4
Incluindo gerência cooperativa de tarefas no sistema operacional TinyOS

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 (Hill/00) — um dos sistemas operacionais mais usados na pesquisa com redes de sensores — prioriza fortemente o tratamento dessas restrições em detrimento da simplicidade oferecida para o desenvolvimento de aplicações. Entretanto, de acordo com Levis e Culler (Levis/02), para que as redes de sensores sejam de fato adotadas é preciso que elas sejam mais fáceis de usar. A experiência de outros pesquisadores (Cheong/03, Han/05, Kasten/05, Dunkels/05), e a nossa experiência particular desenvolvendo componentes TinyOS (Costa/05), confirmam a importância de propor uma interface de programação mais adequada.

Neste capítulo, descrevemos as características principais das redes de sensores e da arquitetura TinyOS, e apresentamos nossa proposta para incluir gerência cooperativa de tarefas usando co-rotinas dentro dessa arquitetura, com o objetivo de simplificar o seu modelo de programação (Rossetto/06).

4.1
Redes de sensores

Redes de sensores são formadas a partir de pequenos dispositivos que se comunicam através de um meio sem fio e são dispostos próximos a um fenômeno físico com a finalidade de monitorá-lo. Esses dispositivos são capazes de medir, processar e transmitir informações sobre o fenômeno físico, tais como temperatura, umidade e luminosidade (Culler/04, Loureiro/03, Akyildiz/02).

A capacidade de processamento efetiva nas redes de sensores é obtida somente com a agregação dos vários dispositivos que individualmente capturam os valores de interesse, e em conjunto transformam informações do mundo físico em dados que podem ser avaliados e usados para finalidades diversas. Tipicamente, os dispositivos processam os dados coletados localmente para extrair as características de interesse, armazenam a informação momentaneamente e usam a conexão via rede sem fio para transmitir a informação para os seus

vizinhos. Quando uma comunicação está sendo recebida via rádio, o atraso de uma fração de segundo pode resultar na perda de toda a transmissão. Além disso, os dispositivos participam de várias interações simultâneas com o mundo físico — sensoriamento, comunicação e roteamento de mensagens — o que os obriga a serem ágeis. Essas questões fazem com que a capacidade de tratar operações concorrentes seja um requisito fundamental.

As redes de sensores sem fio apresentam características que as distinguem das redes tradicionais, como, limitação de recursos computacionais, execução de aplicações específicas, requerimento de vida longa, presença de sensores que requerem acoplamento com o mundo físico e condições ambientais dinâmicas. Os sistemas para os dispositivos da rede devem usar de modo eficiente o processador e a memória, ao mesmo tempo em que devem garantir baixo consumo de energia. O ideal é que esses dispositivos sejam mantidos o máximo de tempo possível em modos de operação que minimizam o consumo de energia, e que sejam solicitados apenas quando algum evento do sistema precisa ser tratado. Como a capacidade de armazenamento interna é baixa, o armazenamento temporário de dados não é uma solução atrativa. A informação capturada pelos sensores ou recebida de outros nós da rede deve ser processada de forma rápida e o resultado retransmitido quando necessário.

Nesse cenário, o estilo de execução reativo, característico da programação dirigida a eventos, aparece como uma solução adequada. As aplicações são estruturadas como pequenos trechos de código não bloqueantes, os quais são executados como resposta à ocorrência de eventos, tais como recepção de mensagens, leitura dos sensores e eventos de temporização. Os dispositivos podem permanecer em estado de espera, com economia de energia, e serem ativados apenas quando um evento ocorrer. Essa é a abordagem adotada pelo sistema operacional TinyOS.

4.2 O sistema operacional TinyOS

O TinyOS implementa um modelo baseado em componentes, com interfaces de duas fases, comunicação baseada em eventos e um mecanismo para postergar a execução de chamadas de procedimentos. O TinyOS pode ser definido como um *framework* de programação para redes de sensores com um conjunto de componentes que permite construir um sistema operacional específico para cada aplicação. Um programa TinyOS consiste de um grafo de componentes de software que usam três abstrações de programação: *comandos*, *eventos* e *tarefas*.

Comandos são usados para requisitar serviços, por exemplo, o envio de

uma mensagem, e terminam imediatamente. Tipicamente, o código associado a um comando armazena os parâmetros de uma requisição e condicionalmente escalona uma tarefa para ser executada posteriormente.

Um *evento* sinaliza o término de um serviço, como por exemplo, o envio de uma mensagem, ou a ocorrência de um evento de hardware. Os componentes de baixo nível possuem tratadores conectados diretamente às interrupções de hardware (por exemplo, temporização ou recepção de mensagens). Um tratador de evento pode depositar informações no ambiente do componente, escalonar tarefas, sinalizar outros eventos ou chamar comandos.

Tarefas são as unidades básicas de execução e permitem postergar a execução das chamadas de procedimentos. O TinyOS executa uma única aplicação com duas linhas de execução: as *tarefas* e os tratadores de *eventos de interrupção*. Uma vez escalonada, uma tarefa executa até terminar, ou seja, não existe preempção entre as tarefas. Os tratadores de eventos de hardware são executados quando uma interrupção de hardware ocorre e também executam até terminar, mas podem interromper as tarefas e outros tratadores de interrupção.

Para garantir que a execução de uma tarefa não postergue indefinidamente a execução de outras tarefas, o código executado por elas deve ser curto, i.e., operações longas devem ser particionadas em várias tarefas. Além disso, como uma tarefa precisa terminar para que a próxima seja tratada, o código de uma tarefa não pode bloquear ou ficar em espera ocupada. O laço principal de uma aplicação TinyOS implementa um *escalonador de tarefas* responsável por escalonar as tarefas para execução sempre que o processador torna-se disponível. A política FIFO (*First In First Out*) é adotada como padrão.

4.2.1 A linguagem nesC

A linguagem *nesC* (Gay/03) — uma extensão da linguagem C — foi projetada para incorporar os conceitos e o modelo de execução do TinyOS. Os programas nesC são construídos a partir de um conjunto de *componentes* com *interfaces bi-direcionais* bem definidas. O comportamento de um componente é especificado em termos das interfaces que ele oferece ou usa. As interfaces oferecidas representam a funcionalidade provida pelo componente aos seus usuários, enquanto as interfaces usadas representam as funcionalidades de que o componente precisa para executar o seu trabalho.

As interfaces definem um canal de interação multi-funcional entre dois componentes: o *provedor* e o *usuário*. O provedor de uma interface deve implementar o conjunto de funções denominadas *comandos*; e o usuário de

uma interface deve implementar o conjunto de funções denominadas *eventos*. Como exemplo, a interface `SendMsg` define o comando `send` para enviar uma mensagem e o evento `sendDone` para sinalizar que a mensagem foi enviada:

```
interface SendMsg {
 command result_t send(uint16_t addr,uint8_t length,TOS_MsgPtr msg);
 event result_t sendDone(TOS_MsgPtr msg,result_t success);
}
```

Essa estrutura permite representar interações entre os componentes, por exemplo, o registro de interesse em algum evento e a informação da função de retorno que deverá ser executada quando o evento acontecer. Todos os comandos TinyOS que codificam computações demoradas, como o envio de mensagens, devem ser não-bloqueantes: o término da execução deve ser sinalizado separadamente por meio de um evento. Os componentes que usam esses comandos devem definir tratadores para os eventos que poderão ser sinalizados em resposta aos comandos chamados. No caso da interface `SendMsg`, um componente não pode chamar o comando `send` se ele não implementar o evento `sendDone`.

A linguagem nesC define dois tipos fundamentais de componentes: `módulo` (*module*) e `configuração` (*configuration*). Os módulos provêem o código da aplicação implementando as interfaces, enquanto as configurações são usadas para conectar outros componentes através das interfaces oferecidas e usadas por eles. Toda aplicação nesC é descrita através de um componente de configuração principal que conecta os demais componentes.

Uma vez que as tarefas e os tratadores de interrupção podem acessar concorrentemente o estado de um componente, os programas nesC são susceptíveis a *condições de corrida*. A linguagem nesC permite duas opções para evitar o acesso inconsistente às variáveis compartilhadas: tratar o acesso à seção crítica somente dentro do código de uma tarefa, ou usar *seções atômicas*. Seções atômicas são pequenos trechos de código definidos através da construção `atomic{ }`, como exemplificado a seguir:

```
atomic {
  for (i=0; i<size; i++)
     sum += (rdata[i] >> 7);
}
```

Nesse exemplo, uma sentença atômica é criada para proteger o acesso ao vetor de dados `rdata`. A linguagem nesC garante que esse trecho de código será executado atomicamente, desabilitando as interrupções.

Comandos e eventos executados como parte de um tratador de interrupção devem ser explicitamente marcados com o identificador `async`. O compilador nesC inclui um mecanismo para detectar possíveis condições de corrida

dentro do código da aplicação e avisar o programador. A informação associada ao identificador `async` é usada para facilitar a detecção de variáveis que poderão ser acessadas concorrentemente por dois tratadores de interrupção ou por um tratador de interrupção e uma tarefa.

4.2.2 Desenvolvendo aplicações para o TinyOS

Uma aplicação comum em redes de sensores consiste em capturar periodicamente os valores físicos monitorados pelos sensores (por exemplo, luminosidade ou temperatura), executar um processamento local sobre esses valores e enão decidir por enviá-los ou não para uma estação base (normalmente um nó conectado com a rede cabeada) usando um esquema de comunicação multisaltos. Para aprofundar a discussão sobre a arquitetura TinyOS, apresentamos a aplicação *Surge* como referência. Trata-se de uma aplicação nesC que solicita periodicamente a leitura dos sensores e usa a rede sem fio para transmitir os valores medidos para uma estação base. *Surge* é intencionalmente simples (não executa operações avançadas como a agregação de dados ou técnicas sofisticadas de roteamento de mensagens) e é frequentemente usada como exemplo na pesquisa com o TinyOS (Gay/03).

Os dois componentes principais da aplicação *Surge* são o módulo que implementa a aplicação — `SurgeM` — e o componente de configuração responsável por conectar os demais componentes — `Surge`. Neste trabalho, apresentamos uma versão simplificada desses componentes para manter o foco nos conceitos mais importantes para a nossa discussão.

A Figura 4.1 mostra o código principal da aplicação. O componente `SurgeM` declara as interfaces oferecidas e usadas por ele. O componente oferece a interface `StdControl` e usa as interfaces `ADC`, `Timer` e `SendMsg`. Isso significa que o componente `SurgeM` pode chamar qualquer um dos comandos oferecidos por essas interfaces e deve implementar os eventos sinalizados por elas.

A interface `StdControl` define os comandos para iniciar, executar e finalizar um componente TinyOS. Todos os componentes que requerem inicialização ou podem ser desligados e religados durante a execução da aplicação devem prover essa interface. A especificação da interface `StdControl` é mostrada a seguir:

```
interface StdControl {
  command result_t init();
  command result_t start();
  command result_t stop();
}
```

```
module SurgeM {
  provides { interface StdControl; }
  uses { interface ADC; interface Timer; interface SendMsg; }
}
implementation {
 TOS_Msg gMsgBuffer;
 uint16_t dest = TOS_BCAST_ADDR;
 norace uint16_t gSensorData;
 bool gSendBusy = FALSE;
 command result_t StdControl.init() {
   return SUCCESS;
 }
 command result_t StdControl.start() {
   return call Timer.start(TIMER_REPEAT, timer_rate);
 }
 command result_t StdControl.stop() {
   return call Timer.stop();
 }
 task void SendData() {
  SurgeMsg *pReading;
  pReading = (SurgeMsg *)((&gMsgBuffer)->data);
  pReading->reading = gSensorData;
  if(!(call SendMsg.send(dest, sizeof(SurgeMsg), &gMsgBuffer)))
    atomic gSendBusy = FALSE;
 }
 event result_t Timer.fired() {
   call ADC.getData();
 }
 async event result_t ADC.dataReady(uint16_t data) {
   atomic if (!gSendBusy) {
     gSendBusy = TRUE;
     gSensorData = data;
     post SendData();
  }
 }
 event result_t SendMsg.sendDone(TOS_MsgPtr pMsg, success) {
   atomic gSendBusy = FALSE;
   return success;
 }
}
```

Figura 4.1: Código principal da aplicação *Surge*

A interface `ADC` é usada para interagir com os sensores requisitando a leitura de valores físicos como, temperatura, luminosidade, ruído, etc.. Ela define o comando `getData` para requisitar uma leitura do sensor, e o evento `dataReady` para sinalizar que o valor requisitado está disponível. A descrição das operações principais da interface é mostrada abaixo:

```
interface ADC {
  async command result_t getData();
  async event result_t dataReady(uint16_t data);
}
```

A interface `Timer` define o comando `start` com dois parâmetros: o tipo do temporizador e o intervalo de sinalização. O comando `stop` encerra o temporizador e o evento `fired` é sinalizado a cada intervalo de tempo.

```
interface Timer {
  command result_t start(char type, uint32_t interval);
  command result_t stop();
  event result_t fired();
}
```

O componente `SurgeM` implementa os comandos da interface `StdControl` (`init, start, stop`) oferecida por ele; e os eventos `Timer.fired`, `ADC.dataReady` e `SendMsg.sendDone` sinalizados pelas interfaces usadas pelo componente. Cada vez que o evento `Timer.fired` é sinalizado, o comando `ADC.getData` é chamado para que o sensor faça uma nova leitura. Na implementação do evento `ADC.dataReady`, o valor medido é armazenado em uma variável global e a tarefa `SendData` é escalonada para que esse valor seja transmitido pela rede.

A linguagem nesC define a construção `call <command>` para chamar um comando e a construção `post <task>` para escalonar a execução de uma tarefa. As funções que implementam tarefas são marcadas com o identificador `task`. Essas funções não recebem parâmetros de entrada e não devolvem valores.

A variável `gSendBusy` é usada para garantir que uma nova leitura seja considerada apenas quando o valor da leitura anterior, armazenado na variável `gSensorData`, já foi enviado pela rede. Como essas variáveis são acessadas dentro de um tratador de evento de hardware, elas são protegidas por seções atômicas. A tarefa `SendData` preenche a variável `gMsgBuffer` com o valor medido. Essa variável é um valor do tipo `TOS_Msg`: a estrutura de dados definida pelo TinyOS para compor as mensagens. O campo `TOS_Msg.data` dessa estrutura é usado para receber o valor que deve ser transmitido.

A Figura 4.2 mostra o componente de configuração da aplicação `Surge`. Ele inclui a especificação dos componentes usados: `Main`, `SurgeM`, `TimerC`, `Photo`,

```
configuration Surge {
}
implementation {
  components Main, SurgeM, TimerC, Photo, GenericCommPromiscuous;
  Main.StdControl -> SurgeM.StdControl;
  Main.StdControl -> TimerC.StdControl;
  Main.StdControl -> Photo.Stdcontrol;
  Main.StdControl -> GenericCommPromiscuous.Stdcontrol;
  SurgeM.ADC   -> Photo.ADC;
  SurgeM.Timer -> TimerC.Timer[1];
  SurgeM.SendMsg -> GenericCommPromiscuous.SendMsg[1];
}
```

Figura 4.2: Versão simplificada do componente de configuração da aplicação *Surge*

e `GenericCommPromiscuous`; e a forma como as interfaces desses componentes são conectadas. O componente `Main` é o componente executado no início de uma aplicação TinyOS, por isso toda aplicação deve incluí-lo.

A interface `StdControl` do componente `Main` é conectada com a interface `StdControl` provida pelos demais componentes da aplicação. Isso significa que quando o comando `StdControl.init` do componente `Main` é chamado, no início da aplicação, o mesmo comando é chamado em todos os outros componentes.

A sentença `SurgeM.ADC -> Photo.ADC` estabelece que a interface `ADC` usada pelo componente `SurgeM` é conectada com a interface `ADC` provida pelo componente `Photo`. Assim, quando o comando `ADC.getData` for chamado na aplicação, a leitura que será recebida corresponderá a uma medida de luminosidade.

Os componentes `TimerC` e `GenericCommPromiscuous` são usados para implementar as interfaces `SurgeM.Timer` e `SurgeM.SendMsg`, respectivamente. Esses componentes implementam versões parametrizadas para as interfaces `Timer` e `SendMsg`. Por isso, um valor precisa ser associado a elas para caracterizar a instância que será usada na aplicação.

4.2.3 Complexidade inerente às aplicações TinyOS

Na aplicação *Surge*, a tarefa principal — requisitar a leitura de um valor do sensor e enviá-lo para os nós vizinhos — foi particionada em quatro pedaços de código distintos: um comando que requisita a leitura do sensor, um evento que sinaliza que o valor da leitura está disponível, uma tarefa para enviar o valor medido para os nós vizinhos e um evento que sinaliza que o valor foi enviado. Nas aplicações TinyOS, as tarefas que requerem interação com dispositivos externos (incluindo a rede de comunicação) são

tipicamente iniciadas através de um comando e o seu término é indicado posteriormente com a sinalização de um evento. O resultado é um tipo de concorrência diretamente associado às interrupções de hardware (leitura dos sensores, recepção de pacotes pela rede ou temporização). Essa abordagem de construir operações em duas fases, típica da arquitetura dirigida a eventos, é o fator principal que torna o desenvolvimento das aplicações TinyOS complexo. Parte do código da aplicação *Surge* é dedicada para controlar o fluxo de execução da aplicação. A medida que a complexidade das aplicações cresce, a tendência natural é o controle do fluxo de execução tornar-se ainda mais difícil de tratar.

Tarefas como a leitura e transmissão de um valor são naturalmente tarefas sequenciais do ponto de vista do programador, mas devem ser obrigatoriamente particionadas. O modelo de programação *multithreading* (com linhas de execução distintas que podem ser automaticamente suspensas e retomadas) seria uma alternativa para lidar com essa dificuldade, mas ele não é adotado pelo TinyOS em razão do seu custo computacional. Nossa proposta, então, consiste em implementar o modelo de programação discutido no capítulo 2 — uma combinação de co-rotinas e comunicação baseada em eventos, com elementos que encapsulam a transferência de controle entre as linhas de execução de uma aplicação — dentro do sistema operacional TinyOS. Mostraremos, assim, que é possível oferecer um caminho intermediário — sem o custo excessivo do uso de *threads* e com uma interface de programação mais simples de usar. Na próxima seção aprofundamos a discussão sobre essa alternativa.

4.3 Uma nova interface de programação para TinyOS

Em linhas gerais, o que propomos é oferecer interfaces TinyOS de nível mais alto, nas quais apenas comandos são definidos, ou seja, o programador não precisa lidar diretamente com o particionamento de uma requisição em uma chamada de comando e em um tratador de evento. Os novos comandos encapsulam as operações de duas fases em uma única requisição com um valor de resposta. As operações que requerem a interação com dispositivos externos, incluindo a comunicação via rádio, permanecem divididas em duas fases, mas dentro de um contexto de execução reservado para elas através do uso de co-rotinas.

Com essa estrutura, a aplicação ainda é capaz de responder a outros eventos do sistema, ou executar tarefas que foram postergadas, enquanto a operação iniciada com a chamada do comando espera a informação necessária para continuar o seu processamento. A transferência de controle entre os con-

textos de execução é feita de forma cooperativa, e é implementada implicitamente através (i) das operações definidas nas interfaces de nível mais alto e (ii) do escalonador de co-rotinas. O escalonador de co-rotinas (ou escalonador de tarefas, como definido no capítulo 2) permite manter o código dos tratadores de interrupção pequenos, pois eles apenas informam ao escalonador que uma determinada co-rotina está pronta para ser executada. Somente quando o escalonador for executado é que o controle poderá ser de fato transferido para essa co-rotina. O resultado é uma interface de programação mais simples, que oferece ao programador uma visão síncrona do código, sem perder o assincronismo da arquitetura TinyOS.

Implementamos esse cenário em três etapas: (a) construção de co-rotinas no ambiente computacional dos sensores; (b) construção de um escalonador de co-rotinas dentro da arquitetura do TinyOS; (c) especificação da estrutura básica para transformar as operações de duas fases do TinyOS em operações de uma fase, redefinindo interfaces básicas em interfaces de nível mais alto. Nas próximas seções discutimos questões relacionadas a cada uma dessas etapas e os resultados obtidos.

4.3.1 Implementando co-rotinas para os sensores

A interface de programação de co-rotinas inclui, além das operações de transferência de controle, uma operação para criar a co-rotina. Essa operação tipicamente reserva o espaço na memória para a pilha da co-rotina e define o procedimento que será executado. Quando a co-rotina termina sua execução, o espaço reservado para a pilha é liberado. Para minimizar o custo computacional com a alocação dinâmica de memória (um requisito importante no contexto dos sensores), particionamos a operação de criação de uma co-rotina em duas fases: a primeira reserva o espaço de memória para a pilha e a segunda define o procedimento que será executado pela co-rotina. Com essa abordagem, o espaço de memória é alocado uma única vez e pode ser reutilizado para executar procedimentos distintos.

Como discutido na seção 2.2.1, co-rotinas podem ser implementadas como co-rotinas *simétricas* ou *assimétricas*. De acordo com Moura (Moura/04), co-rotinas simétricas são mais adequadas para representar unidades de execução independentes, entretanto, quando um escalonador é usado para gerenciar a transferência de controle entre co-rotinas prontas para executar, o uso das co-rotinas assimétricas é mais natural. Com co-rotinas simétricas seria preciso distribuir o escalonamento ou delegá-lo a uma única co-rotina, conhecida por todas as demais. Optamos, então, pelo modelo de co-rotinas assimétricas, com

o seguinte conjunto de operações básicas: (a) reservar espaço de memória para a pilha da co-rotina; (b) associar um procedimento ao espaço de armazenamento da co-rotina; (c) transferir o controle para uma co-rotina; (d) e devolver o controle para a co-rotina anterior.

Uma questão chave relacionada à implementação de co-rotina é como trocar parâmetros entre duas co-rotinas. A dificuldade ocorre porque normalmente as linguagens de programação usam a pilha de execução para passar parâmentos entre chamadas de procedimentos. Como cada co-rotina possui sua própria pilha de execução, essa abordagem não pode ser usada. Além disso, uma co-rotina pode ter vários pontos de entrada, localizados imediatamente após cada operação de transferência de controle. Assim, para manter uma implementação simples (novamente, um requisito importante no contexto dos sensores), optamos por usar variáveis de escopo global, mas com acesso restrito, para transferir dados entre co-rotinas.

Implementamos uma versão de co-rotinas baseada na biblioteca PCL (*Portable Coroutine Library*) (Libenzi/05). Nossa implementação foi especialmente projetada para o microcontrolador ATmega128L (Atmega128), usado como base para as plataformas de redes de sensores Mica2, Mica2Dot e MicaZ (Crossbow). O ATmega128L possui 128KB de memória de código (memória *flash*) e 4KB de memória de dados (SRAM).

A biblioteca AVR Libc (AVRLibc) — um sub-conjunto do padrão ANSI-C especialmente definido para os microcontroladores AVR — implementa as funções `setjmp` e `longjmp` e define uma estrutura de dados particular, denominada `jmp_buf`, para ser usada por essas funções. Definimos uma estrutura básica para armazenar os dados de uma co-rotina com os seguintes campos:

```
typedef struct s_coroutine {
   jmp_buf ctx;
   uint16_t size;
   procedure_t proc;
   struct s_coroutine *caller;
} coroutine;
```

O campo `ctx` é usado para armazenar o contexto da co-rotina (valores dos registradores, ponteiros da pilha e do programa). O campo `size` mantém a informação sobre o tamanho do espaço de memória reservado para a pilha da co-rotina (fixamos um tamanho mínimo de 256 bytes). Essa informação pode ser usada para indicar um erro, caso a pilha da co-rotina cresça além do tamanho reservado para ela. O campo `proc` armazena o ponteiro para o endereço do procedimento que a co-rotina deve executar. Finalmente, o campo `caller` mantém a referência para a co-rotina anterior, para quem o controle

deve ser devolvido quando a co-rotina terminar ou quando ela executar a operação `yield`.

A Figura 4.3 mostra nossa implementação de co-rotinas. A função `create` cria uma co-rotina reservando espaço para a sua pilha na memória *heap*. Essa co-rotina poderá ser usada para executar diferentes procedimentos. Quando um procedimento termina de executar dentro da co-rotina, um novo procedimento pode ser associado a ela. A função `setProcedure` associa um procedimento a uma co-rotina e salva o estado inicial usando a função `setjmp`. A função `setjmp`, por sua vez, usa a estrutura `jmp_buf` para armazenar os valores dos registradores. Os campos com o contador de programa e o ponteiro da pilha são diretamente ajustados dentro da função `setContext`. As operações para a transferência de controle entre as co-rotinas são implementadas através das funções `resume` e `yield` que juntas implementam a semântica de co-rotinas assimétricas. A função `runner` é uma função auxiliar usada para iniciar e finalizar a execução de uma co-rotina. A troca de contexto é efetuada pela função `switchContext`. Ela recebe como argumento as variáveis com a informação do contexto atual e do contexto para onde a execução deve ser transferida.

4.3.2 Incluindo um escalonador de co-rotinas

O TinyOS implementa um *escalonador de tarefas* que executa dentro do laço principal de uma aplicação. Depois que a aplicação executa o código de inicialização, ela entra em um laço infinito, como mostra a Figura 4.4. A função `TOSH_run_task` percorre a fila de tarefas escalonando uma tarefa após a outra. Quando todas as tarefas foram tratadas, o processador entra em um estado que minimiza o consumo de energia e só é ativado novamente quando algum evento ocorre. Como os tratadores de eventos podem desencadear a execução de novas tarefas, após o tratamento dos eventos o escalonador percorre novamente a fila de tarefas.

Quando o programa escalona uma tarefa `t` através da sentença `post t();` a função interna `TOS_post` é chamada. Essa função verifica se é possível escalonar essa tarefa (o número máximo de tarefas no sistema é pré-definido pelo TinyOS em 8). Se for possível, a referência para o procedimento `t` é armazenada na fila de tarefas. A função `TOSH_run_next_task` percorre a fila de tarefas invocando os procedimentos cujas referências foram anteriormente armazenadas.

Nossa primeira tentativa para incluir uma estrutura de gerência cooperativa de tarefas no TinyOS foi tratar todas as tarefas escalonadas pela aplicação

```
typedef void (*procedure_t)();
typedef void *coroutine_t;
static coroutine g_coMain, *g_coCurr = &g_coMain;

void setContext(jmp_buf *ctx,procedure_t proc,char *sbase,int16_t ssize){
  uint16_t p = (uint16_t *) proc;
  char *stack = sbase + ssize - sizeof(int16_t);
  uint16_t s = (uint16_t *) stack;
  setjmp(*ctx);
  ctx[0]->_jb[AVR_PC] = (char) p & 0xff;
  ctx[0]->_jb[AVR_PC+1] = (char) (p >> 8) & 0xff;
  ctx[0]->_jb[AVR_SP] =  (char) s & 0xff;
  ctx[0]->_jb[AVR_SP+1] = (char) (s >> 8) & 0xff;
}

void switchContext
 (jmp_buf *octx, jmp_buf *nctx) {
  if (!setjmp(*octx)) longjmp(*nctx, 1);
}

void runner() {
  procedure_t p; coroutine *co;
  co = g_coCurr; p = co->proc;  p();  co->proc = NULL;
  g_coCurr = co->caller;
  switchContext(&co->ctx, &g_coCurr->ctx);
}

coroutine_t create(int16_t size) {
  coroutine *co; size = size + sizeof(coroutine);
  co = malloc(size); if (!co) return NULL;
  co->size = size; co->proc = NULL;
  return (coroutine_t) co;
}

uint8_t setProcedure (coroutine_t coro, procedure_t proc) {
  coroutine *co = (coroutine *) coro;
  char *stack;
  if (co->proc == NULL) {
   stack = (char *) co; co->proc = proc;
   setContext(&co->ctx, runner, stack, co->size);
  } else return 0;
  return 1;
}

void resume(coroutine_t coro) {
  coroutine *co, *oldco;
  co = (coroutine *) coro; oldco = g_coCurr;
  co->caller = g_coCurr; g_coCurr = co;
  switchContext(&oldco->ctx, &co->ctx);
}

void yield() {
  coroutine *co; co = g_coCurr; g_coCurr = co->caller;
  switchContext(&co->ctx, &g_coCurr->ctx);
}
```

Figura 4.3: Implementação de co-rotinas para o processador ATmega128L.

```
int main() {
  ...codigo inicial...
  while(1) {
     TOSH_run_task();
  }
}
void TOSH_run_task() {
  while (TOSH_run_next_task())
   ;
  TOSH_sleep();
  TOSH_wait();
}
```

Figura 4.4: Laço principal executado por uma aplicação TinyOS.

como co-rotinas. Para isso, alteramos a função `TOSH_run_next_task` de modo que, ao invés de invocar diretamente os procedimentos da fila de tarefas, eles eram associados a uma co-rotina e em seguida o controle era transferido para ela. Desse modo, todas as tarefas eram executadas em contextos distintos como co-rotinas. O problema dessa abordagem foi a sobrecarga de trocas de contextos imposta ao sistema. Nem sempre as tarefas estão diretamente associadas a operações de duas fases nas quais a construção de uma visão síncrona é mais apropriada para o programador. Nesses casos, o custo da troca de contexto não era justificado.

Resolvemos então manter uma fila distinta com os procedimentos que devem ser executados como co-rotinas e construir um escalonador de co-rotinas similar ao escalonador de tarefas do TinyOS. Para lidar com a restrição de espaço de memória, estabelecemos um número máximo de co-rotinas (equivalente ao número máximo de tarefas). Todas as co-rotinas são criadas na inicialização da aplicação. Assim, o espaço para a pilha das co-rotinas na memória *heap* é pré-alocado, reduzindo o custo computacional com alocação e realocação dinâmica de memória.

Para associar procedimentos definidos na aplicação com as co-rotinas pré-criadas, a função `TOS_post_coro`, similar a função `TOS_post`, foi implementada. Essa função recebe um procedimento como argumento e o associa a uma co-rotina disponível. A co-rotina é então marcada como "pronta" e o controle poderá ser transferido para ela pelo escalonador de co-rotinas. Da mesma forma que as tarefas, os procedimentos associados às co-rotinas não devem possuir parâmetros (como discutido na seção 4.3.1, a troca de dados entre as co-rotinas será tratada através de variáveis globais). Implementamos a função `TOSH_run_next_coro`, responsável por verificar quais co-rotinas estão prontas para executar. Quando uma co-rotina "pronta" é encontrada, o controle é transferido para ela. Modificamos o laço principal de execução do TinyOS

```
void TOSH_run_task() {
  while (TOSH_run_next_task())
   ;
  TOSH_run_next_coro();
  TOSH_sleep();
  TOSH_wait();
}
```

Figura 4.5: Laço principal de uma aplicação TinyOS com o escalonador de co-rotinas.

incluindo uma chamada para essa função, como mostra a Figura 4.5.

Finalmente, para permitir o acesso à construção de co-rotinas na implementação dos componentes, as funções descritas a seguir são exportadas para o nível de desenvolvimento das aplicações:

- `bool POST(procedure_t proc)`: escalona um procedimento para ser executado como uma co-rotina;
- `uint8_t GETID()`: devolve o identificador da co-rotina corrente;
- `void SUSPEND()`: suspende a co-rotina corrente e transfere o controle para o escalonador de co-rotinas;
- `void RESTORE(uint8_t id)`: avisa ao escalonador que a co-rotina especificada através do identificador `id` pode ser executada.

A função `POST` é a única função que o desenvolvedor de aplicações precisará usar. As demais funções serão usadas para implementar as interfaces de nível mais alto, cujas operações permitirão encapsular a transferência de controle entre as co-rotinas, i.e., o desenvolvedor de aplicações não precisará lidar diretamente com as trocas de contexto.

4.3.3 Construindo interfaces TinyOS de nível mais alto

Usando o novo escalonador TinyOS e o conjunto de funções exportadas por ele, podemos definir interfaces TinyOS de nível mais alto encapsulando dentro delas a transferência de controle entre as co-rotinas. Como resultado, ao usar essas interfaces no desenvolvimento de aplicações, o programador poderá manter uma visão sequencial do código, mais simples e fácil de entender e implementar. A Figura 4.6 mostra, como exemplo, a definição da interface `newADC` (usada para requisitar a leitura de valores pelos sensores) e o componente `newADCM` que implemementa essa interface.

Na nova versão da interface `ADC`, o comando `newADC.getData` chama o comando original `ADC.getData` e suspende a linha de execução corrente,

```
interface newADC {
 async command result_t getData(uint16_t *data);
 async command result_t getContinuousData(uint16_t *data);
}

module newADCM {
 provides { interface newADC; }
 uses { interface ADC; }
}
implementation {
 uint8_t gBusy, gCoroID;
 uint16_t gData;
 async command result_t newADC.getData(uint16_t *data){
    atomic{
       if (gBusy) return FAIL;
       gBusy = 1; gCoroID = GETID();
    }
    call ADC.getData();
    SUSPEND();
    atomic { *data = gData; gBusy = 0; }
    return SUCCESS;
 }

 async event result_t ADC.dataReady(uint16_t data) {
    atomic gData = data;
    atomic RESTORE(gCoroID);
    return SUCCESS;
 }
}
```

Figura 4.6: *Proxy* para a interface ADC

chamando a operação `SUSPEND`. Nesse ponto, o controle da execução volta para o laço principal e então a aplicação pode tratar outros eventos ou tarefas. O identificador da co-rotina é armazenado na variável `gCoroID` para ser usado posteriormente, quando essa co-rotina volta a estar pronta para executar. Quando o valor requisitado torna-se disponível, o evento `ADC.dataReady` é sinalizado. Isso significa que a co-rotina que foi anteriormente suspensa pode agora continuar sua execução. O escalonador de co-rotinas é notificado através da função `RESTORE`. A próxima vez que essa co-rotina retomar o controle, a execução irá continuar a partir da sentença `SUSPEND()`.

Além de suspender e retomar a execução do comando que requisita uma nova leitura do sensor apropriadamente, a implementação da interface de nível mais alto encapsula todo o acesso à memória compartilhada e evita a sobrecarga de invocação dos comandos oferecidos. Se uma nova invocacação é requisita antes do término da invocação anterior, uma resposta negativa é devolvida. Desse modo, o programador não precisa se preocupar em garantir o acesso atômico à variável que armazena os dados capturados pelo sensor.

Usamos a mesma abordagem para construir uma nova versão para a

```
module SurgeM {
  provides { interface StdControl; }
  uses { interface newADC; interface Timer, interface newSendMsg; }
}
implementation {
 TOS_Msg gMsgBuffer;
 uint16_t dest = TOS_BCAST_ADDR;
 command result_t StdControl.init() {
   return SUCCESS;
 }
 command result_t StdControl.start() {
   return call Timer.start(TIMER_REPEAT, timer_rate);
 }
 command result_t StdControl.stop() {
   return call Timer.stop();
 }
 void newSendData() {
   SurgeMsg *pReading;
   result_t result;
   pReading = (SurgeMsg *)((&gMsgBuffer)->data);
   if (call newADC.getData((&pReading)->reading))
    call newSendMsg.send(dest, sizeof(SurgeMsg), &gMsgBuffer, &result);
 }
 event result_t Timer.fired() {
   POST(newSendData);
 }
}
```

Figura 4.7: Novo código para a aplicação *Surge*

interface `SendMsg`. Essa interface é usada na aplicação *Surge* (Figura 4.1) para enviar a mensagem com o valor capturado pelo sensor para os nós vizinhos. A versão original dessa interface define o comando `SendMsg.send` para enviar uma mensagem, e o evento `SendMsg.sendDone` para informar se a mensagem foi enviada com sucesso ou não. Na nova versão da interface (`newSendMsg`), definimos o comando `newSendMsg.send` e encapsulamos a recepção do evento `SendMsg.sendDone` dentro da implementação desse comando. A informação recebida com a sinalização do evento é devolvida através do parâmetro `results`, incluído na lista de parâmetros do comando `newSendMsg.send`.

Com as novas interfaces, reimplementamos a aplicação *Surge*. A Figura 4.7 mostra o novo código. Na sua versão original (Figura 4.1), o evento `Timer.fired` dispara a requisição de leitura do sensor. O valor recebido com a sinalização do evento `ADC.dataReady` é armazenado em uma variável global e uma tarefa é escalonada para tratá-lo. O código da tarefa insere o valor medido em uma estrutura de dados apropriada, usada para compor a mensagem que deve ser transmitida, e chama o comando `SendMsg.send` para enviá-la. Enquanto a transmissão não é concluída, a estrutura de dados com a mensagem deve ser mantida inalterada. Para garantir esse comportamento, a variável de

controle `gSendBusy` é atualizada no código do evento que sinaliza a conclusão da transmissão.

No novo código (Figura 4.7), a operação completa — incluindo a requisição de leitura do sensor e a transmissão do valor medido pela rede — é codificada dentro de uma única função (`newSendData`). O evento `Timer.fired` dispara a execução dessa função dentro de um contexto de execução distinto, escalonando uma co-rotina para executá-la. Desse modo, quando as operações que requerem interação com dispositivos externos são invocadas, a função pode ser suspensa e retomada posteriormente garantindo a interatividade do restante da aplicação.

A invocação dos comandos originais `ADC.getData` e `SendMsg.send`, e o tratamento dos eventos gerados por eles, é encapsulada dentro dos novos comandos `newADC.getData` e `newSendMsg.send`. Assim, o comportamento baseado em operações de duas fases permanece válido, mas o programador não precisa lidar diretamente com ele, ou seja, embora oferecendo a possibilidade de construir uma visão síncrona das operações naturalmente sequenciais, o modo assíncrono de interação entre os dispositivos não é alterado.

4.4 Avaliação do custo computacional de co-rotinas

Para avaliar a carga computacional adicionada com o uso de co-rotinas dentro do escalonador do TinyOS, simulamos a execução da aplicação *Surge* usando a versão original, mostrada na Figura 4.1, e a nova versão, mostrada na Figura 4.7, no nível de código de máquina.

Como usamos as funções `setjmp/longjmp` para implementar a construção de co-rotinas, foi preciso usar um simulador que permitisse *emular*, instrução por instrução, o funcionamento de um sensor. Para isso, adotamos o simulador ATEMU (Polley/04), um simulador de redes de sensores no nível de instruções. Da mesma forma que o TOSSIM (Levis/03) — o simulador de redes de sensores especialmente projetados para aplicações TinyOS — o ATEMU simula redes de sensores nas quais os nós são microcontroladores AVR e executa o programa do microcontrolador, ao invés de modelos de software. Entretanto, ao gerar o código binário para a simulação, o TOSSIM usa algumas bibliotecas do sistema local — ao invés da biblioteca AVR Libc — e entre essas bibliotecas está aquela que implementa as funções `setjmp/longjmp`. Usamos o emulador *ATmega128L Emulator* (version 0.4), que é parte do ATEMU e a versão 1.1.13 do TinyOS.

As Figuras 4.8 e 4.9 mostram os resultados das simulações. A Figura 4.8 destaca o custo para iniciar a aplicação, enquanto a Figura 4.9 mostra o custo para completar as leituras periódicas do sensor. No primeiro caso, simulamos

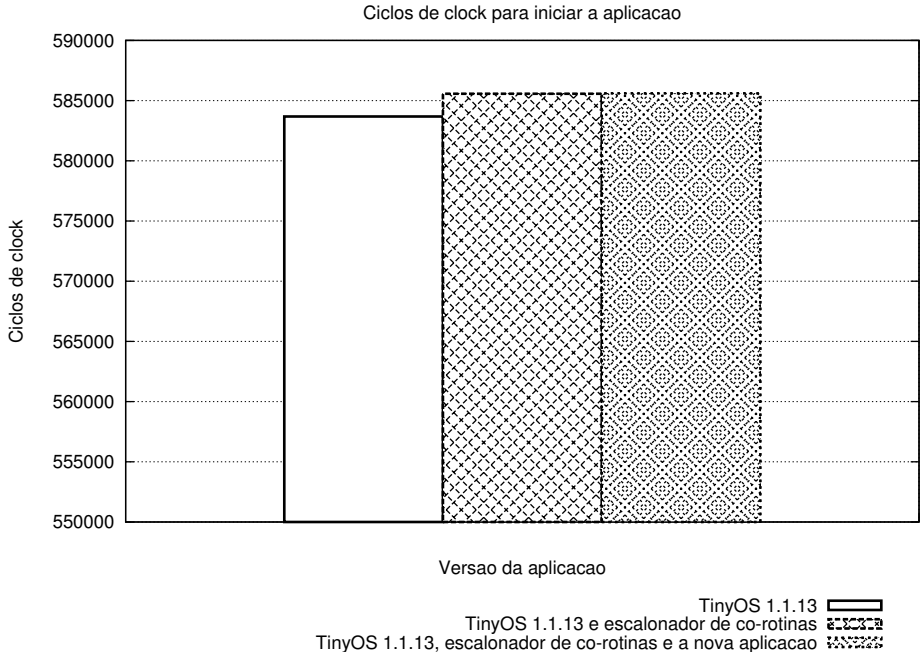

Figura 4.8: Ciclos de clock para iniciar a aplicação.

a versão original da aplicação *Surge* (código mostrado na Figura 4.1), usando a versão original do TinyOS. No segundo caso, incluímos no código do sistema operacional a implementação de co-rotinas, a pré-alocação de espaço na memória *heap* para a pilha das co-rotinas e substituímos o código com o laço principal do TinyOS pela nossa implementação, onde acrescentamos o escalonador de co-rotinas. Nossa meta nesse caso foi medir a sobrecarga imposta pela infra-estrutura adicional, ainda sem usar co-rotinas de fato. No último caso, simulamos a nova versão da aplicação *Surge* (apresentada na Figura 4.7) com a nova versão do laço principal do TinyOS. Nosso objetivo foi comparar o número de ciclos de *clock* necessários para iniciar a aplicação e tratar as leituras do sensor.

A carga computacional adicionada para iniciar a aplicação foi de $0,322\%$ quando incluímos a implementação de co-rotinas no núcleo do TinyOS e de $0,329\%$ quando executamos a nova versão da aplicação. A diferença percebida (de $0,007\%$) entre esses dois casos corresponde ao custo de carregar os componentes que implementam as novas interfaces `newADC` e `newSendMsg`.

Nas leituras do sensor (Figura 4.9), o número de ciclos de clock consumidos pela aplicação original foi de aproximadamente 6800 ciclos de clock, com picos que chegaram a 12800 ciclos. Com a nova versão da aplicação, as leituras consumiram na média 11500 ciclos de clock, com um pico que chegou a 14300 ciclos. Podemos observar uma variação maior nos valores medidos com a nova aplicação. Essa variação se deve ao custo de encontar uma co-rotina disponível para escalonar a leitura do sensor. Percebe-se claramente o incre-

mento consecutivo do número de ciclos de clock até o limite do número de co-rotinas pré-alocadas pelo sistema (no caso dessas simulações esse valor foi 5 co-rotinas). Essa diferença aparece porque o evento de temporização ocorre antes da leitura anterior ser concluída, por isso todo o *pool* de co-rotinas é usado (o escalonador de co-rotinas sempre inicia a busca por uma co-rotina disponível no início do *pool*, uma definição de projeto que pode ser otimizada).

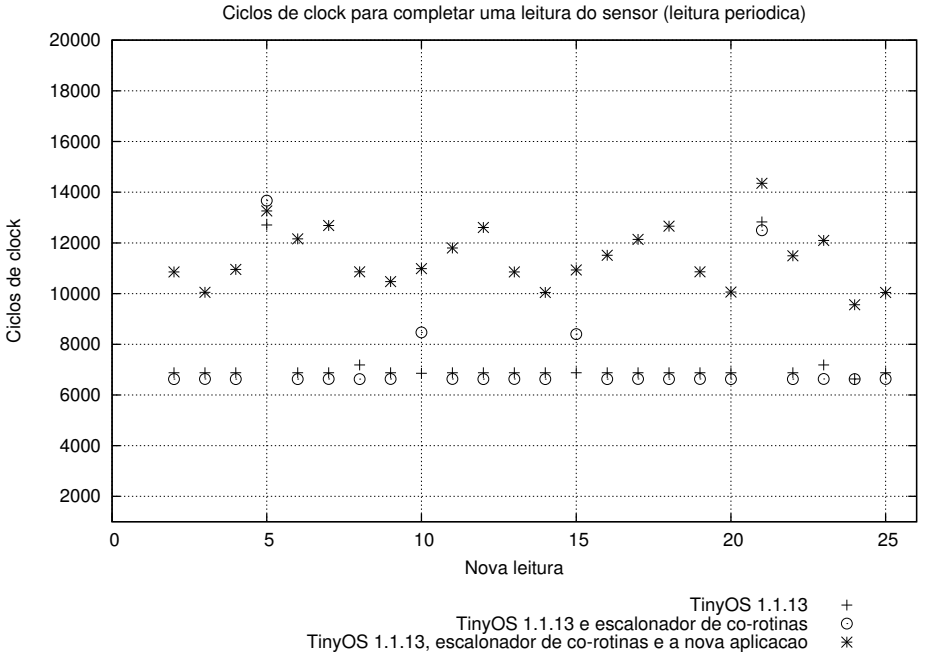

Figura 4.9: Ciclos de clock para completar leituras periódicas do sensor.

Outra informação destacada pela Figura 4.9 é o fato do número de ciclos de clock usados pela versão original do TinyOS ser ligeiramente maior que o número de ciclos de clock usados pela versão do TinyOS que inclui o escalonador de co-rotinas. Seguindo passo-a-passo a execução da simulação descobrimos que o simulador aguarda o evento `ADC.dataReady()` por um período aleatório usando uma lógica que implica na repetição de uma sequência de instruções dentro do procedimento `sleep()`. Na versão com o escalonador de co-rotinas, o tempo gasto percorrendo o *pool* de co-rotinas reduz o número de vezes que a sequência de instruções do procedimento `sleep()` é executada, resultando em um número de ciclos de clock aparentemente menor.

O incremento no custo computacional total usando co-rotinas é um resultado esperado, uma vez que estamos acrescentanto uma funcionalidade ao sistema sem alterar o seu modo de operação básico. O que os valores medidos nos permitem concluir é que o custo adicional está dentro de um limite razoável, considerando a comodidade oferecida para o programador. A lógica de particionar operações em duas fases ainda deve ser codificada, além da transferência correta de controle entre as co-rotinas, mas essa lógica pode

ser implementada uma única vez e usada várias vezes. Esse é o propósito da construção de interfaces de nível mais alto, como a interface `newADC`.

No nível de desenvolvimento das aplicações, o programador pode optar pelo estilo de programação mais adequado para cada problema. As interfaces originais permanecem válidas, permitindo a codificação de soluções usando o estilo reativo do tratamento de eventos. A opção pela execução de uma sequência de operações dentro de uma co-rotina é adequada sempre que essa sequência de operações reflete um comportamento sequencial mas requer a interação com dispositivos externos. Nos demais casos, o tratamento normal dos eventos gerados é a possibilidade com menor custo computacional.

4.5 Aplicabilidade do modelo em outros tipos de aplicações de sensoriamento

A aplicação que usamos como base para a nossa avaliação representa um tipo particular, ainda que frequente, de aplicação de sensoriamento: as leituras dos sensores são requisitadas em intervalos de tempo pré-fixados (no exemplo discutido, a cada 1 segundo). Entretanto, diferentes contextos nos quais as redes de sensores são usadas exigem modelos particulares de aplicacação. Em particular, as aplicações para redes de sensores podem ser categorizadas de acordo com o tipo de sensoriamento e o tipo de disseminação dos dados, como (Loureiro/03):

- **periódica**: coletam os dados em intervalos regulares;
- **reativa**: coletam dados quando ocorrem eventos de interesse ou quando solicitado pelo observador;
- **tempo real**: coletam a maior quantidade de dados possível no menor intervalo de tempo.
- **contínua**: coletam os dados continuamente;

A aplicação *Surge* é um exemplo de aplicação com sensoriamento periódico. Uma versão similar dessa aplicação poderia ser facilmente implementada para permitir a coleta reativa de dados. Para isso, ao invés de escalonar a execução do procedimento `newSendData` dentro do evento `Timer.fired`, esse procedimento passaria a ser escalonado dentro do evento de interesse, ou dentro do evento `ReceiveMsg.receive`, tipicamente implementado para tratar a recepção de mensagens enviadas por outros nós da rede. Para aplicações de tempo real, entretanto, acreditamos que o custo com a troca de contexto entre as co-rotinas possa ter impacto inaceitável na capacidade de coleta de dados dos nós sensores. Nesse caso, a simplicidade da programação representaria um custo não justificável, considerando a finalidade da aplicação.

```
module ReadSendM { ... }
implementation {
 TOS_Msg gMsgBuffer;
 bool read = TRUE;
 uint16_t dest = TOS_BCAST_ADDR;
 void ReadSendData();

 command result_t StdControl.start() {
   POST(ReadSendData);
 }

 void ReadSendData() {
   Msg *pReading;
   result_t result;
   pReading = (Msg *)((&gMsgBuffer)->data);
   while (read) {
    if (call newADC.getData((&pReading)->reading))
    call newSendMsg.send(dest, sizeof(Msg), &gMsgBuffer, &result);
   }
 }
}
```

Figura 4.10: Aplicação com coleta contínua de dados (versão do TinyOS com co-rotinas).

Para o caso em que a coleta contínua de dados é necessária, a Figura 4.10 mostra uma versão simplificada de uma aplicação para coleta e envio contínuo de dados usando as facilidades providas pelas interfaces de alto nível `newADC` e `newSendMsg`. Quando o comando `StdControl.start` é executado, o procedimento `ReadSendData` é escalonado para executar em um novo contexto de execução (usando a operação `POST`). Esse procedimento implementa um laço infinito que invoca a leitura do sensor e em seguida envia o valor lido para a estação base. Nesse caso, a implementação é de fato simplificada pelo fato do programador poder "ver" os comandos `newADC.getData` e `newSendMsg.send` como comandos síncronos.

4.6 Trabalhos relacionados

Outros trabalhos têm investigado alternativas para reduzir a complexidade no desenvolvimento de aplicações para redes de sensores. O trabalho sobre *Protothreads* (Dunkels/05) propõe uma abstração de programação que permite o bloqueio e retomada de linhas de execução. *Protothreads* implementa um tipo de continuação, denominado *continuação local*, usando a sentença *switch* da linguagem C. Diferentemente de co-rotinas, as *protothreads* não requerem uma pilha particular: todas rodam usando a mesma pilha e a troca de contexto é efetuada remontando a pilha de execução. Em consequência disso,

as variáveis dentro do escopo de funções locais não são automaticamente salvas entre as operações bloqueantes. Um espaço de memória precisa ser reservado e gerenciado pelo programador para armazenar todas as variáveis automáticas. No modelo que desenvolvemos cada linha de execução tem sua própria pilha, de modo que é possível manter variáveis locais mesmo com as transferências de controle entre elas.

O modelo de programação TinyGALS (Cheong/03) foi inspirado na arquitetura do TinyOS e visa tornar a concorrência entre as tarefas mais explícita e fácil de tratar pelo desenvolvedor, definindo dois níveis de hierarquia. Os programas TinyGALS são compostos por módulos que interagem assincronamente por meio de troca de mensagens, e cada módulo é constituído por componentes que se comunicam via chamadas de métodos síncronas. Nessa abordagem, as visões síncrona e assíncrona oferecidas para o programador, devem ser construídas estaticamente durante a composição dos módulos e refletem apenas a forma de interação da chamada dos métodos. Na abordagem que propomos, a visão síncrona é construída sobre uma base assíncrona de interação.

O interpretador de *bytecode* Maté (Levis/02), projetado como um componente TinyOS, implementa uma máquina virtual que oferece um conjunto reduzido de primitivas de alto nível. O objetivo principal é reduzir o tamanho das aplicações desenvolvidas para as redes de sensores e então minimizar o custo de energia para transmití-las pela rede. Usando uma representação mais concisa e de alto nível, os autores argumentam que além de facilitar a reprogramação dos nós da rede, é possível simplificar a interface de programação do TinyOS. No nosso trabalho tomamos uma direção oposta: nosso objetivo é simplificar a interface de programação do TinyOS, e uma das consequências dessa simplificação é a possibilidade de construir aplicações mais concisas. O Maté define três contextos distintos de execução que correspondem a três eventos: temporização, recepção de mensagem ou requisição de envio de mensagem. Similar a abordagem que propomos, as instruções definidas pelo Maté escondem o assincronismo da programação com o TinyOS. Por exemplo, quando o comando para enviar uma mensagem é chamado, o contexto de execução é suspenso até que o evento que sinaliza que a mensagem foi enviada seja recebido. Na nossa abordagem, entretanto, não limitamos os contextos de execução para tratar operações específicas. Além disso, embora o conjunto de instruções oferecidas pelo Maté permita lidar de forma mais conveniente com o assincronismo, um esforço adicional é requerido do programador que precisa se familiarizar com a semântica dessas intruções.

OSM (*Object State Model*) (Kasten/05) é uma linguagem de programação para os dispositivos de uma rede de sensores baseada em máquinas de estados

finitos. A proposta consiste em estender o modelo de programação dirigido a eventos com *estados* e *transições*, de modo que a invocação de um tratador de evento leve em conta não apenas a informação do evento ocorrido, mas também o estado do programa. O objetivo é reduzir as dificuldades com a gerência manual da pilha e do fluxo de execução, típicas do paradigma baseado em eventos. O modelo explora a idéia de *variáveis de estado* permitindo o compartilhamento de informações entre as ações executadas pelos tratadores de eventos. Entretanto, embora seja possível associar o estado do programa com as ações executadas em resposta à ocorrência de eventos, partes puramente sequenciais da aplicação ainda precisam ser explicitamente modeladas pelo programador como sequências de estados conectados através de transições. Na nossa abordagem, a informação compartilhada entre os eventos é mantida em variáveis globais com um escopo reduzido (dentro do código que implementa as interfaces redefinidas). As transições entre os estados são encapsuladas na redefinição dos comandos, permitindo que partes sequenciais da aplicação sejam codificadas pelo programador exatamente como uma sequência de comandos.

Welsh e Mainland (Welsh/04) criaram a noção de *regiões abstratas* propondo um conjunto de primitivas de programação que abstraem os detalhes da interação entre os nós de uma rede de sensores. Para simplificar a programação usando *regiões abstratas*, os autores implementaram uma abstração baseada em *threads leves* dentro da arquitetura TinyOS, permitindo a construção de uma interface síncrona de programação. Dois fluxos de execução são mantidos dentro do sistema: o fluxo principal que é dirigido a eventos e não pode bloquear; e o fluxo de controle da aplicação que pode invocar operações bloqueantes. Uma única pilha é compartilhada pelos dois fluxos de execução. Quando a aplicação bloqueia, os valores dos registradores são salvos e o controle é transferido para o sistema. O sistema restaura os valores dos seus registradores mas retoma a execução usando a mesma pilha deixada pela aplicação. A ocorrência de um evento no sistema (como temporização ou recepção de uma mensagem) pode resultar na retomada do controle da aplicação, restaurando os valores dos registradores previamente salvos. Usando essa estrutura, o código da aplicação pode bloquear enquanto o sistema permanece dirigido a eventos. Na nossa proposta, mantemos o fluxo principal de controle dirigido a eventos, e a aplicação pode ser internamente dividida em mais de um fluxo de controle, cada um com sua própria pilha de execução.

Outras abordagens, diferentes daquela proposta pelo TinyOS, têm sido experimentadas no escopo de projeto de sistemas operacionais para dispositivos em redes de sensores. Essas abordagens visam, entre outras metas, ofere-

cer uma interface de programação mais conveniente para o programador. Entre elas: a programação *multithreading*, explorada pelo sistema operacional Mantis (Abrach/03); e uma abordagem híbrida, que combina os benefícios do modelo dirigido a eventos com a programação *multithreading*, adotada pelo sistema operacional Contiki (Dunkels/04).

O objetivo principal de projeto do Mantis é reduzir a curva de aprendizado necessária para o desenvolvimento de protótipos e de aplicações básicas para redes de sensores. Para isso, aderiu-se ao modelo clássico de sistemas operacionais. Em particular, adota-se a programação *multithreading* com filas de prioridade e o mecanismo de semáforos para a sincronização entre os processos. A gerência das *threads* é sustentada por um sub-conjunto da API de *threads* do POSIX. O desafio principal é como adequar esse modelo de programação com as restrições severas de recursos computacionais dos sensores, e integrar o modelo de *multithreading* com a idéia de *threads* que "dormem" enquanto nenhum trabalho é solicitado. O espaço para a pilha de uma *thread* é alocado pelo sistema operacional na memória *heap* quando a *thread* é criada, e esse espaço é liberado quando a *thread* termina. Em cada ponto da aplicação, cada *thread* pertence a uma das duas possíveis listas mantidas pelo sistema: a lista com as *threads* prontas para executar ou a lista associada a um semáforo. Quando um dispositivo gera uma interrupção, o tratador dessa interrupção tipicamente altera um semáforo para que as *threads* na lista de espera possam ser executadas.

Da mesma forma que o Mantis, nossa abordagem opta por uma estrutura que permite a manutenção de linhas de controle distintas. Entretanto, estendendo a arquitetura do TinyOS, mantivemos o estilo reativo da programação dirigida a eventos que reproduz naturalmente o comportamento dos dispositivos (i.e., eles permanecem em estado de espera, com baixo consumo de energia, enquanto nenhum trabalho é solicitado). Além disso, o uso de corotinas como estrutura de controle, ao invés de *threads*, permite que as trocas de contextos aconteçam apenas em pontos conhecidos do código, reduzindo o custo computacional necessário para manter contextos distintos de execução. A transferência de controle embutida nas operações de interação oferece ao programador uma comodidade similar à troca de contexto automática da programação *multithreading*.

Como o TinyOS, o sistema operacional Contiki é baseado em um kernel dirigido a eventos. A diferença principal entre os dois é que o Contiki provê uma estrutura dinâmica que facilita a adaptação das aplicações em tempo de execução. No Contiki, um escalonador de eventos é responsável por despachar os eventos para serem tratados pelos processos em execução. Dois tipos de

eventos são definidos: (a) *eventos assíncronos* caracterizam uma forma de chamada de procedimento postergada, na qual os eventos são enfileirados pelo kernel e despachados posteriormente; (b) *eventos síncronos* são escalonados para execução imediatamente. O controle volta para o processo corrente apenas depois do evento disparado ter sido tratado, caracterizando um tipo de chamada de procedimento entre processos. Todos os processos usam a mesma pilha de execução que é remontada a cada invocação de um tratador de evento.

Diferentemente do TinyOS, o Contiki permite programação *multithreading* para aplicações que explicitamente requerem esse modelo de operação. Para isso, a aplicação é ligada a uma biblioteca que interage diretamente com o kernel e implementa primitivas para preempção e troca de contexto. Na extensão que propomos para o TinyOS, a opção pela gerência cooperativa das tarefas (e a consequente redução do custo computacional necessário para manter linhas de controle distintas) permite embutir a estrutura com multi-tarefa no próprio sistema, oferecendo ao programador uma interface de programação mais conveniente independente da aplicação desenvolvida.