Comunicação em Sistemas Autônomos Críticos

– Proposta de Projeto para INE5424 - Sistemas Operacionais II em 2025/1 –

Este documento apresenta a proposta de padrão de projeto para a disciplina de Sistemas Operacionais II no primeiro semestre de 2025. A proposta gira em torno do desenvolvimento de uma **biblioteca de comunicação confiável e segura para sistemas autônomos críticos,** ambientada no cenário de veículos autônomos.

Dentre os requisitos globais do projeto, temos:

- O desenvolvimento deve ser feito na linguagem de programação C++ utilizando apenas a C Standard Library (libc) e a C++ Standard Library em plataforma POSIX nativa[1].
- Cada sistema autônomo (e.g. veículo) deve ser modelado como um macro-objeto que tem associado a si um processo POSIX específico (e não uma VM ou SBC específica).
- Cada componente de cada um dos sistemas autônomos (e.g. sensor, fusor, modelo de ML) deve ser modelado como um macro-objeto que tem associado a si uma thread POSIX (e não um processo).
- A comunicação via rede se dará sempre por broadcast, com alcance limitado pelo domínio de colisão inerente da respectiva tecnologia (uma célula de rádio 5G ou uma rede local Ethernet).
- Os grupos serão constituídos por até 4 alunos e podem incluir alunos de ambas as turmas desde que os mesmos estejam presentes em todas as apresentações de andamento do projeto e também na apresentação final.
- Depois de desenvolver e testar cada etapa do projeto, os grupos submeterão, via Moodle, um link para um commit específico em seu próprio repositório Git, com todas as especificações e diagramas de projeto localizados em uma pasta chamada “doc”. Junto com o link, envie as credenciais de acesso, para que os avaliadores possam acessar seu código (não use repositório aberto; você pode usar https://codigos.ufsc.br/). Na raiz da sua árvore de desenvolvimento, inclua um Makefile que seja capaz de acionar a **compilação e execução de todos os testes simplesmente com o comando *make*.** Os slides usados nas apresentações de andamento de cada uma das etapas do projeto podem tanto estar na mesma pasta “doc” no Git ou em um documento online referenciado pela segunda URL da tarefa no Moodle. As apresentações devem conter uma avaliação simples de desempenho com latência média observada durante os testes.

1. Comunicação entre Sistemas e entre seus Componentes

A biblioteca de comunicação deve apresentar uma API unificada para todos os agentes, independente de serem sistemas autônomos ou componentes dos mesmos. As mensagens trocadas entre os agentes têm tamanho máximo conhecido e menor do que a MTU da rede, logo, nunca serão fragmentadas.

Propagação de Eventos Assíncronos

Os eventos assíncronos da pilha de protocolos de comunicação em desenvolvimento neste projeto devem ser tratados no prisma do padrão de projeto Observer X Observed, com as entidades observadas notificando seus observadores para que atualizem seus estados. Em particular, a recepção de mensagens deve ser sempre assíncrona, sem protocolos de rendezvous.

Engine para Comunicação entre sistemas autônomos

A classe **Engine** da especificação acima deve ser implementada usando **raw sockets** com frames Ethernet.

---

[1]Linguagens de programação com um nível de abstração da plataforma computacional não superior a este podem ser consideradas.

Mensagens

Nesta etapa, as mensagens são um simples array de bytes:

```
M = {.*}
```

API

Tanto os sistemas autônomos quanto seus componentes se comunicarão através do mesmo conjunto de primitivas básicas, respeitando a mesma API, cuja especificação inicial segue abaixo destacada em azul:

```
// Fundamentals for Observer X Observed

template <typename T, typename Condition = void>
class Conditional_Data_Observer;
template <typename T, typename Condition = void>
class Conditionally_Data_Observed;

// Conditional Observer x Conditionally Observed with Data decoupled by a Semaphore
template<typename D, typename C = void>
class Concurrent_Observer;

template<typename D, typename C = void>
class Concurrent_Observed
{
    friend class Concurrent_Observer<D, C>;
public:
    typedef D Observed_Data;
    typedef C Observing_Condition;
    typedef Ordered_List<Concurrent_Observer<D, C>, C> Observers;

public:
    Concurrent_Observed() {}
    ~Concurrent_Observed() {}

    void attach(Concurrent_Observer<D, C> * o, C c) {
        _observers.insert(o);
    }

    void detach(Concurrent_Observer<D, C> * o, C c) {
        _observers.remove(o);
    }

    bool notify(C c, D * d) {
        bool notified = false;
        for(Observers::Iterator obs = _observers.begin(); obs != _observers.end(); obs++) {
            if(obs->rank() == c) {
                obs->update(c, d);
                notified = true;
            }
        }
        return notified;
    }
private:
    Observers _observers;
};

template<typename D, typename C>
class Concurrent_Observer
{
    friend class Concurrent_Observed<D, C>;
public:
    typedef D Observed_Data;
    typedef C Observing_Condition;
```

```
public:
    Concurrent_Observer(): _semaphore(0) {}
    ~Concurrent_Observer() {}

    void update(C c, D * d) {
        _data.insert(d);
        _semaphore.v();
    }

    D * updated() {
        _semaphore.p();
        return _data.remove();
    }

private:
    Semaphore _semaphore;
    List<D> _data;
};
```

```
// Network

class Ethernet; // all necessary definitions and formats

template <typename Engine>
class NIC: public Ethernet, public Conditional_Data_Observed<Buffer<Ethernet::Frame>,
Ethernet::Protocol>, private Engine
{
public:
    static const unsigned int BUFFER_SIZE =
        Traits<NIC>::SEND_BUFFERS * sizeof(Buffer<Ethernet::Frame>) +
        Traits<NIC>::RECEIVE_BUFFERS * sizeof(Buffer<Ethernet::Frame>);
    typedef Ethernet::Address Address;
    typedef Ethernet::Protocol Protocol_Number;
    typedef Buffer<Ethernet::Frame> Buffer;
    typedef Conditional_Data_Observer<Buffer<Ethernet::Frame>, Ethernet::Protocol> Observer;
    typedef Conditionally_Data_Observed<Buffer<Ethernet::Frame>, Ethernet::Protocol> Observed;

protected:
    NIC();

public:
    ~NIC();

    int send(Address dst, Protocol_Number prot, const void * data, unsigned int size);
    int receive(Address * src, Protocol_Number * prot, void * data, unsigned int size);

    Buffer * alloc(Address dst, Protocol_Number prot, unsigned int size);
    int send(Buffer * buf);
    void free(Buffer * buf);
    int receive(Buffer * buf, Address * src, Address * dst, void * data, unsigned int size);

    const Address & address();
    void address(Address address);
    const Statistics & statistics();

    void attach(Observer * obs, Protocol_Number prot); // possibly inherited
    void detach(Observer * obs, Protocol_Number prot); // possibly inherited

private:
    Statistics _statistics;
    Buffer _buffer[BUFFER_SIZE];
};
```

```
// Communication Protocol

template <typename NIC>
class Protocol: private typename NIC::Observer
{
public:
    static const typename NIC::Protocol_Number PROTO =
        Traits<Protocol>::ETHERNET_PROTOCOL_NUMBER;
    typedef typename NIC::Buffer Buffer;
    typedef typename NIC::Address Physical_Address;
    typedef XXX Port;
    typedef Conditional_Data_Observer<Buffer<Ethernet::Frame>, Port> Observer;
    typedef Conditionally_Data_Observed<Buffer<Ethernet::Frame>, Port> Observed;

    class Address
    {
    public:
        enum Null;
    public:
        Address();
        Address(const Null & null);
        Address(Physical_Address paddr, Port port);
        operator bool() const { return (_paddr || _port); }
        bool operator==(Address a) { return (_paddr == a._paddr) && (_port == a._port); }
    private:
        Physical_Address _paddr;
        Port _port;
    };

    class Header;

    static const unsigned int MTU = NIC::MTU - sizeof(Header);
    typedef unsigned char Data[MTU];

    class Packet: public Header
    {
    public:
        Packet();

        Header * header();

        template<typename T>
        T * data() { return reinterpret_cast<T *>(&_data); }
    private:
        Data _data;
    } __attribute__((packed));

protected:
    Protocol(NIC * nic): _nic(nic) { _nic->attach(this, PROTO); }

public:
    ~Protocol() { _nic->detach(this, PROTO); }

    static int send(Address from, Address to, const void * data, unsigned int size);
        // Buffer * buf = NIC::alloc(to.paddr, PROTO, sizeof(Header) + size)
        // NIC::send(buf)
    static int receive(Buffer * buf, Address from, void * data, unsigned int size);
        // unsigned int s = NIC::receive(buf, &from.paddr, &to.paddr, data, size)
        // NIC::free(buf)
        // return s;

    static void attach(Observer * obs, Address address);
    static void detach(Observer * obs, Address address);
```

```
private:
    void update(typename NIC::Observed * obs, NIC::Protocol_Number prot, Buffer * buf) {
        if(!_observed.notify(buf)) // to call receive(...);
            _nic->free(buf);
    }

private:
    NIC * _nic;

    // Channel protocols are usually singletons
    static Observed _observed;
};
```

```
// Communication End-Point (for client classes)

class Message;

template <typename Channel>
class Communicator: public Concurrent_Observer<typename Channel::Observer::Observed_Data,
    typename Channel::Observer::Observing_Condition>
{
    typedef Concurrent_Observer<typename Channel::Observer::Observed_Data,
        typename Channel::Observer::Observing_Condition> Observer;

public:
    typedef typename Channel::Buffer Buffer;
    typedef typename Channel::Address Address;

public:
    Communicator(Channel * channel, Address address): _channel(channel), _address(address) {
        _channel->attach(this, address);
    }
    ~Communicator_Common() { Channel::detach(this, _address); }

    bool send(const Message * message) {
        return (_channel->send(_address, Channel::Address::BROADCAST, message->data(),
            message->size()) > 0);
    }

    bool receive(Message * message) {
        Buffer * buf = Observer::updated(); // block until a notification is triggered
        Channel::Address from;
        int size = _channel->receive(buf, &from, message->data(), message->size());
        // . . .
        if(size > 0)
           return true;
    }

private:
    void update(typename Channel::Observed * obs, typename
        Channel::Observer::Observing_Condition c, Buffer * buf) {
        Observer::update(c, buf); // releases the thread waiting for data
     }

private:
   Channel * _channel;
   Address _address;
};
```

2. Comunicação entre Componentes de um mesmo Sistema Autônomo

Nesta etapa, a API implementada na etapa anterior para a comunicação entre sistemas autônomos deve ser estendida para suportar a comunicação entre os componentes de um mesmo sistema. Toda a diferenciação deve ser confinada à classe **Engine** referenciada na especificação da API, sendo que esta segunda Engine deve ser implementada usando **memória compartilhada** em um modelo Produtor X Consumidor sincronizado com as primitivas pertinentes de **pthreads**. Além disso, a plena identificação dos atores da comunicação deve ser agora modelada e implementada.

Identificadores

A plena identificação dos atores da comunicação é fundamental para o correto funcionamento da biblioteca sendo desenvolvida e pode ser feita de muitas formas diferentes. Entretanto, há alguns elementos que merecem observação:

- Os endereços na classe NIC (i.e. NIC::Address) são endereços Ethernet (a.k.a. MAC Address), mas é importante lembrar que uma mesma máquina, com mais de uma placa de rede, terá mais de um desses endereços. Além disso, é importante ter-se em mente que os requisitos do projeto exigem que o endereço destino seja o de broadcast.
- Os endereços da classe Protocol (i.e. Protocol::Address) permitem a multiplexação das NICs para suportar múltiplas sessões do protocolo e devem ser únicos neste sentido. Utilizar-se diretamente NIC::Address como Protocol::Physical_Address e os PIDs Protocol::Port é tentador, pois a combinação gera números globalmente únicos e tem um regra de formação clara e de baixíssimo custo computacional. Entretanto, como este esquema seria utilizado para identificar os componentes de um mesmo sistema que se comunicam via memória compartilhada?
- A classe Communicator não usa endereçamento explícito, relegando qualquer forma de identificação de origem e destino das mensagens às próprias mensagens ou às outras entidades na pilha de comunicação. Todavia, como será visto na etapa 3, o recebedor de uma mensagem deve conseguir identificar as entidades necessárias para respondê-la.

Mensagens

Nesta etapa, as mensagens, além do array de bytes, agora denominado payload, devem incluir, ao menos, informações sobre a origem da mesma, de forma que possa ser respondida:

M = {origin, payload}

3. Time-Triggered Publish-Subscribe Messages

Os agentes da comunicação, sejam componentes de um sistema autônomo, sejam sistemas autônomos se comunicando entre si, interagem através de mensagens de **Interesse** e de **Resposta** em um modelo similar ao publish-subscribe, mas sem publicações explícitas. Quando um agente quer um dado, ele envia uma mensagem de Interesse (todas as mensagens são enviadas em broadcast) que designa inequivocamente o tipo do dado no qual ele tem interesse. Os agentes que recebem a mensagem de Interesse e que são capazes de produzir dados daquele tipo, enviam, periodicamente, mensagens de Resposta. Desta forma, cada mensagem deve carregar um código capaz de identificar sua natureza. Isto pode ser alcançado através dos códigos dos Transducer Electronic Data Sheet (TEDS) da norma IEEE 1451. Além disso, cada mensagem de Interesse deve especificar o período no qual os agentes devem enviar respostas e cada agente pertinente deve gerenciar o intervalo de forma correspondente. A comunicação orientada por eventos não está prevista para este projeto. Nesta etapa, a sincronização temporal das máquinas envolvidas na comunicação pode ser presumida.

As mensagens de interesse, portanto, devem ter a seguinte estrutura:

```
I = {origin, type, period} // period in us from now
```

Enquanto as de resposta:

```
R = {origin, type, value}
```

4. Sincronização Temporal

As mensagens (class Message na seção 1) devem ser etiquetadas com um timestamp na origem. Os sistemas autônomos devem sincronizar seus relógios com uma precisão que permita a inequívoca identificação das mensagens através do endereço de origem em combinação com os timestamps (um mesmo agente não pode enviar duas mensagens ao mesmo tempo na percepção de toda a rede).

Os componentes de um mesmo sistema autônomo compartilham a percepção do tempo e concordam com exatidão sobre o tempo transcorrido desde um etiquetamento local (i.e. todas as threads têm acesso ao tempo do etiquetador).

Já a sincronização dos relógios dos sistemas autônomos pode ser feita através de uma implementação simplificada do Precision Time Protocol (IEEE 1588). A escolha de qual dos agentes proverá a base para tal sincronização pode ser, neste momento, arbitrária e fixa.

Mensagens

Nesta etapa, as mensagens seguem o seguinte formato:

```
I = {origin, timestamp, type, period}

R = {origin, timestamp, type, value}

PTP = {origin, timestamp, type}
```

5. Comunicação Segura em Grupos

Cada mensagem, tanto de interesse quanto de resposta, devem agora ser enriquecidas com um *Message Authentication Code* (MAC) de forma que sua autenticidade e integridade possam ser verificadas pelo agente parceiro da comunicação. **A criptografia do algoritmo que gerará o MAC não é relevante para este projeto** e, em último caso, pode ser um simples XOR. Mensagens que falham a verificação do MAC são simplesmente descartadas.

Grupos de sistemas autônomos são formados dinamicamente para efeitos de comunicação no contexto deste projeto, por exemplo, considerando-se a localização geográfica. Cada grupo elege um líder que gera e distribui ao grupo a chave usada para gerar os MACs durante uma sessão[2]. O critério de formação de grupo deve ser especificado por cada projeto. Agentes podem se juntar ao grupo sempre que cumprirem o critério. Podem também deixar o grupo a qualquer momento, por exemplo, em função de falhas. O ingresso deve ser explicitado através de uma **mensagem de interesse de ingresso** (i.e. com um type específico para este propósito). A saída de agentes dos grupos não precisa ser explicitada através de mensagens. O líder eleito de cada grupo passa também a ser a base para a sincronização temporal definida na seção 4.

Mensagens

Nesta etapa, as mensagens seguem o seguinte formato:

```
I = {origin, timestamp, type, period, MAC}

R = {origin, timestamp, type, value, MAC}

PTP = {origin, timestamp, type}

J = {origin, timestamp, type}
```

[2] Interessados em segurança cibernética podem fazer esta etapa com SSL, fora do protocolo, e podem também ajustar a duração das sessões para aumentar a segurança.

6. Comunicação Local Otimizada

Agentes internos de um sistema autônomo, ou seja, seus componentes, não devem pagar o preço da sincronização temporal e nem da comunicação segura, uma vez que, estando no mesmo domínio de proteção da arquitetura do computador (i.e. mesmo *address space*), isto não seria efetivo. Suas mensagens devem ser otimizadas nesta etapa para manter a mesma semântica das mensagens trocadas entre sistemas autônomos, entretanto, com o menor custo computacional possível. Entretanto, caso uma função seja invocada para retornar o timestamp ou o MAC de uma mensagem local, estes devem ser fornecidos de forma consistente (e.g. calculados na hora da invocação, somente quando isto ocorrer).

7. Mobilidade e Simulação Realista

Agentes móveis podem deixar um grupo e passar a integrar outro em função da própria mobilidade. Quando ingressam em um novo grupo, precisam inicialmente autenticar-se para receber a chave de sessão definida na seção 5. Todavia, isto não é suficiente para que a comunicação flua, pois os agentes do novo grupo não receberam suas mensagens de interesse. A repetição de tais interesses ao novo grupo deve ser automatizada nesta etapa. Nesta etapa, uma simulação automotiva realista será disponibilizada para que os grupos integrem seus projetos e tenham assim uma validação realista.

Revisão das Avaliações

Juntamente com a etapa 7 do projeto, a consolidação das correções propostas nas etapas anteriores pode ser feita com o potencial de ajustar em até dois pontos a nota de cada etapa anteriormente avaliada.