Project 3

Unreliable Datagrams

Mayur Patel

Slide heritage: Previous TAs

Cornell CS 4411, October 7, 2016

1 Project Scope

2 Implementation details
- Using the networking pseudo-device
- Interrupts
- Miniports
  - An Example

3 Concluding Thoughts (Grading)

Basics

- How to make computers to communicate?
- **Protocol**: Set of formats and rules to exchange data.
- Different layers of abstraction for different functionality.
- **UDP**: A *transport layer* protocol.

What do unreliable datagrams involve?

- Build a UDP/IP-like networking stack.
- Use the pseudo-network interface `network.h` for "IP".
- Use ports to identify endpoints.
- A *minimessage* layer for thread I/O.

The Interface

```
void minimsg_initialize();
miniport_t* miniport_create_unbound(
        int port_number);
miniport_t* miniport_create_bound(
        network_address_t addr,
        int remote_unbound_port_number);
void miniport_destroy(miniport_t* miniport);
int minimsg_send(miniport_t* local_unbound_port,
        miniport_t* local_bound_port,
        minimsg_t* msg, int len);
int minimsg_receive(miniport_t* local_unbound_port,
        miniport_t** new_local_bound_port,
        minimsg_t* msg, int *len);
```

Overview

The networking device should be treated as the IP layer of your system.

It transparently enables communication between other systems running minithreads.

- `network5.c`
- `network6.c`

Networking is interrupt-driven

- `network_initialize()` installs the handler.
- Should be initialized after `clock_initialize` and before interrupts.
- The prototype/behavior is similar to the clock interrupt.
- Each received packet triggers an interrupt.
- Interrupts are delivered on the current thread's stack.
- This should finish as soon as possible!

Header generation

- *Datagram = Header + Payload*
- Use the interface in `miniheader.h` to pack/unpack.
- Set the protocol field of the header to `PROTOCOL_MINIDATAGRAM`.
- Use `pack_address(...)` to pack source & destination addresses.
- Use `pack_unsigned_short(...)` to pack source & destination ports.

network_handler

```
typedef struct {
    // sender address
    network_address_t sender;
    // header+payload
    char buffer[MAX_NETWORK_PKT_SIZE];
    // size
    int size;
} network_interrupt_arg_t;
```

The header and the data are joined in the buffer; you must strip it off.

Striping the header

- Copy the header from the byte buffer into a `mini_header_t*`.
- Check the protocol field of the header.
- Unpack the source and destination addresses.
- Unpack the destination port.

Sending datagrams

- Call `network_send_pkt()` from `minimsg_send()` to send datagrams.

```
int network_send_pkt(
    network_address_t dest_address,
    int hdr_len, char* hdr,
    int data_len, char* data);
```

- Header contains information about the sender & receiver.
- Reject datagrams larger than `MINIMSG_MAX_MSG_SIZE`.

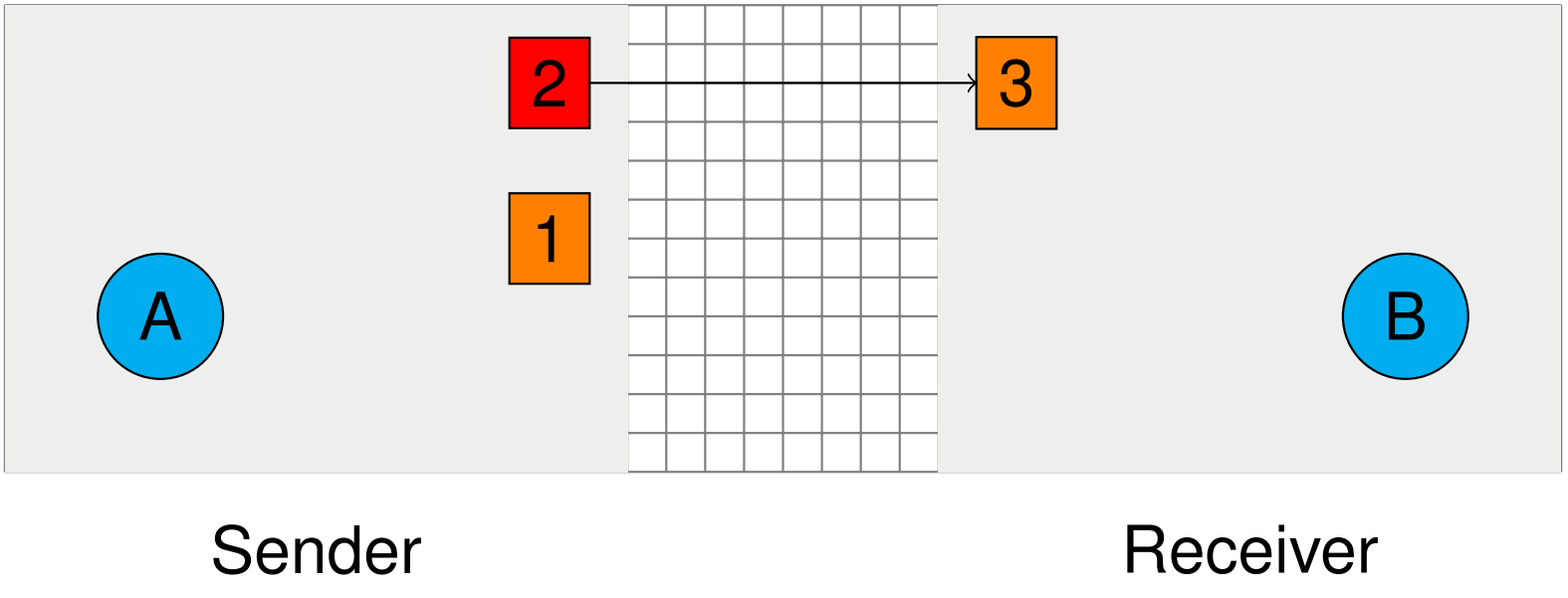
Overview

A miniport is a data structure that represents a one-way communication endpoint.

- **Unbound ports** are used for listening and can receive from any remote port – not *bound* because any (network_address, port) can send data to it.
- **Bound ports** are used for sending data – *bound* because sending data to this port results in a specific (network_address, port) receiving data.

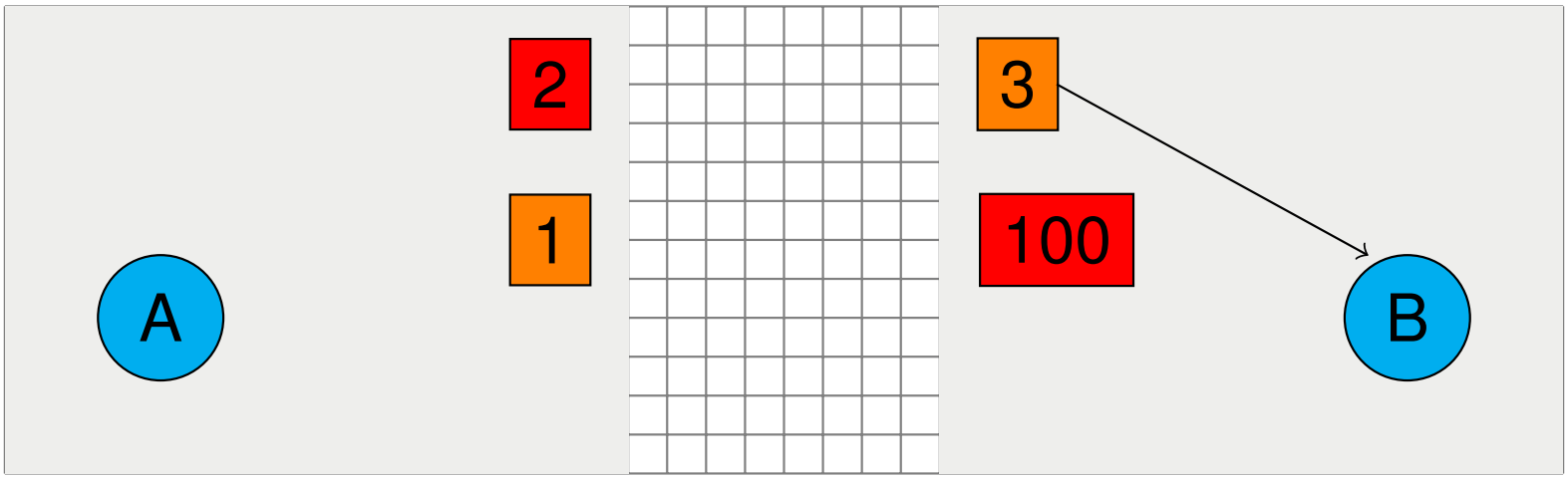
A sends from its port 2 to B's port 3

- Unbound (listening) Port 1 and 3
- Bound (used for sending) Port 2
- Threads: A, B



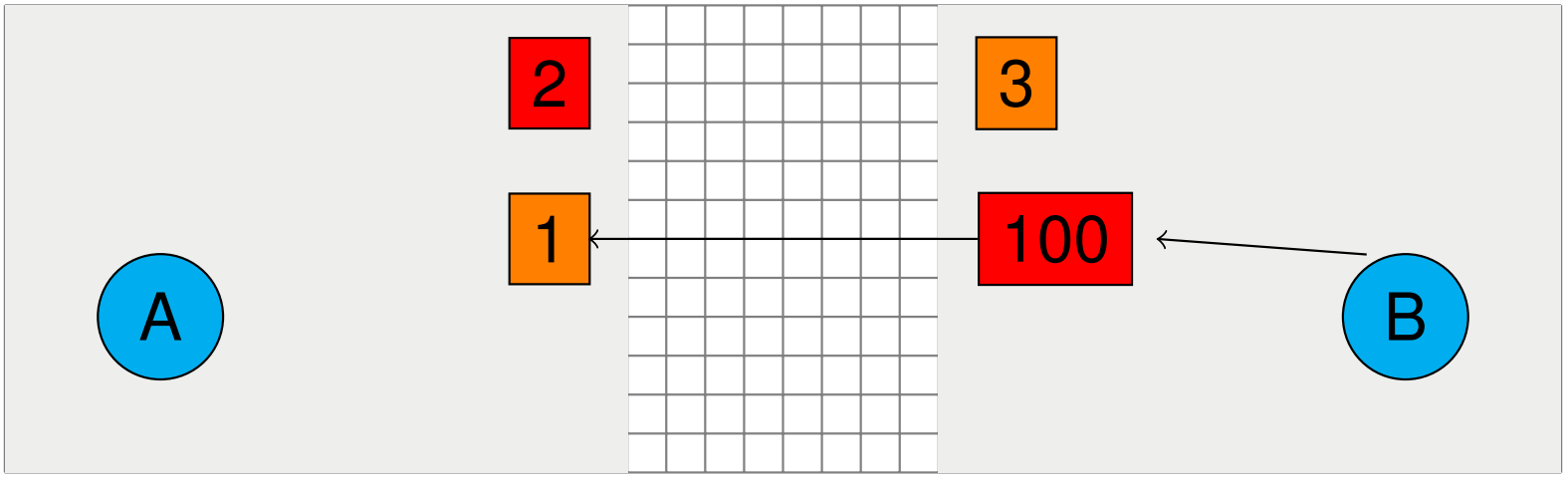
Minimsg layer creates bound port 100 and delivers the message

- A's message is delivered to B's unbound (listening) port 3.
- B is unblocked.
- The bound (used for sending) port 100 is created in order to allow B to respond.



B responds to A over the new bound port.

- B receives a reference to its bound (used for sending) port 100.
- B can send to 100.
- The message will be sent to A's unbound (used for listening) port 1.



What does the datastructure look like?

Conceptually it looks like this*:

```
struct miniport {
    char port_type;
    int port_number;

    queue_t *incoming_data;
    semaphore_t* datagrams_ready;

    network_address_t remote_addr;
    int remote_unbound_port;
}
```

*the next slide should be referenced when implementing

You should use unions

Unions store two overlapping datastructures[†].

```
union {
    struct {
        queue_t *incoming_data;
        semaphore_t* datagrams_ready;
    } unbound;
    struct {
        network_address_t remote_addr;
        int remote_unbound_port;
    } bound;
};
```

[†] You should use this to replace the last 4 variables from the struct on the previous page

Implementation specs

- `miniport_send` sends data to the “remote port”.
- Your computer can also talk to itself – remote port may refer to a local port!
- A miniport is identified by a 16-bit unsigned number (the actual datatype is bigger).
- Unbound miniports are in [0-32767] – can be chosen by the user.
- Bound miniports are in [32768-65535] – assign successive numbers automatically.

Minimsg Layer

- The sender assembles a header that identifies the end points of communication.
- The receiver strips the header to identify the destination, enqueues the packet, and wakes up any sleeping threads.

Minimsg Functions

- `minimsg_send` is non-blocking – i.e. doesn't wait for the send to succeed.
- Sends data using `network_send_pkt()`.
- `minimsg_receive` blocks until a message is received.
- Provides a bound port so a reply may be sent.

Grading

- Port operations must be $O(1)$.
- Do not waste resources.
- Make sure to not reassign ports that are in-use.
- The application destroys remote miniports.
- We will be grading you on your implementation and test cases.