Interrupções no ~~Nanvix~~ RIOT-OS

Ad Nunes Ribeiro (11200620), Eduardo Dias Defreyn (11203780),
Ricardo do Nascimento Boing (14200760)
Bacharelado em Ciência da Computação -- Universidade Federal de Santa Catarina (UFSC)
Campus Reitor João David Ferreira Lima, 88.040-900 -- Florianópolis -- SC -- Brasil

adnunesribeiro@gmail.com; eduardo_dududex@hotmail.com; ricardoboing.ufsc@gmail.com

1. Interrupt Handling

A manipulação de interrupções no RIOT OS é muito direta e mostrada na Figura 1. do Ponto de vista do sistema, após a ocorrência de uma interrupção, o microcontrolador salta para o kernel, que salva o contexto e chama imediatamente o ISR no contexto da interrupção. Portanto, o ISR fica com as operações normais de retorno para ISRs na plataforma de destino. O retorno volta ao kernel novamente onde é tomada uma decisão se uma troca de tarefas tiver que ser feita ou não. Essa decisão é tomada com um mínimo de recursos de agendamento e nenhuma chamada completa do scheduler é necessária. Após restaurar o contexto agendado, o kernel retorna imediatamente para o contexto restaurado.

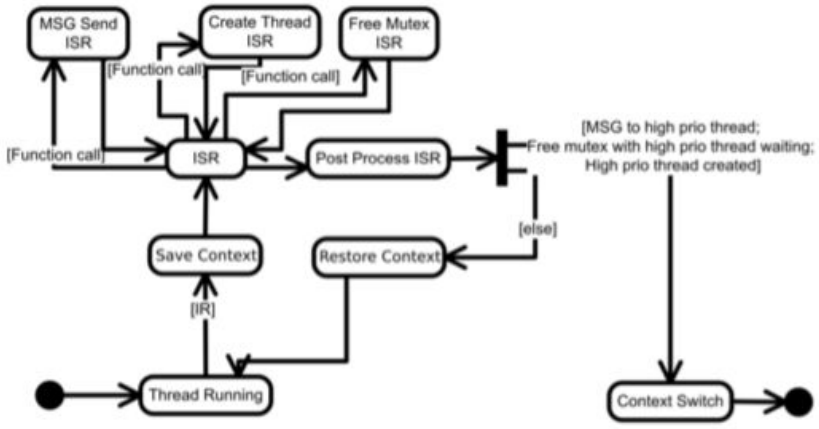


Figura 1. Visão parcial do tratamento de interrupções

1.1. Hardware Interrupt Handling na CPU Native

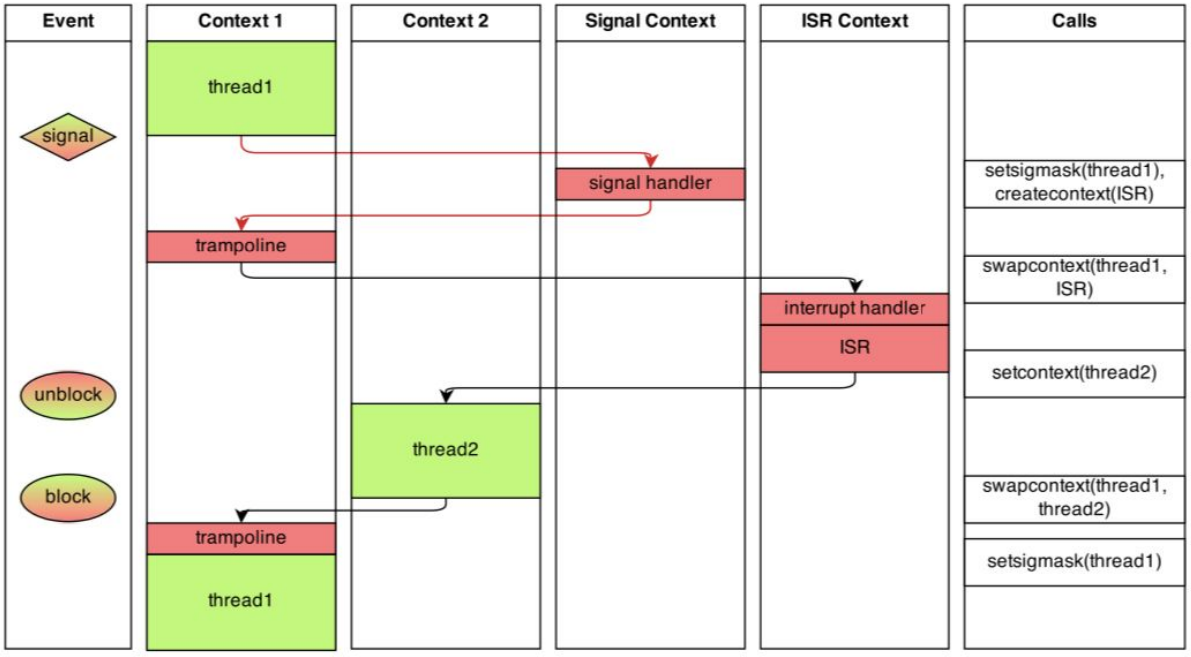


Figura 2. Tratamento de interrupção do native

Sempre que um sinal ocorre, o Riot OS executa uma alternância de contexto para um handler de sinal (transição marcada em vermelho na figura 2). O handler de sinal salva o sinal gravando-o em uma estrutura de dados FIFO (First In First Out). Em seguida, garante que é seguro alternar contextos, ou seja, nenhuma chamada do sistema está sendo executada no momento. Isso é necessário porque não é possível retornar a uma chamada de sistema interrompida do espaço do usuário.

```
/**
 * save signal, return to _native_sig_leave_tramp if possible
 */
void native_isr_entry(int sig, siginfo_t *info, void *context)
{
    (void) info; /* unused at the moment */
    //printf("\n\033[33m\n\t\tnative_isr_entry(%i)\n\n\033[0m", sig);

    /* save the signal */
    if (real_write(_sig_pipefd[1], &sig, sizeof(int)) == -1) {
        err(EXIT_FAILURE, "native_isr_entry: real_write()");
    }
    _native_sigpend++;
    //real_write(STDOUT_FILENO, "sigpend\n", 8);
```

Figura 3. Salvamento do signal na FIFO

```
native_isr_context.uc_stack.ss_sp = __isr_stack;
native_isr_context.uc_stack.ss_size = sizeof(__isr_stack);
native_isr_context.uc_stack.ss_flags = 0;
makecontext(&native_isr_context, native_irq_handler, 0);
_native_cur_ctx = (ucontext_t *)sched_active_thread->sp;

DEBUG("\n\n\t\tnative_isr_entry: return to _native_sig_leave_tramp\n\n");
/* disable interrupts in context */
isr_set_sigmask((ucontext_t *)context);
_native_in_isr = 1;
```

Figura 4. Preparação dos contextos

Após o salvamento do contexto da thread e a criação do contexto para o ISR (figura 4), o handler instala uma função de trampolim a ser chamada quando o handler de sinais retornar a execução ao contexto executado quando o sinal ocorreu. Para evitar outras interrupções durante a transição crítica, ele modifica a máscara de sinal do contexto para bloquear todos os sinais.

```
_native_saved_eip = ((ucontext_t *)context)->uc_mcontext.gregs[REG_EIP];
((ucontext_t *)context)->uc_mcontext.gregs[REG_EIP] = (unsigned int)&_native_sig_leave_tramp;
```

Figura 5. Criação da Função Trampolim

Ao retornar a esse contexto, o código do trampolim salva o contexto na pilha do encadeamento atual e alterna para um contexto de handler de sinal. A rotina de serviço de interrupção (figura 6) que lida com a interrupção do RIOT é então chamada nesse contexto, dependendo do sinal recebido.

```
.globl _native_sig_leave_tramp

_native_sig_leave_tramp:
    pushl _native_saved_eip
    pushfl
    pushal

    pushl _native_isr_ctx
    pushl _native_cur_ctx
    call swapcontext
    addl $8, %esp

    call irq_enable

    movl $0x0, _native_in_isr
    popal
    popfl

    ret
```

Figura 6. Função Trampolim

```
/**
 * call signal handlers,
 * restore user context
 */
void native_irq_handler(void)
{
    DEBUG("\n\n\t\tnative_irq_handler\n\n");

    while (_native_sigpend > 0) {
        int sig = _native_popsig();
        _native_sigpend--;

        if (native_irq_handlers[sig] != NULL) {
            DEBUG("native_irq_handler: calling interrupt handler for %i\n", sig);
            native_irq_handlers[sig]();
        }
        else if (sig == SIGUSR1) {
            warnx("native_irq_handler: ignoring SIGUSR1");
        }
        else {
            errx(EXIT_FAILURE, "XXX: no handler for signal %i\nXXX: this should not have happened!\n", sig);
        }
    }

    DEBUG("native_irq_handler: return\n");
    cpu_switch_context_exit();
}
```

Figura 7. Native Interrupt Handler

Da perspectiva do RIOT, a execução do handler de sinal é transparente e o ISR é executado na mesma ordem, assim como todas as outras plataformas. A Figura 8 ilustra como

o conteúdo da pilha de um contexto interrompido é alterado durante o tratamento de interrupções.

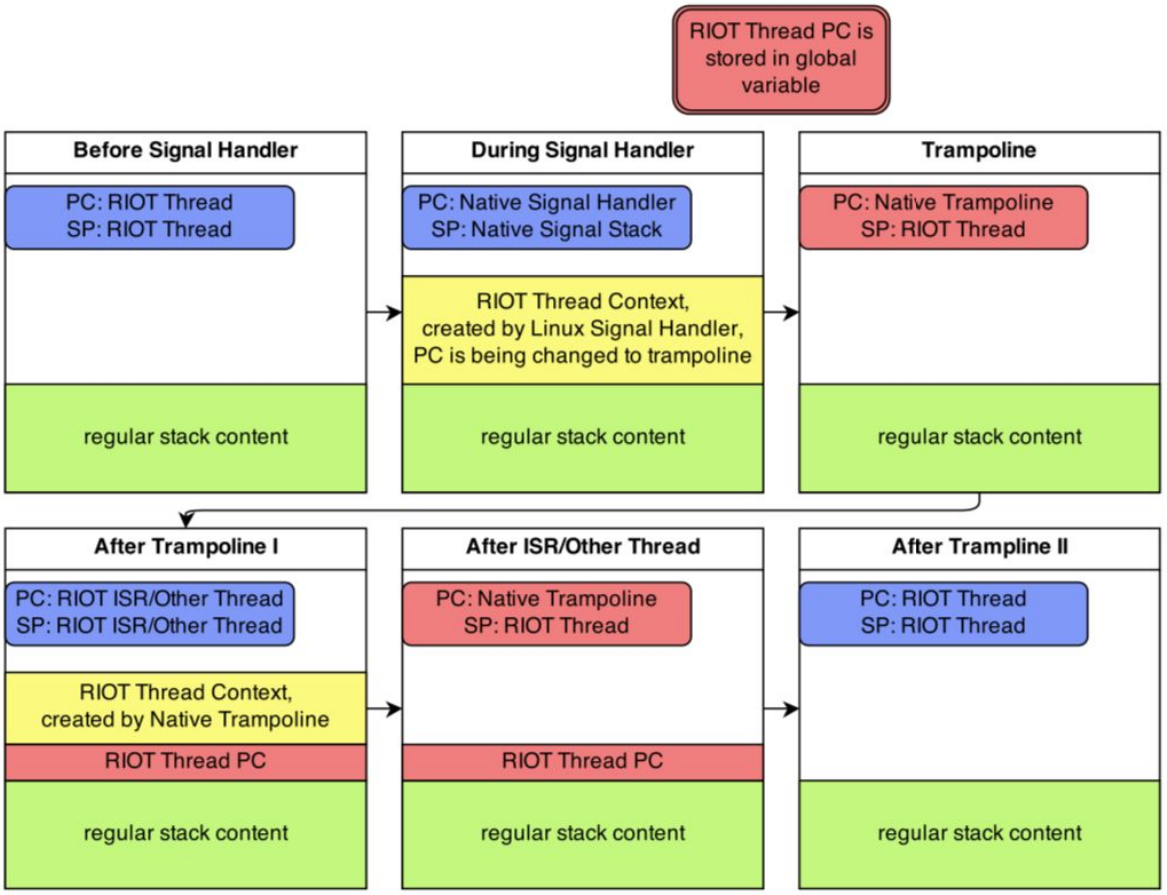


Figura 8. Estados da pilha de Threads do RIOT OS

Primeiro, o manipulador de sinal é executado regularmente, mas altera o contador do programa de retorno. Em seguida, o contador e o contexto originais do programa são armazenados na pilha. Ao reinserir o encadeamento, o contexto original é restaurado e o contador do programa é pulado para o espaço da thread.

1.3- IRQ Handling

O kernel do RIOT OS fornece uma API para controle do processamento de interrupções (IRQ). Esta interface é composta pelos seguintes métodos:

- irq_disable(void): define (seta para 1) o bit de desativação do IRQ no registro de status. Retorna o Valor anterior do registrador de status, não podendo utilizar este como variável booleana. O valor atual é relevante apenas para a função irq_restore().

```c
/**
 * block signals
 */
unsigned irq_disable(void)
{
    unsigned int prev_state;

    _native_syscall_enter();
    DEBUG("irq_disable()\n");

    if (_native_in_isr == 1) {
        DEBUG("irq_disable + _native_in_isr\n");
    }

    if (sigprocmask(SIG_SETMASK, &_native_sig_set_dint, NULL) == -1) {
        err(EXIT_FAILURE, "irq_disable: sigprocmask");
    }

    prev_state = native_interrupts_enabled;
    native_interrupts_enabled = 0;

    DEBUG("irq_disable(): return\n");
    _native_syscall_leave();

    return prev_state;
}
```

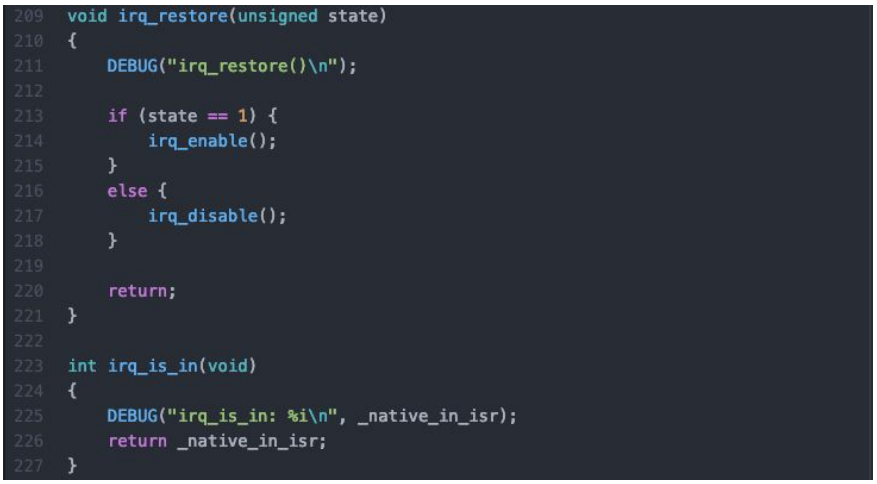
Figura 9.

- irq_enable(void): limpa o bit de desativação do IRQ no registro de status. Retorna o valor anterior do registrador de status. O valor de retorno não deve ser interpretado como um valor booleano. O valor atual é significativo apenas para irq_restore ().

```c
/**
 * unblock signals
 */
unsigned irq_enable(void)
{
    unsigned int prev_state;

    if (_native_in_isr == 1) {
#ifdef DEVELHELP
        real_write(STDERR_FILENO, "irq_enable + _native_in_isr\n", 27);
#else
        DEBUG("irq_enable + _native_in_isr\n");
#endif
    }

    _native_syscall_enter();
    DEBUG("irq_enable()\n");

    /* Mark the IRQ as enabled first since sigprocmask could call the handler
     * before returning to userspace.
     */

    prev_state = native_interrupts_enabled;
    native_interrupts_enabled = 1;

    if (sigprocmask(SIG_SETMASK, &_native_sig_set, NULL) == -1) {
        err(EXIT_FAILURE, "irq_enable: sigprocmask");
    }

    _native_syscall_leave();

    DEBUG("irq_enable(): return\n");

    return prev_state;
}
```

Figura 10.

- irq_is_in(void): verifica se foi chamado dentro de uma rotina de serviço de interrupção. Retorna um valor booleano.
- irq_restore((unsigned state) : restaura o bit de desativação do IRQ no registrador de status para o valor contido no estado passado por parâmetro.



```
209 void irq_restore(unsigned state)
210 {
211     DEBUG("irq_restore()\n");
212 
213     if (state == 1) {
214         irq_enable();
215     }
216     else {
217         irq_disable();
218     }
219 
220     return;
221 }
222 
223 int irq_is_in(void)
224 {
225     DEBUG("irq_is_in: %i\n", _native_in_isr);
226     return _native_in_isr;
227 }
```

Figura 11.

2. Pilha Genérica de Redes (GNRC - Generic Network Stack)

GNRC é a pilha de redes padrão no RIOT. Na Fig. 12 é possível ver a representação dessa pilha, sendo que a primeira camada se comunica com as aplicações dos usuários, através da API Socket, e a última com os drivers de dispositivos, através da API netdev. Já a comunicação entre os protocolos fica por responsabilidade da API netapi. Por fim, existe também a possibilidade da utilização de uma pilha de redes desenvolvida por terceiros, a qual deverá utilizar a API netapi para se comunicar com a API Socket.

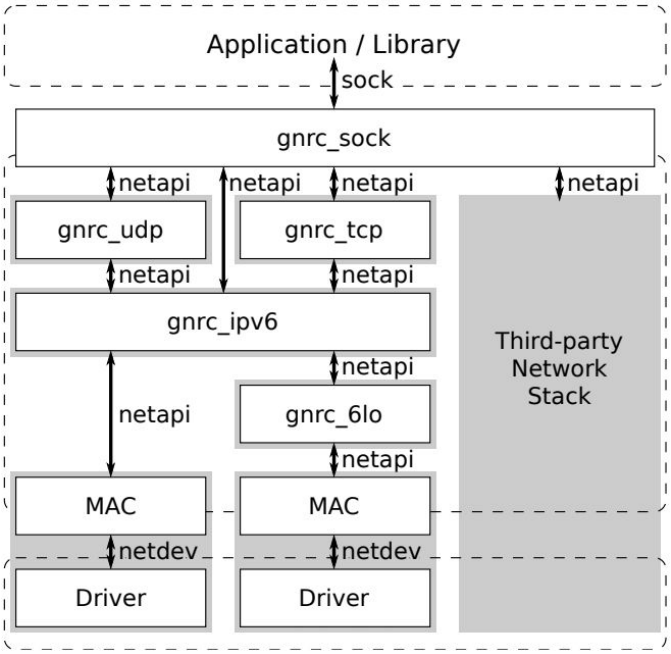


Figura 12. Ilustração da interação entre os protocolos da GNRC.

2.1. Packet Buffer, netapi, netreg e sock

Dentro da GNRC existem algumas ferramentas para o tratamento e repasse de pacotes de rede. O **Packet Buffer** serve como uma abstração de um pacote de rede, e fornece algumas funções para auxiliar na manipulação dos pacotes. Já a **netapi** é uma interface genérica que se destina a intermediar a comunicação entre os protocolos da GNRC, os quais, cada um é executado em uma Thread própria. O **netreg** serve para gerenciar as Threads de cada protocolo, de modo a armazenar o pid da Thread e o tipo de protocolo a qual é destinada. Por fim, a **API sock** é utilizada pelos usuários para enviar e receber pacotes de rede. Um sock pode ser de quatro tipos:

- sock_ip_t: protocolo IP;
- sock_tcp_t: protocolo TCP;
- sock_udp_t: protocolo UDP;
- sock_dtls_t: protocolo DTLS.

2.2. netdev

Além das ferramentas mencionadas anteriormente, a GNRC possui uma interface genérica para realizar a comunicação com os drivers de dispositivos: **netdev**. Cada dispositivo deve possuir sua própria implementação do netdev, sendo que a API é independente da pilha de redes utilizada (não é obrigatório o uso da GNRC). Na Fig. 13 é apresentada uma ilustração que representa a comunicação entre os drivers de dispositivos e a pilha de redes.

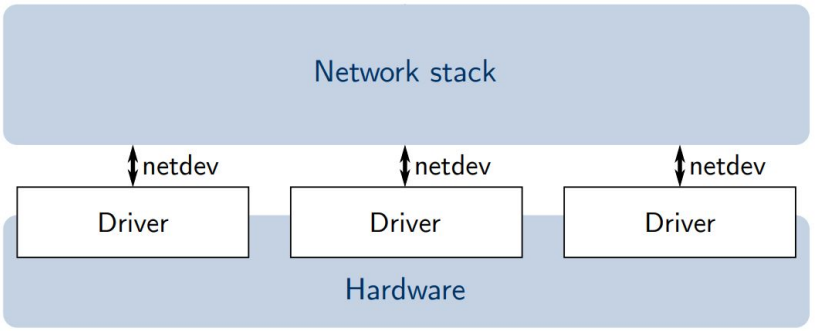


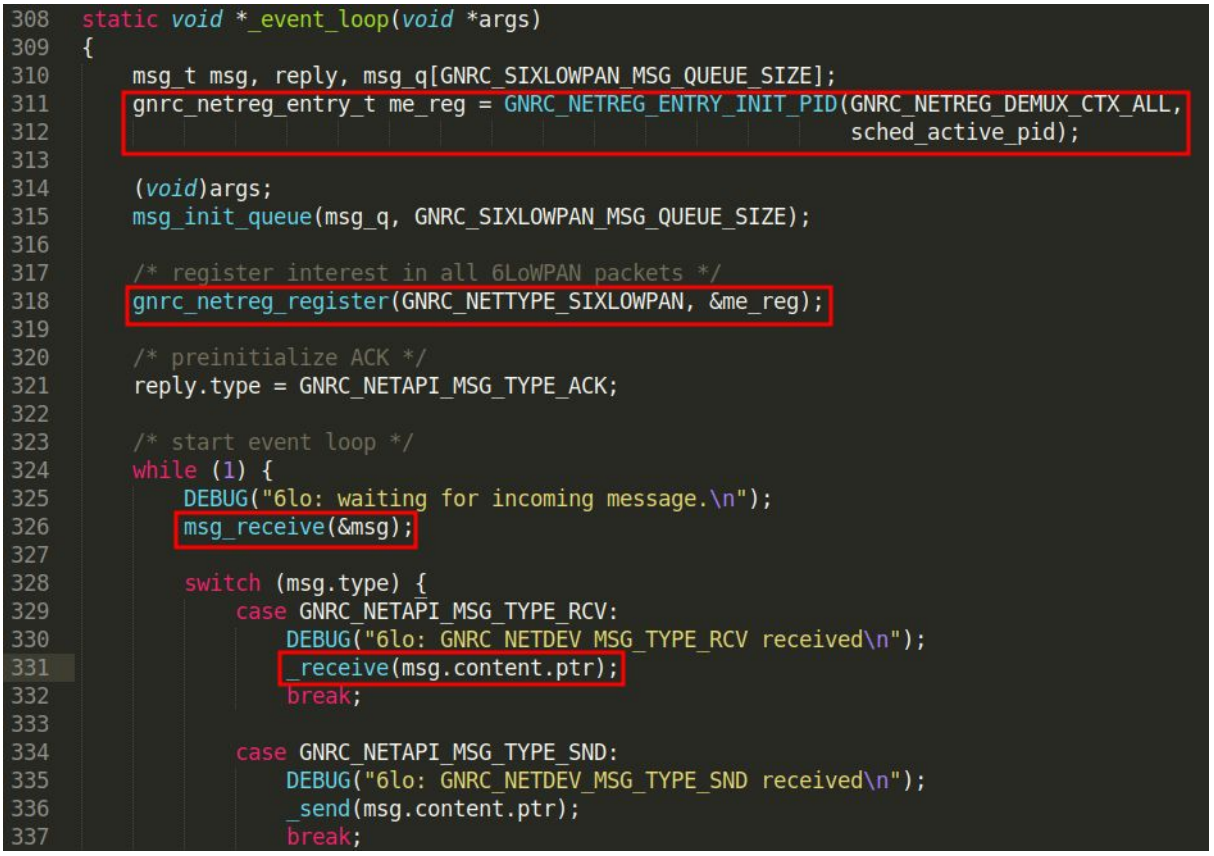
Figura 13. Interação entre os drivers, netdev e a pilha de redes (GNRC ou outra).

2.3. Propagação de um pacote de rede

2.3.1. Recebendo pacotes

As Threads de cada protocolo (6LoWPAN, IPv6, UDP e TCP) devem ser inicializadas para executarem a função *_evento_loop()*, implementada por cada um dos protocolos. Na Fig. 14 é mostrada a parte principal da implementação do *_event_loop()*, para o protocolo 6LoWPAN. Uma descrição dos passos é apresentada abaixo:

1. Registra a Thread na API netreg (linha 311);
2. Registra a intenção da Thread para que receba mensagens destinadas ao protocolo 6LoWPAN (linha 318);
3. Bloqueia a Thread do protocolo até que receba uma mensagem (linha 326);



```
308 static void *_event_loop(void *args)
309 {
310     msg_t msg, reply, msg_q[GNRC_SIXLOWPAN_MSG_QUEUE_SIZE];
311     gnrc_netreg_entry_t me_reg = GNRC_NETREG_ENTRY_INIT_PID(GNRC_NETREG_DEMUX_CTX_ALL,
312                                                             sched_active_pid);
313
314     (void)args;
315     msg_init_queue(msg_q, GNRC_SIXLOWPAN_MSG_QUEUE_SIZE);
316
317     /* register interest in all 6LoWPAN packets */
318     gnrc_netreg_register(GNRC_NETTYPE_SIXLOWPAN, &me_reg);
319
320     /* preinitialize ACK */
321     reply.type = GNRC_NETAPI_MSG_TYPE_ACK;
322
323     /* start event loop */
324     while (1) {
325         DEBUG("6lo: waiting for incoming message.\n");
326         msg_receive(&msg);
327
328         switch (msg.type) {
329             case GNRC_NETAPI_MSG_TYPE_RCV:
330                 DEBUG("6lo: GNRC_NETDEV_MSG_TYPE_RCV received\n");
331                 _receive(msg.content.ptr);
332                 break;
333
334             case GNRC_NETAPI_MSG_TYPE_SND:
335                 DEBUG("6lo: GNRC_NETDEV_MSG_TYPE_SND received\n");
336                 _send(msg.content.ptr);
337                 break;
```

Figura 14. Função definida em sys/net/gnrc/network_layer/sixlowpan/gnrc_sixlowpan.c.

Para que a Thread acorde é necessário que haja o acionamento de um evento através de uma interrupção. Os seguintes passos devem ocorrer, os quais são ilustrados na Fig. 15, para que se inicie o processo de propagação de um pacote até a aplicação dos usuários:

1. A função *netdev::event_callback()* é invocada pela rotina de interrupção do driver (ISR - Interrupt Service Routine), com o argumento event = NETDEV_EVENT_ISR;
2. Na sequência, *netdev::event_callback()* notifica a Thread que espera pelo dispositivo realizando a chamada da função *netdev_driver::isr()*. Desse modo, deixa-se o contexto da Rotina de Interrupção e entra-se no contexto de Threads;
3. *netdev_driver::isr()* fará uma nova chamada da função *netdev::event_callback()*, agora com event = NETDEV_EVENT_RX_COMPLETE, que fará uma chamada da função *netdev_driver::recv()* para que se inicie o processo de propagação do pacote na pilha de redes.

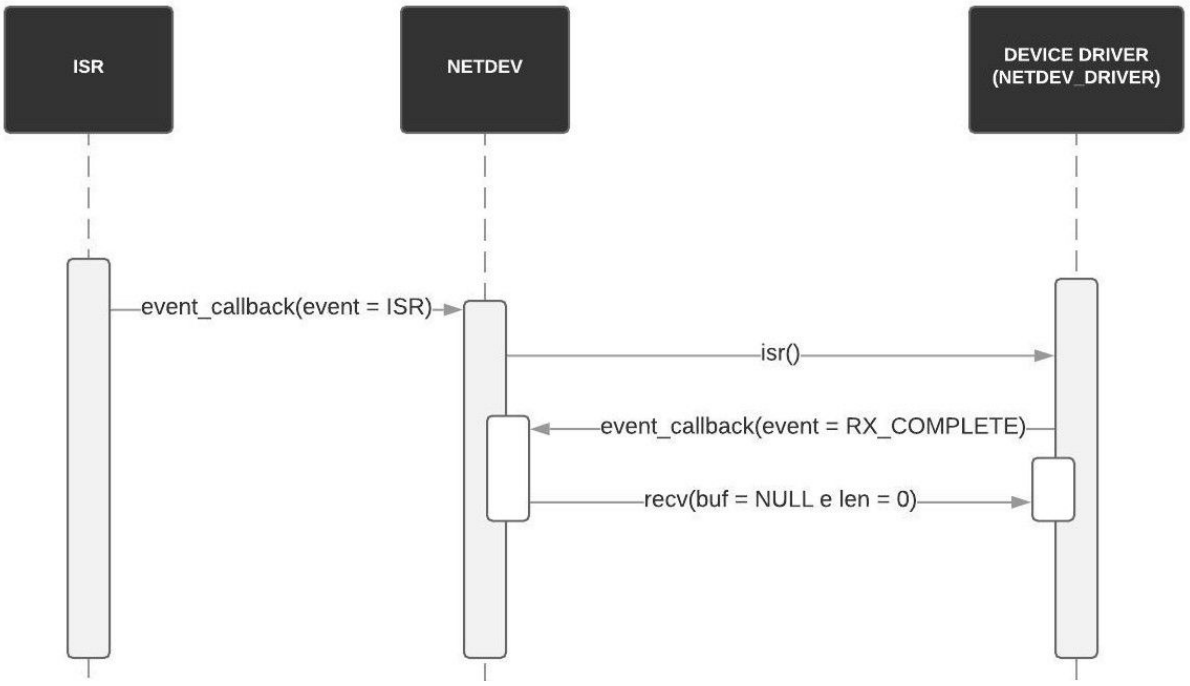


Figura 15. Repasse de um pacote da rotina de interrupção (ISR) para a pilha de redes.

A chamada da função *netdev_driver::recv()*, pela primeira vez, será realizada para obtenção do tamanho do pacote a ser recebido. Em seguida, aloca-se um buffer onde será armazenada a mensagem para que ela possa ser repassada. A seguir são descritos os passos necessários, que são ilustrados na Fig. 16:

4. Após *netdev_driver::recv()* ser invocado pela função *netdev::event_callback(),* com argumentos buf = NULL e len = 0, a função retorna o tamanho do buffer que deverá ser alocado;
5. Aloca-se um buffer que suporte esses dados, de acordo com o tamanho obtido no passo 1;
6. Obtém-se os dados através de uma segunda chamada da função *netdev_driver::recv()*, passando como parâmetros o ponteiro para o buffer onde os dados serão alocados e o tamanho do buffer;
7. Envia o buffer para o protocolo de destino através da função *gnrc_netapi_send_recv()*, que transformará o pacote em um formato para troca de mensagens entre Threads (linhas 55 e 56 da Fig. 17). A mensagem é enviada através da função *msg_try_send()* (linha 58 da Fig. 17).

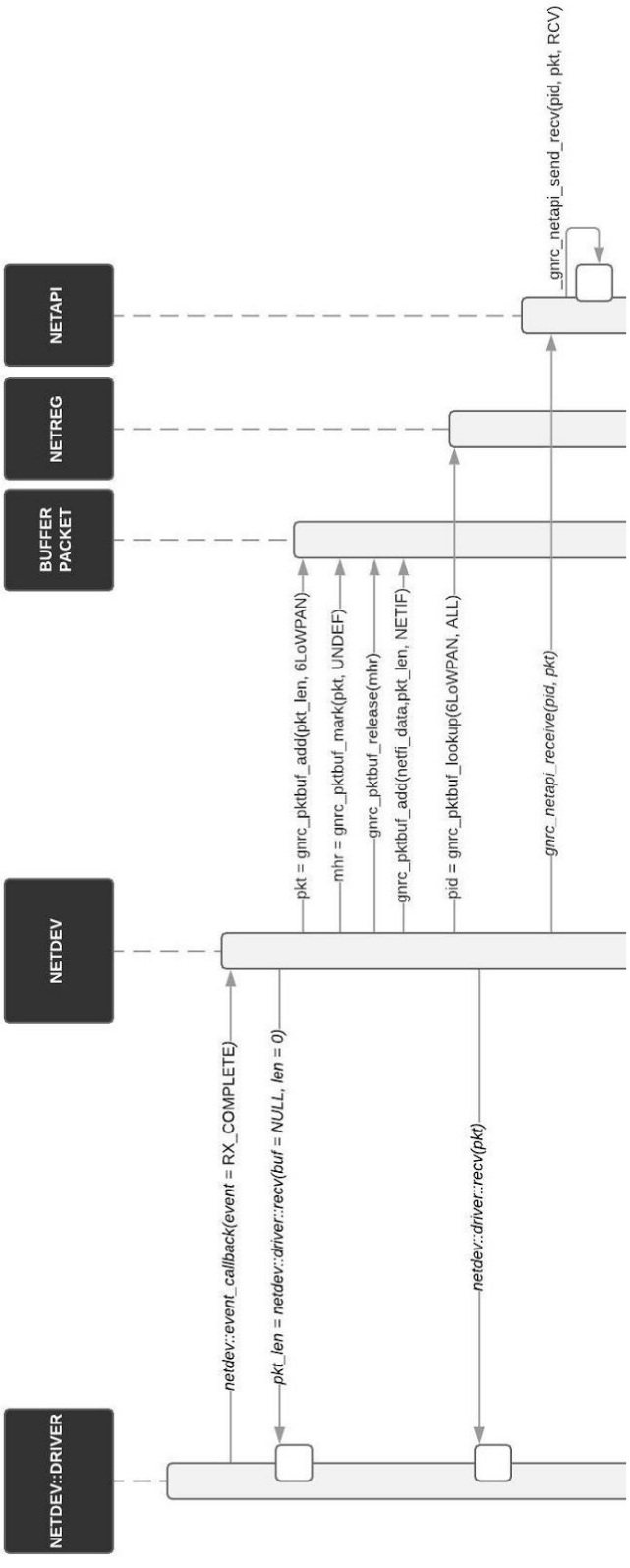


Figura 16. Sequência de chamadas de funções para enviar uma mensagem do driver de rede para o protocolo 6LoWPAN. Por conveniência, GNRC_NETAPI_MSG_TYPE_RCV foi escrito como RCV (método gnrc_netapi_send_recv).

```
51  int _gnrc_netapi_send_recv(kernel_pid_t pid, gnrc_pktsnip_t *pkt, uint16_t type)
52  {
53      msg_t msg;
54      /* set the outgoing message's fields */
55      msg.type = type;
56      msg.content.ptr = (void *)pkt;
57      /* send message */
58      int ret = msg_try_send(&msg, pid);
59      if (ret < 1) {
60          DEBUG("gnrc_netapi: dropped message to %" PRIkernel_pid " (%s)\n", pid,
61                (ret == 0) ? "receiver queue is full" : "invalid receiver");
62      }
63      return ret;
64  }
```

Figura 17. Função definida em sys/net/gnrc/netapi/gnrc_netapi.c.

Após o envio da mensagem, pela função *msg_try_send()*, a Thread do protocolo 6LoWPAN será acordada na linha 326 da Fig. 14. Nesse momento será verificado que a mensagem é do tipo *GNRC_NETAPI_MSG_TYPE_RCV* (receive) e será invocada a função *_receive()*. A Fig. 18 ilustra os passos seguintes, onde é processado o pacote e feito o repasse para o próximo protocolo da pilha.

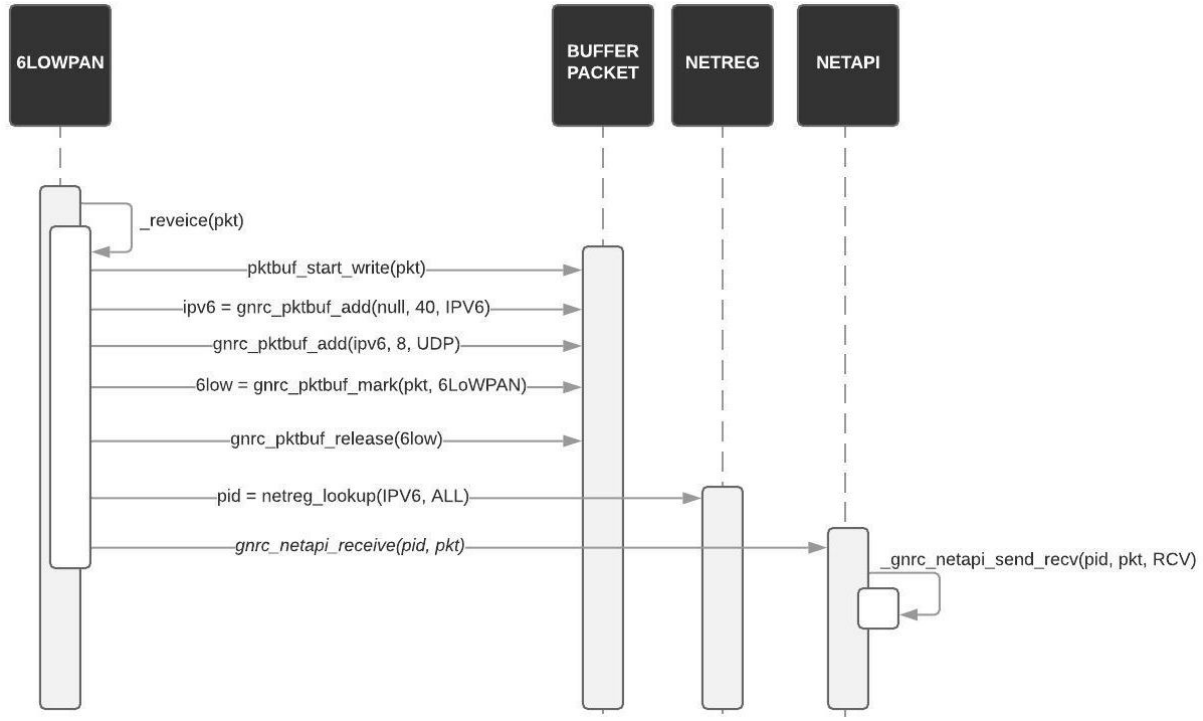


Figura 18. Sequência de chamadas de funções para enviar uma mensagem do protocolo 6LoWPAN para o protocolo IPV6. Por conveniência, GNRC_NETAPI_MSG_TYPE_RCV foi escrito como RCV (método gnrc_netapi_send_recv).

O método receive dos protocolos IPV6, TCP e UDP seguirão os passos mostrados na Fig. 19, até que finalmente chegue na aplicação.

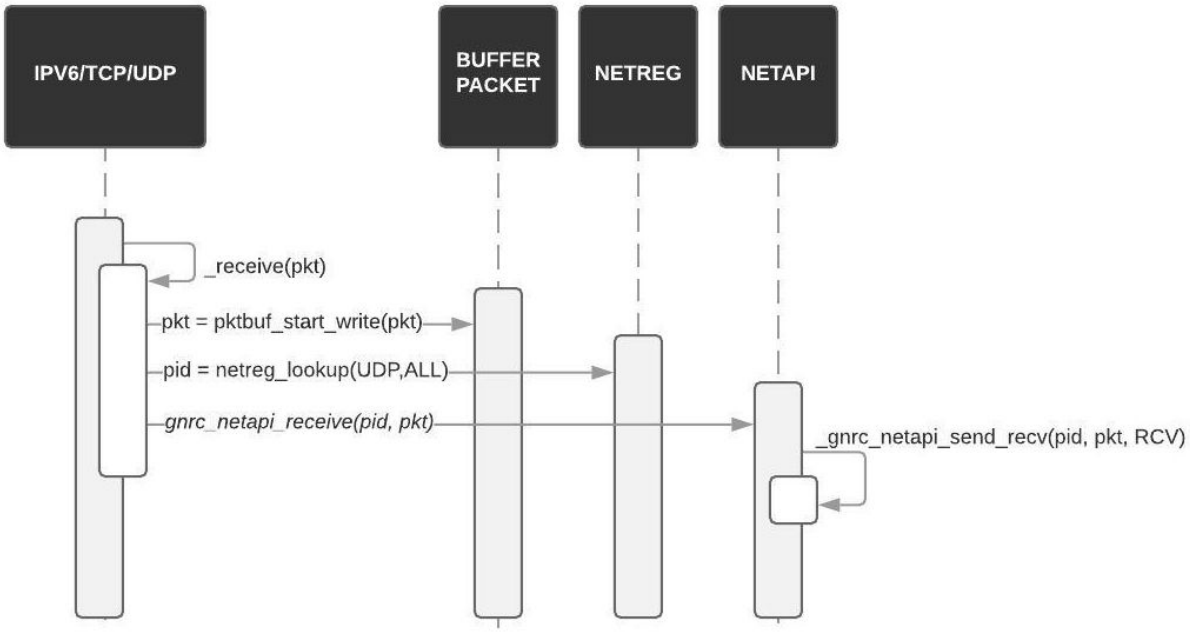


Figura 19. Sequência de chamadas de funções para enviar uma mensagem dos protocolos IPV6, TCP ou UDP para outro protocolo (ou para aplicação). Por conveniência, GNRC_NETAPI_MSG_TYPE_RCV foi escrito como RCV (método gnrc_netapi_send_recv).

Para que o pacote chegue até a aplicação ele deve passar por um socket. Inicialmente o socket deve ser criado, através da função genérica *gnrc_sock_create()*, devendo registrar a Thread no netreg. Após o término da comunicação o socket deve ser destruído, através de uma função close, que deverá ser implementada por cada tipo de socket e remover o registro da Thread do protocolo no netreg. A Fig. 20 apresenta o código da função *gnrc_sock_create()*, enquanto que a Fig. 21 mostra a implementação da função *sock_ip_close()* do protocolo IP.

```
47  void gnrc_sock_create(gnrc_sock_reg_t *reg, gnrc_nettype_t type, uint32_t demux_ctx)
48  {
49      mbox_init(&reg->mbox, reg->mbox_queue, SOCK_MBOX_SIZE);
50      gnrc_netreg_entry_init_mbox(&reg->entry, demux_ctx, &reg->mbox);
51      gnrc_netreg_register(type, &reg->entry);
52  }
```

Figura 20. Função definida em sys/net/gnrc/sock/gnrc_sock.c.

```
63  void sock_ip_close(sock_ip_t *sock)
64  {
65      assert(sock != NULL);
66      gnrc_netreg_unregister(GNRC_NETTYPE_IPV6, &sock->reg.entry);
67  }
```

Figura 21. Função definida em sys/net/gnrc/sock/ip/gnrc_sock_ip.c.

Por fim, as funções *gnrc_sock_recv()* e *gnrc_sock_send()* são utilizadas para receber e enviar pacotes, respectivamente.

3. Demonstração

3.1 Ambiente

Para fins demonstrativos o próprio RioT fornece um ambiente em máquina virtual para testes (configurada com virtualbox e vagrant), onde é necessário apenas baixar os arquivos do github e executar o vagrant. Com o ambiente configurado a máquina pode ser acessada por ssh e já contém todos os arquivos do sistema, além de exemplos e tutoriais.

Para esta demonstração ainda devem ser criados taps para executar dois nodos de rede independente. Para isso é necessário executar o comando "./tapsetup -c 2", acessível em "~/RIOT/dist/tools/tapsetup/". Caso seja necessário realizar o debug, da aplicação, pode-se utilizar o GDB, contudo deve-se fazer uma chamada do GDB passando o PID da aplicação com o comando "gdb program PID". O PID da aplicação pode ser obtido através do comando "pgrep -l task". No caso de aplicações de rede pode ser necessário, ainda, a utilização de permissões de super usuário (sudo), tanto na aplicação quanto na execução do GDB.

3.2 Aplicação

O modo mais simples de observar o funcionamento das interrupções é através das operações de rede. Devido a simplicidade dos tutoriais foi escolhida a tarefa de envio de pacotes UDP (task-06), que pode ter seu funcionamento observado entre nodos assim como entre sistemas.

Além do código da tarefa, o RioT permite adicionar pacotes modulares as aplicações. Na tarefa escolhida foi utilizado apenas o necessário para a troca de mensagens UDP. Os módulos podem ser vistos na Fig. 22 assim como já foram apresentados na Fig. 12.

```
21  # Modules to include:
22  USEMODULE += shell
23  USEMODULE += shell_commands
24  USEMODULE += ps
25  USEMODULE += gnrc_netdev_default
26  USEMODULE += auto_init_gnrc_netif
27  USEMODULE += gnrc_ipv6_default
28  USEMODULE += gnrc_icmpv6_echo
29  USEMODULE += gnrc_sock_udp
```

Figura 22. Módulos da aplicação UDP (pacotes em uso)

Para executar a tarefa é necessário utilizar o comando "make all term PORT=tap" onde o tap depende do nodo, neste caso onde foram criados dois taps uma instância roda em tap0 e outra em tap1.

Com a tarefa executando, antes de iniciar o serviço de recebimento de pacotes, pode ser observado na Fig. 23 os módulos do sistema para tratar cada etapa da aplicação são representados por threads individuais de forma condizente com um dos caso de uso apresentado por [1] para as trocas de mensagens na rede. As thread de rede(ipv6, udp e gnrc) tem sua inicialização idêntica a descrita na seção 2.3.1 que basicamente informa o sistema informa ao sistema qual o tipo de pacote que será tratado, a Fig. 17 destaca as operações necessárias para isso.

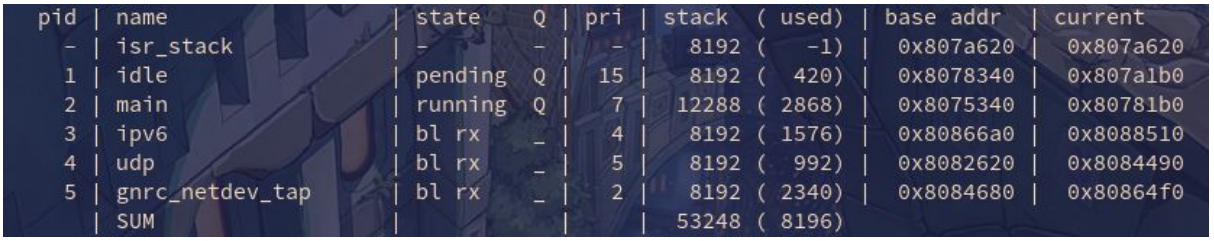


Figura 23. Processos sem udp server

Ainda que o sistema esteja configurado para realizar operações de rede ainda é necessário, dentro da aplicação, iniciar o sock entre a aplicação e o sistema de modo que os pacotes sejam acessíveis pela aplicação, de outro modo toda a sequência de rede aconteceria e no final o pacote iria ser descartado.

A Fig. 24 mostra a lista de processos necessária para a troca de pacotes UDP, como pode ser visto é iniciada uma thread especialmente para o UDP Server pois o mesmo representa um sock udp entre o sistema e aplicação.

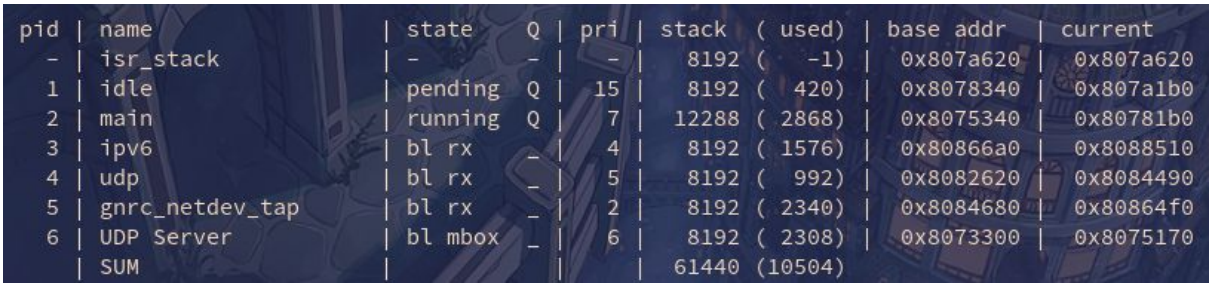


Figura 24. Processos com udp server

Assim como as outras thread que tratam a rede o sock UDP também inicializa uma queue, como pode ser visto na Fig. 25, e em seguida chama um método próprio para a criação de uma sock UDP. Este método(sock_udp_create) configura o sock e garante que o método para criar sock's, mostrado na Fig. 20, seja chamado de forma correta pela aplicação. Por fim o servidor fica esperando o recebimento de um pacote destinado ao sock UDP criado.

```
msg_init_queue(server_msg_queue, SERVER_MSG_QUEUE_SIZE);

if(sock_udp_create(&sock, &server, NULL, 0) < 0) {
    return NULL;
}

server_running = true;
printf("Success: started UDP server on port %u\n", server.port);

while (1) {
    int res;

    if ((res = sock_udp_recv(&sock, server_buffer,
                             sizeof(server_buffer) - 1, SOCK_NO_TIMEOU
                             NULL)) < 0) {
```

Figura 25. Código aplicativo do UDP Server

Por fim o resultado observado pode ser visto iniciando dos nodos em portas (tap) diferentes onde um deles realiza o envio do pacote como mostra a Fig. 26 onde é informada a porta e o endereço.

```
> udp fe80::88f5:f0ff:fe05:fab9 8888 demo
udp fe80::88f5:f0ff:fe05:fab9 8888 demo
Success: send 4 byte to fe80::88f5:f0ff:fe05:fab9
```

Figura 26. Envio de um pacote udp para o nodo alvo

O receptor por sua vez, precisa iniciar o sock na porta designada como já descrito, e esperar o recebimento do pacote na rede como mostra a Fig. 27.

```
> udps 8888
udps 8888
Success: started UDP server on port 8888
> ifconfig
ifconfig
Iface  5  HWaddr: 8A:F5:F0:05:FA:B9
          L2-PDU:1500 MTU:1500  HL:64  Source address length: 6
          Link type: wired
          inet6 addr: fe80::88f5:f0ff:fe05:fab9  scope: local  VAL
          inet6 group: ff02::1
          inet6 group: ff02::1:ff05:fab9

> Recvd: demo
```

Figura 27. recebimento de uma pacote udp.

Referências

https://ieeexplore.ieee.org/stamp/stamp.jsp?tp=&arnumber=5355049&tag=1
https://riot-os.org/docs/riot-infocom2013-abstract.pdf
http://cst-pub.imp.fu-berlin.de/papers/knuepfer-diploma-thesis.pdf
https://riot-os.org/api/group__net__gnrc.html
https://doc.riot-os.org/group__net__gnrc__netreg.html
https://doc.riot-os.org/group__drivers__netdev__api.html
https://doc.riot-os.org/group__net__sock.html
[1] http://doc.riot-os.org/mlenders_msc.pdf
https://doc.riot-os.org/group__net__gnrc__netapi.html
https://riot-os.org/api/group__net__gnrc__pktbuf.html
https://arxiv.org/pdf/1906.12143.pdf
http://doc.riot-os.org/mlenders_msc_def.pdf
https://github.com/RIOT-OS/Tutorials
https://doc.riot-os.org/index.html#riot-in-a-nutshell