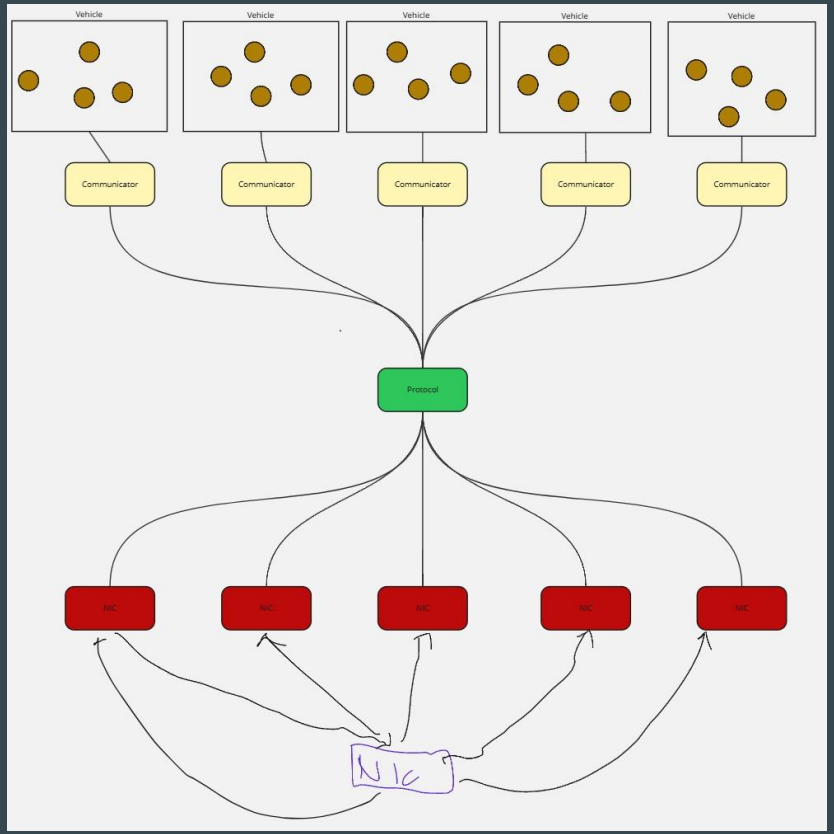
Sistemas Operacionais II - P1

João Vitor dos Santos
Pablo Lopes Teixeira
Fábio Henrique Antunes Coelho
Tiago Oliveira da Luz

Architecture



SIGIO - raw_socket_engine

```
struct ifreq ifr;
memset(&ifr, 0, sizeof(ifr));
strcpy(ifr.ifr_name, "wlp4s0"); // Default interface, can be configurable

if(ioctl(_socket, SIOCGIFINDEX, &ifr) < 0) {
    ConsoleLogger::error("SIOCGIFINDEX");
    close(_socket);
    throw std::runtime_error("Falha ao obter índice da interface");
}
_ifindex = ifr.ifr_ifindex;
```

NIC - constructor

```
NIC(const std::string& id) : _buffer_count(0) {
    ConsoleLogger::print("Starting NIC...");

    MacAddressGenerator::generate_mac_from_seed(id, _address);

    for (unsigned int i = 0; i < BUFFER_SIZE; i++) {
        _buffer[i] = new Buffer<Ethernet::Frame>(Ethernet::MTU);
    }

    // Register this NIC instance
    active_nics.push_back(this);

    // Set up SIGIO handling if this is the first NIC
    if (active_nics.size() == 1) {
        // Register the signal handler for SIGIO
        struct sigaction sa;
        memset(&sa, 0, sizeof(sa));
        sa.sa_flags = SA_RESTART;
        sa.sa_handler = &NIC::sigio_handler;
        if (sigaction(SIGIO, &sa, NULL) < 0) {
            ConsoleLogger::error("sigaction");
            exit(EXIT_FAILURE);
        }
        ConsoleLogger::print("NIC: SIGIO handler set");
    }

    int flags = fcntl(Engine::_socket, F_GETFL, 0);
    fcntl(Engine::_socket, F_SETFL, flags | O_ASYNC | O_NONBLOCK);
    fcntl(Engine::_socket, F_SETOWN, getpid());

    std::cout << "PROCESS ID: " << getpid() << std::endl;
}
```

NIC - send and receive

```
int send(NICBuffer * buf) {
    ConsoleLogger::print("NIC: Sending frame.");
    Ethernet::Frame* frame = buf->frame();
    int result = Engine::raw_send(
        frame->header()->h_dest,
        ntohs(frame->header()->h_proto),
        frame->data(),
        buf->size() - sizeof(Ethernet::Header)
    );

    ConsoleLogger::print("NIC: Frame sent.");
    return result;
}
```

```
int receive(NICBuffer * buf, Address * src, Address * dst, void * data, unsigned int size) {
    Ethernet::Frame* frame = buf->frame();
    memcpy(src, frame->header()->h_source, ETH_ALEN);
    memcpy(dst, frame->header()->h_dest, ETH_ALEN);

    unsigned int data_size = buf->size() - sizeof(Ethernet::Header);
    unsigned int copy_size = (data_size > size) ? size : data_size;

    memcpy(data, frame->data(), copy_size);

    return copy_size;
}
```

NIC - sigio_handler and process_incoming_data

```cpp
// SIGIO handler (static function shared by all NICs)
static void sigio_handler(int signum) {
    std::cout << "SIGIO RECEIVED: " << signum << std::endl;
    // Check all active NICs for data
    for (auto nic : active_nics) {
        nic->process_incoming_data();
    }
}
```

```cpp
void process_incoming_data() {
    // Keep reading while there's data available (non-blocking)
    while (true) {
        Address src;
        Protocol_Number prot;

        // Get a free buffer
        NICBuffer* buf = alloc(address(), 0, Ethernet::MTU - sizeof(Ethernet::Header));
        if (!buf) {
            // No buffers available, we'll have to try again later when buffer is freed
            break;
        }

        Ethernet::Frame* frame = buf->frame();
        int size = Engine::raw_receive(&src, &prot, frame->data(), Ethernet::MTU - sizeof(Ethernet::Header));

        if (size > 0) {
            // Successful read
            buf->size(size + sizeof(Ethernet::Header));
            notify(prot, buf);
        } else if (size == 0 || (size < 0 && errno == EAGAIN)) {
            // No more data available
            free(buf);
            break;
        } else {
            // Error
            free(buf);
            perror("Error reading from socket");
            break;
        }
    }
}
```

Protocol - send and receive

```
int send(Address from, Address to, const void * data, unsigned int size) {
    if (size > MTU) {
        return -1;
    }
    ConsoleLogger::print("Protocol: Sending message.");
    std::cout << "SIZES -> " << sizeof(Header) + size << " " << sizeof(Header) << " " << size << std::endl;

    NIC* nic = get_nic(from.paddr());
    std::cout << "MAC ADDRESS SEND BEFORE: " << mac_to_string(from.paddr()) << std::endl;
    NICBuffer* buf = nic->alloc(to.paddr(), PROTO, sizeof(Header) + size);
    if (!buf) {
        return -1;
    }

    ConsoleLogger::print("Protocol: Buffer allocated.");

    Packet* packet = reinterpret_cast<Packet*>(buf->frame()->data());
    packet->Header::operator=(Header(from, to, size));

    std::cout << "MAC ADDRESS SEND: " << mac_to_string(packet->from_paddr()) << std::endl;
    std::cout << "MAC ADDRESS SEND: " << mac_to_string(packet->to_paddr()) << std::endl;

    memcpy(packet->template data<void>(), data, size);

    int result = nic->send(buf);
    nic->free(buf);

    return result;
}
```

```
int receive(NICBuffer * buf, Address from, void * data, unsigned int size) {
    Packet* packet = reinterpret_cast<Packet*>(buf->frame()->data());

    if (packet->length() > size) {
        return -1;
    }

    NIC* nic = get_nic(from.paddr());

    Physical_Address paddr;
    nic->receive(buf, &paddr, nullptr, nullptr, 0);

    from = Address(paddr, packet->from_port());
    memcpy(data, packet->template data<void>(), packet->length());

    return packet->length();
}
```

Conditional_Data_Observer

```
template <typename T, typename Condition = void>
class Conditional_Data_Observer
{
public:
    typedef T Observed_Data;
    typedef Condition Observing_Condition;

    virtual void update(Condition c, T* d) {};

    void set_condition(Condition condition) {
        _condition = condition;
    }
    Condition rank() {
        return _condition;
    }
private:
    Condition _condition;
};
```

Conditionally_Data_Observed

```
template <typename T, typename Condition = void>
class Conditionally_Data_Observed
{
public:
    typedef T Observed_Data;
    typedef Condition Observing_Condition;
    typedef Ordered_List<Conditional_Data_Observer<T, Condition>, Condition> Observers;

    Conditionally_Data_Observed() {
        ConsoleLogger::print("Conditionally_Data_Observed: Initializing instance.");
    }
    ~Conditionally_Data_Observed() {}

    void attach(Conditional_Data_Observer<T, Condition>* o, Condition c) {
        std::cout << "Protocol condition set: " << c << std::endl;
        o->set_condition(c);
        _observers.insert(o);
    }

    void detach(Conditional_Data_Observer<T, Condition>* o, Condition c) {
        _observers.remove(o);
    }
```

```
    bool notify(Condition c, T* d) {
        ConsoleLogger::print("Conditionally_Data_Observed: Notifying observers.");
        bool notified = false;
        for(typename Observers::Iterator obs = _observers.begin(); obs != _observers.end(); ++obs) {
            std::cout << "PROTO: " << c << std::endl;
            if ((*obs)->rank() == c) {
                (*obs)->update(c, d);
                notified = true;
            }
        }
        return notified;
    }

private:
    Observers _observers;
};
```