Programando aplicações para redes de sensores sem fio usando uma linguagem reativa

Aluno: Carlo Melo Caratori
Orientador: Noemi Rodriguez
Co-orientador: Francisco Sant'anna

Introdução

Redes de Sensores sem Fio (RSSF) consistem de sensores autônomos (motes) capazes de monitorar o ambiente no qual eles estão inseridos através de sensores de luz, temperatura, som, entre outros. Estes motes também são capazes de se comunicar, permitindo que este monitoramento abranja grandes áreas. Seu desenvolvimento inicial foi motivado por aplicações militares como vigilância em campos de batalha, mas a constante queda no custo dos motes tem incentivado outras áreas a pesquisar utilidades para os mesmos.

Objetivos

Softwares desenvolvidos para RSSF geralmente envolvem uma grande quantidade de medidas dos sensores, processamento, atuação e comunicação. Essas atividades são em grande parte reativas, tendo em vista que elas envolvem envio e recebimento de mensagens, sensoriamento, etc. O objetivo desta Iniciação Cientifica é avaliar a utilidade de Céu neste mundo de programação em sensores. Céu é uma linguagem de programação reativa que está sendo desenvolvida pelo grupo de RSSF da PUC-Rio. Esta avaliação deve ser executada durante o desenvolvimento da linguagem, dando suporte ao mesmo.

Metodologia

Antes que se possa defender o uso de outra linguagem em RSSF, é preciso explicar o porquê de não estarmos interessados em usar as já existentes. Atualmente, a linguagem de programação mais difundida em aplicações com sensores é nesC [1], que nada mais é do que uma extensão de C desenvolvida com o intuito de englobar os conceitos estruturais e modelo de execução do TinyOS [2], o sistema operacional criado para os motes.

Contudo, apesar de ser a linguagem mais usada neste seguimento, nesC apresenta alguns contratempos. O mais notável diz respeito à dificuldade em se programar com ela. Por ser uma linguagem dirigida a eventos, nesC se baseia em uma cadeia de chamadas e retornos de funções, tornando a programação de problemas não triviais bastante difícil e propícia a erros.

Com isso chegamos a Céu, uma linguagem reativa baseada em um pequeno conjunto de primitivas com funcionalidades similares as de Esterel [3]. Céu é uma linguagem síncrona com sintaxe e semântica completamente diferentes de nesC, mas ao ser compilada gera códigos em C, levando a programas eficientes tanto em termos de memória quanto processamento. Céu também é baseada no uso de eventos internos e externos ao programa, que controlam seu fluxo de execução.

Para que pudéssemos comparar Céu a nesC e avaliar sua performance em aplicações de RSSF, decidimos que os testes seriam feitos baseados nos exemplos demonstrados pelo tutorial [4] do TinyOS e que alguns parâmetros deveriam ser levados em consideração. Estes testes são aplicações reais com diferentes níveis de dificuldade, além de serem muito bem documentados, facilitando a avaliação em si.

Antes de detalhar o desenvolvimento do projeto em si, é preciso uma pequena introdução à Céu, assim como às ferramentas oferecidas pela linguagem.

Céu

Céu é uma linguagem reativa que permite a execução de múltiplas linhas de comando de maneira concorrente. A comunicação com o ambiente é feita através de eventos externos, podendo ser de entrada ou saída. O seguinte exemplo ilustra bem como essa comunicação é feita.

```
( ~Radio_receive ~> Leds_set) *
```

Este programa espera ("~") o evento externo de entrada *Radio_receive*, e ao recebê-lo, gera ("~>") o evento externo de saída *Leds_set* passando como parâmetro o valor recebido pelo radio. Em seguida, ele entra em loop ("*"), repetindo o processo. Vale à pena citar que eventos externos de entrada não podem ser gerados, assim como eventos externos de saída não podem ser esperados.

Em Céu existe também o conceito de eventos internos, que são visíveis somente dentro do próprio programa. Eventos internos são o principal meio de comunicação entre linhas de execução concorrentes. O seguinte exemplo ilustra esta característica.

```
( ~1000 ; val ~> add ) *
||
( ~add -> inc => val) *
```

Este programa contém duas linhas de código que são executadas em paralelo ("||"). Na primeira região, o programa espera 1000 milissegundos e gera o evento interno *add* passando como parâmetro o valor da variável *val* e repete este processo. Na segunda região, o programa espera o evento interno *add*. Ao "acordar", a variável *add* vai conter o valor de *val* gerado pela primeira região, incrementará este valor, e o atribuirá ("=>") novamente a *val*. Em seguida volta a esperar por *add.*

Eventos externos começam com letra maiúscula, e os internos com letra minúscula. Estes conceitos de eventos internos e externos são o suficiente para entender como programas mais completos funcionam, apesar de a linguagem ainda possuir outras funcionalidades.

Desenvolvimento

Para começar os testes foi escolhido um exemplo muito simples, o primeiro do tutorial, que simplesmente pisca os LEDs dos motes um a um com diferentes freqüências. Já neste teste foi percebida uma grande vantagem de Céu, chamada por nós de expressividade. Com poucos comandos e linhas de código, Céu foi capaz de realizar o mesmo trabalho do código em nesC.

Com isso continuamos com os testes baseados nos exemplos do TinyOS, e novamente os códigos em Céu se mostraram muito mais simples e menores. Nestes testes foram implementados protocolos de comunicação simples, comunicação entre motes e computadores através de portas seriais, sensoriamento, etc. Vale a pena dizer que desde o início houve a preocupação de ser extremamente fiel aos exemplos abordados para que a comparação e avaliação posterior tivessem algum significado pertinente.

Por fim chegamos a um dos exemplos do tutorial que se mostra mais completo.

Antitheft

Antitheft é uma aplicação de detecção de furtos que leva em conta os valores lidos pelos sensores de luz e movimento dos motes para dizer se aquele mote foi roubado ou não. É uma aplicação completamente configurável, onde o usuário informa aos motes como eles

detectam o furto e como devem agir ao detectá-lo, e que engloba as principais funções dos motes.

A aplicação em si se divide em três partes. A primeira é em Java, e é na verdade a interface da aplicação com o usuário. Através desta interface, o usuário é capaz de configurar o intervalo de tempo em que os motes verificarão o furto, como será feita essa verificação (pelo sensor de luz ou pelo de aceleração) e como os motes devem reagir assim que forem roubados (piscando um led, avisando aos motes vizinhos via radio, ou apitando). Qualquer configuração escolhida pelo usuário é enviada à rede de sensores através da porta serial. A segunda parte é programada em Céu, mas na verdade foi desenvolvida em outro tutorial. É chamada de Basestation, e nada mais é do que uma ponte entre os sensores e o computador. O mote com Basestation instalado é conectado a um computador via porta serial, e sempre que recebe uma mensagem a transmite para frente. Ou seja, quando recebe uma mensagem serial a transmite pelo radio, e quando recebe pelo radio a transmite pela porta serial.

A terceira e mais importante parte é o código de detecção de furto em si. Também programado em Céu, este é o programa que recebe as preferências do usuário e configura o mote para atendê-la. Alem disso, o código é responsável por verificar os sensores do mote e decidir se o mesmo foi roubado ou não, reagindo da forma adequada. O código a seguir é um trecho deste programa, usado somente para exemplificar a simplicidade do mesmo.

```
~radio_started ;
 (
   ( ~radio_received ~> config )*
   ||
   ( ~(interval) ; ~>check_theft )*
 )
```

O programa inicialmente espera o rádio do mote ser inicializado. Assim que o evento interno *radio_started* é gerado em outra região do código, o programa abre duas regiões paralelas responsáveis por tarefas específicas. A primeira fica esperando uma mensagem via rádio ser recebida (*radio_received)*, e após gera o evento interno *config* passando o valor recebido como parâmetro. Em outra parte do código, em paralelo com esta, uma linha de execução fica permanentemente esperando o evento *config* e assim que ele é gerado configura o mote de acordo com as preferências do usuário. A segunda região gerada pelo código acima espera um certo intervalo *interval* passar, e depois gera o evento interno *check_theft.* Em outra parte do código, em paralelo com esta, uma linha de execução fica permanentemente esperando este evento e quando ele é gerado verifica se aquele determinado mote foi roubado ou não e reage de acordo.

O programa completo é razoavelmente menor e extremamente mais simples do que seu similar em nesC. Isto se deve às ferramentas extremamente uteis oferecidas pela linguagem como, por exemplo, a propagação de valores de variáveis (variáveis reativas) pelo programa, usada diversas vezes neste programa.

Ferramenta de Testes

Outra grande funcionalidade de Céu é a possibilidade de testar o programa sem a necessidade de usar o hardware (mote) em si. Através da própria linguagem é possível criar o que chamamos de regiões assíncronas de código (“@{código}”) que permitem ao programador simular basicamente tudo desde envio e recebimento de mensagens, valores lidos pelos sensores e até condições de erro, dentre outros.

```
@{
        1~>Radio_startDone
        0~>Radio_startDone
        25~>Radio_receive
        ~>(10000)
}
```

A região assíncrona acima é apenas um exemplo de como esta ferramenta pode ser usada. Assim que a região síncrona do programa não tem mais nada a executar (todas as linhas de execução do programa principal terminaram ou estão esperando), a região assíncrona é executada. Vale à pena frisar que diferentemente da região síncrona, eventos externos de entrada podem ser gerados na região assíncrona. Isto é o que possibilita a simulação em si. No caso do teste acima, a primeira linha gera o evento externo *Radio_startDone* com o valor 1. Como neste caso o valor 1 significa erro, ou seja, o rádio não foi inicializado corretamente, o próprio programa deve reagir de maneira adequada e tentar iniciá-lo novamente. Em seguida, a região assíncrona gera o evento *Radio_startDone* com o valor 0, que neste caso significa sucesso. Com isso o programa principal pode prosseguir sabendo que o rádio está corretamente inicializado. Este é um exemplo da verificação de condições de erro, extremamente difíceis de serem testadas usando o hardware. Em seguida vemos como o recebimento de mensagem pode ser simulado. O evento externo *Radio_Receive* é gerado com o valor 25 e qualquer linha de execução do código principal que esteja esperando receber uma mensagem via rádio é acordada.

A última linha demonstra outra grande utilidade das regiões assíncronas. Ela simula a passagem de 10000 milissegundos. Ou seja, a simulação de tempo real é possível e extremamente útil, tendo em vista que podemos simular a reação de programas a passagem de tempo e verificar se ela foi correta.

Todos estes artifícios inseridos nas regiões assíncronas de código tornam a programação em si muito menos propícia a erros, tendo em vista que os testes no programa podem ser feitos independentemente do hardware, isolando a ocorrência e facilitando a correção de eventuais erros.

Conclusões

A pesquisa desenvolvida durante esta Iniciação Cientifica foi extremamente válida. Foi uma experiência agregadora, onde tive a oportunidade de trabalhar conceitos vistos em sala de aula, como protocolos de comunicação, programação orientada a objetos, linguagens de programação, dentre outros.

Vale também salientar que o objetivo do programa de Iniciação Cientifica foi cumprido. Durante esses meses de testes usando Céu a linguagem evoluiu consideravelmente e em seu atual estagio oferece ao programador um ambiente simples e seguro de programar. Apesar de não ser uma linguagem completa ainda, pode-se perceber que ela oferece inúmeras vantagens em relação à nesC.

Parte desta grande evolução de Céu se deve a este trabalho de pesquisa e testes desenvolvido por nós durante o último ano.

Referências

[1] Gay, D., Levis, P., Behren, R., Welsh, M., Brewer, E., and Culler, D. **The nesC Language: A Holistic Approach to Networked Embedded Systems**, In *Proceedings of Programming Language Design and Implementation* , June 2003.

[2] Hill, J., Szewczyk, R., Woo, A., Hollar, S., Culler, D., and Pister, K. **System architecture directions for networked sensors**. SIGPLAN Not. 35 (November 2000), 93-104.

[3] Berry, G., and Gonthier, G. **The ESTEREL synchronous programming language: design, semantics, implementation**. Science of Computer Programming 19, 2 (1992), 87-152

[4] TinyOS Tutorials. Disponível em: http://docs.tinyos.net/tinywiki/index.php/TinyOS_Tutorials