Extensão dos mecanismos de gerência de tarefas do sistema operacional TinyOS

Bolsista: Pedro Rosanes Orientador: Silvana Rossetto

Departamento de Ciência da Computação

11 de abril de 2011

Sumário

Resumo

Resumo Redes de Sensores Sem Fio (RSSFs) são formadas por pequenos dispositivos de sensoreamento, com espaço de memória e capacidade de processamento limitados, fonte de energia esgotável e comunicação sem fio. O sistema operacional mais usado na programação desses dispositivos é o TinyOS, um sistema leve, projetado especialmente para consumir pouca energia, um dos requisitos mais importante para RSSFs. O modelo de programação adotado pelo TinyOS prioriza o atendimento de interrupções. Em função disso, as operações são normalmente divididas em duas fases: uma para envio do comando, e outra para o tratamento da resposta (evento sinalizado via interrupção). Esse modelo de programação, baseado em eventos, quebra o fluxo de execução normal, dificultando a tarefa dos desenvolvedores de aplicações. Para que os tratadores de eventos (interrupções) sejam curtos, tarefas maiores são postergadas para execução futura e, para evitar concorrência entre elas, as tarefas são executadas em sequência, uma após a outra (i.e., uma tarefa só é iniciada após a tarefa anterior ser concluída). O objetivo deste trabalho é propor e implementar políticas alternativas de escalonamento de tarefas para o TinyOS visando a construção de abstrações de programação de nível mais alto que facilitem o desenvolvimento de aplicações nessa área.

1 Introdução

Redes de Sensores Sem Fio (RSSFs) caracterizam-se pela formação de aglomerados de pequenos dispositivos que, atuando em conjunto, permitem monitorar ambientes físicos ou processos de produção com elevado grau de precisão. O desenvolvimento de aplicações que permitam explorar o uso dessas redes requer o estudo e a experimentação de protocolos, algoritmos e modelos de programação que se adequem às suas características e exigências particulares, entre elas, uso de recursos limitados, adaptação dinâmica das aplicações, e a necessidade de integração com outras redes, como a Internet.

Sistemas projetados para os dispositivos que formam as redes de sensores devem lidar apropriadamente com as restrições e características particulares desses ambientes. A arquitetura adotada pelo TinyOS [1] — um dos sistemas operacionais mais usados na pesquisa nessa área — prioriza fortemente o tratamento dessas restrições em detrimento da simplicidade oferecida para o desenvolvimento de aplicações. A linguagem de programação usada é o nesC [2], uma extensão de C que provê um modelo de programação baseado em componentes e orietado a eventos. Para lidar com as diversas operações de entrada e saída, o TinyOS utiliza um modelo de execução em duas fases, evitando bloqueios e, consequentemente, armazenamento de estados. A primeira fase da operação é um comando que pede ao hardware a execução de um serviço (ex.: sensoreamento). Este comando retorna imediatamente dando continuidade à execução. Quando o serviço é terminado, o hardware envia uma interrupção, sinalizada como um evento pelo TinyOS. Então, o tratador do evento recebe as informações (ex.: valor sensoreado) e trata/processa essas informações conforme programado. O problema gerado por essa abordagem é a falta da visão de um fluxo contínuo de execução na perspectiva do programador.

O modelo de concorrência divide o código em dois tipos: assíncrono e síncrono. Um código assíncrono pode ser alcançável a partir de pelo menos um tratador de interrupção. Em função disso, a execução desses trechos do programa pode ser interrompida a qualquer momento e é necessário tratar possíveis condições de corrida. Um código síncrono é alcançável somente a partir de tarefas (*tasks*) que são procedimentos adiados (postergados). Tarefas executam até terminar (não existe concorrência entre elas), por isso as condições de corrida, neste contexto, são evitadas. As tarefas são todas escalonadas por um componente do TinyOS que usa uma política padrão de escalonamento do tipo *First-in First-out* [3].

Com o objetivo de oferecer maior flexibilidade aos desenvolvedores de aplicações, a versão mais atual do TinyOS (versão 2.1.x) trouxe novas facilidades. Uma delas é a possibilidade de substituir o componente de escalonamento de tarefas para implementar diferentes políticas de escalonamento [4]. A outra é a possibilidade de usar o modelo de programação multithreading, mais conhecido pelos desenvolvedores de aplicações e que pode ser usado como alternativa para lidar com as dificuldades da programação orientada a eventos.

Neste trabalho avaliamos essas novas facilidades do TinyOS e propomos extensões que visam oferecer facilidade adicionais para os desenvolvedores de aplicações. Inicialmente propusemos novos escalonadores de tarefas, implementando diferentes políticas de escalonamento por prioridade e avaliamos o modelo de multithreading oferecido, comparando diferentes formas de implementação de uma aplicação básica e o custo da gerência de threads. Em seguida, tomando como base o modelo multithreading oferecido, projetamos um mecanismo de gerência cooperativa de tarefas para o TinyOS, visando uma solução alternativa entre o modelo de escalonamento de tarefas que executam até terminar e evitam condições de corrida, e o modelo de execução alternada entre as tarefas que permite maior flexibilidade durante a execução, mas com custo de gerência alto.

O modelo de gerência cooperativa de tarefas é uma solução viável para as redes de sensores sem fios devido simplicidade do hardware. Como os processadores têm somente um núcleo, e não possuem tecnologia hyperthreading, não é possível existir duas unidades de execução rodando em paralelo. Portanto uma gerência cooperativa de tarefas seria uma solução menos custosa, por eliminar a necessidade de mecânismos de sincronização, e diminuir o número de trocas de contexto.

2 Conceitos Básicos

2.1 Rede de Sensores Sem Fio

Uma rede de sensores sem fio (RSSF) é um conjunto de dispositivos formando uma rede de comunicação *ad-hoc*. Cada sensor tem a capacidade de monitorar diversas propriedades físicas, como intensidade luminosa, temperatura, aceleração, entre outras. Através de troca de mensagens, esses dispositivos podem agregar todas essas informações para detectar um evento importante no local, como um incêndio. Essa conclusão é então encaminhada para um nó com maior capacidade computacional, conhecido como estação base. Este nó pode decidir uma

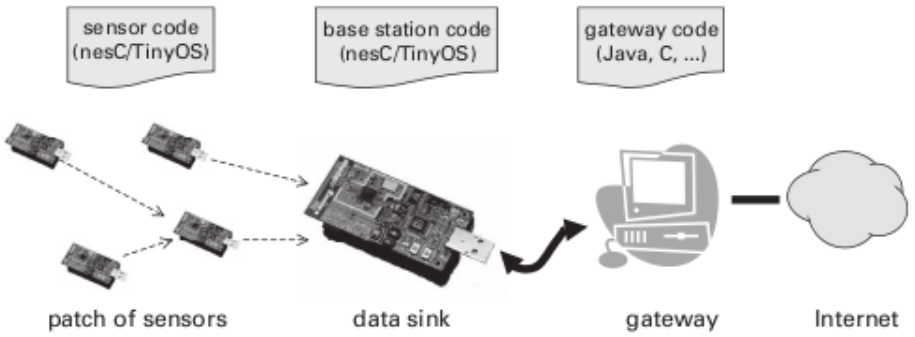


Figura 1: Rede de sensores sem fio

ação a ser tomada, ou enviar a informação pela Internet.

Estes sensores são desenvolvidos para monitoriar ambientes de difícil acesso, portanto, devem ser pequenos e utilizar comunicação sem fio para facilitar a instalação no ambiente e minimizar o custo financeiro. Para evitar manutenções frequêntes, eles também devem consumir pouca energia. Devido a estas características, o hardware destes dispositivos tende a ter recursos computacionais limitados. Ao invés de utilizar CPUs, são usados microcontroladores de 8 ou 16 bits, normalmente, com baixas frequências de relógio. Para armazenar o código da aplicação é utilizada uma pequena memória flash, da ordem de 100kB, e para as variáveis existe uma memória RAM, da ordem de 10kB. Os circuitos de rádio também têm uma capacidade reduzida de transferência, da ordem de kilobytes por segundos [3]. Aliado ao hardware, o software também deve ser voltado para o baixo consumo de energia e de memória. Detalhes serão vistos na próxima seção.

Uma RSSF é usada para monitorar ambientes de difícil acesso, onde uma rede cabeada seria inviável ou custosa. Alguns exemplos reais do uso de RSSF são: monitoramento da ponte Golden Gate em São Francisco, e dos vulcões Reventador e Tungurahua no Equador [3].

2.2 TinyOS e nesC

O TinyOS é o sistema operacional mais usado para auxiliar os programadores a desenvolverem aplicações para rede de sensores sem fio de baixo consumo. O modelo de programação provido é baseado em componentes e orientado a eventos. Os componentes são pedaços de código reutilizáveis, onde são definidas claramente suas dependências e os serviços oferecidos, por meio de interfaces. A linguagem *nesc* implementa esse modelo estendendo a linguagem C. É através da conexão (*wiring*) de diversos componentes que o sistema é montado.

Já o modelo de programação orientado a eventos permite que o TinyOS rode uma aplicação, com somente uma linha de execução, respondendo a diferentes interrupções de sistema, sem a necessidade de ações bloqueantes. Para isso todas as operações de entrada e saída são realizadas em duas fases. Na primeira fase, o comando de E/S sinaliza para o hardware o que deve ser feito, e retorna imediatamente, dando continuidade a execução. A conclusão da operação é sinalizada através de um evento, que será tratado pela segunda fase da operação

de E/S.

O modelo de programação baseada em compontenes está intimamente ligada à programação orientada a eventos: Um componente oferece a interface, implementando os comandos e sinalizando eventos relacionandos, enquanto outro componente utiliza esta interface, através do uso dos comandos e da implementação dos tratadores de evento.

A divisão em componentes também facilita a implementação da camada de abstração de hardware. De forma que cada plataforma tem um conjunto diferente de componentes para lidar com as instruções de cada hardware. As abstrações providas são de serviços como sensoreamento, comunicação por rádio e armazenamento na memória flash.

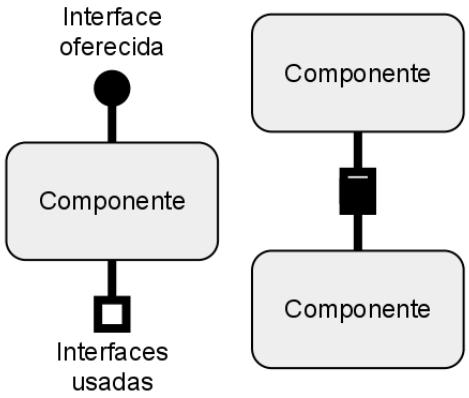


Figura 2: Ilustração de componentes e suas interfaces

Exemplo A aplicação *Blink*, no anexo B.1, é utilizada para ilustrar estes conceitos básicos. Esta aplicação faz os LEDs da plataforma piscarem continuamente. Toda aplicação utiliza uma configuração para descrever os componentes que serão usados, e quais são as conexões entre as interfaces.

Listing 1: Configuração (BlinkAppC.nc)

```
configuration BlinkAppC
{}
implementation
{
  components MainC, BlinkC, LedsC;
  components new TimerMilliC() as Timer0;
  components new TimerMilliC() as Timer1;
  components new TimerMilliC() as Timer2;

  BlinkC.Boot -> MainC.Boot;
  BlinkC.Timer0 -> Timer0;
  BlinkC.Timer1 -> Timer1;
  BlinkC.Timer2 -> Timer2;
  BlinkC.Leds -> LedsC.Leds;
}
```

Na listagem 1, pode-se ver que alguns dos componentes utilizados são *MainC*, *BlinkC*, *LedsC*. *MainC* é o responsável pela inicialização do sistema. E indica o termino deste processo através do evento *booted*, da interface *Boot* (mais detalhes na seção 2.3). *BlinkC* implementa a lógica da aplicação. E *LedsC* implementa as operações necessárias para acender ou apagar os Leds da plataforma.

O outro componente utilizado é um temporizador. O comando *new* é usado para criar instâncias de componentes genéricos. Isso permite a criação de cópias distintas de uma mesma funcionalidade. Neste caso são criados três temporizadores diferentes. Alguns componentes não podem ter cópias distintas, como o *LedsC* que representa uma estrutra física do sensor, logo não pode ser multiplicada.

Na listagem 1 também são definidas as conexões das interfaces de cada componente. A construção da linha 14, por exemplo, indica que o módulo *BlinkC* utiliza a interface oferecida por *LedsC* para apagar ou acender os LEDs.

A lógica da aplicação está implementada no componente *BlinkC*, através da contrução de um módulo..

Listing 2: Interfaces usadas pelo módulo (BlinkC.nc)

```
#include "Timer.h"

module BlinkC @safe()
{
  uses interface Timer<TMilli> as Timer0;
  uses interface Timer<TMilli> as Timer1;
  uses interface Timer<TMilli> as Timer2;
  uses interface Leds;
  uses interface Boot;
}
```

Na listagem 2, estão especificadas as interfaces utilizadas, que formaram as conexões vistas na configuração.

Listing 3: Eventos, Comandos, Postagem de tarefas (BlinkC.nc)

```
implementation
{
  event void Boot.booted()
  {
    call Timer0.startPeriodic( 250 );
    call Timer1.startPeriodic( 500 );
    call Timer2.startPeriodic( 1000 );

    post tarefa();
  }

  event void Timer0.fired()
  {
```

```
24      call Leds.led0Toggle();
    }
```

Finalmente, na listagem 3, é definida a lógica do componente principal da aplicação. O tratador do evento *Boot.booted* é o primeiro a ser ativado após a inicialização do sistema. Dentro deste, é invocado o comando para inicializar os temporizadores. A periodicidade de cada um é definida pelo parâmetro passado. E por último, é postada uma tarefa, cujos detalhes serão vistos a seguir. O tratador do evento *Timer0.fired*, toda vez que é ativado, invoca o comando responsável por acender/apagar o LED 0. O mesmo acontece para os outros temporizadores.

Tasks O TinyOS também utiliza o conceito de procedimento postergados, chamados de tarefas (*tasks*). As próprias tarefas, comando e tratadores de eventos podem postar uma nova tarefas, a qual é enfileirada para execução posterior. Esta será atendida, de forma síncrona, pelo escalonador. Ser executada de forma síncrona, significa que tarefas não são preemptiva entre si. Portanto, se diversas tarefas compartilha uma mesma variável, não haverá condições de corrida. Mais detalhes serão vistos na seção 2.4.

Listing 4: Implementação de tarefas (BlinkC.nc)

```
  task void tarefa()
38  {
        dbg("BlinkC", "tarefa\n");
40  }
}
```

Na listagem 3, existe um exemplo de uma postagem de tarefa. E na listagem 4, um exemplo da implementação de uma tarefa, que neste caso somente emite uma mensagem de depuração.

Interfaces Ao desenvolver aplicações mais complexas, é preciso desenvolver componentes intermediários. Estes devem além de usar, também devem oferecer interfaces. Para oferecer novas interfaces, o programador deve declará-las, implementar seus comandos, e sinalizar seus eventos. O componente que usar estas interfaces, será responsável por utilizar os comandos e implementar os tratadores de eventos.

Listing 5: interface

```
1  interface Send {
       command error_t send(message_t* msg, uint8_t len);
3      command error_t cancel(message_t* msg);
       event void sendDone(message_t* msg, error_t error);
5      command uint8_t maxPayloadLength();
       command void* getPayload(message_t* msg, uint8_t len);
7  }

9  module SendExample {
       provides interface Send;
11 }
   implementation {
```

```
13      command error_t Send.send(message_t* msg, uint8_t len) {
            //Implementacao do comando send.
15          ...

17          signal Send.sendDone(msg_var, SUCCESS);
        }
19      ...
}
```

2.3 Sequência de inicialização do TinyOS

O principal componente do TinyOS, responsável por inicializar o sistema, é chamado *MainC*. Ele inicializa os componentes de hardware e software e o escalonador de tarefas. Para isso, *MainC* se liga aos componentes *RealmainP*, *PlataformC*, *TinySchedulerC*, e utiliza a interface *SoftwareInit*.

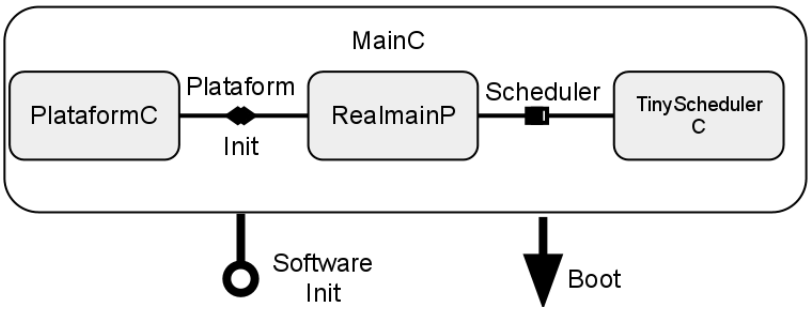


Figura 3: MainC

Primeiro é configurado o sistema de memória e escolhido o modo de processamento. Com esses pré-requisitos básicos estabelecidos, o escalonador de tarefas é inicializado para permitir que as próximas etapas possam postar tarefas. O segundo passo é inicializar os demais componentes de hardware, permitindo a operabilidade da plataforma. Alguns exemplos são configuração de pinos de entrada e saída, calibração do clock e dos LEDs. Como esta etapa exige códigos específicos para cada tipo de plataforma, o MainC se liga ao componente *PlataformC* que implementa o tratamento requerido por cada tipo de plataforma.

O terceiro passo trata da inicialização dos componentes de software. Além de configurar os aplicativos básicos do sistema, como os *timer*s, nesta etapa são executados também os procedimentos de inicialização dos componentes da aplicação. Para isso, os componentes da aplicação que precisam ser inicializados devem oferecer a interface *SoftwareInit*. Assim, durante a etapa de inicialização do sistema, os códigos de inicialização dos componentes da aplicação são automaticamente chamados.

Por último, quando todas as etapas foram concluídas, o MainC avisa a aplicação que a inicialização terminou, através do evento *Boot.booted()*. O TinyOS entra no seu laço principal, no qual o escalonador espera por tarefas e as executa. É importante notar que durante todo o processo de inicialização as interrupções do sistema ficam desabilitadas [5].

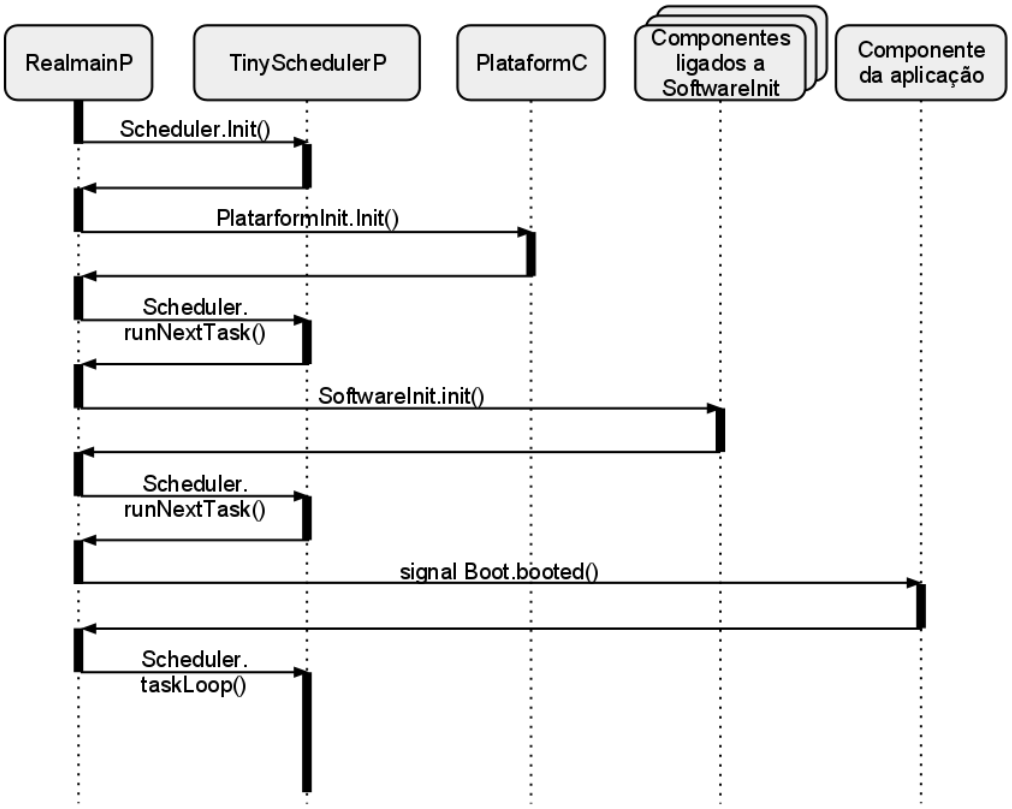


Figura 4: Sequência de Inicialização

Listing 6: Código de inicialização

```
module RealMainP {
    provides interface Booted;
    uses {
        interface Scheduler;
        interface Init as PlatformInit;
        interface Init as SoftwareInit;
    }
}
implementation {
     int main() __attribute__ ((C, spontaneous)) {
         atomic {
             platform_bootstrap();
             call Scheduler.init();
             call PlatformInit.init();
             while (call Scheduler.runNextTask());
             call SoftwareInit.init();
             while (call Scheduler.runNextTask());
         }
         __nesc_enable_interrupt();
         signal Boot.booted();
         call Scheduler.taskLoop();
```

```
22          return -1;
        }
```

2.4 Modelo de concorrência do TinyOS

O TinyOS define o conceito de *tasks* (tarefas) como mecanismo central para lidar com as questões de concorrência nas aplicações. Tarefas têm duas propriedades importantes. Elas não são preemptivas entre si, e são executadas de forma adiada. Isso significa que ao postar uma tarefa, o fluxo de execução continua, sem desvio, e ela só será processada mais tarde. Na definição básica do TinyOS, as tarefas não recebem parâmetros e não retornam resultados.

O TinyOS minimiza os problemas clássicos de concorrência garantindo que qualquer possível condição de corrida gerada a partir de tratadores de interrupção, seja detectada em tempo de compilação. Para que isso seja possível, o código em nesC é dividido em dois tipos:

Código Assíncrono Código alcançável a partir de pelo menos um tratador de interrupção.

Código Síncrono Código alcançável somente a partir de tarefas.

Como visto na seção 2.3, ao final do processo de inicialização, as interrupções são ativadas, e o evento *Boot.booted* é sinalizado. O tratador deste evento, implementado no módulo principal da aplicação, é executado. E por último, o TinyOS entra em um laço infinito, onde as tarefas passam a ser atendidas. Este fluxo, quando não interrompido, é chamado de fluxo principal do TinyOS, e corresponde ao código síncrono. Como não há preempção entre as tarefas, variáveis compartilhadas entre elas são imunes a condições de corrida. Porém quando há uma interrupção de hardware, o tratador da interrupção assume o controle, e o fluxo principal é congelado até o termino daquele. Qualquer variável compartilhada, quando acessada por estes códigos assíncronos, está sujeita a condições de corrida.

Como tarefas são postergadas, e atendidas pelo fluxo de execução principal, elas são usadas para fazer uma transição de contexto assíncrono para síncrono. Para fazer isto, um tratador de interrupção deve fazer somente o processamento mínimo, como transferência de dados entre o *buffer* e a memória. Após isto, deve postar uma tarefa para sinalizar o evento de conclusão da operação de E/S. Quando a tarefa for atentida, ela sinalizará o evento de forma síncrona. E portanto, seu tratador também será um código síncrono.

Para auxiliar no controle de condições de corrida, qualquer código assíncrono devem ser marcados como *async* no código fonte. Para contornar isto, deve-se usar o comando *atomic* ou *power locks*.

O comando *atomic* garante exclusão mútua desabilitando interrupções. Dois fatos importantes surgem com o seu uso, primeiro a ativação e desativação de interrupções consome ciclos de CPU. Segundo, longos trechos atômicos podem atrasar outras interrupções, portanto é preciso tomar cuidado ao chamar outros componentes a partir desses blocos.

Algumas vezes é preciso usar um determinado hardware por um longo tempo, sem compartilhá-lo. Como a necessidade de atomicidade não está no processador e sim no hardware, pode-se conceder sua exclusividade a somente um usuário (componente) através de *Power locks*. Para isso, primeiro é feito um pedido através de um comando, depois quando o recurso desejado

estiver disponível, um evento é sinalizado. Assim não há espera ocupada. Existe a possibilidade de requisição imediata. Nesse caso nenhum evento será sinalizado: se o recurso não estiver locado por outro usuário (componente), ele será imediatamente cedido, caso contrário, o comando retornará falso. [3, Cap.11].

3 Escalonamento de tarefas

Nesta seção é feita uma abordagem teórica sobre escalonamento de tarefas. Depois apresentamos a implementação do escalonador padrão de tarefas do TinyOS. Também apresentamos o projeto e as etapas de implementação de novos escalonadores de tarefas para o TinyOS. Implementamos três propostas: escalonador EDF (*Earliest Deadline First*), escalonador com prioridades, e escalonador multi-nível. Por último mostramos os experimentos e resultados obtidos, usados para comprar os diferentes escalonadores.

3.1 Abordagem teórica sobre escalonamento de tarefas

As tarefas, por serem procedimentos adiados, necessitam de algoritmos de escalonamento. Estes algoritmos também não podem ser preemptivos, devido a natureza das tarefas do TinyOS. O algoritmo mais simples, e também o padrão do TinyOS, é o *First-Come, First-Served*, onde as tarefas são atendidas segundo a ordem de chegada. O *overhead* gerado é mínimo, e não há possibilidade de *starvation*. Porém o tempo de resposta pode ser alto, se houver uma grande quantidade de tarefas na fila.

Escalonamento utilizando *deadline* é muito usado em sistemas operacionais de tempo real [6]. Neste algoritmo a próxima tarefa a ser executada é aquela com menor prazo (*deadline*). As diversas variações deste algoritmo utilizam parâmetros como: tempo de entrada na fila de prontos, prazo para começar a tarefa, prazo para terminar a tarefa, tempo de processamento, recursos utilizados, prioridade, existência de preempção. Porém, o principal parâmetro utilizado pelos algoritmos é a existência ou não de preempção. Caso não exista preempção, faz mais sentido utilizarmos, no escalonamento, o prazo para começar a tarefa. Caso exista preempção, o prazo para terminar a tarefa é utilizado [6]. Um *overhead* maior passa a existir, devido à ordenação das tarefas na fila e à preempção, caso exista. Porém o tempo de resposta pode ser aproximadamente estipulado pela própria tarefa.

Em um escalonamento de prioridade fixa, cada tarefa indica, no momento de entrada para fila de prontos (tempo de execução), sua importância em relação às outras tarefas. Nestes algoritmos podemos ter preempção por parcela de tempo, na entrada de outras tarefas, ou não ter preempção. No primeiro tipo, pode existir um *overhead* desnecessário quando o *time-slice* da tarefa atual terminou, porém não existe nenhuma outra com prioridade maior. O segundo tipo resolve este problema: se existe uma ordem de tarefas na fila, esta ordem só pode ser alterada caso uma nova tarefa entre. Quando não há preempção, a troca de tarefa só ocorre no término da execução de uma tarefa. Neste escalonamento também há um *overhead* maior, devido a ordenação das tarefas na fila. A possibilidade de *starvation* passa a existir, e o tempo

de resposta varia de acordo com a prioridade das tarefas.

O escalonamento de multi-nível é um caso especial do escalonamento de prioridade fixa. Cada tarefa determina seu nível de prioridade em tempo de compilação. Onde cada nível de prioridade tem uma fila, com política *First-in First-out*, e as filas mais importantes devem ser atendidas por completo para que outra sejam atendidas.

O escalonamento de prioridade dinâmica visa eliminar a possibilidade de *starvation*. Neste caso, a tarefa ainda indica sua importância no momento de entrada para fila de prontos. Porém, as tarefas que estão esperando para executar aumentam de prioridade toda vez que não são atendidas. Apesar disto aumentar significativamente o *overhead*, o *starvation* é eliminado.

3.2 Escalonador padrão de tarefas do TinyOS

O componente responsável por gerenciar e escalonar tarefas no TinyOS é o componente *TinySchedulerC*. O escalonador padrão adota uma política *First-in First-out* para agendar as tarefas. Ele também cuida de parte do gerenciamento de energia, colocando a CPU em um estado de baixo consumo quando não há nada para ser executado.

O programador, ao codificar uma tarefa, utiliza duas contruções:

```
1    post nome_da_tarefa();
     task void nome_da_tarefa() { //Definicao da tarefa }
```

Essas duas contruções são transformadas pelo compilador, fazendo com que a aplicação implemente uma interface chamada *TaskBasic*. A primeira é transformada em um comando, usado para indicar ao escalonador que esta tarefa deve entrar na fila. O escalonador por sua vez, quando decidir que esta tarefa será a próxima a executar, sinalizará o evento relacionado a este comando. A segunda sintaxe é transformada no tratador deste evento, que implementa o que a tarefa deverá executar quando escalonada. É esta interface que permite a conexão das tarefas ao escalonador [3].

```
interface TaskBasic {
2    async command error_t postTask();
     event void runTask();
4 }
```

Todo escalonador, além de prover a interface *TaskBasic*, também deve prover a interface *Scheduler*.

```
interface Scheduler {
2    command void init();
     command bool runNextTask();
4    command void taskLoop();
}
```

A implementação dessas interfaces se dá da seguinte forma:

command postTask() Decide onde a tarefa será inserida na fila.

event runTask() Indica que a tarefa deve executar.

command runNextTask() Retira a primeira tarefa da fila e sinaliza sua execução com o evento *runTask()*

command taskLoop() Laço infinito que executa o comando *runNextTask()*. Caso não haja tarefa para executar, coloca a CPU em modo de baixo consumo.

Para criar novos tipos de tarefas, é preciso definir uma interface nova, com o comando *postTask* e o evento *runTask* que será provida pelo escalonador como visto acima. Por exemplo:

```
1 interface TaskDeadline<precision_tag> {
    async command error_t postTask(uint32_t deadline);
3   event void runTask(); }
```

Na versão 2.1.x do TinyOS é possível mudar a política de gerenciamento de tarefas substituindo o componente escalonador padrão. Qualquer novo escalonador tem de aceitar a interface de tarefa padrão (*TaskBasic*), e garantir a execução de todas as tarefas (ausência de *starvation*) [4].

O componente *TinySchedulerC* é uma configuração que conecta as interfaces de tarefa à implementação do escalonador. Para alterar o escalonador basta definir um novo componente *TinySchedulerC* e adicioná-lo ao diretório da aplicação. Neste novo componente, a interface *Scheduler* deve ser associada ao componente que implementa o escalonador proposto, como ilutrado abaixo.

Por último, deve-se amarrar a interface da tarefa com a interface do escalonador. Por exemplo:

```
1 configuration TinySchedulerC {
    provides interface Scheduler;
3   provides interface TaskBasic[uint8_t id];
    provides interface TaskDeadline<TMilli>[uint8_t id];
5 }
  implementation {
7   components SchedulerDeadlineP as Sched;
    ...
9   Scheduler = Sched;
    TaskBasic = Sched;
11  TaskDeadline = Sched;
  }
```

Um exemplo de aplicação que utiliza um escalonador novo pode ser visto no anexo B.2. Para que o escalonador funcione corretamente no simulador TOSSIM é preciso adicionar funções que lidam com eventos no simulador. Um exemplo desta extensão poder ser achada no arquivo *tinyOS-root-dir/tos/lib/tossim/SimSchedulerBasicP.nc*, ou no apêndice A

3.3 Escalonador EDF (*Earliest Deadline First*)

Este escalonador [1] aceita tarefas com deadline e elege aquelas com menor *deadline* para executar. A interface usada para criar esse tipo de tarefas é *TaskDeadline*. O *deadline* é passado

[1] O TEP 106 [4] disponibiliza um protótipo

por parâmetro pela função *postTask*. As tarefas básicas (*TaskBasic*) também são aceitas, como recomendado pelo TEP 106[4].

Em contraste, o escalonador não segue outra recomendação: não elimina a possibilidade de *starvation* pois as tarefas básicas só são atendidas quando não há nenhuma tarefa com *deadline* esperando para executar. A fila de prioridades é implementada da mesma forma que a do escalonador padrão3.2, a única mudança está na inserção. Para inserir, a fila é percorrida do começo até o fim, procurando-se o local exato de inserção. Portanto, o custo de inserir é $\bigcirc(n)$, e o custo de retirar da fila é $\bigcirc(1)$.

3.4 Escalonador por prioridades

Desenvolvemos um escalonador onde é possível estabelecer prioridades para as tarefas. A prioridade é passada como parâmetro através do comando *postTask*. Quanto menor o número passado, maior a preferência da tarefa, sendo 0 a mais prioritária e 254 a menos prioritária. As *Tasks* básicas também são aceitas, e são consideradas as tarefas de menor prioridade.

Foram encontrados dois problemas de *starvation*. O primeiro relacionado com as tarefas básicas, onde elas só seriam atendidas caso não houvesse nenhuma tarefa de prioridade na fila. Para resolver isso, foi definido um limite máximo de tarefas prioritárias que podem ser atendidas em sequência. Caso esse limite seja excedido, uma tarefa básica é atendida. O segundo é relacionado às próprias tarefas de prioridade. Se entrar constantemente *tasks* de alta prioridade, é possível que as de baixa prioridade não sejam atendidas. A solução se deu através do envelhecimento de tarefas. Ou seja, *tasks* que ficam muito tempo na fila, têm sua importância aumentada.

Dois tipos de estrutura de dados foram usadas para a organização das tarefas, uma fila comum e uma *heap*. Com isso, totalizou-se quatro diferentes versões do escalonador:

1. Fila comum sem envelhecimento
2. Fila comum com envelhecimento
3. Heap sem envelhecimento
4. Heap com envelhecimento

A seguir uma tabela com a complexidade de inserção e remoção para cada escalonador:

Escalonador	Inserção	Remoção
Fila, sem envelhecimento	$\bigcirc(n)$	$\bigcirc(1)$
Heap, sem envelhecimento	$\bigcirc(\log(n))$	$\bigcirc(\log(n))$
Fila, com envelhecimento	$\bigcirc(n)$	$\bigcirc(n)$
Heap, com envelhecimento	$\bigcirc(\log(n))$	$\bigcirc(n)$

3.5 Escalonador multi-nível

No TinyOS, percebe-se uma divisão clara dos tipos de serviços:

Rádio Comunicação sem fio entre diferentes nós da rede através de ondas de rádio.

Sensor Sensoriamento de diferentes características do ambiente.

Serial Comunicação por fio entre um nó e uma estação base (PC).

Básica Outros serviços, como por exemplo temporizador.

Por isso, desenvolvemos um escalonador que divide as tarefas de acordo com os tipos definidos acima. Cada tipo de tarefa tem sua própria fila com política *First-in First-out*, e as filas mais importantes devem ser atendidas por completo para que as outras sejam antendidas.

3.6 Experimentos e resultados obtidos

Experimentos com o escalonador de tarefas padrão Antes de começar a desenvolver outros escalonadores de tarefas, foi feito um experimento com o escalonador padrão que utiliza a política *First in, First Out*. Para medir a complexidade na prática, foi desenvolvida uma aplicação de teste (anexo B.2). Nela cada tarefa executa um loop de 65000 iterações, fazendo uma simples multiplicação em cada iteração. O número de tarefas variou entre 20, 50 e 100. O tempo de execução foi medido em uma plataforma *MicaZ*, utilizando o temporizador *Counter<TMicro,uint32_t>*, utilizando uma precisão de microsegundos. Os valores medidos não variaram mais de uma unidade entre diferentes execuções.

Escalonador	20 Tarefas	50 Tarefas	100 Tarefas
Escalonador Padrão	1366	1849	2652

Experimentos com o escalonador com prioridades Para avaliar o desempenho com o escalonador com prioridades foi desenvolvida a mesma aplicação de teste, onde cada tarefa executa um loop de 65000 iterações, fazendo uma simples multiplicação em cada iteração. O número de tarefas variou entre 20, 50 e 100. O tempo de execução foi medido em uma plataforma *MicaZ*, utilizando o temporizador *Counter<TMicro,uint32_t>*, utilizando uma precisão de microsegundos. Os valores medidos não variaram mais de uma unidade entre diferentes execuções.

Escalonador	20 Tarefas	50 Tarefas	100 Tarefas
Escalonador Padrão	1366	1849	2652
Fila, sem envelhecimento	1733	4660	13721
Heap, sem envelhecimento	2603	4308	7486
Fila, com envelhecimento	2278	7887	26066
Heap, com envelhecimento	2665	4510	7887

Podemos perceber que, para um número pequeno de tarefas, a fila é mais eficiente que a heap. Isso acontece pois não é compensado o *overhead* do algoritmo da heap.

4 Modelos de programação

Nesta seção, primeiro é feita uma abordagem teórica sobre Multithreading e co-rotinas. Também apresentamos a biblioteca *TinyOS Threads*, que oferece um modelo de programação em threads, como alternativa ao modelo orientado a eventos. Depois apresentamos o projeto e as etapas de implementação de co-rotinas para o TinyOS. Por último mostramos os experimentos e resultados obtidos, usados para comprar o modelo de threads com o de co-rotinas.

4.1 Abordagem teórica sobre Multithreading e co-rotinas

Multithreading refere-se a capacidade do sistema operacional e/ou do hardware de suportar diversas linhas de execução, chamadas de *threads*. Cada *thread* contém um contexto que inclui instruções, variáveis, uma pilha de execução, e um bloco de controle. O suporte de diversar unidades de execução se dá por meio de paralelismo real ou aparente. O primeiro tipo ocorre quando diferentes *threads* executam em diferentes processadores, núcleos, ou em processadores superescalares com multiplos bancos de registradores. O segundo tipo ocorre quando as *threads* intercalam o uso da CPU, por meio da gerência de um escalonador.

Em *multithreading*, o escalonador faz uso de um artifício chamado preempção. Isso significa que uma thread em excução pode ser interrompida, após qualquer instrução, para ceder a CPU a outra *thread*. Esta técnica permite que a CPU seja usada por todos, sem intervenção do programador. Ou seja, a alternância de uso da CPU entre as *threads* ocorre de forma independente ao código implementado por elas.

Quando diferentes linhas de execução compartilham dados, o uso de preempção pode causar problemas de integridade destes dados. Este problema, conhecido como condição de corrida, ocorre quando a preempção modifica a sequência de instruções de uma operação. Para permitir que a operação execute sem interrupções, são utilizadas primitivas que desabilitam a preempção temporariamente, garantindo a exclusão mútua de tais regiões. Quando diversas *threads* estão trabalhando em conjunto, as vezes é preciso garantir uma ordem de execução. Isso é garantido com o uso de primitivas de sincronização. Porém essas primitivas que gerenciam o uso concorrente de recurso são custosas. [6]

Rotinas coopertativas, ou co-rotinas, têm as mesmas características das *threads*, quando classificadas como completas [7, s. 2.4]. Porém elas cooperam no uso da CPU através de transferência explícita de controle. Com isso elimina-se a necessidade de preempção, e consequentemente de gerêcia do uso concorrente de recursos.

Co-rotinas podem ser classificadas de acordo com o tipo de transferência de controle: simétricas e assimétricas. Co-rotinas do primeiro tipo têm a capacidade de ceder o controle para outra co-rotina explicitamente nomeada. As assimétricas só podem ceder o controle para a co-rotina que lhes ativou e possuêm um comportamente semelhante ao comportamento de funções. [7]

4.2 TinyOS Threads

4.2.1 Modelo de threads do TinyOS

TOSThreads é uma biblioteca que permite programação com threads no TinyOS sem violar ou limitar o modelo de concorrência do sistema. O TinyOS executa em uma única thread — a thread do kernel — enquanto a aplicação executa em uma ou mais threads — nível de usuário. Em termos de escalonamento, o kernel tem prioridade máxima, ou seja, a aplicação só executa quando o núcleo do sistema está ocioso. Ele é responsável pelo escalonamento de todas as tarefas e execução das chamadas de sistemas.

Três tipos de contextos de execução passam a existir: tarefas, interrupções e threads. Tarefas e interrupções podem interromper threads de aplicação, mas não o contrário. Threads tem preempção entre elas e podem compartilhar variáveis, por isso é necessário o uso de primitivas de sincronização para o controle de seções críticas. As opções fornecidas são *mutex*, semáforos, barreiras, variáveis de condição, e contador bloqueante. Esta última foi desenvolvida especialmente para o TinyOS. Seu uso se dá de forma que a thread fica bloqueada até o contador atingir um número arbitrário, enquanto outras threads podem incrementar ou decrementar esta variável através de uma interface específica. O escalonador de threads utiliza uma política *Round-Robin* com um tempo padrão de 5 milisegundos. É ele que oferece toda a interface para manipulação de threads, como pausar, criar e destruir.

As threads podem ser estáticas ou dinâmicas. A diferença está no momento de criação da pilha e do bloco de controle da thread. Nas threads estáticas a criação é feita em tempo de compilação, enquanto nas threads dinâmicas a criação é feita em tempo de execução. O bloco de controle, também chamado de *Thread Control Block* (TCB), contém informações sobre a thread, como seu identificador, seu estado de execução, o valor dos registradores (para troca de contexto), entre outras[8]. A troca de contexto é feita por códigos específicos para cada plataforma.

4.2.2 Implementação

A seguir descrevemos detalhes da implementação do *TOSThread*. Mostraremos a organização dos diretórios e os códigos fonte mais importantes.

Organização dos diretórios: O diretório raiz do *TOSThread* é */opt/tinyos-2.1.1/tos/lib/tosthreads/*. Abaixo descrevo sua estrutura básica de subdiretórios e as respectivas descrições[2]:

chips: Código específico de chips.

interfaces: Interfaces do sistema.

lib: Extensões e subsistemas.

- **net:** Protocolos de rede (protocolos *multihop*).
- **printf:** Imprime pequenas mensagens através da porta serial (para depuração).
- **serial:** Comunicação serial.

platforms: Código específico de plataformas.

sensorboards: Drivers para placas de sensoreamento.

system: Componentes do sistema.

types: Tipos de dado do sistema (arquivos header).

Sequência de Boot: Na inicialização do *TinyOS* com threads, primeiro há um encapsulamento da thread principal. Depois o curso original é tomado. A função *main()* está implementada em *system/RealMainImplP.nc*. A partir dela, o escalonador de threads é chamado através de um signal.

[2]Todos os arquivos serão referenciados a partir do diretório raiz */opt/tinyos-2.1.1/tos/lib/tosthreads/*. i.e. *types/thread.h*

```
module RealMainImplP {
    provides interface Boot as ThreadSchedulerBoot;}
implementation {
    int main() @C() @spontaneous() {
        atomic signal ThreadSchedulerBoot.booted();}
}
```

O escalonador de threads, implementado em *TinyThreadSchedulerP.nc* encapsula a atual linha de execução como a thread do kernel. A partir de então, o curso normal de inicialização é executado.

```
event void ThreadSchedulerBoot.booted() {
    num_runnable_threads = 0;
    //Pega as informacoes da thread principal, seu ID.
    tos_thread = call ThreadInfo.get[TOSTHREAD_TOS_THREAD_ID]();
    tos_thread->id = TOSTHREAD_TOS_THREAD_ID;
    //Insere a thread principal na fila de threads prontas.
    call ThreadQueue.init(&ready_queue);

    current_thread = tos_thread;
    current_thread->state = TOSTHREAD_STATE_ACTIVE;
    current_thread->init_block = NULL;
    signal TinyOSBoot.booted();
}
```

Na fase final do *boot*, é feita a inicialização do hardware, do escalonador de tarefas, dos componentes específicos da plataforma, e de todos os componentes que se ligaram a *SoftwareInit*. É então sinalizado que o *boot* terminou, permitindo que o compontente do usuário execute. Por ultimo, o kernel passa o controle para o escalonador de tarefas.

```
void TinyOSBoot.booted() {
    atomic {
        //Inicializa hardware
        platform_bootstrap();
        call TaskScheduler.init();
        call PlatformInit.init();
        //Executa tarefas postas pela funcao a cima
        while (call TaskScheduler.runNextTask());
        call SoftwareInit.init();
        //Executa tarefas postas pela funcao a cima
        while (call TaskScheduler.runNextTask());
    }
    __nesc_enable_interrupt();
    //Sinaliza boot para o usuario
    signal Boot.booted();
    call TaskScheduler.taskLoop();
}
```

No escalonador de tarefas, quando não houver mais *tasks* para executar, o controle é passado para o escalonador de threads.

```
command void TaskScheduler.taskLoop() {
    for (;;) {
        uint8_t nextTask;

        atomic {
            while((nextTask = popTask()) == NO_TASK) {
                call ThreadScheduler.suspendCurrentThread();
            }
        }
        signal TaskBasic.runTask[nextTask]();
    }
}
```

***types/thread.h*:** Este arquivo contém os tipos de dados e constantes excenciais para threads. A seguir estão listados esses dados, e seus respectivos códigos. Estados que uma thread pode assumir, como ativo, inativo, pronto e suspenso.

```
enum {
    TOSTHREAD_STATE_INACTIVE = 0,   //This thread is inactive and
                                    //cannot be run until started
    TOSTHREAD_STATE_ACTIVE = 1,     //This thread is currently running
                                    //on the cpu
    TOSTHREAD_STATE_READY = 2,      //This thread is not currently running,
                                    //but is not blocked and has work to do
    TOSTHREAD_STATE_SUSPENDED = 3, //This thread has been suspended by a
                                    //system call (i.e. blocked)
};
```

Constantes que controlam a quantidade máxima de threads, e o periodo de preempção. Estrutura da thread que contém dados como identificador, ponteiro para pilha, estado, ponteiro para função, registradores.

```
struct thread {
volatile struct thread* next_thread;
    //Pointer to next thread for use in queues when blocked
thread_id_t id;
    //id of this thread for use by the thread scheduler
init_block_t* init_block;
    //Pointer to an initialization block from which this thread was spawned
stack_ptr_t stack_ptr;
    //Pointer to this threads stack
volatile uint8_t state;
    //Current state the thread is in
volatile uint8_t mutex_count;
    //A reference count of the number of mutexes held by this thread
uint8_t joinedOnMe[(TOSTHREAD_MAX_NUM_THREADS - 1) / 8 + 1];
```

```
    //Bitmask of threads waiting for me to finish
void (*start_ptr)(void*);
    //Pointer to the start function of this thread
void* start_arg_ptr;
    //Pointer to the argument passed as a parameter to the start
    //function of this thread
syscall_t* syscall;
    //Pointer to an instance of a system call
thread_regs_t regs;
    //Contents of the GPRs stored when doing a context switch
};
```

Estrutura para controle de chamadas de sistema. Contém seu identificador, qual thread está executando, ponteiro para função que a implementa.

```
struct syscall {
struct syscall* next_call;
    //Pointer to next system call for use in syscall queues when
    //blocking on them
syscall_id_t id;
    //client id of this system call for the particular syscall_queue
    //within which it is being held
thread_t* thread;
    //Pointer back to the thread with which this system call is associated
void (*syscall_ptr)(struct syscall*);
    //Pointer to the the function that actually performs the system call
void* params;
    //Pointer to a set of parameters passed to the system call once it is
    //running in task context
};
```

***interfaces/Thread.nc*:** Contém os comandos de gerênciamento da thread e um evento para executá-la.

```
interface Thread {
    command error_t start(void* arg);
    command error_t stop();
    command error_t pause();
    command error_t resume();
    command error_t sleep(uint32_t milli);
    event void run(void* arg);
    command error_t join();
}
```

***interfaces/ThreadInfo.nc*:** Contém um comando *get()* para receber as informações da thread.

```
interface ThreadInfo {
    async command error_t reset();
```

```
3     async command thread_t* get();
}
```

***interfaces/ThreadScheduler.nc*:** Contém os comandos para gerênciar todas as threads. Essas funções servem para pegar informações das threads, inicializá-las e trocar de contexto.

```
interface ThreadScheduler {
2     async command uint8_t currentThreadId();
      async command thread_t* currentThreadInfo();
4     async command thread_t* threadInfo(thread_id_t id);

6     command error_t initThread(thread_id_t id);
      command error_t startThread(thread_id_t id);
8     command error_t stopThread(thread_id_t id);

10    async command error_t suspendCurrentThread();
      async command error_t interruptCurrentThread();
12
      async command error_t wakeupThread(thread_id_t id);
14    async command error_t joinThread(thread_id_t id);
}
```

***system/ThreadInfoP.nc*:** Contém o vetor que representa a pilha, as informações da thread, como visto em 4.2.2 e a função que sinaliza a execução.

```
1  generic module ThreadInfoP(uint16_t stack_size, uint8_t thread_id) {
   provides {
3     interface Init; // Para Inicializar as informacoes
      interface ThreadInfo; // Para exportar as Informacoes da thread
5     interface ThreadFunction; // Sinaliza para a thread executar
   }}
7
   implementation {
9    uint8_t stack[stack_size];
     thread_t thread_info;
11
     void run_thread(void* arg) __attribute__((noinline)) {
13     signal ThreadFunction.signalThreadRun(arg);
     }
15
     error_t init() {
17     thread_info.next_thread = NULL;
       thread_info.id = thread_id;
19     thread_info.init_block = NULL;
       thread_info.stack_ptr = (stack_ptr_t)(STACK_TOP(stack, sizeof(stack)));
21     thread_info.state = TOSTHREAD_STATE_INACTIVE;
       thread_info.mutex_count = 0;
23     thread_info.start_ptr = run_thread;
```

```
    thread_info.start_arg_ptr = NULL;
25    thread_info.syscall = NULL;
    return SUCCESS;
27  }

29  ... Comandos de interface ...
}
```

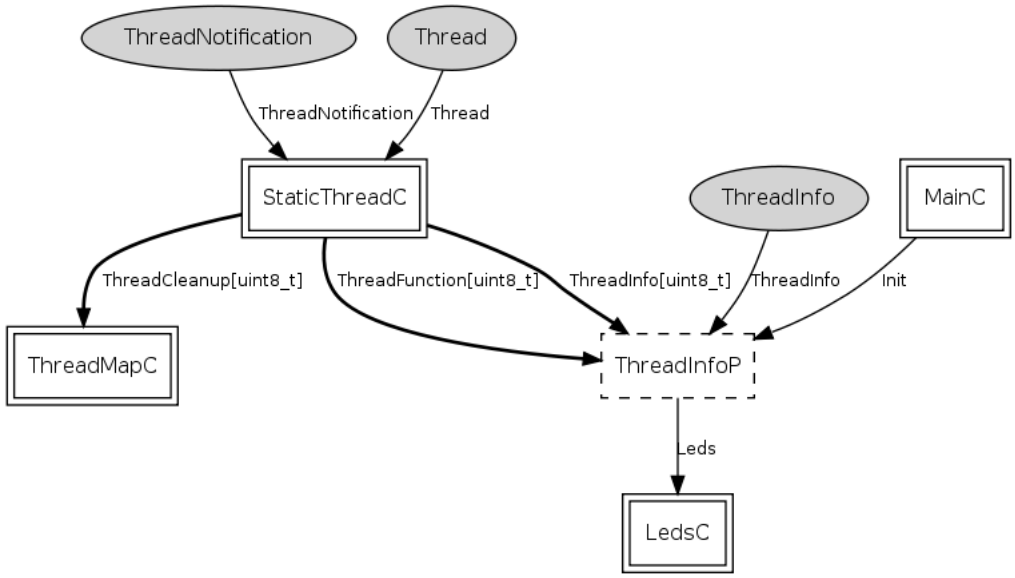
***system/StaticThreadP.nc*:** Tem como principal objetivo servir de interface entre uma thread específica e o escalonador. Por exemplo, se StaticThreadC recebe um comando de pausa, este é repassado para o escalonador executar. Também termina de inicializar a thread e sinaliza o evento *Thread.run*.

```
module StaticThreadP.nc { ... }
2 implementation {

4 error_t init(uint8_t id, void* arg) {
    error_t r1, r2;
6   thread_t* thread_info = call ThreadInfo.get[id]();
    thread_info->start_arg_ptr = arg;
8   thread_info->mutex_count = 0;
    thread_info->next_thread = NULL;
10  r1 = call ThreadInfo.reset[id]();
    r2 = call ThreadScheduler.initThread(id);
12  return ecombine(r1, r2);
}
14
event void ThreadFunction.signalThreadRun[uint8_t id](void *arg) {
16  signal Thread.run[id](arg);
}
18
command error_t Thread.start[uint8_t id](void* arg) {
20  atomic {
      if( init(id, arg) == SUCCESS ) {
22      error_t e = call ThreadScheduler.startThread(id);
        if(e == SUCCESS)
24        signal ThreadNotification.justCreated[id]();
        return e;
26    }
    }
28  return FAIL;

30  ... Continuacao da implementacao da interface thread ...
    ... Todos os comandos sao simplesmente passados para o ...
32  ... equivalente no ThreadScheduler ...
}
```

***system/ThreadC.nc*:** Esta configuração é a "interface" da thread com o usuário e com o escalonador. Primeiramente, é ela que prove a interface *interfaces/Thread.nc*, portanto o programador deve codificar o tratador do evento *Thread.run* e amarrá-lo a este componente. Em segundo lugar, conecta entre si todos os componentes importantes para o gerenciamento. Os principais são *system/MainC* para inicialização da thread no *boot* do sistema, *system/ThreadInfoP.nc* como visto em 4.2.2, e *system/StaticThreadC.nc* como visto em 4.2.2. A figura abaixo permite uma melhor visualização. As elipses são interfaces, os retângulos são componentes e as setas indicam qual interface liga os dois componentes.



***chips/atm128/chip_thread.h*:** Antes de expor as funções do escalonador de threads, é importante expor algumas macros de baixo nível que realizam a troca de contexto. Para guardar o contexto de hardware da thread, criaram a estrutura *thread_regs_t*.

```
typedef struct thread_regs {
    uint8_t status;
    uint8_t r0;
    ...
    uint8_t r31;
} thread_regs_t;
```

Existem também algumas macros para salvar e restaurar estes registradores.

```
#define SAVE_STATUS(t)                                          \
    __asm__("in %0,__SREG__ \n\t" : "=r" ((t)->regs.status) : );

//Save General Purpose Registers
#define SAVE_GPR(t)                                             \
    __asm__("mov %0,r0 \n\t" : "=r" ((t)->regs.r0) : );   \
    ...

//Save stack pointer
#define SAVE_STACK_PTR(t)                \
```

```
    __asm__("in %A0, __SP_L__\n\t"     \
    "in %B0, __SP_H__\n\t"             \
    :"=r"((t)->stack_ptr) : );

#define SAVE_TCB(t) \
   SAVE_GPR(t);       \
   SAVE_STATUS(t);    \
   SAVE_STACK_PTR(t)

//Definicao das macros de restauracao
...

#define SWITCH_CONTEXTS(from, to) \
   SAVE_TCB(from);                \
   RESTORE_TCB(to)
```

Por último, são definidas duas macros para preparação da thread.

```
#define SWAP_STACK_PTR(OLD, NEW) \
   __asm__("in %A0, __SP_L__\n\t in %B0, __SP_H__":"=r"(OLD):);\
   __asm__("out __SP_H__,%B0\n\t out __SP_L__,%A0"::"r"(NEW))

#define PREPARE_THREAD(t, thread_ptr)                          \
{
   uint16_t temp;                                              \
   SWAP_STACK_PTR(temp, (t)->stack_ptr);                       \
   __asm__("push %A0\n push %B0"::"r"(&(thread_ptr)));         \
   SWAP_STACK_PTR((t)->stack_ptr, temp);                       \
   SAVE_STATUS(t)                                              \
}
```

***system/TinyThreadSchedulerP.nc*:** Durante a inicialização do sistema muitas inicializações são feitas através da interface *Init* amarrada ao compontente *MainC*. Isso ocorre com a *system/StaticThreadP.nc*. Como visto acima, durante a execução desta função, o escalonador é chamado através do comando a seguir.

```
command error_t ThreadScheduler.initThread(uint8_t id) {
    thread_t* t = (call ThreadInfo.get[id]());
    t->state = TOSTHREAD_STATE_INACTIVE;
    t->init_block = current_thread->init_block;
    call BitArrayUtils.clrArray(t->joinedOnMe, sizeof(t->joinedOnMe));
    PREPARE_THREAD(t, threadWrapper);
        //O codigo abaixo e' definicao da macro PREPARE_THREAD,
        //inserido aqui para facilitar o entendimento do codigo.
        //uint16_t temp;                                        \
        //SWAP_STACK_PTR(temp, (t)->stack_ptr);                 \
        //__asm__("push %A0\n push %B0"::"r"(&(threadWrapper)));   \
        //SWAP_STACK_PTR((t)->stack_ptr, temp);                 \
```

```
        //SAVE_STATUS(t)
    return SUCCESS;
}
```

É importante notar que na macro *PREPARE_THREAD()*, o endereço da função *threadWrapper* está sendo empilhado na pilha da thread. Está função encapsula a chamada para a execução da thread.

```
void threadWrapper() __attribute__((naked, noinline)) {
    thread_t* t;
    atomic t = current_thread;

    __nesc_enable_interrupt();
    (*(t->start_ptr))(t->start_arg_ptr);

    atomic {
        stop(t);
        sleepWhileIdle();
        scheduleNextThread();
        restoreThread();
    }
}
```

No laço principal do escalonador de tarefas, quando não há mais nada para executar, a thread atual é suspensa. Com isso o controle é passado para o escalonador de threads através do comando *suspendCurrentThread()*. Na demostração de código abaixo, algumas chamadas a funções são substituídas pelo seus corpos, para facilitar o entendimento.

```
async command error_t ThreadScheduler.suspendCurrentThread() {
    atomic {
        if(current_thread->state == TOSTHREAD_STATE_ACTIVE) {
            current_thread->state = TOSTHREAD_STATE_SUSPENDED;
            //suspend(current_thread);
            #ifdef TOSTHREADS_TIMER_OPTIMIZATION
                num_runnable_threads--;
                post alarmTask();
            #endif
            sleepWhileIdle();
            //interrupt(current_thread);
            yielding_thread = current_thread;
            //scheduleNextThread();
            if(tos_thread->state == TOSTHREAD_STATE_READY)
                current_thread = tos_thread;
            else
                current_thread = call ThreadQueue.dequeue(&ready_queue);

            current_thread->state = TOSTHREAD_STATE_ACTIVE;
            //fim scheduleNextThread();
```

```
22              if(current_thread != yielding_thread) {
                    //switchThreads();
24                  void switchThreads() __attribute__((noinline)) {
                        SWITCH_CONTEXTS(yielding_thread, current_thread);
26                  }
                    //fim switchThreads();
28              }
                //fim interrupt(...)
30              //fim suspend(current_thread);
                return SUCCESS;
32          }
            return FAIL;
34      }
}
```

É muito importante notar que a função *switchThreads()* não é *inline*. Isso significa que os valores dos registradores serão empilhados. Haverá então uma troca de contexto e o registrador SP apontará para a pilha da nova thread. Por último, a função *switchThreads()* retornará para o endereço que está no topo da nova pilha. Este novo endereço, como visto acima, aponta para a função *threadWrapper()*. Esta por sua vez, através de uma função e duas sinalizações executa a thread.

Trocas de Contextos acontecem por três motivos diferentes: ocorrência de uma interrupção, termino do tempo de execução da thread, ou chamadas bloqueantes ao sistema. Para implementar o primeiro caso, é inserida a função *postAmble* ao final de todas as rotinas de processamento de interrupção. Esta função verifica se foi postada uma nova tarefa, e caso positivo, o controle é passado para o *kernel*. Caso contrário, a thread continua a executar logo após o termino do tratador de interrupção. Para implementar o segundo caso, é utilizado um temporizador que provoca uma interrupção ao final de cada *timeslice*. Esta posta uma tarefa, forçando o *kernel* a assumir o controle e escalonar a próxima thread.

As chamadas de sistema transformam os serviços de duas fases dos TinyOS em chamadas bloqueantes. Para o programador isso facilita a programação pois torna o fluxo do código continuo. Para permitir isto, essa nova estrutura posta uma tarefa que executará a primeira fase do serviço (*command*/*call*), suspende a thread e a acorda o *kernel* para escalonar outra thread. Ao chegar o evento (através de uma tarefa), a *syscall* acorda a thread e lhe repassa o dado.

4.3 Implementação de co-rotinas para o TinyOS

O modelo que decidimos implementar foi um descrito por Ana Moura em sua tese de doutorado[7, s. 6.2]. Neste modelo existe uma co-rotina principal que é responsável por escalonar as outras co-rotinas.

Nossa implementação utilizou como base a extensão *TOSThreads*, vista na seção 4.2. O primeiro passo foi criar uma cópia do diretório desta extensão e um novo *target* referente a

este diretório para o *make* do TinyOS.

Na implementação do *TOSThreads* existem dois casos em que ocorre preempção: termino do *timeslice* e acontecimento de uma interrupção de hardware. Portanto foi preciso modificar esses dois casos. A primeira alteração foi retirar o limite de tempo de execução de cada thread. Para isso o temporizador responsável por essa contagem foi desabilitado. A segunda alteração foi criar um novo tipo de interrupção, que chamamos de interrupção curta. Originalmente, no *TOSThreads*, quando o tratador de interrupção postava uma tarefa, o kernel assumia o controle, executava a tarefa e escalonava a próxima thread da fila. Na nossa implementação, após o kernel executar a tarefa, a thread que foi originalmente interrompida volta a executar. Para isso, foi criado um novo comando no escalonador de threads: *brieflyInterruptCurrentThread()*

```
async command error_t ThreadScheduler.brieflyInterruptCurrentThread() {
    atomic {
        if(current_thread->state == TOSTHREAD_STATE_ACTIVE) {
            briefly_interrupted_thread = current_thread;
            briefly_interrupted_thread->state =
                TOSTHREAD_STATE_BRIEFLYINTERRUPTED;
            interrupt(current_thread);
            return SUCCESS;
        }
        return FAIL;
    }
}

/* schedule_next_thread()
 * This routine does the job of deciding which thread should run next.
 * Should be complete as is.  Add functionality to getNextThreadId()
 * if you need to change the actual scheduling policy.
 */
    void scheduleNextThread() {
        if(tos_thread->state == TOSTHREAD_STATE_READY)
            current_thread = tos_thread;
        else if (briefly_interrupted_thread != NULL)
        {
            current_thread = briefly_interrupted_thread;
            briefly_interrupted_thread = NULL;
        }
        else
            current_thread = call ThreadQueue.dequeue(&ready_queue);

        current_thread->state = TOSTHREAD_STATE_ACTIVE;
    }
```

Uma vez excluída a preempção, o próximo passo foi modificar a interface da thread para permitir passagem de controle ao escalonador. Para isso, foi criado o comando *yield()*. É importante notar que este comando pode ser chamado em qualquer ponto do programa, porém

só deve ser chamado dentro da thread ao qual o comando se refere.

```
//Arquivo: interfaces/Thread.nc
interface Thread {
    ...
    command error_t yield();
    ...
}

//Arquivo: system/StaticThreadP.nc
module StaticThreadP {
    ...
    command error_t Thread.yield[uint8_t id]() {
        return call ThreadScheduler.interruptCurrentThread();
    }
    ...
}
```

4.4 Experimentos e resultados obtidos

Com o objetivo de comparar o desempenho da implementação de co-rotinas com a biblioteca *TOSThread*, foram desenvolvidas duas aplicações para implementar o problema do produtor-consumidor. Uma utilizando *threads* (anexo B.3), e outra utilizando co-rotinas (anexo B.4). Foram utilizados uma linha de execução para o produtor, e outra para o consumidor, e um *buffer* de tamanho único. Para simular o tempo de processamento da produção e do consumo de uma unidade, foi implementado um laço de cem iterações, onde cada passo executa uma operação aritmética. Após consumir mil produtos, uma nova linha de execução é ativada, para calcular o tempo de execução.

Para variar a carga, foram utilizadas diferentes operações aritiméticas. O tempo de execução foi medido em uma plataforma *MicaZ*, utilizando o temporizador *Counter<TMicro,uint32_t>*, utilizando uma precisão de microsegundos. Os valores medidos não variaram mais de uma unidade entre diferentes execuções.

Operação	x += 1	x *= 2	x *= 3	x *= 5
Threads	380000	490000	530000	563000
Co-rotinas	151000	252000	289000	314000

5 Conclusões

As redes de sensores sem fio podem ser aplicadas em diversas áreas, por exemplo, monitoramento de oscilações e movimentos de pontes, observação de vulcões ativos, previsão de incêndio em florestas, entre outras. Muitas dessas aplicações podem atingir alta complexidade, exigindo a construção de algoritmos robustos, como roteamento de pacotes diferenciado. Os escalonadores desenvolvidos neste trabalho poderão ajudar os desenvolvedores dessas aplicações complexas, oferecendo maior flexibilidade no projeto das soluções, como, por exemplo,

a possibilidade de priorizar certas atividades da aplicação (comunicação via rádio ou serial, sensoriamento, etc.). Porém é preciso analisar se o ganho em flexibilidade, oferecido pelo escalonador, irá compensar o *overhead* gerado. Pretendemos realizar ainda outros experimentos com os escalonadores desenvolvidos, considerando diferentes tipos de aplicações.

Sem um fluxo contínuo de execução, sobre a perspectiva do programador, as aplicações grandes ficam difíceis de implementar e entender. O modelo de *threads* oferecido no TinyOS 2.1.X[8] facilita este problema. Entretanto, por ser um modelo preemptivo, o custo de gerência das threads pode implicar em queda de desempenho das aplicações. Com a implementação de um mecanismo de cooperação baseado em co-rotinas pretendemos oferecer uma alternativa a mais para o programador.

6 Trabalhos Futuros

7 Apêndice

A Extensão para o Simulador TOSSIM

Para que o escalonador funcione corretamente no simulador TOSSIM é preciso adicionar funções que lidam com eventos no simulador. Essas funções foram retiradas do arquivo *tinyOS-root-dir/tos/lib/tossim/SimSchedulerBasicP.nc*. Primeiro é preciso alterar a implementação das interfaces de tarefa e da interface *Scheduler*, criando as funções e variáveis do código abaixo:

```
1  bool sim_scheduler_event_pending = FALSE;
   sim_event_t sim_scheduler_event;
3
   int sim_config_task_latency() {return 100;}
5
   void sim_scheduler_submit_event() {
7    if (sim_scheduler_event_pending == FALSE) {
       sim_scheduler_event.time = sim_time() + sim_config_task_latency();
9      sim_queue_insert(&sim_scheduler_event);
       sim_scheduler_event_pending = TRUE;
11   }
   }
13
   void sim_scheduler_event_handle(sim_event_t* e) {
15   sim_scheduler_event_pending = FALSE;
     if (call Scheduler.runNextTask())
17     sim_scheduler_submit_event();
   }
19
   void sim_scheduler_event_init(sim_event_t* e) {
21   e->mote = sim_node();
```

```
    e->force = 0;
    e->data = NULL;
    e->handle = sim_scheduler_event_handle;
    e->cleanup = sim_queue_cleanup_none;
  }
```

Depois, no comando *init()* deve-se adicionar:

```
  sim_scheduler_event_pending = FALSE;
  sim_scheduler_event_init(&sim_scheduler_event);
```

E, por último, nos comandos *postTask()*, deve-se adicionar:

```
  sim_scheduler_submit_event();
```

B Anexos

B.1 Blink

B.1.1 BlinkC.nc:

```
#include "Timer.h"

module BlinkC @safe()
{
  uses interface Timer<TMilli> as Timer0;
  uses interface Timer<TMilli> as Timer1;
  uses interface Timer<TMilli> as Timer2;
  uses interface Leds;
  uses interface Boot;
}
implementation
{
  event void Boot.booted()
  {
    call Timer0.startPeriodic( 250 );
    call Timer1.startPeriodic( 500 );
    call Timer2.startPeriodic( 1000 );

    post tarefa();
  }

  event void Timer0.fired()
  {
    call Leds.led0Toggle();
  }

  event void Timer1.fired()
  {
```

```
29      call Leds.led1Toggle();
    }
31
    event void Timer2.fired()
33  {
      call Leds.led2Toggle();
35  }

37  task void tarefa()
    {
39      dbg("BlinkC", "tarefa\n");
    }
41  }
```

B.1.2 BlinkAppC.nc:

```
1  configuration BlinkAppC
   {}
3  implementation
   {
5    components MainC, BlinkC, LedsC;
     components new TimerMilliC() as Timer0;
7    components new TimerMilliC() as Timer1;
     components new TimerMilliC() as Timer2;
9
     BlinkC.Boot -> MainC.Boot;
11   BlinkC.Timer0 -> Timer0;
     BlinkC.Timer1 -> Timer1;
13   BlinkC.Timer2 -> Timer2;
     BlinkC.Leds -> LedsC.Leds;
15 }
```

B.2 Aplicação de Teste do Escalonador com Prioridades

B.2.1 aplicacaoTesteC.nc:

```

2  /**
    * Implementa aplicativo de teste do Scheduler de prioridade
4   **/

6  #include "MsgSerial.h"
   #include "Timer.h"
8  #include "printf.h"

10 module aplicacaoTesteC @safe()
   {
```

```
12      uses interface Boot;
        uses interface Leds;
14      uses interface TaskPrioridade as Tarefa1;
        uses interface TaskPrioridade as Tarefa2;
16      uses interface TaskPrioridade as Tarefa3;
        uses interface TaskPrioridade as Tarefa4;
18      uses interface TaskPrioridade as Tarefa5;
        // ...
20      uses interface TaskPrioridade as Tarefa98;
        uses interface TaskPrioridade as Tarefa99;
22
        uses interface Counter<TMicro, uint32_t> as Timer1;
24  }
    implementation
26  {
        /* Variaveis */
28      unsigned int t1;
        bool over;
30
        async event void Timer1.overflow()
32      {
            over = TRUE;
34      }

36      /* Boot
        */
38      event void Boot.booted()
        {
40          over = FALSE;
            t1 = call Timer1.get();
42          printf("tempo inicial: %u\n", t1);
            printfflush();
44
            call Tarefa1.postTask(20);
46
            call Tarefa2.postTask(10);
48          call Tarefa3.postTask(10);
            // ...
50          call Tarefa98.postTask(10);
            call Tarefa99.postTask(10);
52      }

54      /* Tarefas
        */
56      event void Tarefa1.runTask()
        {
58          uint16_t i = 0;
```

```
        uint16_t k = 1;
60      for (i = 0; i < 65000; i++)
        {
62          k = k * 2;
        }
64      // Calculo do tempo de execucao
        t1 = call Timer1.get();
66      printf("tempo final: %u\n", t1);
        if (over == TRUE)
68          printf("Ocorreu Overflow\n");
        printfflush();
70  }
    event void Tarefa2.runTask()
72  {
        uint16_t i = 0;
74      uint16_t k = 1;
        for (i = 0; i < 65000; i++)
76      {
            k = k * 2;
78      }
    }
80
    // ...
82
    event void Tarefa99.runTask()
84  {
        uint16_t i = 0;
86      uint16_t k = 1;
        for (i = 0; i < 65000; i++)
88      {
            k = k * 2;
90      }
    }
92 }
```

B.2.2 aplicacaoTesteAppC.nc:

```
/**
2 * Aplicativo de teste do Scheduler de prioridade
 *
4 **/

6 #include "MsgSerial.h"
 #include "Timer.h"
8 #include "printf.h"

10 configuration aplicacaoTesteAppC
```

```
{
}
implementation
{
  components MainC, aplicacaoTesteC, LedsC, TinySchedulerC;
  components CounterMicro32C as Timer1;

  aplicacaoTesteC.Timer1 -> Timer1;

  aplicacaoTesteC-> MainC.Boot;
  aplicacaoTesteC.Leds -> LedsC;

  aplicacaoTesteC.Tarefa1->
  TinySchedulerC.TaskPrioridade[unique("TinySchedulerC.TaskPrioridade")];
  aplicacaoTesteC.Tarefa2->
  TinySchedulerC.TaskPrioridade[unique("TinySchedulerC.TaskPrioridade")];
  aplicacaoTesteC.Tarefa3->
  TinySchedulerC.TaskPrioridade[unique("TinySchedulerC.TaskPrioridade")];
  aplicacaoTesteC.Tarefa4->
  TinySchedulerC.TaskPrioridade[unique("TinySchedulerC.TaskPrioridade")];
  aplicacaoTesteC.Tarefa5->
  TinySchedulerC.TaskPrioridade[unique("TinySchedulerC.TaskPrioridade")];

// ...

  aplicacaoTesteC.Tarefa98->
  TinySchedulerC.TaskPrioridade[unique("TinySchedulerC.TaskPrioridade")];
  aplicacaoTesteC.Tarefa99->
  TinySchedulerC.TaskPrioridade[unique("TinySchedulerC.TaskPrioridade")];
}
```

B.3 Aplicação de Teste de Threads

B.3.1 BenchmarkAppC.nc:

```
configuration BenchmarkAppC{
}
implementation {
  components MainC, BenchmarkC,  LedsC;
  components new ThreadC(800) as Produtor;
  components new ThreadC(800) as Consumidor;
  components new ThreadC(800) as SerialSender;

  components CounterMicro32C as Timer;
  BenchmarkC.Timer -> Timer;

```

```
13    components BlockingSerialActiveMessageC;
      components new BlockingSerialAMSenderC(228);
15    BenchmarkC.AMControl -> BlockingSerialActiveMessageC;
      BenchmarkC.BlockingAMSend -> BlockingSerialAMSenderC;
17    BenchmarkC.Packet -> BlockingSerialAMSenderC;

19    MainC.Boot <- BenchmarkC;

21    BenchmarkC.Produtor -> Produtor;
      BenchmarkC.Consumidor -> Consumidor;
23    BenchmarkC.SerialSender -> SerialSender;

25    BenchmarkC.Leds -> LedsC;

27    components MutexC;
      BenchmarkC.Mutex -> MutexC;
29
      components SemaphoreC;
31    BenchmarkC.Semaphore -> SemaphoreC;
  }
```

B.3.2 BenchmarkC.nc:

```
1 #include "mutex.h"
  #include "semaphore.h"
3
  module BenchmarkC {
5   uses {
      interface Boot;
7     interface Thread as Produtor;
      interface Thread as Consumidor;
9     interface Thread as SerialSender;
      interface Leds;
11
      interface BlockingStdControl as AMControl;
13    interface BlockingAMSend;
      interface Packet;
15
      interface Counter<TMicro, uint32_t> as Timer;
17
      interface Mutex;
19    interface Semaphore;
    }
21 }

23 implementation {
      uint32_t t1;
```

```
25      uint32_t * tempo;
        uint8_t buffer;
27      mutex_t mutex_buffer;
        semaphore_t sem_produto, sem_termino, sem_buffer_cheio;
29
        async event void Timer.overflow()
31      {}

33      event void Boot.booted() {
          buffer = 0;
35        t1 = call Timer.get();

37        call Mutex.init(&mutex_buffer);
          call Semaphore.reset(&sem_produto, 0);
39        call Semaphore.reset(&sem_buffer_cheio, 1);
          call Semaphore.reset(&sem_termino, 0);
41
          call Produtor.start(NULL);
43        call Consumidor.start(NULL);
          call SerialSender.start(NULL);
45      }

47      event void SerialSender.run(void* arg)
        {
49          message_t msg;

51          call Semaphore.acquire(&sem_termino);

53          t1 = call Timer.get() - t1;

55          while( call AMControl.start() != SUCCESS );

57          tempo = call Packet.getPayload(&msg, sizeof(uint32_t));
            (*tempo) = t1;
59
            while( call BlockingAMSend.send(AM_BROADCAST_ADDR,
61                          &msg, sizeof(uint32_t)) != SUCCESS );

63          //Para conferir a corretude da execucao
            (*tempo) = buffer;
65          while( call BlockingAMSend.send(AM_BROADCAST_ADDR,
                            &msg, sizeof(uint32_t)) != SUCCESS );
67
            call Leds.led1Toggle();
69      }


71
```

```
    event void Produtor.run(void* arg) {
73      uint16_t counter = 1;
        uint16_t num_prods, j;
75
        for (num_prods = 0; num_prods < 1000; num_prods++)
77      {
            call Semaphore.acquire(&sem_buffer_cheio);
79
            //Tempo simulado para criar um produto
81          for (j = 0; j < 100; j++)
                counter *= 3;
83
            call Mutex.lock(&mutex_buffer);
85              buffer = counter;
            call Mutex.unlock(&mutex_buffer);
87
            call Semaphore.release(&sem_produto);
89      }
    }
91
    event void Consumidor.run(void* arg) {
93      uint16_t num_prods, j;
        uint16_t counter = 0;
95
        for (num_prods = 0; num_prods < 1000; num_prods++)
97      {
            call Semaphore.acquire(&sem_produto);
99
            call Mutex.lock(&mutex_buffer);
101             counter = buffer;
            call Mutex.unlock(&mutex_buffer);
103
            //Tempo simulado para consumir produto
105         for (j = 0; j < 100; j++)
                counter *= 3;
107
            call Semaphore.release(&sem_buffer_cheio);
109     }
        call Semaphore.release(&sem_termino);
111 }

113 }
```

B.4 Aplicação de Teste de Co-rotinas

B.4.1 BenchmarkAppC.nc:

```
configuration BenchmarkAppC{
}
implementation {
  components MainC, BenchmarkC,  LedsC;
  components new ThreadC(800) as Produtor;
  components new ThreadC(800) as Consumidor;
  components new ThreadC(800) as SerialSender;

  components CounterMicro32C as Timer;
  BenchmarkC.Timer -> Timer;


  components BlockingSerialActiveMessageC;
  components new BlockingSerialAMSenderC(228);
  BenchmarkC.AMControl -> BlockingSerialActiveMessageC;
  BenchmarkC.BlockingAMSend -> BlockingSerialAMSenderC;
  BenchmarkC.Packet -> BlockingSerialAMSenderC;

  MainC.Boot <- BenchmarkC;

  BenchmarkC.Produtor -> Produtor;
  BenchmarkC.Consumidor -> Consumidor;
  BenchmarkC.SerialSender -> SerialSender;

  BenchmarkC.Leds -> LedsC;


}
```

B.4.2 BenchmarkC.nc:

```
module BenchmarkC {
  uses {
    interface Boot;
    interface Thread as Produtor;
    interface Thread as Consumidor;
    interface Thread as SerialSender;
    interface Leds;

    interface BlockingStdControl as AMControl;
    interface BlockingAMSend;
    interface Packet;

    interface Counter<TMicro, uint32_t> as Timer;
  }
}

implementation {
```

```
18      uint32_t t1;
        uint32_t * tempo;
20      uint8_t buffer;

22      async event void Timer.overflow()
        {}
24
        event void Boot.booted() {
26        buffer = 0;
          t1 = call Timer.get();
28
          call Produtor.start(NULL);
30        call Consumidor.start(NULL);
        }
32
        event void SerialSender.run(void* arg)
34      {
            message_t msg;
36
            t1 = call Timer.get() - t1;
38
            while( call AMControl.start() != SUCCESS );
40
            tempo = call Packet.getPayload(&msg, sizeof(uint32_t));
42          (*tempo) = t1;

44          while( call BlockingAMSend.send(AM_BROADCAST_ADDR,
                        &msg, sizeof(uint32_t)) != SUCCESS );
46
            //Para conferir a corretude da execucao
48          (*tempo) = buffer;
            while( call BlockingAMSend.send(AM_BROADCAST_ADDR,
50                      &msg, sizeof(uint32_t)) != SUCCESS );
        }
52

54      event void Produtor.run(void* arg) {
            uint16_t counter = 1;
56          uint16_t num_prods, j;

58          for (num_prods = 0; num_prods < 1000; num_prods++)
            {
60              //Tempo simulado para criar um produto
                for (j = 0; j < 100; j++)
62                  counter *= 3;

64              buffer = counter;
```

```
66              call Produtor.yield();
            }
68      }

70      event void Consumidor.run(void* arg) {
            uint16_t num_prods, j;
72          uint16_t counter = 0;

74          for (num_prods = 0; num_prods < 1000; num_prods++)
            {
76              counter = buffer;

78              //Tempo simulado para consumir produto
                for (j = 0; j < 100; j++)
80                  counter *= 3;

82              call Consumidor.yield();
            }
84          call SerialSender.start(NULL);
        }
86
}
```

Referências

[1] J. Hill, R. Szewczyk, A. Woo, S. Hollar, D. Culler, and K. Pister. System architecture directions for networked sensors. In *9th International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 93–104, New York, NY, USA, 2000. ACM Press.

[2] David Gay, Phil Levis, Rob von Behren, Matt Welsh, Eric Brewer, and David Culler. The nesC language: A holistic approach to networked embedded systems. *Proceedings of Programming Language Design and Implementation (PLDI)*, June 2003.

[3] Philip Levis and David Gay. *TinyOS Programming*. Cambridge University Press, 2009.

[4] Philip Levis and Cory Sharp. Tep106: Schedulers and tasks. http://www.tinyos.net/tinyos-2.1.0/doc/html/tep106.html.

[5] Philip Levis. Tep107: Tinyos 2.x boot sequence. http://www.tinyos.net/tinyos-2.1.0/doc/html/tep107.html.

[6] William Stallings. *Operating Systems: Internals and Design Principles*. Prentice Hall, 5a edition, 2004.

[7] Ana L. de Moura. *Revisitando co-rotinas*. PhD thesis, PUC-Rio, Rio de Janeiro, Brasil, 2004.

[8] Kevin Klues, Chieh-Jan Liang, Jeongyeup Paek, Razvan Musaloiu-E, Ramesh Govindan, Andreas Terzis, and Philip Levis. Tep134: The tosthreads thread library. http://www.tinyos.net/tinyos-2.1.0/doc/html/tep134.html.