket can be used for communication with multiple peers
  - "sendto" and "recvfrom" calls have a socket address as parameter
- Two levels
  - Multiple RUDP sockets
  - Multiple peers per socket
- Need to
  - Maintain state for per-socket "peers"
  - Have a way to look up peer state
  - Maintain queues with outbound packets
    - See "RUDP Sender Side" slide

Sample Application: VSFTP



- 32-bit type field
  - BEGIN, DATA, END
- Length implicit from RUDP length

Applications

vs_send [-d] host1:port1 [host2:port2 ...] file1 [file2 ...]

- Unlike many other file transfer programs, VSFTP can send many files concurrently to many receivers
- List of "host:port" pairs defines recipients
- List of file names defines files
- Uses RUDP sockets
  - One RUDP socket per file
  - On each RUDP socket, VSFTP packets are duplicated to all recipients
  - So with five recipients:
    - Five "BEGIN" packets are sent first
    - Then five "DATA" packets for each segment of data
    - Finally five "END" packets are sent

Applications

vs_recv [-d] port

- Create a VSFP receiver on port
- Will receive and process all incoming VSFTP file commands
- Copies of received files placed in current directory
  - So don't run it in the same directory as where the sender is!
  - That would open the same file for reading and writing
    - With unexpected results...

File package

- **"rudp.tar"**
- **RUDP**
  - **rudp_api.h** Declarations for the RUDP API.
  - **rudp_proto.h** Declarations for the RUDP protocol.
  - **rudp.c** Stub functions and prototypes for the RUDP protocol.
- **Event Manager**
  - **event.h** Declarations for the event manager.
  - **event.c** The event manager.
- **Sample Application**
  - **vsftp.h** Declarations for the sample application.
  - **vs_send.c** Sender for the sample application.
  - **vs_recv.c** Receiver for the sample application.
  - **Makefile** For building the sample application.

Grades

- Basic: Single socket, single peer
  - vs_send with one file, one destination
- Medium: Multiple sockets, single peer
  - vs_send with multiple files, one destination
- Advanced: Multiple sockets, multiple peers
  - vs_send with multiple files, multiple destinations

Notes and Hints

- One of the main challenges in this assignment is understanding exactly how the code should work
  - Think through the protocol carefully
    - Closing a socket will require much attention
    - Just have a look at what "vs_send" does…
  - Think through the dynamic behaviour of the RUDP library
    - What happens, and when
    - Check the slides!
  - Define the protocol states and transitions
    - <current state, event, action, new state>
    - In fact, you should include a state diagram in your report
      - Finite state machine
- Do not attempt to use threads
  - Even though it may seem tempting at first
- It may be a good idea to decide early on which grade to aim for
- May we do this in a different language?
  - No

Due Date

- Sunday May 11
- Code package with instructions for compiling and running
- Short written report
  - Including protocol specification