



Universidade Federal de Pernambuco

Graduação em Engenharia da Computação

Centro de Informática

Uma extensão da máquina virtual tinyreef para adaptação dinâmica de redes de sensores sem fio

Trabalho de Graduação

Aluno: Hudson Ummem veloso (huv@cin.ufpe.br)

Orientador: nelson souto rosa (nsr@cin.ufpe.br)

RECIFE-PE

01 de dezembro de 2009

Universidade Federal de Pernambuco

Centro de Informática

Graduação em Engenharia da Computação

Hudson Ummem Veloso

*Uma Extensão da Máquina Virtual TinyReef para Adaptação Dinâmica em Redes de Sensores Sem Fio*

*Este trabalho foi apresentado ao Centro de Informática da Universidade Federal de Pernambuco como requisito parcial para a obtenção do Grau de Engenheiro da Computação.*

Orientador: Nelson souto rosa

Orientador

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

Nelson Souto Rosa

Aluno

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

Hudson Ummem Veloso

Recife, 01 de Dezembro de 2009

“Todas as coisas devem ser feitas da forma mais simples possível, porém não mais simples que o possível.”

(Albert Einstein)

AGRADECIMENTOS

Agradeço a Deus por ter me dado forças para alcançar o feito de ingressar em uma universidade e aos meus pais pelo apoio e esforço para garantir a mim e meu irmão uma educação de qualidade. Ao meu irmão, Valter pelo companheirismo.

A minha namorada Amanda pelo carinho, apoio e compreensão em todos os momentos, sempre com muito amor e sinceridade. Aos meus colegas de turma por dividir um pouco do seu conhecimento durante as aulas, projetos e exames prestados ao longo do curso. Aos colegas da iniciação científica Brazil-IP pelos momentos de descontração e pelo conhecimento compartilhado no dia-a-dia do nosso trabalho.

Ao professor Nelson pelo seu esforço e dedicação durante a orientação deste trabalho e pela ajuda na estruturação e revisão para torná-lo conciso e interessante para os leitores. Aos colegas Jobson e Igor pela ajuda com o ambiente de desenvolvimento e pelo esclarecimento de dúvidas que surgiram ao longo do estudo sobre redes de sensores.

Enfim, agradeço de coração aos meus pais e meu irmão pela formação do meu caráter, a minha professora de reforço, Jane, com quem aprendi a ter disciplina com relação aos estudos, a minha namorada pelo incentivo incondicional aos meus estudos e a todos os meus familiares e amigos pela compreensão diante da minha ausência em muitas comemorações devido ao meu compromisso com o curso de Engenharia da Computação. A todos vocês, muito OBRIGADO!

RESUMO

Devido às restrições de recursos, tais como pouca energia, ou à influência do ambiente na qual estão inseridos, os nós de uma rede de sensores sem fio (RSSF) estão constantemente sujeitos a falhas, o que pode gerar a necessidade de substituição do hardware ou reprogramação do software.

Na maioria das aplicações, não é possível ter acesso físico aos nós sensores, pois eles são implantados em locais de difícil acesso ou com agentes nocivos ao homem. Então para garantir a tolerância a falhas em redes de sensores é importante a utilização de mecanismos eficientes de gerenciamento de recursos e atualização da aplicação.

O uso de máquinas virtuais é uma solução interessante para resolver o problema de adaptação, pois elas dão suporte a linguagens com alto nível de abstração e a mecanismos de reprogramação. Linguagens de alto nível facilitam o desenvolvimento rápido de novas aplicações, de tamanho menor e, consequentemente, diminuem o custo energético do processo de reprogramação da aplicação, uma vez que menos pacotes serão enviados pela rede.

Neste contexto, o objetivo deste trabalho é apresentar uma extensão da máquina virtual TinyReef para permitir a atualização dinâmica das aplicações executando em nós das RSSFs.

**Palavras-Chaves: Redes de Sensores Sem Fio, Adaptação Dinâmica, Middleware, Máquinas Virtuais.**

ABSTRACT

Due to resource constraints such as low power, or influence the environment in which they belong, the nodes of a wireless sensor network (WSN) are constantly subject to failures, which may necessitate the replacement of hardware or reprogramming the software.

In most applications, you cannot have physical access to sensor nodes, as they are deployed in areas of difficult access or agents harmful to humans. Therefore, to ensure fault tolerance in sensor networks is important to use of efficient management of resources and upgrade the application.

The use of virtual machines is an interesting solution to solve the problem of adaptation, because they support languages with a high level of abstraction and mechanisms of reprogramming. High-level languages facilitate rapid development of new applications, smaller, and thus reduce the energy cost of reprogramming process the application, since fewer packets are sent over the network.

In this context, the objective of this study is an extension of the virtual machine TinyReef to allow the dynamic update of applications running on nodes of WSNs.

**Keywords: Wireless Sensor Networks, Dynamic Adaptation, TinyReef, TinyOS, Middleware, Virtual Machines.**

SUMÁRIO

[1 Introdução 1](#_Toc248572214)

[2 Conceitos básicos de redes de sensores sem fio 3](#_Toc248572215)

[2.1 Elementos Básicos das RSSFs 3](#_Toc248572218)

[2.2 Projeto de RSSF 5](#_Toc248572219)

[2.3 Classificação das RSSFs 7](#_Toc248572220)

[2.3.1 Quanto à Configuração 7](#_Toc248572221)

[2.3.2 Quanto ao Sensoriamento 8](#_Toc248572222)

[2.3.3 Quanto à Comunicação 9](#_Toc248572223)

[2.3.4 Quanto ao Processamento 11](#_Toc248572224)

[2.4 Componentes de uma RSSF 11](#_Toc248572225)

[2.5 *Middleware* para RSSF 13](#_Toc248572226)

[2.6 Considerações Finais 14](#_Toc248572227)

[3 Sistema Operacional TinyOS 15](#_Toc248572228)

[3.1 Funcionalidades do TinyOS 15](#_Toc248572230)

[3.1.1 Fluxo de Execução das Tarefas 16](#_Toc248572231)

[3.1.2 Conjunto de Serviços 16](#_Toc248572232)

[3.2 Modelo dos Componentes 17](#_Toc248572233)

[3.2.1 Comandos 19](#_Toc248572234)

[3.2.2 Eventos 20](#_Toc248572235)

[3.2.3 Tarefas 20](#_Toc248572236)

[3.3 Ambiente de Programação 20](#_Toc248572237)

[3.3.1 Linguagem de Programação NesC 20](#_Toc248572238)

[3.3.2 Simulador TOSSIM 25](#_Toc248572239)

[3.4 Considerações Finais 26](#_Toc248572240)

[4 Máquina Virtual TinyReef 27](#_Toc248572241)

[4.1 Objetivos 27](#_Toc248572243)

[4.2 Arquitetura da TinyReef 27](#_Toc248572244)

[4.2.1 Módulos 28](#_Toc248572245)

[4.2.2 Instruções de Máquina 30](#_Toc248572246)

[4.2.3 Implementação da TinyReef 30](#_Toc248572247)

[4.2.4 Estrutura dos Programas 31](#_Toc248572248)

[4.3 Considerações Finais 32](#_Toc248572249)

[5 Extensão à TinyReef 33](#_Toc248572250)

[5.1 Objetivos 33](#_Toc248572252)

[5.2 Modificações na Máquina Virtual TinyReef 35](#_Toc248572253)

[5.3 Definição do Protocolo de Comunicação 36](#_Toc248572254)

[5.3.1 Formato, Tipos e Campos das Mensagens 37](#_Toc248572255)

[5.3.2 Comportamento dos Nós 40](#_Toc248572256)

[5.4 Fluxo de Desenvolvimento 42](#_Toc248572257)

[5.4.1 Ambiente de Verificação 42](#_Toc248572258)

[5.4.2 Desenvolvimento e Validação do Protocolo 45](#_Toc248572259)

[5.4.3 Exemplo de Funcionamento 56](#_Toc248572260)

[5.5 Considerações Finais 57](#_Toc248572261)

[6 Trabalhos relacionados 58](#_Toc248572262)

[6.1 Maté: *a tiny virtual machine for sensor networks* 58](#_Toc248572264)

[6.2 Davim: *a dynamically adaptable virtual machine for sensor networks* 59](#_Toc248572265)

[6.3 Considerações Finais 60](#_Toc248572266)

[7 Conclusão e Trabalhos Futuros 61](#_Toc248572267)

[Referências 62](#_Toc248572268)

LISTA DE FIGURAS

[Figura 2.1: Arquitetura básica de um nó sensor 12](#_Toc248572269)

[Figura 2.2: Exemplo de rede de sensores sem fio com nós *gateway* e *data sink.* 13](#_Toc248572270)

[Figura 3.1: Hierarquia de componentes em uma aplicação de roteamento *ad hoc* baseada no TinyOS (Buonadonna, Hill e Culler 2001). 18](#_Toc248572271)

[Figura 3.2: Comunicação entre componentes através de comandos e eventos (Ruiz, et al. 2004). 19](#_Toc248572272)

[Figura 3.3: Exemplo da descrição de um componente em NesC (Gay, et al. 2003) 21](#_Toc248572273)

[Figura 3.4: Exemplo de descrição da interface de um componente em NesC (Gay, et al. 2003) 22](#_Toc248572274)

[Figura 3.5: Exemplo de código de configuração *top level* em NesC 23](#_Toc248572275)

[Figura 3.6: Exemplo de implementação de um componente do tipo módulo 24](#_Toc248572276)

[Figura 4.1: Exemplo comparativo entre instruções baseadas em registradores e instruções baseadas em pilha (Marques, Ronan e Rosa 2009) 28](#_Toc248572277)

[Figura 4.2: Visão lógica da arquitetura da máquina virtual TinyReef (Marques, Ronan e Rosa 2009) 28](#_Toc248572278)

[Figura 4.3: Exemplos de formato de instruções da TinyReef (Marques, Ronan e Rosa 2009) 30](#_Toc248572279)

[Figura 4.4: Exemplo de programa TinyReef que coleta dados de um sensor e os envia via rádio (Marques, Ronan e Rosa 2009). 31](#_Toc248572280)

[Figura 5.1: Exemplo de uso do serviço de adaptação 34](#_Toc248572281)

[Figura 5.2: Representação gráfica da interconexão dos componentes da TinyReef 35](#_Toc248572282)

[Figura 5.3: Formato padrão do *frame* 802.15.4 do TinyOS 37](#_Toc248572283)

[Figura 5.4: Formato dos pacotes do protocolo de atualização 38](#_Toc248572284)

[Figura 5.5: Fluxo de tarefas do nó sensor durante a atualização do software aplicativo 40](#_Toc248572285)

[Figura 5.6: Plataforma de teste de hardware da TinyReef 43](#_Toc248572286)

[Figura 5.7: Interface do programa *programaSender* 44](#_Toc248572287)

[Figura 5.8: Assinatura das interfaces do componente *UpdateHandler* 46](#_Toc248572288)

[Figura 5.9: Estrutura do pacote de requisição de atualizações 46](#_Toc248572289)

[Figura 5.10: Comando *updateProgram* provido pelo componente *UpdateHandler* 47](#_Toc248572290)

[Figura 5.11: Estrutura do pacote de recebimento de atualização 47](#_Toc248572291)

[Figura 5.12: Trecho de código do comando *programAssemble* 48](#_Toc248572292)

[Figura 5.13: Código do evento *TimeoutTimer* 49](#_Toc248572293)

[Figura 5.14: Código da função *checkPacketSequence* 50](#_Toc248572294)

[Figura 5.15: *Loop* do comando *prepareProgram* que monta a nova versão do programa aplicativo 51](#_Toc248572295)

[Figura 5.16: Estrutura que representa uma instrução da máquina virtual TinyReef 52](#_Toc248572296)

[Figura 5.17: Código da função *receive* do componente *ReceiveBytecode* 53](#_Toc248572297)

[Figura 5.18: Programa usado como exemplo para validação do serviço de adaptação 56](#_Toc248572298)

LISTA DE TABELAS

[Tabela 3.1: Subconjunto das interfaces providas pelo TinyOS (Levis, Madden, et al. s.d.) 17](#_Toc248572299)

[Tabela 5.1: Comparativo de utilização de memória da máquina virtual TinyReef executando em diferentes versões do TinyOS 55](#_Toc248572300)

[Tabela 5.2: Número de linhas de código de alguns componentes da TinyReef antes e depois da introdução do serviço de atualização. 55](#_Toc248572301)

# Introdução

O desenvolvimento das tecnologias de transmissão sem fio aliado aos avanços no processo de fabricação de circuitos integrados permitiu a miniaturização e a produção em larga escala de chips com baixo custo e foram determinantes para o surgimento e o desenvolvimento das redes de sensores sem fio (RSSF).

As RSSF são formadas por centenas ou milhares de dispositivos pequenos chamados de nós sensores que possuem capacidades limitadas de processamento, armazenamento, comunicação e energia. Desse modo eles devem cooperar entre si para juntos realizar uma tarefa específica.

Devido às restrições de recursos como pouca energia bem como a influência do ambiente na qual estão inseridos, os nós, constantemente, estão sujeitos a falhas o que pode gerar a necessidade de reparo ou substituição.

Na maioria das aplicações não é possível acessar nós sensores fisicamente, havendo nesse caso a necessidade da adoção abordagens preventivas baseadas no gerenciamento de recursos e de adaptação.

Para dar suporte as características mencionadas existem algumas soluções baseadas no uso de *middleware* que disponibilizam um conjunto de serviços que facilitam o gerenciamento de recursos e técnicas de agregação de dados como forma de diminuir o consumo de energia.

O grande desafio para este tipo de solução é a implementação desses serviços com a quantidade de memória disponível na maioria dos nós sensores (cerca de 4KB) e também a queda de desempenho com relação a processamento e economia de energia com a adição de uma nova camada de software .

Alguns sistemas de middleware para RSSF têm sido construídos sobre máquinas virtuais, pois elas provêem um eficiente paradigma de programação que permite o desenvolvimento com alto nível de abstração e dão suporte a mecanismos que promovem o dinamismo da rede .

Máquinas virtuais como Maté e Davim possuem como características comuns um modelo de execução de aplicações baseado na interpretação de *bytecodes*, arquitetura de instruções baseadas em manipulações de dados na pilha e suporte a atualização da aplicação.

Esse modelo de execução permite definir um repertório de instruções que abstrai os comandos do sistema operacional executando na camada abaixo da máquina virtual. Eles são interessantes, pois permitem especificar aplicações menores com o uso dessa abstração e por outro lado diminuem o custo de envio de atualização da aplicação via pacotes.

Baseado nesse modelo de execução foi construído a máquina virtual TinyReef cuja arquitetura seguiu uma abordagem diferente das anteriores, pois seu repertório de instruções é baseado em manipulações de dados em registradores .

O objetivo da TinyReef é promover as mesmas funcionalidades de abstração e atualização, mas buscando diminuir os impactos referentes à interpretação de *bytecodes* através da abordagem baseada em registradores.

Neste contexto, o objetivo deste trabalho é apresentar uma extensão da máquina virtual TinyReef para permitir a atualização dinâmica das aplicações executando em nós das RSSFs.

O texto está organizado da seguinte forma: o Capítulo 2 apresenta os principais conceitos relacionados a redes de sensores sem fio, o Capítulo 3 faz um resumo do funcionamento do sistema operacional TinyOS. O Capítulo 4 descreve a máquina virtual TinyReef através da análise de sua arquitetura e o Capítulo 5 mostra as etapas de desenvolvimento do suporte a atualizações. O Capítulo 6 faz um resumo do funcionamento das máquinas virtuais Maté e Davim e, por fim, o Capítulo 7 encerra o texto apresentando as conclusões do trabalho e as direções para trabalhos futuros.

# Conceitos básicos de redes de sensores sem fio



Neste capítulo será apresentada uma visão geral sobre os principais conceitos relacionados à tecnologia de Redes de Sensores e *middleware* para RSSF.

## Elementos Básicos das RSSFs

As RSSFs têm como características principais: capacidade de auto-organização, formada por um grande número de nós sensores, os quais são implantados, de maneira planejada ou aleatória, em ambientes de difícil acesso ou hostis ao ser humano.

Os nós sensores devem possuir capacidade de comunicação, de processamento dos dados coletados e deve existir cooperação entre eles .

**- Capacidade de auto-organização:**

A rede deve ser capaz de se auto-organizar estabelecendo as rotas de encaminhamento de pacotes ao longo da rede.

Caso um nó sensor, que compõe alguma rota de pacotes, tenha sido inutilizado devido a alguma falha (por exemplo, avarias físicas, pouca energia ou interferências do ambiente), novas rotas devem ser criadas e a topologia da rede, ou seja, a forma como os nós estão ligados, poderá mudar.

**- Grande quantidade de nós:**

Dependendo do tamanho da área monitorada, do fenômeno em observação e da forma como os nós sensores são implantados, a quantidade de nós pode ser muito grande.

Algumas limitações de hardware como, por exemplo, o alcance do *link* de comunicação pode ser usado como parâmetro para determinar a distância máxima entre nós que se comunicam.

**- Modo de implantação:**

Nós sensores são implantados dentro da área onde ocorre o fenômeno em observação e dependendo do ambiente que será monitorado e do tipo da aplicação, esta tarefa poderá ser feita de maneira aleatória ou planejada.

Na implantação aleatória, eles são “soltos” no ambiente, devido a empecilhos, tais como dificuldade de acesso ou presença de agentes nocivos ao homem, impedindo a sua implantação da maneira tradicional. Na implantação planejada, os nós são posicionados de forma a facilitar a comunicação e, por conseguinte o funcionamento da rede.

**- Capacidade de comunicação:**

Os nós sensores devem possuir um mecanismo de comunicação por onde os dados coletados poderão ser transmitidos.

**- Processamento dos dados:**

Os dados transmitidos devem ser processados antes de serem enviados, de modo a ser enviado apenas o que for estritamente necessário a rede. Por exemplo, pode-se configurar um nó sensor para enviar um dado somente quando o valor amostrado pertencer a uma faixa de valor pré-configurada na aplicação.

**- Cooperação entre os nós:**

Os nós sensores comunicam-se de modo a definir uma rota para que os dados possam ser trafegados ao longo da rede. Além disso, eles podem utilizar um modelo de comunicação cooperativo para disseminar informações pela rede sobre o fenômeno. Um exemplo desse tipo de abordagem é usado no algoritmo *Low Energy Adaptive Clustering Hierarchy*.(Heinzelman e Chandrakasan 2000).

Diante das características citadas é de extrema importância que os nós sensores possuam um baixo custo de produção, baixo consumo de energia, sejam pequenos e multifuncionais (além de sensoriamento, forneçam informações de posicionamento ou se adaptem a condições adversas, por exemplo). .

Apesar de possuírem tais restrições, os avanços nas tecnologias de desenvolvimento de dispositivos eletrônicos e pesquisas em técnicas eficientes de transmissão sem fio permitem a utilização de RSSF em uma grande variedade de áreas, tais como: medicina, militar, agronegócio, ecologia, transportes, monitoração e controle industrial, áreas de desastre, monitoramento da qualidade de produtos, engenharia naval, segurança pública e de ambientes em geral e assim por diante. .

## Projeto de RSSF

Segundo Akyildiz , os principais fatores de influência sobre o projeto de RSSF são: tolerância a falha, escalabilidade, custo de produção dos nós, ambiente de operação, topologia de rede, restrições de hardware, meios de transmissão e consumo de potência.

**- Tolerância a falhas:**

As principais causas de falha em RSSF decorrem de avarias físicas, pouca energia e interferência do ambiente. O objetivo da capacidade de tolerar falhas consiste em manter as tarefas da rede funcionando mesmo depois de sofrer uma falha. Ela pode ser usada para medir a confiabilidade de uma rede.

**- Escalabilidade:**

A rede deve ser capaz de trabalhar com um grande número de nós e suportar a adição de novos nós ao sistema.

**- Custo de produção:**

Devido ao grande número de nós empregados, a soma do custo de todos eles deve ser baixa o bastante para justificar a implantação da rede de sensores ao invés de implantar sensores da maneira tradicional e coletar os dados de forma manual ou usando robôs.

**- Ambiente de operação:**

Em geral, nós sensores são implantados dentro da área onde ocorre o fenômeno em observação.

**- Topologia de Rede:**

A grande quantidade de nós sensores requer um cuidado adicional para manutenção da topologia de rede, ou seja, o modo como os nós estão ligados. A manutenção dela passa por algumas fases.

A fase inicial cuida da implantação dos nós definindo a forma como eles serão implantados. Isto pode ser feito de maneira aleatória ou planejada. Depois de implantada, a topologia pode mudar devido a fatores inerentes ao ambiente ou ao tipo de aplicação, ou a restrições de energia. Após a modificação da topologia, pode ser necessário reimplantar novos nós para manter o funcionamento da rede.

**- Restrições de Hardware:**

O hardware dos nós sensores deve ser o mais eficiente possível de modo a consumir pouca energia, o que permitirá um tempo de vida maior para a rede. Além das limitações energéticas, eles possuem baixa capacidade de processamento e armazenamento devido ao seu tamanho, que deve ser pequeno, e ao custo de produção, que deve ser baixo.

**- Meios de transmissão:**

O meio de transmissão é definido pela forma de transmitir os dados e o tipo de *link*. A forma de transmissão mais usada em RSSF é a multi-salto. O *link* sem fio pode ser do tipo: rádio freqüência, infravermelho ou óptico.

Transmissão multi-salto consiste em definir uma rota passando por diversos nós até chegar ao nó de destino. A transmissão de dados de um nó para outro ao longo de uma rota é considerada um salto.

**- Consumo de Potência:**

Fatores energéticos são bastantes críticos em RSSF uma vez que os nós não são facilmente acessíveis e por isso o reabastecimento de bateria não é possível. Desse modo, durante o projeto dos protocolos e algoritmos que irão executar no nó sensor deve-se sempre buscar a forma mais eficiente de realizar a tarefa tendo em vista a conservação de energia como prioridade.

## Classificação das RSSFs

De acordo com Ruiz , a aplicação influenciará diretamente no modo como a rede será construída devendo o projetista considerar, além dos requisitos funcionais da aplicação, aspectos relacionados tanto ao ambiente onde os sensores serão inseridos quanto às características e restrições dos nós sensores a serem utilizados. Baseado nessa constatação, as RSSF foram classificadas quanto à configuração, ao sensoriamento, à comunicação e ao processamento.

### Quanto à Configuração

Quanto à configuração podemos subdividir as RSSF em cinco subclasses que são: composição, organização, mobilidade, densidade, distribuição.

Segundo a composição, a configuração da RSSF pode ser homogênea ou heterogênea. A tipificação se dá de acordo com o tipo de hardware utilizado na rede. Se toda a rede é composta por nós com a mesma especificação de hardware dizemos que ela é uma rede homogênea caso contrário será considerada heterogênea. Eventualmente, podem estar executando softwares diferentes em nós de uma rede homogênea.

Segundo a organização, a configuração da RSSF pode ser hierárquica ou plana. Em ambas, os nós estão dispostos em grupos, porém se existe alguma hierarquia dentro de um grupo, dizemos que ela é uma rede hierárquica, caso contrário, será considerada plana. Um exemplo de configuração de RSSF hierárquica pode ser visto em Heinzelman .

Segundo a mobilidade, a configuração da RSSF pode ser móvel ou estacionária. A tipificação se dá de acordo com o posicionamento dos nós sensores durante o tempo de vida da rede. Se um nó permanece no mesmo local desde a sua deposição na área de monitoramento até o fim da vida da rede dizemos que é uma rede estacionária caso contrário será considerada móvel.

Segundo a densidade, a configuração da RSSF pode ser balanceada, ou densa, ou esparsa. A tipificação se dá de acordo com a distribuição e concentração dos nós sensores por unidade de área de monitoramento.

Se a distribuição foi realizada de maneira considerada ideal para o objetivo da aplicação dizemos que a rede é balanceada. Se a distribuição apresenta alta concentração de nós por unidade de monitoramento a rede é densa. Se ela apresenta baixa concentração de nós por unidade de área estamos diante de uma rede esparsa.

Segundo a distribuição, a configuração da RSSF pode ser irregular ou regular. A tipificação se dá de acordo com a uniformidade de distribuição dos nós sensores na área monitorada. Se os nós estão distribuídos ao longo da área monitorada uniformemente a rede será dita regular caso contrario será considerada irregular.

### Quanto ao Sensoriamento

Quanto ao sensoriamento podemos subdividir as RSSFs de acordo com a regularidade em que a tarefa de coleta de dados é realizada. O sensoriamento dos nós da RSSF pode ser periódico, ou contínuo, ou reativo, ou tempo real. A tipificação se dá de acordo com intervalos de tempo entre as coletas de dados.

Se a coleta é feita em intervalos regulares diz-se que o sensoriamento é regular, se os dados são coletados continuamente para, por exemplo, formação de base dados para pesquisas temos uma rede de sensoriamento contínuo.

Caso a coleta ocorra devido à ocorrência de um determinado evento ou quando solicitado por um observador, neste caso temos um sensoriamento reativo e quando os nós coletam a maior quantidade de informações no menor intervalo de tempo possível temos um sensoriamento de tempo real.

### Quanto à Comunicação

Quanto à comunicação podemos subdividir as RSSF em cinco subclasses que são: disseminação, tipo de conexão, transmissão, alocação de canal e fluxo de informação.

Segundo a disseminação de dados, a comunicação na RSSF pode ser programada, ou contínua, ou sob demanda. A tipificação se dá de acordo com intervalos de tempo entre disseminações de dados.

Se a disseminação é feita em intervalos regulares diz-se que a comunicação é programada, se disseminação ocorre de forma contínua à comunicação será dita contínua. Caso a disseminação ocorra devido à ocorrência de um determinado evento ou quando solicitado por um observador, neste caso temos uma RSSF com comunicação sob demanda.

Segundo o tipo de conexão, a comunicação na RSSF pode ser simétrica ou assimétrica. A tipificação se dá de acordo com o alcance da conexão entre os nós sensores. Se o alcance varia durante a transferência de informações entre os nós sensores, a rede é dita de comunicação assimétrica, exceto no caso da comunicação com o nó sorvedouro, caso contrário, a rede é classificada como simétrica.

Segundo o tipo de conexão, a comunicação na RSSF pode ser *simplex*, *half-duplex* ou *full-duplex*. A tipificação se dá de acordo com a capacidade de transmissão e recepção da unidade transceptora dos nós sensores.

Se os nós sensores possuírem uma unidade capaz apenas de enviar dados classifica-se a rede como de comunicação simplex. Caso a unidade consiga enviar e receber dados em intervalos distintos diz-se que é uma rede de comunicação duplex.

Se for possível enviar e receber dados ao mesmo tempo a rede será dita de comunicação *full-duplex*.

Segundo a alocação de canais, a comunicação na RSSF pode ser estática ou dinâmica. A tipificação ocorre considerando-se o modo como é dividida a largura de banda da rede, ou seja, a quantidade máxima de dados que podem ser transmitidos ao longo do link de comunicação.

Se a divisão for fixa, todos os nós devem possuir partes iguais para transmitir (largura de banda dos nós igual), desse modo dizemos que a comunicação da rede é estática. Caso não exista atribuição fixa de largura de banda para os nós e eles disputem o canal de transmissão, neste caso a comunicação será considerada dinâmica.

Segundo o fluxo de informação, a comunicação na RSSF pode ser *flooding*, *multicast*, *unicast*, *gossiping*, *bargaining*. A tipificação ocorre considerando-se o modo como os pacotes são roteados ao longo da rede.

O fluxo de informação de uma RSSF será considerado do tipo *flooding* (também conhecido por algoritmo de inundação), quando os nós sensores fazem *broadcast* de suas informações (enviam mensagens iguais a todos os seus vizinhos). Por sua vez, os vizinhos também o fazem para todos os outros nós até que essas informações cheguem ao ponto de acesso da rede.

Quando os nós sensores estão agrupados e o fluxo de informação é feito por meio daqueles pertencentes ao grupo, diz-se então que a comunicação da rede é do tipo *multicast* segundo o fluxo de informações.

Nesta abordagem, os nós pertencentes ao grupo podem ser agregados de acordo com algum atributo como, por exemplo, seu posicionamento na área monitorada ou então depois de enviar uma solicitação de agregação ao nó *sink*. Os tipos de nós de RSSF serão vistos na Seção 2.4.

Caso a comunicação entre os nós sensores e o ponto de acesso seja feita diretamente usando protocolos de roteamento *multihop* (multi-saltos) a rede terá comunicação do tipo *unicast* segundo o fluxo de informações.

O fluxo de informações do tipo *gossiping* é uma derivação do *flooding*. A diferença é que um nó sensor não retransmite a informação em *broadcast*, mas para um nó vizinho escolhido de maneira randômica . Neste caso, temos uma rede classificada segundo a comunicação como *gossiping*.

Por fim, o fluxo de informações na rede será do tipo *bargaining*, somente quando o nó destino manifestar interesse em receber as informações, desse modo, deve existir um processo de negociação.

### Quanto ao Processamento

Quanto ao processamento podemos subdividir as RSSF de acordo com a cooperação existente entre os nós que compõem a rede. O processamento na RSSF pode ser deinfra-estrutura, localizado, ou correlativo. A tipificação ocorre considerando-se os procedimentos executados pelos nós sensores.

Se os nós sensores executam apenas procedimentos relacionados à infra-estrutura da rede, como por exemplo, algoritmos de roteamento, de controle de acesso ao meio, descoberta de localização, criptografia. Neste caso, a rede será considerada de cooperação do tipo infra-estrutura.

Caso os componentes da RSSF desempenhem tarefas além das relacionadas à infra-estrutura da rede, como por exemplo, processamento local para tradução de dados coletados segundo algum critério pré-definido na aplicação, a rede será classificada segundo a cooperação como de processamento local.

Se as tarefas forem relacionadas à correlação dos dados como agregação, compressão, contagem, supressão seletiva a rede será classificada como de processamento correlativo.

## Componentes de uma RSSF

Uma RSSF é composta por vários nós sensores, os quais se comunicam formando a rede: nós sensores e estações bases.

Nós sensores são dispositivos autônomos equipados com capacidade de sensoriamento, processamento e comunicação. Quando estes nós são dispostos em rede em um modo *ad hoc*, formam as RSSF. Eles coletam dados via sensores, os processam localmente ou coordenadamente entre vizinhos podendo enviar a informação para o usuário ou, em geral para um *data* *sink* (também conhecidos como sorvedouros).

Os nós podem possuir tarefas diferentes na rede como sensoriamento do ambiente, processamento de informação e tarefas associadas ao roteamento de dados. Os nós *data sink* (também conhecidos como sorvedouros) são aqueles cuja tarefa é processar as informações vindas de vários nós de sensoriamento.

Um exemplo de nó sorvedouro pode ser usado em abordagens cooperativas onde existe uma agregação dos dados vindos de alguns nós cuja função é fazer a coleta dos dados e enviá-los para aqueles.

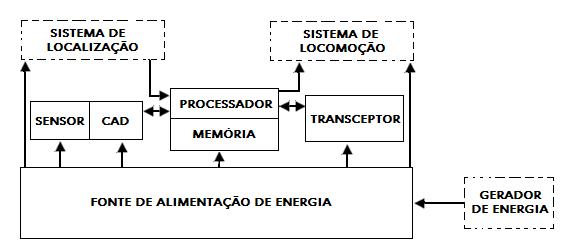


Figura .: Arquitetura básica de um nó sensor

Os componentes básicos de um nó sensor são mostrados na Figura 2.1: unidade de sensoriamento (compreende o sensor e o conversor analógico digital - CAD), processador, transceptor (usado para realizar a transmissão e recepção de dados), bateria (fonte de alimentação) e memória.

Os componentes com bloco tracejado são dependentes da aplicação: sistema de localização, gerador de energia e sistema de locomoção.

Os *gateways* (estações base) são os nós responsáveis pela ligação da RSSF com redes externas como a Internet, outras RSSF ou sistemas de controle e comando.

Os *gateways* possibilitam a troca de informações entre os nós comuns da rede e o mundo exterior, através de um canal de comunicação sem fio ou cabeado com a estação base. A estação base é um computador capaz de se comunicar com o nó *gateway*, e por onde os usuários analisam os dados coletados.

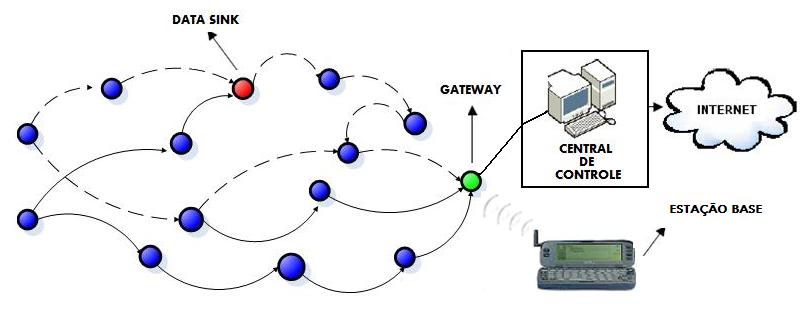


Figura .: Exemplo de rede de sensores sem fio com nós *gateway* e *data sink.*

A Figura 2.2 mostra um exemplo de rede formada por nós sensores (bolas azuis), um nó sorvedouro (bola vermelha) e um *gateway* (bola verde) ligado a uma estação base (via comunicação sem fio) e a uma central de controle (via interface cabeada).

## *Middleware* para RSSF

Plataformas de *Middleware* vêm sendo desenvolvidas para RSSF para tratar alguns dos problemas relacionados ao projeto e gerenciamento desse tipo de rede, como mencionados na Seção 2.2.

Seu principal objetivo é dar suporte à abstração de serviços como: gerenciamento da topologia de rede e da heterogeneidade dos nós sensores, adaptabilidade ao ambiente de operação, mecanismos de tolerância a falhas e de cooperação entre os nós e gerenciamento de recursos (energia, memória, comunicação).

Heterogeneidade e adaptabilidade são características essenciais em um projeto de *Middleware* (de Freitas 2008). A primeira, porque visa prover uma visão homogênea da rede abstraindo os detalhes da plataforma de hardware utilizada o que simplifica a atualização e substituição dela.

A segunda devido à dinamicidade das redes de sensores que devem se adaptar de acordo com as condições de operação (por exemplo, pouca energia) e mudanças no ambiente de monitoramento (por exemplo, influências temporais) para se manter funcionando. Dito isto, é extremamente importante se ter um mecanismo confiável de reprogramação .

Alguns exemplos de middleware que provêem mecanismos de coleta de dados através de consultas de alto nível semelhantes com as da linguagem SQL são o TinyDB e COUGAR (Demers, et al. 2003).

Um modelo de *middleware* que contempla serviços como suporte a programação em alto nível são as máquinas virtuais como a TinyReef , a qual se fala em detalhes no capítulo 4.

## 

## Considerações Finais

Ao longo deste capítulo viram-se as principais características e classificações das RSSF, bem como os aspectos que influenciam seu planejamento e implantação. Viu-se também que elas são altamente dependentes da aplicação e que possuem sérias restrições de recursos.

Propostas para vencer os desafios impostos a seu funcionamento foram apresentadas como, por exemplo: o uso de plataformas de middleware para abstrair serviços e facilitar o desenvolvimento das aplicações.

# Sistema Operacional TinyOS

O TinyOS é um sistema operacional projetado especificamente para RSSF. Ele vem sendo usado como uma plataforma para pesquisas em diversos projetos ao redor do mundo. Seus principais objetivos são atender aos aspectos críticos de RSSF como recursos limitados, concorrência de eventos reativos, plataforma de desenvolvimento flexível e promover economia de potência .

A abordagem escolhida para atender tais requisitos foi utilizar um modelo de funcionamento baseado em eventos com arquitetura modular personalizável, onde novos componentes podem ser adicionados ao núcleo básico do sistema de acordo com a necessidade da aplicação. O núcleo é composto basicamente de um escalonador de tarefas e um tratador de interrupções e ocupa cerca de 400 *bytes* da memória.

Sua extensa utilização deve-se em parte ao suporte que ele provê a uma grande variedade de plataformas de hardware que incluem diversos processadores e chips de rádio. Além disso, ele possui um ambiente de programação que incorpora ferramentas de depuração, visualização, simulação e de suporte a integração de RSSF com redes externas. A arquitetura modular e o fato do código ser aberto permitem uma constante evolução do sistema .

Ao longo deste capítulo serão apresentados detalhes sobre os aspectos funcionais do TinyOS. A Subseção 3.1 apresenta o seu funcionamento básico, a Subseção 3.2 explica como é organizado o modelo de componentes e, por fim, a Subseção 3.3 apresenta o ambiente de programação e as principais ferramentas disponíveis.



## Funcionalidades do TinyOS

O funcionamento do TinyOS é baseado num conjunto de eventos e serviços. Os eventos são estímulos, vindos do ambiente monitorado, que o sistema deve ser capaz de detectar e realizar alguma tarefa.

Os serviços são um conjunto de tarefas, comuns a aplicações de RSSF, que são adicionadas ao núcleo do sistema operacional através de interfaces e componentes. As Subseções 3.1.1 e 3.1.2 apresentam maiores detalhes sobre o fluxo de execução e os serviços disponibilizados.

### Fluxo de Execução das Tarefas

O núcleo do sistema operacional é composto por um gerenciador de interrupções e um escalonador de tarefas. O gerenciador de interrupções é responsável por notificar o escalonador de tarefas sobre a ocorrência e o tipo de evento sinalizado pelo hardware. O escalonador, por sua vez, executa as tarefas de acordo com o evento e a ordem de chegada.

O escalonamento é baseado em uma fila (FIFO - *First In First Out*) com tamanho limitado onde as tarefas são executadas até o fim sem que haja interrupção para execução de outras tarefas, exceto quando esta invocar um evento ou ocorrer um evento devido à interrupção de hardware .

Este modelo foi usado, pois suporta concorrência sem ocupar muito espaço na memória, além de economizar mais energia se comparado a outras abordagens baseadas, por exemplo, em troca de contexto e pilha. Esta última consome mais memória e energia devido ao processamento inerente as operações de escrita e leitura de dados do contexto (valores dos registradores) na memória.

Neste modelo de eventos, os ciclos ociosos do processador, depois que as tarefas são finalizadas e a fila fica vazia, são utilizados no estado de economia de energia e assim permanecem até que uma nova tarefa seja colocada na lista.

### Conjunto de Serviços

Os serviços providos pelo TinyOS têm o objetivo de abstrair os recursos de hardware através de componentes e interfaces. Os componentes são entidades independentes que executam serviços e possuem uma ou mais interfaces de comunicação. Um componente possui duas classes de interfaces: aquelas que ele provê e aquelas que ele usa. Interfaces definem como os componentes interagem.

Cada componente é composto de comandos, eventos e tarefas. Os comandos e eventos são mecanismos de comunicação entre componentes, e as tarefas são usadas para resolver questões internas de processamento concorrente. Mais detalhes sobre a arquitetura interna dos componentes serão vistos na Seção 3.2.

Tabela 3.1: Subconjunto das interfaces providas pelo TinyOS (Levis, Madden, et al. s.d.)

|  |  |
| --- | --- |
| Interface | Descrição |
| ADC | Comunicação com sensores |
| *Clock* | Controle do *clock* |
| *Leds* | Acender e apagar os *LEDs* |
| *ReceiveMsg* | Recebimento de mensagens |
| *SendMsg* | Envio de mensagens |
| *StdControl* | Inicialização e Parada de componentes |
| *Time* | Manipulação de temporizador |

A mostra algumas interfaces disponíveis no TinyOS. A versão mais recente do sistema (2.1.0) provê um grande número de componentes que garantem abstração de sensores, protocolos de roteamento, suporte a gerenciamento de energia, acesso a contadores, diferentes modos de armazenamento e assim por diante.

## Modelo dos Componentes

O modelo de componentes em conjunto com a descrição das suas interfaces visa facilitar a modularidade do sistema. Este modelo é provido pela linguagem NesC . NesC encapsula no componente um conjunto de serviços especificados em suas interfaces.

As interfaces de componentes especificam um conjunto de comandos e eventos. Os ações de um comando são implementadas no componente que provê a interface e os eventos têm seu comportamento definido pelo componente usuário da interface.

Uma aplicação NesC é composta de um ou mais componentes interligados para formar um código executável. A conexão entre componentes da aplicação e do TinyOS é feita através de uma configuração de ligação. Ela define quais módulos serão usados durante o processo de compilação do código do sistema. Serão adicionados ao código binário gerado pelo compilador de NesC somente os componentes que estiverem ligados.

Esta configuração será usada para compor uma hierarquia de componentes que pode ser representada em camadas como mostra o exemplo da Figura 3.1. Neste exemplo, a configuração da aplicação pode usar a interface provida pelo componente *Active Messages* para envio de mensagens através de duas interfaces de comunicação diferentes (serial e rádio).

As três setas para baixo no componente *Active Messages* indicam que ele usa três comandos providos por cada um dos componentes abaixo dele enquanto que as duas setas para cima indicam os eventos que ele deve definir o comportamento.

Esse conjunto de comandos e eventos é definido para que o componente *Active Messages* consiga atingir seu objetivo de abstrair, para os componentes que usam os comandos providos por ele, o meio de transmissão das mensagens.

Desse modo, o componente *Active Messages* instancia as interfaces dos componentes *RadioPacket* e *SerialPacket* e estes se comunicam com a camada mais baixa de software composta pelos os componentes do TinyOS responsáveis pela comunicação com o hardware.

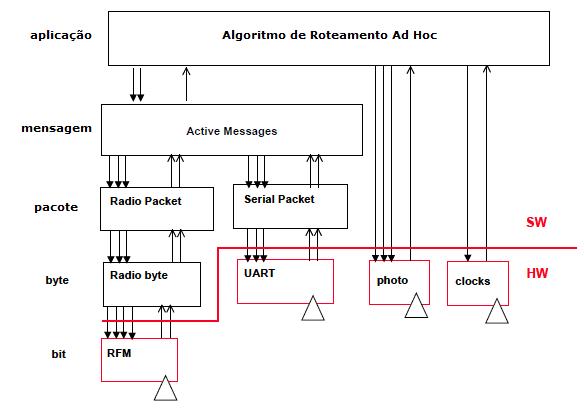
i

Figura .: Hierarquia de componentes em uma aplicação de roteamento *ad hoc* baseada no TinyOS .

A arquitetura de um componente é formada por comandos, eventos e tarefas. Um componente pode prover e usar uma mesma interface várias vezes contanto que defina um nome diferente para cada instância. A seguir são apresentados os conceitos sobre comandos, eventos e tarefas.

### Comandos

Os comandos são pedidos assíncronos feitos de um componente de uma camada mais alta a outros de camadas inferiores. Eles tipicamente requisitam que um componente execute alguma tarefa, como por exemplo, inicializar um temporizador ou acender um *led*. Comandos não bloqueiam o funcionamento do sistema, ou seja, eles apenas invocam tarefas de outros componentes e devem retornar a execução para seu chamador.

Essas tarefas serão colocadas na fila de tarefas para serem executadas e quando terminadas deverão ser gerados eventos indicando sua conclusão. A Figura 3.2 ilustra a comunicação entre componentes através de comandos e eventos.

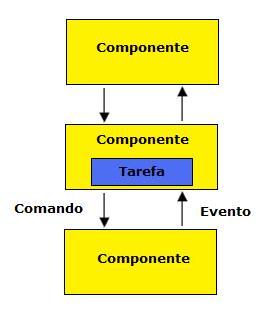


Figura .: Comunicação entre componentes através de comandos e eventos .

### Eventos

Eventos podem ser sinalizados devido a alguma interrupção de hardware ou quando uma tarefa foi finalizada notificando o componente, pertencente a uma camada de nível mais alto, que a requisitou. Um evento pode também iniciar tarefas ou chamar comandos de componentes de nível inferior.

Comandos não podem sinalizar eventos para que o sistema não corra o risco de entrar em loop. Eles utilizam uma pequena quantidade de processamento que está relacionada a interrupções de relógio e interrupções do conversor analógico digital.

### Tarefas

Tarefas são usadas em computações básicas não críticas quanto ao tempo. Elas não podem bloquear a execução de outra tarefa, mas podem ser interrompidas por eventos. As tarefas podem invocar comandos de nível inferior, sinalizar eventos de um nível superior e programar outras tarefas dentro de um componente.

## Ambiente de Programação

Além do núcleo do sistema operacional, existe um ambiente de programação que diminui o tempo de desenvolvimento de aplicações para RSSF baseados no TinyOS. Estas ferramentas são usadas para depuração, simulação e compilação, que geram código de máquina para diferentes plataformas de hardware a partir da linguagem de programação NesC (pronuncia-se Nes-see) . Essa linguagem e o simulador TOSSIM, serão abordados nas Seções 3.3.1 e 3.3.2 a seguir.

### Linguagem de Programação NesC

NesC é uma linguagem de programação projetada para sistemas de rede embarcados que possuem características como execução concorrente, programas reativos a eventos e que devem operar sobre restrições de memória e energia. Dentro dessa classe de sistemas encontram-se as redes de sensores.

Duas das principias motivações para o desenvolvimento da linguagem foram o suporte e a implementação do modelo de programação para o TinyOS. Para atender tais requisitos os projetistas adotaram um modelo de programação com execução baseada em eventos, suporte à concorrência e código orientado a componentes.

Esta abordagem é baseada em três propriedades. A primeira, todos os recursos devem ser conhecidos em tempo de compilação. A segunda, as aplicações são construídas juntando um conjunto de componentes de sistema reusáveis com um componente de código específico. A terceira, devido à variedade de plataformas de hardware é importante ter uma arquitetura de software com decomposição flexível .

Este modelo impõe restrições ao programador que permite ao compilador NesC fazer uma análise completa do programa para melhorar o desempenho da aplicação e economiza recursos. Dentre as restrições estão à proibição do uso de alocação dinâmica de memória e ponteiros para funções. Esta análise inclui a identificação de conflitos de dados e a realização de melhorias no código.

As melhorias consistem em instanciação estática de componentes, expansão de sub-rotinas, eliminação de código morto (por exemplo, variável instanciada, mas nunca usada na aplicação será removida), eliminação de sub-expressões e propagação de constantes.

Em NesC um componente deve especificar as interfaces que ele provê e usa como mostra a Figura 3.3.

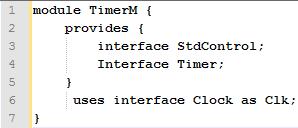


Figura .: Exemplo da descrição de um componente em NesC

Este código apresenta a especificação das interfaces do componente TimerM. As interfaces que ele provê e usa devem ser declaradas seguidas das palavras reservadas provides e uses como exibido nas linhas 2 e 6. Pode-se definir também um “apelido” para uma interface através do uso da palavra reservada as seguido do novo nome como foi feito na linha 6 com a interface Clock.

Interfaces são bidirecionais e definem um conjunto de comandos e eventos. Um exemplo de interface é mostrado na .

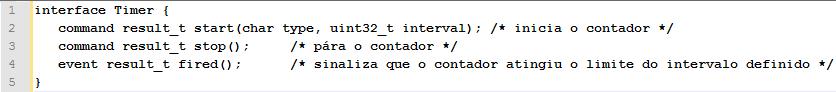


Figura .: Exemplo de descrição da interface de um componente em NesC

Neste código vemos a declaração da interface Timer que provê os comandos start e stop (usados para iniciar e desligar o temporizador) nas linhas 2 e 3, e o evento fired que é usado para sinalizar que o temporizador chegou ao valor final do intervalo.

Os componentes podem ser de dois tipos: módulos ou configurações. Módulos descrevem o funcionamento da aplicação através do provimento (definem o comportamento de comandos e eventos) e uso (invocam serviços providos por outros componentes) de uma ou mais interfaces.

Configurações são usadas para interconectar as interfaces entre componentes. Toda aplicação descrita em NesC possui uma configuração *top level* que define a ligação entre as interfaces dos componentes que a compõem. O código fonte mostrado na Figura 3.5 ilustra um exemplo de configuração *top level* do componente TimerC.

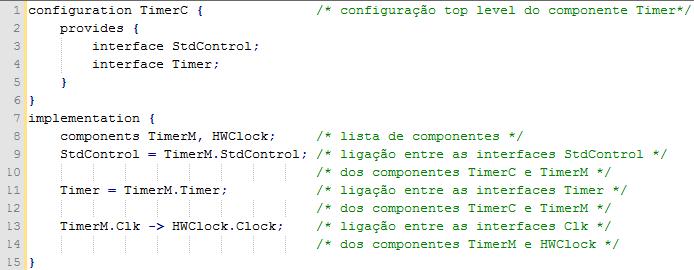


Figura .: Exemplo de código de configuração *top level* em NesC

A linha 8 mostra os componentes TimerMe HWClock que fazem parte da configuração *top level* do componente TimerC. Em seguida, nas linhas 9, 11 e 13 são realizadas as conexões entre eles.

O sinal de igual nas linhas 9 e 11 indicam que as interfaces StdControl e Timer providas pelo componente de configuração TimerCsãoimplementadas por um componente da aplicação, neste caso, TimerM.

Na linha 13, a “seta” usada para ligar as interfaces Clock dos componentes TimerM e HWClock indica que esta interface é implementada por um componente externo da aplicação. A “seta” aponta na direção do componente que usa a interface para o daquele que provê a interface. Neste exemplo, o componente TimerM usa a interface Clock do componente HWClock.

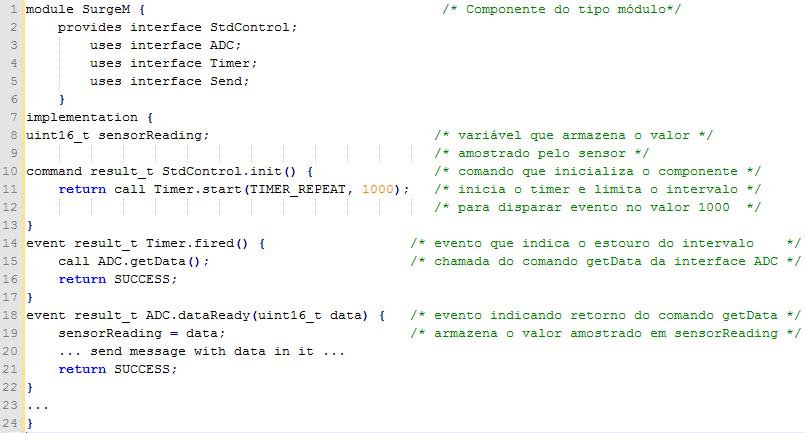


Figura .: Exemplo de implementação de um componente do tipo módulo

O corpo de um módulo é escrito em linguagem C com algumas extensões como pode ser visto na . Um comando ou evento *c* em uma interface *f* é invocado através de *f.c*.A chamada a um comando é feita utilizando a palavra reservada call (linhas 11 e 15). A chamada a eventos é feita utilizando a palavra reservada signal.

A definição do comando ou do evento f.c é feita usando as palavras reservadas command e event. A ilustra um exemplo de um componente do tipo módulo escrito em NesC que implementa o comando StdControl.init (linha 9) e é sinalizado pelos eventos Timer.fired e ADC.dataReady (linhas 12 e 16).

### Simulador TOSSIM

TOSSIM é um simulador baseado em eventos discretos projetado para simulação de redes de sensores baseadas no TinyOS. Seu objetivo é permitir que o programador possa testar e verificar o código da aplicação para diferentes arquiteturas de hardware, algoritmos, pilhas de protocolos, processamentos distribuídos e ambientes de implantação.

O TOSSIM é capaz de gerenciar milhares de nós sensores executando aplicações completas. Seu funcionamento é baseado em uma *engine* simples que mapeia os eventos do TinyOS em um conjunto de eventos discretos que simulam inúmeras interações entre a aplicação e o hardware com nível de fidelidade programável.

A modelagem da comunicação de rádio, por exemplo, pode ser configurada pelo desenvolvedor de acordo com a complexidade e a precisão necessária para a aplicação. O link de comunicação entre os nós é representado internamente por um grafo direcionado de probabilidades de *bit error*.

Cada aresta *(u, v)* no grafo representa a taxa de erro quando um mote *u* envia dado para *v*, e esta é diferente da aresta *(v, u)* o que permite especificar links assimétricos. Se uma condição de *bit error* ocorrer o simulador inverte o valor do bit.

TOSSIM possui dois modos especiais de modelagem de rádio. O primeiro, chamado de simples, é usado para testar protocolos em canais sem erros usando comunicação ponto-a-ponto e o outro, chamado estático, usado para teste de protocolos em comunicação *multihop* .

O simulador possui alguns serviços de comunicação que permitem monitorar e atuar sobre a *engine* através de interfaces seriais, de rede e por interrupções de software. Alguns *plug-ins* são disponibilizados caso o projetista esteja interessado em construir aplicações, executando no PC, que se comunica com o simulador.

Exemplos de eventos enviados pelo simulador são mensagens de debug e exemplos de comandos enviados pelo programador é a mudança da probabilidade de *bit error* na comunicação entre dois nós, desligamento e ligamento de um mote, entre outros.

Para garantir a fidelidade da simulação da rede, a emulação da comunicação de rádio foi feita utilizando uma análise no nível de bit da pilha de protocolos do TinyOS . Esta análise permitiu que os projetistas modelassem o comportamento do hardware com alta fidelidade. Entre os possíveis erros que podem ser simulados estão: problema do nó escondido, pacotes corrompidos, aumentar o atraso quando os nós tentam repetidamente escutar um sinal da rede através de CSMA, detecção de *single bit error*, entre outros .

## Considerações Finais

Neste capítulo foi apresentado o funcionamento, a arquitetura e o ambiente de programação e depuração do sistema operacional TinyOS. Mostrou-se como um programa pode ser construído e estruturado em sua arquitetura baseada em componentes de acordo com a linguagem de programação NesC.

Discutiu-se também o funcionamento do simulador TOSSIM e como ele pode ser usado para depuração abordando suas diferentes interfaces de comunicação. Estas interfaces podem ser do tipo serial, de rede ou via interrupções de software com escrita de mensagens na linha de comando.

­­

# Máquina Virtual TinyReef

Este capítulo apresenta a máquina virtual TinyReef , projetada especificamente para plataformas de redes de sensores sem fio e que executa sobre o sistema operacional TinyOS.



## Objetivos

O principal objetivo da TinyReef é a construção de uma camada de software que dá suporte ao desenvolvimento de aplicações para RSSF utilizando linguagens de programação com alto nível de abstração. Além disso, ela permite a atualização do software aplicativo executando sobre a máquina virtual em tempo de execução sem afetar o funcionamento da aplicação. Essa característica visa suprir à necessidade de adaptação nós sensores em decorrência da mudança das condições normais de operação do sistema.

O projeto da TinyReef considera as restrições de memória e processamento inerentes ao hardware utilizado nos nós sensores. A solução encontrada para suprir tais problemas foi a análise cuidadosa da arquitetura. Essa arquitetura será o objeto de estudo da Seção 4.2.

## Arquitetura da TinyReef

TinyReef é uma máquina virtual cuja arquitetura é baseada em registradores de 32 *bits* e executa sobre o sistema operacional TinyOS. Sua principal diferença com relação a outras máquinas virtuais projetadas para redes de sensores como Maté e Davim é o fato de sua arquitetura ser baseada em registradores ao invés de usar pilha.

Segundo Marques , o uso de uma abordagem baseada em registradores diminui o tamanho do programa, pois um conjunto de instruções para realizar uma operação na abordagem de pilha pode ser substituído por uma única instrução na abordagem utilizando registradores.



Figura .: Exemplo comparativo entre instruções baseadas em registradores e instruções baseadas em pilha

Um exemplo dessa análise é mostrado na Figura 4.1 onde se observa a diferença no número de instruções necessárias para realizar uma operação de soma. Na abordagem com registradores, instrução da esquerda, seria necessária apenas uma instrução, enquanto que na outra técnica seriam preciso 4. Isso acarreta uma diminuição na utilização de memória e também uma melhora no desempenho do sistema com a redução do número de *bytecodes* que serão interpretados.

### Módulos

A arquitetura da TinyReef é composta por 5 componentes: *Program State*, *Loader*, *Instruction Set*, *Event Handler* e *Interpreter*. A mostra esta arquitetura através de uma representação de blocos lógicos.



Figura .: Visão lógica da arquitetura da máquina virtual TinyReef

O componente *Program State* gerencia todas as estruturas necessárias para o funcionamento de um processador comum como a memória, o contador de programa (*Program Counter* - PC) e 16 registradores de 32 bits para propósito geral.

A memória é dividida pelos segmentos de dados e de instruções, ambos endereçáveis por 32 bits. O PC é responsável por guardar o endereço da próxima instrução que será executada na sequência lógica do programa.

O componente *Loader* é responsável pelo carregamento dos programas na máquina virtual. Ele gerencia quando um novo programa será carregado bem como se encarrega da comunicação com a estação base para pedir atualizações do software em execução.

O *Interpreter Set* contém a definição de todas as instruções do repertório disponível para descrição de programas. Ele é responsável por identificar, a partir do opcode (código da operação), qual delas deverá ser executada.

O componente *Event Handler* é responsável pelo gerenciamento das tarefas assíncronas da máquina virtual. Ele é chamado por instruções específicas e tem a responsabilidade de armazenar o endereço da rotina que será executada quando o evento registrado for sinalizado.

Finalmente, o componente *Interpreter* é o responsável pela lógica de funcionamento da máquina virtual. Assim que um programa é carregado, ele começará a executar. Seu fluxo de execução começa com a leitura da instrução cujo endereço está armazenado no PC.

Depois de obter o *bytecode* na memória, a instrução é separada em *opcode* e operandos e o valor do PC é incrementado. A partir desse instante ele usa o componente *Interpreter Set* para executar a instrução de acordo com *opcode* e os operandos do *bytecode*.

O *Interpreter Set* então aguarda o retorno da execução da instrução, ou seja, quando ela terminar de ser executado o controle volta para o *Interpreter* que , por sua vez, refaz o ciclo novamente até encontrar o opcode de uma instrução *halt*.

### Instruções de Máquina

Cada instrução possui 32 *bits*, onde o primeiro byte representa o *opcode*, ou seja, o código da operação que será executada e os demais bytes são reservados para os operandos.



Figura .: Exemplos de formato de instruções da TinyReef

A mostra 3 exemplos de instruções. A primeira é uma instrução de manipulação de dados da memória, a segunda é uma operação aritmética de soma e a terceira é um desvio incondicional (JUMP).

Observe que dependendo do tipo da instrução os operandos podem variar. Nesse caso, as 3 instruções possuem formatos diferentes. O JUMP só possui um operando que é o endereço de destino. A instrução ADD possui 3 operandos que são o endereço dos registradores que serão somados e o registrador de destino do resultado.

No caso da operação MOV são necessários 5 parâmetros. O primeiro identifica opcode, o segundo o endereço base de memória, o terceiro o registrador fonte e os últimos representam o offset. Essa operação guarda o valor de um registrador na memória.

### Implementação da TinyReef

A TinyReef foi implementada sobre o sistema operacional TinyOS utilizando como plataforma de hardware o nó sensor MicaZ. MicaZ foi escolhida por ser uma plataforma amplamente difundida e suportada pelo TinyOS.

A máquina virtual corresponde a um componente TinyOS que opera acima dos componentes padrões de abstração do hardware como rádio, contadores, sensores, entre outros. Seu funcionamento básico já foi validado através de exemplos simples escritos no formato de bytecodes. Atualmente, ela se encontra em desenvolvimento onde os projetistas estão buscando melhorar o gerenciamento de memória e dá suporte a propagação de programas.

### Estrutura dos Programas

A estrutura dos programas é formada por dois segmentos: um segmento de dados e outro de código. O primeiro representa as variáveis que serão usadas no programa e o segundo as instruções cuja sequência lógica exprimem a funcionalidade do sistema.



Figura .: Exemplo de programa TinyReef que coleta dados de um sensor e os envia via rádio .

A mostra um exemplo do formato de um programa TinyReef. Os segmentos são separados pelas palavras reservadas DATASEG e CODESEG. A primeira linha do segmento de dados indica que deverá ser armazenada uma variável estática do tipo DWORD que corresponde a 32 bits.

O segmento de código é formado por um conjunto de instruções onde os operandos podem ser registradores representados através do símbolo $, ou endereços de memória, ou constantes. Além disso, no segmento de código são definidos rótulos como tfired, rdfired e sdfired que serão usados para armazenar o endereço base de procedimentos que tratam, respectivamente, dos eventos assíncronos relacionados à temporizadores, recebimento e confirmação de envio de mensagens.

Esses eventos são registrados no início do segmento de código que corresponde ao processo de inicialização do programa até que se encontre a instrução halt. Nesse momento, o programa ficará parado esperando a ocorrência de um evento.

## Considerações Finais

Este capítulo apresentou a máquina virtual TinyReef cujo principal objetivo é o desenvolvimento de uma camada de software que permita o desenvolvimento de aplicações em alto nível de abstração.

# Extensão à TinyReef

A extensão à máquina virtual TinyReef de que trata este capítulo é o suporte ao serviço de adaptação em tempo de execução do software de aplicação. Como mencionado na Seção 2.5, esta é uma das principais características que um middleware para RSSF deve suportar.

Neste capítulo serão abordados os objetivos pretendidos com esta extensão, as modificações realizadas na TinyReef para suportar tal funcionalidade, o fluxo de desenvolvimento e verificação que inclui as ferramentas e métodos utilizados e, por fim, um exemplo do funcionamento deste serviço.



## Objetivos

Esta extensão visa à construção, validação e adição à máquina virtual TinyReef de um componente capaz de prover a infra-estrutura de comunicação necessária ao serviço de atualização da aplicação em tempo de execução.

Este serviço consiste na abstração da requisição de atualizações de programas feitas pelos nós sensores à estação-base (para mais detalhes sobre os componentes de uma RSSF ver Subseção 2.4). A máquina virtual se encarrega de identificar certas condições operacionais ou mudança no ambiente que necessitem de uma atualização.

Essa infra-estrutura de comunicação é formada por um protocolo que define o formato dos pacotes de requisição e recebimento dos dados, as ações executadas pelo nó sensor e pela estação-base e o modo como as informações são trafegadas na rede.

Um cenário onde este serviço poderia ser usado é mostrado na Figura 5.1. Supondo que os nós azuis fazem parte de uma rede de sensores que monitora a umidade de duas regiões de uma grande floresta que tem o risco de pegar fogo se a umidade estiver abaixo de um limite.

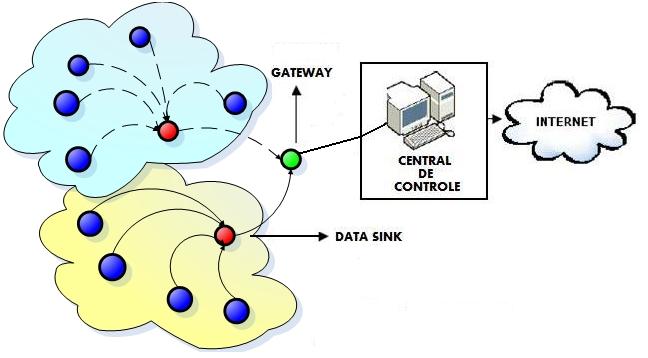


Figura .: Exemplo de uso do serviço de adaptação

Supondo que os nós vermelhos são do tipo *data sink* e têm a responsabilidade de monitorar os nós azuis e fornecer uma atualização do software aplicativo para que eles emitam sinais de localização para facilitar a descoberta de um possível foco de incêndio.

Além disso, os nós data sink estão ligados a um nó gateway (nó verde). Por sua vez, ele está ligado à central de controle da guarda florestal e emite sinais de aviso caso uma atualização de baixa umidade seja requerida pelos sensores vermelhos.

A máquina virtual executando nos nós azuis pode enviar a requisição para os nós data sink pedindo a atualização devido à condição de umidade baixa. Caso haja uma mudança no clima e a umidade volte ao normal, a máquina virtual pedirá o programa antigo, por exemplo, por que ele consome menos energia.

## Modificações na Máquina Virtual TinyReef

As principais modificações na máquina virtual TinyReef foram realizadas nos módulos *Loader* responsável por carregar os programas e *ReceiveByteCode*, encapsulado no componente *Interpreter Set*,que deve ser capaz de identificar a chegada de pacotes de atualização e encaminhá-los para o componente criado para gerenciar as atualizações chamado de *UpdateHandler*. Para mais detalhes sobre os módulos da TinyReef ver Subseção 4.2.1.

Além de requisitar e montar o *buffer* com o programa atualizado, o componente *UpdateHandler* deve gerenciar o *timeout* de pacotes com o intuito de recuperar pacotes perdidos. Quando houver *timeout*, ele requisitará novamente o pacote perdido. Mais detalhes sobre o protocolo serão dados na Seção 5.3.

As alterações no componente *Loader* foram necessárias para tornar a função de carregamento de programas parametrizável. Quando uma atualização terminar de ser recebida, o *UpdateHandler* invoca o comando provido pelo *Loader* para que a nova aplicação possa ser carregada. Esse comportamento será explicado na Subseção 5.3.2.

Além dos componentes *Loader* e *ReceiveByteCode*, foram modificados também os arquivos de configuração responsáveis pela conexão entre os módulos da TinyReef e adicionado um comando no componente *EventHandler* para desabilitar eventos. A é uma representação gráfica da interconexão entre os componentes da TinyReef depois da introdução do componente UpdateHandler.

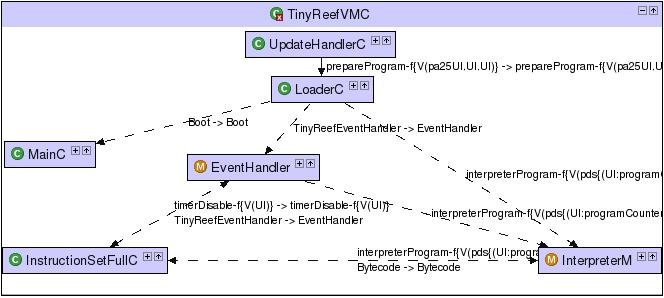


Figura 5.2: Representação gráfica da interconexão dos componentes da TinyReef

Também foram realizadas alterações nos arquivos cabeçalhos TinyReef.h e TinyReefMessage.h. O primeiro contém as estruturas de dados e constantes utilizados na máquina virtual TinyReef e o segundo contém as estruturas de dados que definem os tipos de pacotes.

Foi adicionada uma nova estrutura para encapsular os pacotes de atualização recebidos e constantes para os tipos de pacotes no arquivo TinyReef.h. No arquivo TinyReefMessage.h encontra-se a declaração das estruturas dos pacotes de requisição e de recebimento das atualizações. Mais detalhes sobre o formato desses pacotes são dados na Subseção 5.3.1.

No componente *EventHandler*, a função do comando adicionado é desabilitar os eventos de temporizadores registradores na tabela de eventos da TinyReef. Este comando é invocado pelo componente *Loader* antes do carregamento do novo programa.

## Definição do Protocolo de Comunicação

A definição do protocolo de requisição e recebimento de pacotes de atualização consistiu na especificação do formato das mensagens e das regras de troca de mensagens entre as camadas.

O protocolo definido para este serviço se baseia em uma comunicação ponto-a-ponto entre o nó sensor e a estação-base, por isso nenhum protocolo de roteamento foi utilizado.

Nas camadas de enlace e física utilizou-se a pilha de protocolos do TinyOS que gerencia o acesso ao meio e a transmissão simples ponto-a-ponto juntamente com o protocolo de abstração de pacotes chamado de *Active Message* (AM). (von Eicken e Goldstein 1992) (Levis, Lee, et al. 2003).

AM é um protocolo de enlace não confiável, pois não possui mecanismo de recuperação de pacotes perdidos. Ele provê interfaces de comunicação unificadas tanto para enlace sem fio via rádio quanto para enlace cabeado via porta serial.

Sua comunicação é síncrona baseada em *acknowlegdes*. Ele fornece 256 tipos diferentes de mensagens AM que podem ser associados a diferentes tipos de protocolos e também possui um mecanismo de detecção de pacotes corrompidos via CRC que descarta aqueles que possuírem um ou mais *bits* errados.

Se não há erro nos dados do pacote o receptor verifica o endereço de destino. Se este endereço coincide com o dele, ele encaminha o pacote para o componente que gerência mensagens do tipo especificado no campo tipo do pacote AM.

|  |  |  |
| --- | --- | --- |
| Cabeçalho 802.15.4 | Carga útil | 802.15.4 CRC |

Figura 5.3: Formato padrão do *frame* 802.15.4 do TinyOS

O formato padrão do *frame* do TinyOS é mostrado na

Figura **5**.**3**. Ele segue o padrão IEEE 802.15.4 para enlaces sem fio e é formado por cabeçalho onde fica as informações de controle, espaço de carga útil (*payload*), e o campo de CRC usado para detecção de erros.

Dentro da carga útil do frame são reservados 28 *bytes* para dados. Além disso, os pacotes AM possuem os campos Endereço de destino(2 *byte*s), Tipo AM (1 *byte*s), grupo AM (1 *byte*), número de *bytes* válidos (1 *byte*), entre outros.

O endereço de *broadcast* é 0xFFFF e o endereço reservado para comunicação serial é 0x007E. O campo grupo do pacote AM pode ser usado para criar redes virtuais onde um conjunto de nós possui o mesmo endereço.

Nas Subseções 5.3.1 e 5.3.2 serão abordados os detalhes com relação ao formato, tipos e campos das mensagens bem como o comportamento dos nós sensores e da estação-base durante o processo de atualização.

### Formato, Tipos e Campos das Mensagens

O tamanho dos pacotes de atualização ficou limitado ao tamanho do campo *DATA* dos pacotes AM (28 *bytes*). Foram definidos dois tipos de pacotes: um para envio de requisição e outro para recepção de atualização. O primeiro é usado tanto na requisição de atualização quanto na recuperação de pacotes perdidos. O segundo é usado no recebimento de mensagens de atualização. A Figura 5.4.a e a Figura 5.4.b mostram o formato desses pacotes, respectivamente.

pacote_request.jpg pacote_UPDATE_RECEIVE.jpg

(a)Requisição e recuperação (b) Recebimento

Figura .: Formato dos pacotes do protocolo de atualização

O pacote de requisição ocupa 6 *bytes* da área de dados do pacote AM e é formado pelos campos *VERSION* (4 *bits*), *TYPE* (4 *bits*) e *DATA* (5 *bytes*). O campo *TYPE* indica o tipo da requisição que poderá ser: *PROGRAM\_LOSTPACKET* ou *PROGRAM\_UPDATE*. Os demais valores possíveis para *TYPE* são reservados para outros tipos de requisições caso seja necessário no futuro.

A semântica dos campos *VERSION* e *DATA* variam de acordo com o campo *TYPE*. Se ele for do tipo *PROGRAM\_UPDATE*, os campos *VERSION* e *DATA* representam, respectivamente, a versão atual executando na máquina virtual e *DATA* não terá nenhum significado neste caso.

Quando *TYPE* for do tipo *PROGRAM\_LOSTPACKET*,o valor do campo *VERSION* indicará o número da versão para a qual se deseja recuperar pacotes e *DATA* irá conter o número dos pacotes perdidos. Cada número de pacote usará 4 *bits* do campo *DATA*.

O pacote de recepção é formado pelos campos INFO (2 *bytes*), LENGTH (1 *byte*) e DATA (25 *bytes*). INFO encapsula as informações de versão, número do pacote e total de pacotes. LENGTH indica a quantidade de bits válidos no pacote e DATA contém os dados. O campo INFO é dividido da seguinte forma: 4 *bits* para a versão, 6 *bits* para o número do pacote e 6 *bits* para o número total de pacotes.

Como a TinyReef reserva apenas 256 *bytes* para código de aplicação, seriam necessários, no máximo, 11 pacotes para o envio de um programa completo, pois cada um pode conter até 25 *bytes* de dados totalizando 275 *bytes*. Desse modo, os campos que representam o número de pacotes poderiam ter apenas 4 *bits* que dariam para representar 16 pacotes e uma carga total de dados de 400 *bytes*.

Nessa configuração, mantendo-se o tamanho para o campo versão, sobrariam 4 *bits* que poderiam ser utilizados para representar dados. No entanto, isso não diminuiria o número máximo de pacotes necessário para encher a memória de instruções, pois 204 x 10 pacotes = 2040 *bits* é menor que os 2048 *bits* (256 x 8) da memória.

Uma solução seria diminuir o número de versões para 4, representados por apenas 2 *bits*, que somados aos 4 *bits* comentados anteriormente totalizariam 206 *bits*/pacote. Isto seria interessante, pois diminuiria o número máximo de pacotes enviados para 10, uma vez que 206 x 10 = 2060 *bits* ocasionando uma diminuição no consumo de energia.

A configuração com 5 *bits* para representação do número dos pacotes não seria viável, pois se diminuíssemos o número de versões para 2 sobrariam apenas 4 *bits* ( 2 da versão mais 2 dos campos *packet\_number* e *total\_packet*) totalizando 204 *bits* no espaço para dados, o que não diminuiria o número de pacotes máximo como vimos há pouco.

Apesar da solução utilizando 4 *bits* para representar pacotes ser a ideal por ser compatível com a configuração atual de memória da máquina virtual e também por diminuir o número de pacotes, ela se mostra muito restrita quanto à possibilidade de mudança da configuração da TinyReef.

Como ela representa apenas 16 pacotes daria para se ter no máximo um programa de 16 x 206 *bits* = 412 *bytes* e com um número máximo de versões igual a 4. Desse modo optou-se pela configuração com 6 *bits* que permite representar até 64 pacotes suportando programas de até 64 x 200 bits = 1,6 Kbytes e com 4 *bits* para representar até 16 versões diferentes de programas.

### Comportamento dos Nós

Nesta subseção será apresentado o comportamento das partes envolvidas no processo de atualização que são os nós sensores e a estação-base. O comportamento do nó sensor é representado através do diagrama de fluxo apresentado na Figura **5**.**5**.

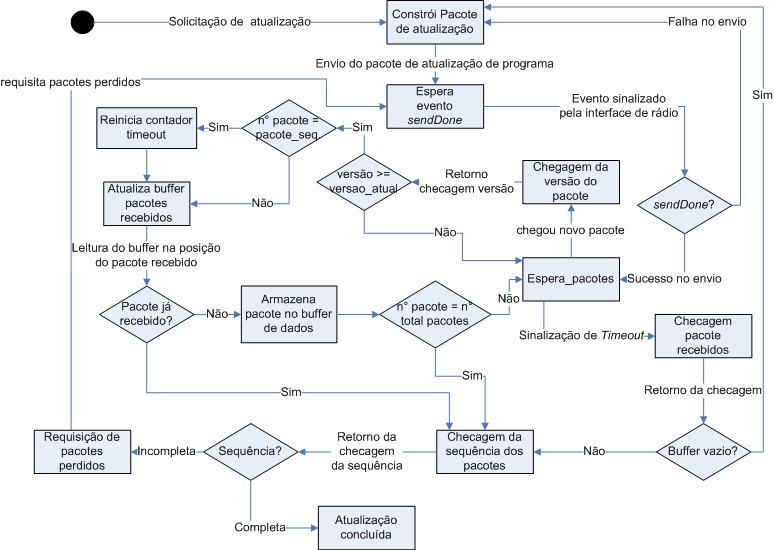


Figura 5.5: Fluxo de tarefas do nó sensor durante a atualização do software aplicativo

O ponto de partida desse processo é a ocorrência de algum evento pré-estabelecido na lógica da máquina virtual que desencadeia a necessidade de atualização do software.

Esse evento pode ser devido a uma grande diversidade de possibilidades como mudanças no ambiente de operação ou devido a falhas entre outras. Diante dessa ocorrência o componente *Loader*, responsável pelo carregamento de programas da TinyReef, envia um comando para o componente *UpdateHandler* solicitando uma atualização.

O *UpdateHandler* então verifica a versão da aplicação executando na máquina virtual e constrói o pacote de requisição de atualização passando como parâmetro a versão atual e o tipo *PROGRAM\_UPDATE* e envia o pacote de atualização através da sua interface de rádio.

Depois de recebida a confirmação que o pacote foi enviado, deve-se inicializar o contador de *timeout* para aquela requisição. Após ligar o contador, o componente deve esperar pelo evento indicando que houve *timeout* ou que chegou algum pacote de atualização de programa.

Se o evento sinalizado for a chegada de um novo pacote de atualização, o *UpdateHandler* deve verificar a versão do pacote recebido determinando se ela é mais recente do que a versão atual executando na máquina virtual e a guarda.

Se a verificação da versão for positiva, ele reiniciará o contador de *timeout*, se o pacote estiver na sequência correta. Caso contrário, configura o *buffer* de pacotes recebidos e aloca um *buffer* baseado no número total de pacotes, caso já não exista um buffer.

O *UpdateHandler* armazenará o pacote recebido no buffer de dados se o pacote já não existia no *buffer* de pacotes recebidos. À medida que novos pacotes vão chegando estes vão sendo armazenados no *buffer* de dados se pertencerem à mesma versão. Se o campo número do pacote for igual ao campo total de pacotes será verificada a sequência dos pacotes.

Se ela estiver correta invoca-se um comando do componente *Loader* para indicar que uma nova versão da aplicação está disponível finalizando o ciclo de atualização.

Se acontecer um *timeout* ou a verificação da sequência estiver incorreta o *UpdateHandler* deve verificar o estado da última requisição feita analisando se o buffer de pacotes recebidos está vazio.

Se a verificação for positiva ele envia uma requisição do tipo *PROGRAM\_UPDATE* novamente. Caso contrário irá fazer uma requisição do tipo *PROGRAM\_LOSTPACKET* passando a versão dos pacotes armazenados no buffer e o número dos pacotes que ainda faltam.

O tempo de *timeout* foi estimado empiricamente através de simulações utilizando o simulador TOSSIM. O ambiente de teste considerou a concorrência no uso do canal de comunicação através do envio de pacotes de aplicação a cada 250 ms e as requisições de atualização feitas em paralelo e no pior caso o tempo foi de 900 ms.

O comportamento da estação base fica transparente para o nó gateway. O papel do gateway é somente receber as requisições e encaminhá-las para o software host executando no PC, bem como enviar os pacotes de atualização para os nós sensores remotos. Mais detalhes sobre esse comportamento são dados na Subseção 5.4.1.

## Fluxo de Desenvolvimento

Esta Seção aborda o fluxo de desenvolvimento da infra-estrutura de comunicação para dar suporte ao serviço de atualização dinâmica da máquina virtual TinyReef. O fluxo foi dividido em duas fases: construção do ambiente de verificação e o desenvolvimento e validação do protocolo. Estes tópicos são discutidos nas Subseções 5.4.1, 5.4.2, respectivamente.

### Ambiente de Verificação

A construção de um ambiente de verificação é essencial para a garantia da funcionalidade do sistema bem como a qualidade da aplicação. Em redes de sensores a verificação é ainda mais crítica, pois talvez não seja possível ter acesso ao sistema para repará-lo após a sua implantação no ambiente.

No contexto de RSSF, a modelagem de protocolos de comunicação deve ser realizada não só com o intuito de verificar a funcionalidade mais também de aperfeiçoá-lo o máximo possível através de simulação matemática ou empírica .

O ambiente de verificação projetado neste trabalho englobou o desenvolvimento do aplicativo chamado de *ProgramSender* cujo objetivo é auxiliar na interação entre o nó *gateway* e o PC, e também a realização de simulações utilizando o simulador TOSSIM. Para a execução do teste em hardware, a comunicação entre o nó *gateway* e o PC foi realizada através da interface serial da placa de desenvolvimento e a porta USB de um notebook como mostra a Figura 5.6.



Figura .: Plataforma de teste de hardware da TinyReef

Na interface do *ProgramSender* há um campo onde deve ser especificada a porta por onde se dará a comunicação serial, pois dependendo do sistema operacional esta porta poderá mudar.

O aplicativo foi desenvolvido na linguagem Java devido à existência de APIs bem testadas e estáveis que facilitam a integração de aplicações customizadas com o ambiente TinyOS. Ele executa no sistema operacional Ubuntu versão 8.04 . Sua interface permite enviar, editar e carregar programas de atualização escritos no formato da linguagem de máquina da máquina virtual como se pode observar na Figura 5.7.

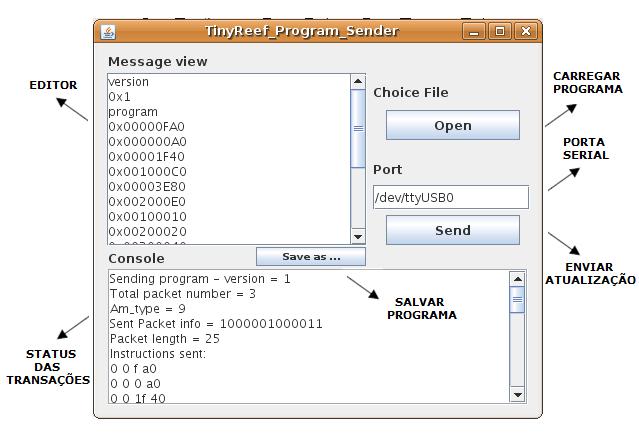


Figura .: Interface do programa *programaSender*

Além das características já mencionadas, o *ProgramSender* possui também uma área onde as mensagens enviadas e recebidas são impressas para facilitar o acompanhamento das transações. A aplicação do nó *gateway* é fornecida junto com o TinyOS. Sua tarefa é receber os dados enviados por nós sensores por um enlace de rádio e encaminhar essas mensagens através da porta serial para o PC. Da mesma forma, ela repassa os pacotes recebidos via serial vindo do PC para os nós remotos através do enlace de rádio no modo *broadcast*.

Além dos testes realizados com o auxílio do *ProgramSender* também foram feitas depurações utilizando o simulador TOSSIM. Com ele pode-se corrigir alguns problemas funcionais do protocolo como: ordenação das instruções, inicialização de variáveis, requisição de pacotes perdidos e assim por diante, utilizando sua interface de depuração para imprimir os valores de variáveis na tela de console do PC.

Com o ambiente de simulação funcionando, foi possível fazer os testes para calcular o tempo médio entre o envio e o recebimento dos pacotes. Esses tempos serviram de parâmetro para estimar o tempo de *timeout* mencionado na Seção 5.3.2.

Como o TOSSIM possui a limitação de só carregar um programa para todos os nós instanciados na simulação, houve a necessidade de desenvolver um ambiente de teste no TinyOS. Neste ambiente, só havia os componentes da interface de rádio, temporizadores, o *UpdateHandler* e um componente simulando o comportamento do *Loader*.

O ambiente de teste emula o comportamento tanto do nó estação-base quanto do nó sensor remoto. Desse modo pode-se testar a funcionalidade do protocolo tanto sem interferências de ruído quanto considerando um enlace ruidoso. Mais detalhes sobre o simulador TOSSIM foram dados na Subseção 3.3.2.

### Desenvolvimento e Validação do Protocolo

O desenvolvimento do protocolo foi realizado em 3 etapas: construção do comportamento do nó sensor, construção do comportamento do nó estação-base e em seguida houve a integração na TinyReef. A validação ocorreu de modo incremental ao longo dessas fases.

O comportamento do nó sensor, como se viu na Subseção 5.3.2, engloba a comunicação entre os componentes da máquina virtual quando ocorre um evento indicando a necessidade de atualização da aplicação. Durante sua construção levou-se em consideração as características que RSSF possuem com relação a limitações de recurso como: tamanho da memória, capacidade de processamento e canais de comunicação. Os componentes envolvidos nesse processo são *UpdateHandler*, *Loader*, *ReceiveByteCode*. Desses módulos somente o *UpdateHandler* é um componente novo que foi criado para atender a nova funcionalidade.

A assinatura das interfaces do componente *UpdateHandler* é mostrada naFigura 5.8. Nela pode-se observar as interfaces providas e usadas pelo componente, bem como a assinatura dos comandos prepareProgram, programAssemble e updateProgram (linhas 12, 16, 15, respectivamente).

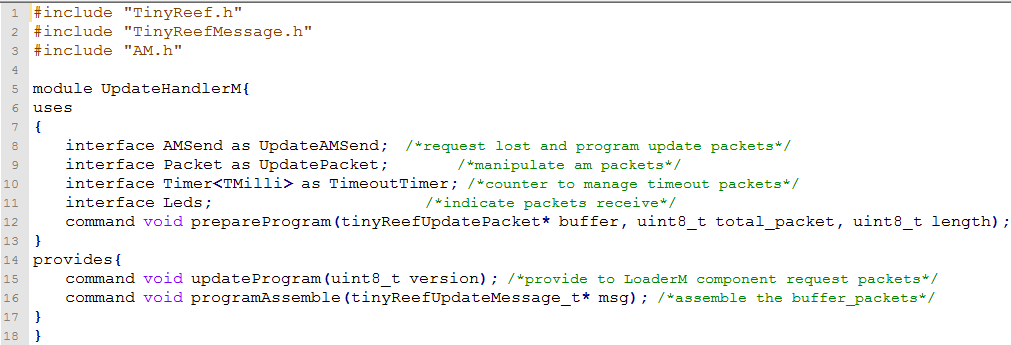


Figura .: Assinatura das interfaces do componente *UpdateHandler*

O *UpdateHandler* usa o comando prepareProgram (linha 12), provido pelo componente *Loader*, para repassar o *buffer* com o novo programa, e a interface *Timer* (linha 10),provida pelo TinyOS, para controlar o *timeout* dos pacotes requisitados. Por sua vez, o *UpdateHandler* usa os comandos da interface AMSend e Packet, ambos providos também pelo TinyOS (linhas 8 e 9), para enviar os pacotes de requisição. A estrutura do pacote de requisição é mostrada na Figura 5.9.

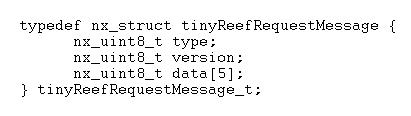


Figura .: Estrutura do pacote de requisição de atualizações

O *UpdateHandler* provê os comandos updateProgram(linha 15), usado pelo componente *Loader* para sinalizar a necessidade de atualização, e o comando programAssemble (linha 16), chamado pelo componente *ReceiveByteCode* para sinalizar a chegada de pacotes de atualização.

O comando updateProgram recebe como parâmetro a versão atual do programa executando na máquina virtual. A mostra o código deste comando. A sua função é inicializar as variáveis globais do componente *UpdateHandler* e requisitar a atualização (linha 196).

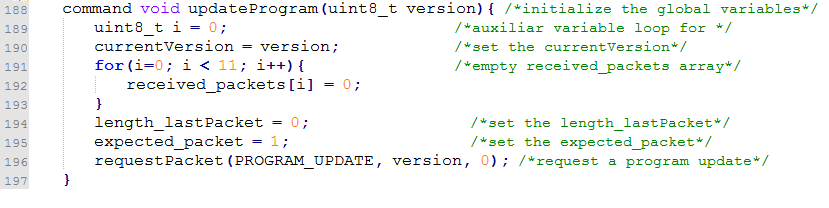


Figura 5.10: Comando *updateProgram* provido pelo componente *UpdateHandler*

O comando programAssemble recebe como parâmetro um ponteiro para a estrutura do tipo tinyReefUpdateMessage que representa o pacote de atualização recebido. Essa estrutura é mostrada na Figura 5.11. As estruturas exibidas nas Figuras 5.9 e 5.11 são formadas, respectivamente, pelos campos dos pacotes de requisição e recebimento de atualização descritos na Subseção 5.3.1.

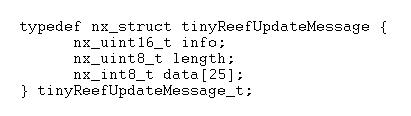


Figura .: Estrutura do pacote de recebimento de atualização

Um trecho do código do comando programAssemble é mostrado na Figura 5.12. Sua função é montar os pacotes sendo recebidos em um *buffer* cujo tamanho será igual ao número total de pacotes multiplicado por 25 *bytes* que é o tamanho total de um pacote (linha 54). O total de pacotes é informado dentro do campo *info* do pacote de atualização.



Figura .: Trecho de código do comando *programAssemble*

Inicialmente, os pacotes são verificados quanto à versão. Se a versão do pacote for maior ou igual à versão atual armazenada, esta será atualizada (linha 51). Caso contrário o pacote será descartado. Após aceitar o pacote o algoritmo verifica se o *buffer* já está alocado (linha 53).

Depois de verificada a validade do pacote quanto à versão, é verificado se o pacote já foi recebido (linha 55). Caso ele ainda não tenha sido recebido, o tempo de *timeout* do pacote esperado na sequência normal é atualizado (linha 56), o pacote é armazenado no *buffer* de dados (buffer\_packets) (linha 58) e o *buffer* de pacotes recebidos (received\_packets) é atualizado (linha 60).

Então o valor do pacote esperado será atualizado para a posição do próximo pacote na sequência ainda não recebido (linhas 62 a 67). Depois de atualizar o *array* de pacotes recebidos e armazenar o pacote, o algoritmo verificará a sequência de pacotes invocando a função checkPacketSequence(linha 70) se este for o último pacote (linha 68). A checagem da sequência é feita no último pacote para evitar que várias mensagens sejam enviadas a cada pacote perdido. Essa abordagem visa evitar problemas de *flooding* onde vários pacotes são enviados ao mesmo tempo e acabam congestionando a rede.

Quando o pacote já foi recebido (linha 76), será feita a checagem da sequência para verificar se existem pacotes perdidos. Se a sequência não estiver completa serão requisitados os pacotes perdidos. Caso contrário será finalizada o processo de atualização com o desligamento do contador de *timeout* (linha 80) e a chamada do comando prepareProgram (linha 81).

A mostra o código do evento de sinalização de *timeout* de pacotes. Se o *buffer* de pacotes estiver vazio serão requisitados todos os pacotes novamente (linha 135). Caso ocorra um *timeout* do pacote esperado, a sequência será checada para verificar se existem outros pacotes perdidos, de posição anterior a dele (linha 137). Como visto nas linhas 62 a 67 da Figura 5.12, o pacote esperado será atualizado sempre para o próximo pacote na sequência de recebimento de pacotes com relação ao último recebido. Desse modo, quando houver *timeout*, serão requisitados todos os pacotes perdidos de uma só vez.

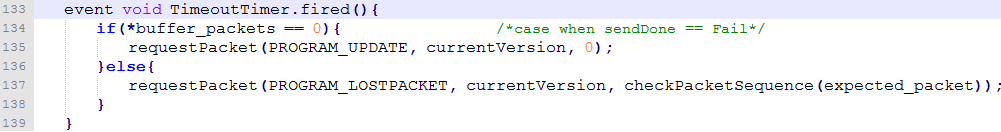


Figura 5.13: Código do evento *TimeoutTimer*

O código da função checkPacketSequence é mostrado na Figura 5.14. Esta função recebe como parâmetro um *byte* que representa o número do pacote em que ocorreu o *timeout* (linha 137 da ), ou o número total de pacotes quando for recebido o último pacote da sequência, ou pacotes repetidos (linhas 70 e 77, respectivamente, da Figura 5.12).

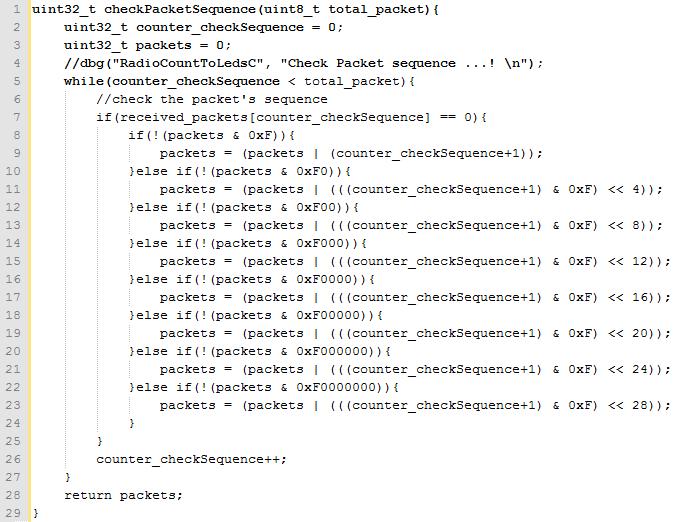


Figura .: Código da função *checkPacketSequence*

A função então varre o *array* de booleanos received\_packets que indica quais pacotes foram recebidos até o número do pacote que deu *timeout*. Caso existam pacotes perdidos além daquele passado como parâmetro ela vai adicionando o número do pacote a variável de retorno packets. Cada número de pacote ocupa 4 *bits* dessa variável e como ela é de 32 *bits* cabem até 8 números de pacotes diferentes. Essa limitação foi implantada com o intuito de diminuir a utilização de memória RAM.

No momento em que a checagem da sequência dos pacotes reportar um inteiro igual a zero, indicando que a sequência está completa (linha 71 da Figura 5.12), a função programAssemble invocará o comando prepareProgram (linha 73 da Figura 5.12) passando como parâmetro o *buffer* com os pacotes, o total de pacotes e o tamanho do último pacote.

O comando prepareProgram se encontra no componente *Loader*. Sua função é usar o *buffer* recebido e construir o programa de acordo com as instruções armazenadas no *buffer*.

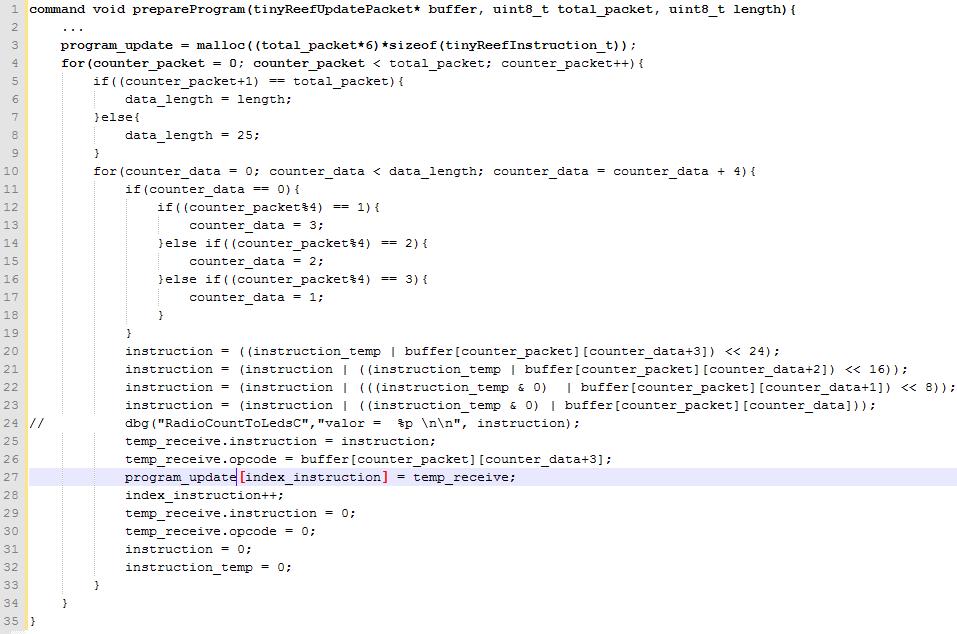


Figura .: *Loop* do comando *prepareProgram* que monta a nova versão do programa aplicativo

A Figura 5.15 mostra o trecho de código do *loop* que prepara o programa. Na linha 3 vê-se a alocação da variável que representa o novo programa. Seu tamanho é calculado de acordo com o número total de pacotes multiplicado por 6 e pelo tamanho de uma instrução. O número 6 representa o número máximo de instruções que podem estar contidas em um pacote.

Depois de alocar a variável, será preparado todo o programa através de dois laços (linhas 4 e 10). O primeiro indica o índice do número do pacote sendo lido e o segundo representa o *byte* dentro do pacote. As verificações feitas nas linhas 11 a 18 são para manter a sequência de leitura correta uma vez que o pacote pode conter 6 instruções mais 1 *byte*.

São usadas duas variáveis auxiliares do tipo inteiro (instruction e instruction\_temp) para armazenar os bytes da instrução na ordem correta e também a variável temp\_receive que é do tipo estrutura definido para instruções TinyReef. Ela é formada pelo opcode e pelos 4 *bytes* que formam a instrução. A Figura 5.16 mostra a estrutura que representa uma instrução da máquina virtual TinyReef.

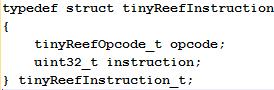


Figura .: Estrutura que representa uma instrução da máquina virtual TinyReef

As linhas 25 e 26 mostram a atribuição da variável instruction e do opcode aos respectivos campos da variável temp\_receive. Então o valor de temp\_receive é atribuído a variável program\_update (linha 27). Ao final do *loop*, o novo programa estará pronto para ser carregado. Então será invocada a função loadProgram passando como parâmetro o ponteiro program\_update para o novo programa finalizando assim o ciclo de atualização.

As modificações no componente *ReceiveByteCode* foram feitas para torná-lo apto a identificar as mensagens destinadas ao componente *UpdateHandler*. A apresenta o código da função receive do componente *ReceiveByteCode*.

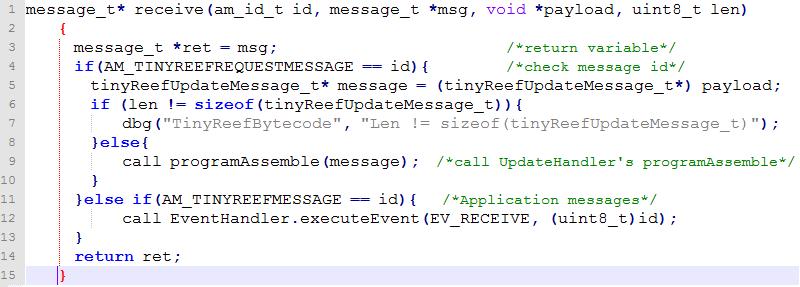


Figura 5.17: Código da função *receive* do componente *ReceiveBytecode*

Na linha 4 é verificado o tipo AM do pacote para determinar se se trata de um pacote de atualização. Caso a verificação seja positiva e o tamanho do pacote esteja correto (linha 6), o comando programAssemble será invocado passando como parâmetro a mensagem recebida (linha 9).

Finalizada a implementação do componente *UpdateHandler* e feitas as modificações nos componentes *Loader* e *ReceiveByteCode* foram iniciadas os testes de verificação funcional do sistema. Para isso foi utilizado o simulador TOSSIM usando o ambiente de teste mencionado na Subseção 5.4.1.

Como se pode observar na Figura 5.15 (linha 24) deixou-se de propósito o comando de *debug* (dbg) da interface de depuração do TOSSIM para mostrar como ele foi utilizado na validação do protocolo. Utilizando esses comandos de depuração foi possível analisar as variáveis do sistema ao longo da execução. Essa validação incluiu diferentes casos de testes como: variação do tamanho dos pacotes, perdas de pacotes, pacotes fora de sequência e diferentes versões de programas.

O comportamento na estação-base, por conseguinte, o aplicativo servidor *ProgramSender* é bastante simples. Ele apenas imprime as mensagens de requisição recebidas na sua área de texto que mostra o *status* das transações realizadas. A partir dessas mensagens é possível identificar o tipo da requisição, que é impressa no console da interface gráfica e escolher ou criar uma versão que será enviada através do nó *gateway* que se comunica via serial com o *host*.

Esta comunicação serial foi construída utilizando uma biblioteca fornecida pelo TinyOS para integração com aplicações customizadas para monitoramento no PC. Ela pode ser encontrada no diretório *raiz\_tinyos/support/sdk/java/tinyos.jar*.

A modificação realizada nessa biblioteca foi que as mensagens que chegam à estação-base ao invés de serem impressas no console são direcionadas para interface gráfica.

Além disso, as mensagens que ele recebe foram customizadas para refletirem o formato das mensagens definidas no protocolo. Para isso utilizou-se a ferramenta chamada de MIG (*Message Interface Generator*) que é um *parser* que gera código para as linguagens Java, C e Python .

Ele recebe três argumentos que são a linguagem para qual será gerado o código, o arquivo cabeçalho que descreve a estrutura da mensagem e o nome da estrutura. Um exemplo do comando na linha de código é mig java -target = micaz -I./ -java-classname = TinyReefUpdateMessage TinyReefMessage.h TinyReefUpdateMessage.

O ambiente de teste do *ProgramSender* permitiu a validação do protocolo num ambiente real usando os nós sensores. A depuração do programa executando no nó foi possível através da biblioteca *printf* fornecida pelo TinyOS que permite imprimir mensagens no console, contendo inclusive o valor de variáveis, a partir de uma interface serial .

A integração com a máquina virtual TinyReef implicou em algumas alterações nos seus componentes como foi mencionado na Seção 5.2. Além dos testes tradicionais comentados a pouco a funcionalidade foi validada com um exemplo que será explicado na Subseção 5.4.3.

A apresenta a utilização das memórias RAM e ROM em bits e percentual com relação à RAM de 4KB e a ROM de 128 KB da plataforma MICAZ. Nela pode-se observar que o aumento do uso de memória RAM da máquina virtual TinyReef com a introdução do serviço de atualização foi de apenas 1,4% e de 1,9%, respectivamente, executando nas versões 2.0.2 e 2.1.0 do TinyOS.

Os números indicam uma melhora de 20% para a utilização da memória ROM com a utilização da versão 2.1.0, porém com um aumento da utilização de memória RAM de quase 14% se comparados os dados para as aplicações executando sobre a máquina virtual TinyReef integrada ao componente *UpdateHandler*.

Tabela 5.1: Comparativo de utilização de memória da máquina virtual TinyReef executando em diferentes versões do TinyOS

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Aplicação | versão TinyOS | RAM (*bits*) | ROM (*bits*) | RAM(%) | ROM(%) |
| TinyReef\_Sem\_Updatehandler | v2.0.2 | 3098 | 46758 | 75,63 | 35,67 |
| TinyReef\_Sem\_Updatehandler | v2.1.0 | 3651 | 21040 | 89,14 | 16,05 |
| TinyReef\_Com\_Updatehandler | v2.0.2 | 3167 | 52728 | 77,32 | 40,23 |
| TinyReef\_Com\_UpdateHandler | v2.1.0 | 3724 | 23624 | 90,92 | 18,02 |

A compara o número de linhas de código dos componentes *UpdateHandler*, *ReceiveByteCode*, *Loader* e *EventHandler*  antes e depois da integração do serviço de atualização da TinyReef.

Tabela 5.2: Número de linhas de código de alguns componentes da TinyReef antes e depois da introdução do serviço de atualização.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Versão da TinyReef | *UpdateHandler* | *Loader* | *ReceiveBytecode* | *EventHandler* |
| Com o serviço de atualização | 242 | 170 | 110 | 45 |
| Sem o serviço de atualização | \*\*\* | 146 | 100 | 72 |

### Exemplo de Funcionamento

O exemplo de funcionamento foi realizado mudando-se o comportamento de uma aplicação simples cuja tarefa era acender os 3 *leds* do nó sensor de acordo com o valor de um contador que é incrementado a cada 250 ms.

A apresenta o código do programa usado como exemplo para validação do serviço de adaptação que inicialmente está executando na máquina virtual. As instruções (linha 2-11) são usadas para configurar a aplicação. A execução da instrução HALT (linhas 11, 14, 17 e 20) indica que o programa ficará em estado ocioso esperando a ocorrência de eventos. As instruções nas linhas 2, 5 e 7 consistem na ativação de 3 contadores que dispararão eventos em 250, 500 e 1000 ms. Esses valores são carregados na memória pelas instruções MOV\_CONST nas linhas 2, 4 e 6.

O evento disparado em 250 ms atribui o valor 10 ao contador de programa (PC) da máquina virtual que corresponde a execução da instrução da linha 13 da . Os eventos disparados em 500 ms e 1000 ms atribuem os valores 12 e 14 ao PC que correspondem, respectivamente, a execução das instruções nas linhas 16 e 19 da .

As instruções nas linhas 13, 16, e 19 acendem os *leds* de acordo com o valor armazenado nos registradores $1, $2, $3, cujos valores são atribuídos pelas instruções MOV\_CONST apresentadas nas linhas 8, 9 e 10.

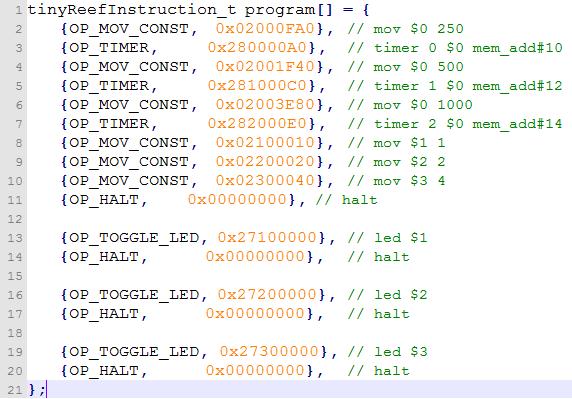


Figura 5.18: Programa usado como exemplo para validação do serviço de adaptação

O novo programa enviado fará a mesma tarefa sendo que a ordem de acendimento dos *leds* será feita de forma diferente. As instruções das linhas 8, 9 e 10 foram modificadas para atribuir os valores 4, 2 e 1 aos registradores $1, $2, $3.

Desse modo, manteve-se o comportamento da aplicação de acender os *leds* de acordo com o incremento de um contador, mas a ordem de acendimento dos *leds* foi trocada. Esse exemplo valida o funcionamento do serviço de atualização integrado a máquina virtual TinyReef.

## Considerações Finais

Ao longo deste capítulo foi explicado o processo de desenvolvimento e validação da extensão da TinyReef para suporte do serviço de atualização dinâmica dos nós. Através de diagramas de fluxos e exemplos de código pode-se entender a estratégia utilizada para garantir a entrega de forma confiável das atualizações sem comprometer o desempenho da rede nem os recursos de memória da plataforma de hardware.

Também foi explicado como foi feita a escolha do tamanho dos pacotes de modo a garantir flexibilidade caso haja necessidade de mudanças na arquitetura da TinyReef ou este serviço seja ampliado para outras funcionalidades como adição de políticas de adaptação.

Para validar o funcionamento do serviço, um teste de atualização de um programa foi realizado com sucesso validando seu algoritmo. O Capítulo 6 apresenta alguns exemplos de projetos de máquinas virtuais para redes de sensores que possuem serviço de atualização semelhante.

# Trabalhos relacionados

Este capítulo apresenta dois exemplos de projetos de máquinas virtuais que possuem a funcionalidade de reprogramação de seus nós através de atualizações enviadas via mensagens assim como foi implementado na TinyReef.



## Maté: *a tiny virtual machine for sensor networks*

Maté é uma máquina virtual que funciona de acordo com o paradigma de interpretação de *bytecodes* e possui arquitetura baseada na manipulação de dados em pilhas . Esta arquitetura foi escolhida porque permite definir um conjunto de instruções independentes do tipo de operando, pois este se encontra na pilha de operandos. Seu principal objetivo é facilitar o desenvolvimento de aplicações para redes de sensores e promover a reprogramação da rede de maneira fácil e sem comprometer o desempenho do sistema.

Ela foi implementada como um componente do sistema operacional TinyOS que interage com os componentes responsáveis pela comunicação do hardware de forma transparente para o projetista. A transparência está relacionada com o modo como as instruções são executadas e o formato dos programas.

Os programas consistem de uma ou mais capsulas de 24 *bytes* compostos por instruções de um *byte* de tamanho. O tamanho das capsulas foi definido de modo que possam caber em um pacote TinyOS tornando as capsulas atômicas. Cada capsula possui um identificador e uma versão.

Maté possui 4 tipos de capsulas: recebimento de mensagens, envio de mensagens, clock e sub-rotinas. As sub-rotinas são usadas para programas mais complexos onde os tipos citados não cabem em uma única capsula. O número de sub-rotinas é limitado a 4.

Sua execução se baseia na manipulação de dois tipos de pilhas. Uma para operandos e outra para endereços de retorno definidas para cada tipo de capsula. A troca de informações entre as capsulas é feita através de uma *heap* onde são definidas as prioridades.

Cada instrução é executada como uma tarefa do TinyOS o que permite o sistema executa várias instruções sem que haja bloqueio quando a máquina virtual esperar um evento. Nesse caso ela guarda o endereço da sub-rotina que irá tratar o evento na pilha de endereços.

Com relação à reprogramação, os pacotes possuem dois campos: o tipo que possui sete valores diferentes (sub-rotinas de 0-3, recebimento, envio e *clock* ) e uma versão. Se a máquina virtual recebe uma versão mais recente da capsula que está executando em execução, ela a atualiza para a nova versão. A versão é representada como um contador de 32*bits*.

Além disso, Maté possui uma instrução para retransmitir essa atualização para outros nós na rede através de mensagem de *broadcast* para os seus vizinhos. Este tipo de abordagem é chamado de programação viral. A atualização se espalha pela rede como um vírus.

Dentro da nova capsula deve ter essa instrução de propagação. Existe também uma instrução que transmiti outros tipos de capsula. Os desenvolvedores testaram esse sistema de encaminhamento de atualizações para reprogramação da rede e chegaram à conclusão de que ele funciona, mas que o encaminhamento controlado pela aplicação não é eficiente. À medida que a taxa de pacotes aumenta na rede começam a ocorrer congestionamento o que acaba degradando o desempenho devido ao atraso de entrega.

Diante desse contexto, os autores sugerem métodos alternativos como adicionar informação nos pacotes informando se o nó deve ou não repassar a atualização ou então só repassar a determinados grupos utilizando o id de grupos disponibilizado pelos pacotes AM.

## Davim: *a dynamically adaptable virtual machine for sensor networks*

Davim é um *middleware* configurável que suporta a customização de sistemas para domínios de aplicações complexas . Ele se baseia no modelo de atualização da máquina virtual Maté com a diferença de que apenas a parte do código que mudou será enviada. Ele permite a extensão do sistema dinamicamente através da adição e ou atualização de bibliotecas do sistema operacional e suporta múltiplas aplicações executando em máquinas virtuais isoladas.

Para permitir tal funcionalidade sua arquitetura é composta por módulos que gerenciam diferentes máquinas virtuais e diferentes bibliotecas do sistema operacional. As bibliotecas servem para prover serviços enquanto que as máquinas virtuais servem para isolar as aplicações.

Davim executa sobre o sistema operacional SOS que possui a funcionalidade de adicionar, remover e atualizar módulos do sistema em tempo de execução . Ela também possui a característica de adicionar ou modificar instruções das máquinas virtuais em tempo de execução através da adição de código nativo em bibliotecas. Esses códigos são transformados em instruções da máquina virtual.

Apesar de se mostrarem inviáveis para os nós sensores mais recentes que possuem ainda restrições de memória como, por exemplo, um nó do tipo MicaZ não seria capaz de suportar esta arquitetura com seus 4KB de memória de dados.

Segundo Horre o SOS reserva 2KB de memória dinâmica na plataforma MicaZ, mas para suportar o core do Davim seria necessário o dobro desse valor. Portanto, ele não é indicado para domínios de aplicação com funcionalidade simples.

## Considerações Finais

A análise desses trabalhos serviu de base para construção do serviço de atualização para a TinyReef. Neles pode-se ver uma análise mais detalhada de duas técnicas diferentes de atualização.

Na Maté os programas são enviados de maneira parecida com o projeto implementado na TinyReef a grande diferença é que os projetistas não se preocuparam em definir uma estratégia de recuperação de pacotes perdidos que evite a inundação de pacotes na rede.

Os desenvolvedores da Davim buscaram tornar sua arquitetura mais flexível possível permitindo diferentes tipos de atualização como, por exemplo, atualização de programas e também de instruções da máquina virtual. No entanto, esta abordagem é inviável devido à quantidade de memória requerida para suportar esta flexibilidade.

# Conclusão e Trabalhos Futuros

Este trabalho abordou o processo de desenvolvimento e validação de uma extensão da máquina virtual TinyReef para dar suporte ao serviço de atualização dinâmica de redes de sensores sem fio através de uma estratégia baseada em um protocolo de envio de mensagens de atualização confiável.

Para definir tal protocolo foi realizado um estudo detalhado das características das redes de sensores buscando entender a real necessidade desse tipo de serviço e qual seria a melhor solução para ser aplicada a uma rede que executa sobre uma máquina virtual.

A partir dessa análise percebeu-se que o serviço de adaptação da rede é muito importante em virtude da limitação dos recursos que as plataformas de hardware possuem e da heterogeneidade do ambiente em que estes sistemas estão inseridos. Portanto, a característica de reprogramação da rede pode ser uma alternativa para superar estes problemas.

O uso de máquina virtual visa diminuir o tamanho dos programas com o intuito de facilitar o desenvolvimento de aplicações e diminuir o custo de propagação de programas grandes ao longo da rede, porém isto tem um custo sobre o uso da memória e do desempenho do processador.

Pensando nisso, o projeto do protocolo definido buscou diminuir o número de pacotes a serem enviados através de uma técnica que garante a entrega das mensagens de atualização, mas que retarda ao máximo a requisição de pacotes perdidos com o intuito de não sobrecarregar a rede com pacotes inúteis. Além disso, este protocolo foi pensado de modo a ser flexível com relação a mudanças na arquitetura da TinyReef e com a possibilidade de adição de tipos de pacotes que permitam um gerenciamento mais apurado dos recursos da rede.

Uma melhoria futura seria tornar possível a atualização de apenas parte do código como, por exemplo, definir programas modularizados de modo que apenas alguns módulos precisem ser enviados. Outra extensão seria a adoção de um protocolo de propagação dos programas para os demais nós da rede utilizando uma técnica de disseminação por multi-saltos coordenada pela estação-base.

Referências

1. Akyildiz, I. F., W. Su, Y. Sankarasubramaniam, e E. Cayirci. “A Survey on Sensor Networks.” *IEEE Communication Magazine* 40 (Aug 2002): 102-114.
2. ATMEL. “IEEE 802.15.4 Mac User Guide.” 2006. http://www.atmel.com/dyn/resources/prod\_documents/doc5182.pdf (acesso em 25 de Novembro de 2009).
3. Buonadonna, P., J. Hill, e D Culler. “Active Message Communication for Tiny Networked Sensors.” *20th Annual Joint Conference of the IEEE Computer and Communications Societies*, Abril de 2001.
4. Chipcom, Products. “CC2420: 2.4 GHZ IEEE 802.15.4 ZigBee-ready RF Transceiver.” *Texas Instruments.* 2008.
5. Comunity, TinyOS. *An open-source OS for the networked sensor regime.* http://www.tinyos.net/ (acesso em 26 de Novembro de 2009).
6. *Crossbow Technology.* http://www.xbow.com (acesso em 25 de Novembro de 2009).
7. Crossbow, Technology. “MicaZ: WIRELESS MEASUREMENT SYSTEM.” *Crossbow Technology.* http://www.xbow.com/Products/Product\_pdf\_files/Wireless\_pdf/MICAZ\_Datasheet.pdf (acesso em 25 de Novembro de 2009).
8. Culler, D., J. Hill, M. Horton, K. Pister, R. Szewczyk, e A. Woo. “MICA: The Commercialization of Microsensor Motes.” 2002. http://www.sensorsmag.com/bookmark\_migration.php?page=360782 (acesso em 25 de Novembro de 2009).
9. Da Silva, Ivanovitch Medeiros Dantas. “Redes de Sensores sem Fio aplicadas em Ambientes Industriais de Petróleo e Gás.” *Agência Nacional de Petróleo (ANP).* Dezembro de 2006. http://www.anp.gov.br/CapitalHumano/Arquivos/PRH22/Ivanovitch-Medeiros-Dantas-da-Silva\_PRH22\_UFRN\_G.pdf (acesso em 27 de Novembro de 2009).
10. de Freitas, E. P. “A Survey on Adaptable Middleware for Wireless Sensor Networks.” School of Information Science, Computer and Electrical Engineering, Halmstad University, Box 823, SE-30118 Halmstad, Sweden, 2008.
11. Demers, A., J. Gehrke, R. Rajaraman, e N. & Yao, Y. Trigoni. “The Cougar Project: A Work In Progress Report.” *In Sigmod Record*, 2003.
12. Gay, D., P. Levis, R. von Behren, M. Welsh, E. Brewer, e D. Culler. “The nesC Language: A Holistic Approach to Networked Embedded Systems.” *Programming Language Design and Implementation (PLDI)*, 2003.
13. Guimarães, G., E. Souto, M. Vieira, G. Vasconcelos, N. Rosa, e C. & Kelner, J. Ferraz. “Middleware para Redes de Sensores Sem-Fio: Projeto, Implementação e Avaliação de Consumo de Energia.” *24° Simpósio Brasileiro de Redes de Computadores (SBRC)*, 2006: 1-16.
14. Hadim, S., e N. Mohamed. “Middleware for Wireless Sensor Networks: A Survey.” *Communication System Software and Middleware*, 2006: 1-7.
15. Han, C.-C., R. Kumar, R. Shea, E. Kohler, e M. Srivastava. “A Dynamic Operating System for Sensor Nodes.” *3rd international conference on Mobile systems, applications, and services*, 2005: 163-176.
16. Heinzelman, W. R., e A. & Balakrishnan, H. Chandrakasan. “Energy-efficient communication protocol for wireless microsensor networks.” *33rd Annual Hawaii International Conference on System Sciences*, Janeiro de 2000.
17. Horre, W., S. Michiels, e W. & Verbaeten, P. Joosen. “DAVIM: Adaptable Middleware for Sensor Networks.” *Distributed Systems Online, IEEE*, 2008.
18. Levis, P., e D. Culler. “Maté: A Tiny Virtual Machine for Sensor Networks.” *10th international conference on Architectural support for programming languages and operating systems*, Outubro de 2002.
19. Levis, P., et al. “TinyOS: An Operating System for Sensor Networks.” *TinyOS.* http://www.dbis.ethz.ch/education/ss2007/tatbul/hotdms/papers/tinyos\_chapter.pdf (acesso em 27 de Novembro de 2009).
20. Levis, P., N. Lee, M. Welsh, e D. Culler. “TOSSIM: Accurate and Scalable Simulation of Entire.” *First ACM Conference on Embedded Networked Sensor Systems (SenSys)*, 2003.
21. Loureiro, A. A. F., J. M. S. Nogueira, L. B. Ruiz, R. A. de Freitas Mini, e E. F. & Figueiredo, C. M. S. Nakamura. “Redes de Sensores Sem Fio.” *Simpósio Brasileiro de Redes de Computadores*, 2003: 179-226.
22. MADDEN, S. R., M. J. FRANKLIN, e J. M. & HONG, W. HELLERSTEIN. “TinyDB: An Acquisitional Query Processing System for Sensor Networks.” *ACM Transactions on Database Systems*, 2005: 122–173.
23. Mainwaring, A., J. Polastre, R. Szewczyk, e D. & Anderson, J. Culler. “Wireless Sensor Networks for Habitat Monitoring.” *ACM International Workshop on Wireless Sensor Networks and Applications*, Sept de 2002.
24. Marques, I. L., J. Ronan, e N. S. Rosa. “TinyReef: a Register-Based Virtual Machine for Wireless Sensor Networks.” *8th Annual IEEE Conference on Sensors*, Outubro de 2009: 1-6.
25. Pereira, M. R., e C. L. & de Castro, M. C. S. de Amorim. “Tutorial sobre Redes de Sensores.” http://magnum.ime.uerj.br/cadernos/cadinf/vol14/3-clicia.pdf (acesso em 25 de Novembro de 2009).
26. Roca, I., F. de Moraes, e S. T. Kofuji. “APLICAÇÕES DAS REDES DE SENSORES SEM FIO (RSSF) NA ENGENHARIA NAVAL E OCEÂNICA.” *INSTITUTO PAN-AMERICANO DE ENGENHARIA NAVAL (IPEN).* 2007. http://www.ipen.org.br/downloads/XX/2G/APLICACOES.pdf (acesso em 27 de Novembro de 2009).
27. Ruiz, L. B., et al. “Arquitetura de Redes Sensores Sem Fio.” In: *Minicurso do XXII Simpósio Brasileiro de Redes de Computadores*, 167-218. Gramado, RS, 2004.
28. Silva, F. A., T. R. de Moura Braga, e L. B. & Nogueira, J. M. S. Ruiz. “Publicações Sensornet.” *Sensornet.* Janeiro de 2003. http://www.sensornet.dcc.ufmg.br/Publicacoes/pesquisanodosRT2003.pdf (acesso em 26 de Novembro de 2009).
29. Sinha, A., e A. Chandrakasan. “Dynamic power management in wireless sensor networks.” *Design & Test of Computers, IEEE*, March de 2001: 62-74.
30. *Sistema Operacional Ubuntu.* http://www.ubuntu.com/ (acesso em 28 de 11 de 2009).
31. Tanembaum, Andrews S. “CRC.” In: *Redes de Computadores*, por A. S. Tanembaum, tradução: Vandenberg D. de Souza, 208-212. Rio de Janeiro: Campus, 2003.
32. Verdone, R. & Buratti, C. “Modelling for Wireless Sensor Network Protocol Design.” *International Workshop on Wireless Ad-hoc Networks*, 2005.
33. von Eicken, T., e S. C. & Schauser, K. E Goldstein. “Active Messages: a Mechanism for Integrated Communication and Computation.” *9th Annual International Symposium on Computer Architecture*, 1992.
34. Wang, Q., Y. Zhu, e L. Cheng. “Reprogramming Wireless Sensor Networks: Challenges and Approaches.” *IEEE Network*, 2006: 48-55.
35. wiki, TinyOS. *Mote-PC serial communication and Serial Fowarder.* http://docs.tinyos.net/index.php/Mote-PC\_serial\_communication\_and\_SerialForwarder#MIG:\_generating\_packet\_objects (acesso em 25 de Novembro de 2009).
36. Wiki, TinyOS. *The TinyOS printf Library.* http://docs.tinyos.net/index.php/The\_TinyOS\_printf\_Library (acesso em 28 de 11 de 2009).