Extenção dos mecanismos de gerência de tarefas do sistema operacional TinyOS

Bolsista: Pedro Rosanes Orientador: Silvana Rossetto

Departamento de Ciência da Computação

Sumário

1 Resumo

Redes de Sensores Sem Fio (RSSFs) caracterizam-se pela formação de aglomerados de pequenos dispositivos que, atuando em conjunto, permitem monitorar ambientes físicos ou processos de produção com elevado grau de precisão. O desenvolvimento de aplicações que permitam explorar o uso dessas redes requer o estudo e a experimentação de protocolos, algoritmos e modelos de programação que se adequem às suas características e exigências particulares, entre elas, uso de recursos limitados, adaptação dinâmica das aplicações, e a necessidade de integração com outras redes, como a Internet. O sistema operacional mais usado nesses sensores é o TinyOS, um sistema leve, feito especialmente para consumir pouca energia, a característica mais importante para RSSFs. Sua programação é orientada a componentes para facilitar a reutilização de código, e a eventos para economizar memória. Isto cria um modelo de programação em duas fases, uma para envio do comando, e outra para espera da resposta (evento). O que causa a falta de um único fluxo de execução, levando a dificuldades no desenvolvimento de aplicações. O objetivo deste projeto é criar abstrações de programação para o sistema operacional TinyOS, visando facilitar a construção de abstrações de programação distribuida de nível mais alto que auxiliem o desenvolvimento de aplicações nessa área.

2 Introdução

Sistemas projetados para os dispositivos que formam as redes de sensores devem lidar apropriadamente com as restrições e características particulares desses ambientes. A arquitetura adotada pelo TinyOS prioriza fortemente o tratamento dessas restrições em detrimento da simplicidade oferecida para o desenvolvimento de aplicações. A linguagem de programação usada é o nesC, uma adaptação de C que provê baixo consumo de memória, optimizações, e previne condições de corrida. Para lidar com as diversas operações de entrada e saída, o TinyOS utiliza um modelo de execução em duas fases, evitando bloqueios e consequentemente armazenamento de estados. A primeira fase da operação é um comando que pede ao hardware a execução de um serviço (ex.: sensoreamento). Este comando retorna imediatamente dando continuidade à execução. Quando o serviço é terminado, o hardware envia uma interrupção, sinalizada como um evento pelo TinyOS. Então, o seu tratador recebe as informações (ex.: valor sensoreado) e lida com elas conforme programado. O problema gerado por isso é a falta de um fluxo continuo de execução, na perspectiva do programador. *o* citar livro* O modelo de concorrência divide o código em dois tipos, assíncrono e síncrono. O ultimo é composto por código alcançável somente a partir de tarefas (*tasks*) que são precedimentos adiados. Elas não podem se interromper, mas podem ser interrompidas por códigos assíncronos. O primeiro tipo é composto por código alcançável a partir de pelo menos um tratador de interrupção. Podem se interromper, e interromper códigos síncronos. As tarefas são todas escalonadas por um componente. Apesar da política de escalonamento ser do tipo *First-in*

First-out, este componente pode ser substituído para implementar outras políticas. De acordo com Levis e Culler *o* citar*, para que as redes de sensores sejam de fato adotadas é preciso que elas sejam mais fáceis de usar. Portanto no TinyOS 2.1.X foi implementado um modelo de threads, permitindo um novo paradigma de programação a um custo de gerenciamento das threads. No nosso projeto, propomos novos escalonadores de tarefas e um novo modelo de programção baseado em corotinas, para diminuir os custos de gerenciamento.

3 Objetivos do projeto

Os principais objetivos do projeto são aprofundar o conhecimento sobre o modelo de concorrência do TinyOS, avaliar os novos componentes e a flexibilidade de alteração, introduzidos nas versão 2.1.X, incluindo a implementação de threads. Propor e desenvolver diferentes políticas de escalonamento de tarefas, além de implementar e avaliar um mecânismo de gerência cooperativa de tarefas via corotinas.

4 Metodologia

A seguir a metodologia usada.

4.1 Primeiro ciclo (out, nov)

Estudar material introdutório sobre as ferramentas TinyOS, nesC e TOSSIM.

Instalar as ferramentas e experimentá-las em aplicações básicas.

Estudar/revisar conceitos fundamentais sobre concorrência e gerência de tarefas em sistemas operacionais.

4.2 Segundo ciclo (nov, dez, jan)

- Estudar o modelo de concorrência e os mecanismos de gerência de tarefas do TinyOS.
- Redigir texto sobre o modelo de concorrência e os mecanismos de gerência de tarefas do TinyOS.
- Projetar, implementar e avaliar outros componentes básicos de escalonamento de tarefas para o TinyOS.
- Redigir relatório técnico sobre os componentes de escalonamento desenvolvidos.
- Estudar o modelo de threads do TinyOS.
- Implementar aplicações básicas usando o modelo de threads do TinyOS.
- Redigir texto sobre o modelo de threads do TinyOS.

4.3 Terceiro ciclo (fev, mar, abr, mai)

- Estudar o conceito de co-rotinas (mecanismo de gerência cooperativa de tarefas)
- Pensar e elaborar proposta de extensão do modelo de concorrência do TinyOS e dos seus mecanismos de gerência de tarefas.
- Descrever resumidamente a proposta
- Projetar a solução e os experimentos que deverão ser realizados.
- Descrever o projeto e modelagem da solução e dos experimentos
- Implementar a solução.
- Executar os experimentos e avaliar os resultados.
- Redigir texto com proposta, projeto, modelagem, implementação e experiemntos realizados.

4.4 Quarto ciclo (mai, jun, jul, ago)

- Avaliar os resultados obtidos e refinar a proposta de solução.
- Propor trabalhos futuros.
- Redigir relatório final sobre a Iniciação Científica: atividades realizadas, conheci- mentos adquiridos, dificuldades encontradas.
- Redigir artigo completo.

5 Teoria

A seguir os estudos feitos.

5.1 Fundamentos das RSSFs com o TinyOS

Estudo feito sobre o sistema operacional TinyOS, sua linguagem de programação nesC e o simulador TOSSIM. Como fonte do aprendizado foram usados o minicurso desenvolvido pela professora Silvana Rosseto[1], o material oferecido pela página do TinyOS[2], e o livro *TinyOS Programming*, de Levis e Gay[3].

5.2 Concorrência e Gerência de Tarefas em Sistemas Operacionais

Antes de estudar o modelo de concorrência e gerência de tarefas específica do TinyOS, foi feita uma revisão para sistemas operacionais em geral. O matérial utilizado foi o livro do professor Carlos Maziero (PUCPR)[4].

5.3 Concorrência e Gerência de Tarefas no TinyOS

Neste momento foram estudados o modelo de concorrência e os mecanismos de gerência de tarefas específicos do TinyOS como a sequência de inicialização dos sensores, e o escalonador padrão.[**?**]

5.3.1 Sequência de Inicialização

*o*citar TEP* O principal componente do TinyOS responsável por botar o sistema no ar é o *MainC*. Para isso, ele inicializa o escalonador de tarefas, os componentes de hardware e software. Primeiro é configurado o sistema de memória e escolhido o modo de processamento. Com esses pré-requisitos básicos estabelecidos é inicializado o escalonador, para permitir que as proximas etapas possam postar tarefas.

O segundo passo é inicializar o hardware como um todo, permitindo a operabilidade da plataforma. Alguns exemplos são configuração de pinos de entrada e saida, calibração do clock e dos LEDs. Como esta etapa exige códigos específicos para cada tipo de plataforma, o MainC se liga ao componente *PlataformC* que implementa o programa necessário.

O terceiro passo trata o software. Além de configurar os aplicativos básicos do sistema, como o temporizador, também é preciso inicializar os programas do usuário. Portanto, se um componente do usuário precisa ser inicializado basta amarrá-lo ao SoftwareInit. Assim o TinyOS se responsabiliza por executar este código. Por ultimo, quando tudo é concluido, o MainC sinaliza que a inicialização concluiu, através do sinal *Boot.booted()*. Isso permite que os componentes rodem. E finalmente, o TinyOS entra no seu laço principal, no qual o escalonador espera por tarefas, e as executa. E importante notar que durante todo este processo, as interrupções do sistema ficam desabilitadas.

5.3.2 Mecânismos de Gerência de Tarefas

Tasks, ou tarefas, têm duas propriedades importantes. Elas não são preemptivas entre si, e são executadas de forma adiada. Isso significa que ao postar uma tarefa, o fluxo de execução continua, sem desvio, e ela só será processada depois. As tasks básicas não tem parâmetro, apesar de ser possível fazê-las receber parâmetros, criando uma interface e amarrando-a ao componente do escalonador.

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

É possível mudar a política de gerenciamento de tarefas substituindo o escalonador padrão. Para isto, basta adicionar uma configuração com o nome TinySchedulerC no diretório da aplicação e amarrá-la ao componente responsável pela implementação. Qualquer novo escalonador tem de aceitar a interface das tasks padrões, e garantir a execução de todas as tarefas, sem permitir Starvation

5.3.3 Modelo de Concorrência

O TinyOS mantém os problemas de concorrência bem simples, qualquer possível condição de corrida é detectada em tempo de compilação. Para que isso seja possível, o código em nesC é dividido em dois tipos:

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

Código Síncrono Código alcançável somente a partir de tasks.

Eventos e comandos que podem ser sinalizados ou chamados a partir de um trador de interrupção são códigos assíncronos. Eles podem interromper outros eventos, comandos e *tasks*. Por isso devem ser marcados como *async* no código fonte. O problema aparece quando variáveis compartilhadas são acessadas por esse tipo de código. Para contornar isso, deve-se usar o comando *atomic* ou *Power locks*.

O comando *atomic* permite que um trecho de instruções possa ser executado sem ser interrompido. Dois fatos importantes surgem com o seu uso, primeiro a ativação e desativação de interrupções consome ciclos de CPU. Segundo, longos trechos atômicos podem atrasar outras interrupções, portanto é preciso tomar cuidado ao chamar outros componentes a partir desses blocos.

Algumas vezes é preciso usar um certo hardware por um longo tempo, sem compartilhá-lo. Como a necessidade de atomicidade não está no processador e sim no hardware, pode-se conceder sua exclusividade a somente um 'usuário' através de *Power locks*. Para isso, primeiro é feito um pedido através de um comando, depois quando o recurso desejado estiver disponível, um evento é sinalizado. Assim não há bloqueio de execução, como em semáforos. Existe a possibilidade de

requisição imediata. Nesse caso nenhum evento será sinalizado: se o recurso não estiver protegido, ele será imediatamente cedido, caso contrário, o comando retornará falso. *Power Locks* têm três sub-componentes: Um abitrador que gerência as prioridades dos pedidos, um gerênciador de energia e um configurador que ajusta o hardware de acordo com o cliente.

5.4 Componentes de Escalonamento

Neste momento foram análisados e desenvolvidos diversos gerênciadores de tarefas.

5.4.1 Análise do Escalonador Padrão

O escalonador padrão adota uma política FIFO. A estrutura de dados utilizada é uma lista encadeada. Ele provê as interfaces *Scheduler* e *TaskBasic*. As tarefas se conectam ao escalonador através da *TaskBasic*. Ao compilar um programa em NesC, todas tarefas básicas viram uma interface desse tipo. Porém, para se diferenciarem é criado um parâmetro na interface [1].

5.4.2 Análise do Escalonador *Earliest Deadline First*

Este escalonador *o*citar de onde surgiu esse escalonador* aceita tarefas com deadline e elege as que tem menor *deadline* para executar. A interface usada para criar esse tipo de tarefas é *TaskDeadline*. O *deadline* é passado por parâmetro pela função *postTask*. As *TaskBasic* também são aceitas como recomendado pelo TEP 106[**?**].

Em contraste, o escalonador não segue outra recomendação: não elimina a possibilidade de *starvation* pois as tarefas básicas só são atendidas quando não há nenhuma com *deadline* esperando para executar. A fila de prioridades é implementada da mesma forma que a do escalonador padrão**??**, a única mudança está na inserção. Para inserir, a fila é percorrida do começo até o fim, procurando-se o local exato de inserção. Por tanto o custo de inserir é $\bigcirc(n)$, e o custo de retirar da fila é $\bigcirc(1)$. Uma possível modificação seria utilizar uma *heap*. Mudando o custo de inserção e de retirada para $\bigcirc(\log n)$.

A princípio tive problemas com o componente *Counter32khzC*, pois ele não existe para o *micaz*. Para poder compilar o escalonador tive de tirá-lo. Ele era usado para calcular a hora atual, e somar ao deadline. Sem esse componente, temos um escalonador de prioridades (mínimo).

5.4.3 Configuração de um Escalonador não Padrão

Para substituir o escalonador padrão é preciso colocar uma configuração com o nome *TinySchedulerC* no diretório da aplicação. Dentro desta configuração, amarra-se a interface *Scheduler* a implementação do escalonador. Por exemplo:

```
configuration TinySchedulerC
{
```

[1] Para mais informações sobre interfaces parametrizadas olhar o livro TinyOS Programming[3, s. 8.3 e 9].

```
    provides interface Scheduler;
    ...
}
implementation
{
    components SchedulerDeadlineP;
    ...

    Scheduler = SchedulerDeadlineP;
    ...
}
```

É preciso também criar a interface para o novo tipo de tarefa, com o comando *postTask* e o evento *runTask*. Por exemplo:

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

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

```
configuration TinySchedulerC
{
    provides interface Scheduler;
    provides interface TaskBasic[uint8_t id];
    provides interface TaskDeadline<TMilli>[uint8_t id];
}
implementation
{
    components SchedulerDeadlineP;
    ...

    Scheduler = SchedulerDeadlineP;
    TaskBasic = Sched;
    TaskDeadline = Sched;
}
```

Para que o escalonador funcione corretamente no simulador é preciso adicionar funções que lidam com eventos no *tossim*. Essas funções foram retiradas do arquivo *opt/tinyos-2.1.1/tos/lib/tossim/SimSchedulerBas*
Primeiro é preciso adicionar ao *Scheduler*:

```
  bool sim_scheduler_event_pending = FALSE;
  sim_event_t sim_scheduler_event;
```

```
int sim_config_task_latency() {return 100;}

void sim_scheduler_submit_event() {
  if (sim_scheduler_event_pending == FALSE) {
    sim_scheduler_event.time = sim_time() + sim_config_task_latency();
    sim_queue_insert(&sim_scheduler_event);
    sim_scheduler_event_pending = TRUE;
  }
}

void sim_scheduler_event_handle(sim_event_t* e) {
  sim_scheduler_event_pending = FALSE;
  if (call Scheduler.runNextTask()) {
    sim_scheduler_submit_event();
  }
}

void sim_scheduler_event_init(sim_event_t* e) {
  e->mote = sim_node();
  e->force = 0;
  e->data = NULL;
  e->handle = sim_scheduler_event_handle;
  e->cleanup = sim_queue_cleanup_none;
}
```

Depois, no *Scheduler.init()* adicione:

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

E por ultimo, no *Scheduler.postTask()*, caso a tarefa tenha sido colocada na fila, adicione:

```
sim_scheduler_submit_event();
```

5.4.4 Escalonador de Prioridades

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

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

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

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

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

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

5.4.5 Escalonador multi-nível

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

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

Sensor Sensoriamento de diferentes características do ambiente.

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

Básica Outros serviços, como por exemplo temporizador.

Portanto desenvolvemos um escalonador que divide as tarefas de acordo com os tipos definidos acima. Cada tipo de tarefa tem sua própria fila com política *First-in First-out*, e as filas mais importantes devem ser atendidas por completo para que as outras sejam antendidas. Definiu-se que a ordem de prioridade seria serial, radio, sensor e por ultimo básica.

5.5 Modelo de Threads do TinyOS

Por ser uma abstração de programação já implementada no TinyOS (opcional), foi feito um estudo neste modelo. Os conceitos já utilizados/implementados por threads que ajudarão na criação de uma interface do corotinas são: Chamadas bloqueantes ao sistema, re-aproveitamento dos serviços oferecidos, bloco de controle de threads (TCB), e a da troca de contexto.

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

A interface entre as threads a nível de usuário e o kernel é feito através de chamadas de sistema bloqueantes. Essas chamadas são implementadas aproveitando os serviços já disponíveis do TinyOS. São responsáveis por manter o estado do serviço *split-phase* que será usado, bloquear a thread que a invocou e acordar a thread do kernel.

Passam a existir três tipos de contextos de execução: tarefas, interrupções e threads. Tarefas e interrupções podem interromper threads de aplicação, mas não o contrário. Threads tem preempção entre elas, de modo que é necessário o uso de primitivas de sincronização. As opções fornecidas são *Mutex*, semáforos, barreiras, variáveis de condição, e contador bloqueante. Esta ultima foi desenvolvida especialmente para o TinyOS. Seu uso se dá de forma que a thread fica bloqueada até o contador atingir um número arbitrário, enquanto outras threads podem incrementar ou decrementar esta variável através de uma interface.

O TinyOS retoma o controle sobre a aplicação de dois modos diferentes. No primeiro, uma aplicação faz uma chamada de sistema que posta uma tarefa para processar o serviço. No segundo modo, um manipulador de interrupção posta uma tarefa. Porém, neste caso o TinyOS só acorda depois de terminada a execução da interrupção.

O escalonador de threads utiliza uma política *Round-Robin* com um tempo de 5 milisegundos. É ele que oferece toda a interface para manipulação de threads, como pausar, criar e destruir. É interessante nota que o escalonador não existe em um contexto de execução específico, seu contexto depende de quem utilizou sua interface.

As threads podem ser estáticas ou dinâmicas. A diferença é o momento de criação da pilha e do bloco de controle da thread. O primeiro tipo em tempo de compilação, o segundo em tempo de execução. O bloco de controle, também chamado de *Thread Control Block* (TCB) contém informações essenciais da thread, como seu identificador, seu estado de execução, o valor dos registradores (para troca de contexto), entre outras.

A troca de contexto é feita por códigos específicos para cada plataforma. É utilizada a linguagem assembly junto com C, para armazenar o valor dos registradores importantes na TCB. Exemplo do código utilizado para o chip *atm218*:

```
//Define on platform specific basis for inclusion in
// the thread control block
typedef struct thread_regs {
    uint8_t status;
    uint8_t r0;
    uint8_t r1;
    uint8_t r2;
    ...
}

//Save General Purpose Registers
#define SAVE_GPR(t)                                          \
  __asm__("mov %0,r0 \n\t" : "=r" ((t)->regs.r0) : );    \
  __asm__("mov %0,r1 \n\t" : "=r" ((t)->regs.r1) : );    \
  __asm__("mov %0,r2 \n\t" : "=r" ((t)->regs.r2) : );    \
```

5.6 Corotinas

Primeiramente foram revisados conceitos de corotinas em sistemas operacionais. Depois foi feita uma análise na implementação de corotinas para a antiga versão do TinyOS (1.0), feita por Silvana Rosseto em sua tese de doutorado[5].

6 Resultados Obtidos

Seguem os experimentos e seus respectivos resultados.

6.1 Experimentos com o Escalonador Padrão

Antes de começar a desenvolver outros escalonadores de tarefas, foi feito um experimento com o padrão que utiliza uma política *First in, First Out*. Para medir a complexidade na prática, foi desenvolvida uma aplicação de teste. Nela cada tarefa executa um loop de 65000 iterações, fazendo uma simples multiplicação em cada iteração. O número de tarefas variou entre 20, 50 e 100. Na tabela a seguir pode-se ver o tempo de execução em microsegundos:

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

6.2 Experimentos com o Escalonador de Prioridades

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

Dois tipos de estrutura de dados foram usadas para a organização das tarefas, uma fila comum e uma *heap*. E devido aos problemas de *starvation* como visto acima**??**, foram criadas políticas de envelhecimento.

Com isso, totalizou-se quatro diferentes versões do escalonador:

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

Analisando as estruturas de dados usadas em cada escalonador, criamos uma tabela com a complexidade de inserção e remoção para cada um:

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

Para medir a complexidade na prática, foi desenvolvida uma aplicação de teste. Nela cada tarefa executa um loop de 65000 iterações, fazendo uma simples multiplicação em cada iteração. O número de tarefas variou entre 20, 50 e 100. Na tabela a seguir pode-se ver o tempo de execução em microsegundos:

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

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

7 Conclusões

8 Relatório de Atividades

Referências

[1] Minicurso sobre TinyOS, Silvana Rosseto, http://www.dcc.ufrj.br/~silvana/curso-tinyos-ufes2010/

[2] TinyOS Documentation Wiki, http://docs.tinyos.net/index.php/Main_Page

[3] TinyOS Programming, Philip Levis, David Gay.

[4] Livro de Sistemas Operacionais, Carlos Maziero, http://www.ppgia.pucpr.br/~maziero/doku.php/so:livro_de_sistemas_operacionais

[5] Citar tese de doutorado!**********