1
Introdução

A necessidade de se privilegiar interações assíncronas aparece em vários cenários (Cardelli/99, Weiser/93, Kleinrock/96, Akyildiz/02). Os modelos de programação mais adotados para permitir esse tipo de interação — em particular, o modelo baseado em *threads* e o modelo dirigido a eventos — tipicamente refletem os mecanismos de comunicação não-bloqueante deixando a cargo do programador a tarefa de construir abstrações de nível mais alto e mais adequadas para cada tipo de problema. Neste trabalho aprofundamos a discussão sobre como oferecer uma interface de programação mais apropriada para o programador que precisa lidar com contextos assíncronos de execução, combinando operações de comunicação não-bloqueante com um modelo de gerência cooperativa de tarefas.

1.1
Motivação

Nas aplicações distribuídas a execução de um processo depende da interação com outros processos que normalmente executam em máquinas ou dispositivos distintos. Os processos interagem para trocar informações ou requisitar a execução de tarefas necessárias para a continuidade do processamento local. Uma interação é *assíncrona* quando ela não requer a sincronização das partes envolvidas. Se um processo faz uma requisição a outro processo, o processo requisitante não precisa ficar bloqueado até que a resposta esteja disponível: a resposta poderá ser recebida e tratada posteriormente. Enquanto isso, outras respostas, ou tarefas internas do processo podem ser executadas.

O modelo de programação *multithreading*, introduzido no escopo dos sistemas operacionais e posteriormente incorporado no desenvolvimento de aplicações, oferece um caminho para favorecer interações assíncronas. A aplicação é dividida em linhas de execução distintas (*threads*) que podem evoluir de forma independente. Em máquinas com mais de um processador, as *threads* podem executar em paralelo, cada uma em um processador distinto. Quando essa infra-estrutura não é oferecida, as *threads* compartilham o mesmo processador e a transferência de controle entre elas acontece auto-

PUC-Rio - Certificação Digital Nº 0210641/CA

maticamente, permitindo que todas as linhas de execução possam evoluir. A transferência automática do controle também garante o uso apropriado dos recursos disponíveis: quando uma *thread* deve esperar pela conclusão de uma operação bloqueante (como a recepção de uma mensagem), outra *thread* é escalonada para executar. Assim, é possível sobrepor a execução do algoritmo principal de uma aplicação com as operações que requerem acesso aos dispositivos externos (como a rede de comunicação). Uma das principais vantagens desse modelo, quando comparado à arquitetura dirigida a eventos, é que cada tarefa pode ser inteiramente codificada dentro de um contexto de execução cujo estado é implicitamente mantido, ou seja, não é necessário lidar explicitamente com a continuação de uma tarefa ao longo de transações distintas da aplicação. Por outro lado, embora oferecendo uma interface de programação mais conveniente para o programador, a transferência automática do controle entre as *threads* exige o uso de mecanismos adicionais para proteger o acesso aos recursos compartilhados. Em determinados contextos, a sobrecarga computacional associada às trocas de contextos e a consequente necessidade de sincronização na execução das *threads* torna-se excessivamente custosa e complexa de codificar.

Na arquitetura dirigida a eventos o estilo de programação baseado nos próprios eventos de comunicação favorece naturalmente as interações assíncronas. As aplicações são modeladas como trechos de código distintos — os tratadores de eventos — executados em resposta à ocorrência de eventos. Um único contexto de execução é mantido, o qual tipicamente permanece em um laço infinito processando eventos de diferentes tipos. Quando cada tarefa de uma aplicação pode ser inteiramente codificada como a resposta à ocorrência de um evento, o desenvolvimento da aplicação é simplificado. Entretanto, quando o tratamento de uma tarefa envolve a ocorrência de mais de um evento, essa tarefa precisa ser codificada como uma máquina de estados finitos, onde a transição entre os estados é disparada pela ocorrência dos eventos. A consequência direta dessa abordagem é a necessidade de preservar o estado de continuação de cada tarefa entre cada transição. Outra dificuldade é que, para manter a interatividade do sistema, considerando a infra-estrutura básica com um único contexto de execução, os tratadores de eventos não podem ser bloqueantes: quando o código de resposta a um evento precisa executar uma operação bloqueante (como a recepção de uma mensagem), um tratamento especial deve ser adotado. Normalmente o código do tratador do evento é particionado em duas partes: a primeira faz a requisição para a operação bloqueante; e a segunda implementa a continuação do tratador original através da implementação de outro evento sinalizado quando a resposta da operação

PUC-Rio - Certificação Digital Nº 0210641/CA

estiver disponível. Uma vez que, de modo geral, as linguagens de programação não oferecem mecanismos adequados para salvar e restaurar o estado de parte de uma aplicação, cabe ao programador preservar o contexto necessário para que as tarefas da aplicação sejam apropriadamente executadas.

As vantagens e dificuldades características dos modelos baseado em *threads* e dirigido a eventos são discutidas em alguns trabalhos (Behren/03, Ousterhout/96). Normalmente, o modelo mais adequado para um determinado problema exige que o programador se ocupe diretamente com as desvantagens inerentes a esse modelo, tornando a tarefa de desenvolvimento de aplicações mais complexa. Em alguns contextos, uma alternativa apropriada é a combinação dos modelos. A arquitetura SEDA (Welsh/01), por exemplo, adota essa alternativa, combinando o uso de *threads* com o modelo de execução dirigido a eventos para construir uma plataforma de gerência de serviços na Internet.

1.2 Objetivo

O objetivo desse trabalho é aprofundar o estudo sobre modelos de programação distribuída que permitam simplificar a tarefa do desenvolvedor de aplicações em contextos que requerem interações assíncronas entre processos. Investigamos modelos que combinam o estilo de comunicação baseado em eventos com uma arquitetura multi-tarefa com gerência cooperativa. Em particular, exploramos os modelos que usam *co-rotinas* (Knuth/97) como estrutura de controle para a gerência de tarefas, com o objetivo principal de manter contextos de execução distintos com um custo computacional baixo.

Co-rotina é uma construção de programação que representa uma linha de execução distinta, com sua própria pilha. Diferente das *threads* que podem executar simultaneamente, as co-rotinas são colaborativas: um programa com co-rotinas executa apenas uma co-rotina em cada momento e a transferência de controle entre elas aparece explicitamente no algoritmo da aplicação. O nosso objetivo é prover uma interface de programação mais conveniente para o programador, usando uma infra-estrutura de multi-tarefa mais leve que o modelo tradicional da programação *multithreading*.

1.3 Contribuições

A contribuição principal deste trabalho consiste em investigar uma forma de combinar operações que encapsulam primitivas de comunicação não-bloqueante com um mecanismo de gerência cooperativa de tarefas em cenários

PUC-Rio - Certificação Digital Nº 0210641/CA

distintos da computação distribuída. As operações propostas permitem que o programador explore diferentes paradigmas de programação dentro de uma mesma aplicação, com a finalidade de modelar da forma mais apropriada a solução para problemas específicos. Com o mecanismo de gerência cooperativa de tarefas, partes distintas — e naturalmente sequenciais — de uma aplicação podem ser inteiramente codificadas dentro de um único contexto de execução, simplificando a tarefa de programação sem o custo excessivo da gerência preemptiva de tarefas.

Para conduzir nossa investigação, escolhemos dois casos particulares nos quais a necessidade de assincronismo requer a construção de abstrações de programação adequadas para facilitar o desenvolvimento de aplicações: a migração da computação paralela de ambientes restritos (como *clusters*) para as redes geográficas (Foster/01); e os sistemas projetados para executar em dispositivos que formam as redes de sensores (Culler/04, Loureiro/03). No primeiro caso, tratamos do assincronismo na interação entre processos que executam em máquinas distintas e participam da realização de uma atividade global. No segundo caso, tratamos do assincronismo necessário dentro de um sistema que precisa interagir simultaneamente com vários dispositivos conectados diretamente com o ambiente físico. A opção por dois cenários completamente distintos tem por objetivo mostrar que a combinação que estamos investigando para facilitar o desenvolvimento de aplicações pode servir como base para diferentes contextos que requerem interações assíncronas.

A computação paralela trata de problemas que requerem computação intensiva e longa, como por exemplo, o processamento de imagens codificadas em matrizes de pixels. Para reduzir o tempo de processamento requerido e otimizar o desempenho das aplicações, a solução para um problema é tipicamente construída a partir de sub-tarefas que podem ser executadas em paralelo, usando processadores distintos. Em boa parte das aplicações, as sub-tarefas executam o mesmo algoritmo sobre uma parte distinta do conjunto de dados (*Simple Program Multiple Data - SPMD*). Normalmente, um processo *mestre* é responsável por distribuir as sub-tarefas para um conjunto de processos *trabalhadores*, coletar os resultados processados e concluir a aplicação. O desenvolvimento de aplicações paralelas usando a infra-estrutura das redes geográficas permite aproveitar recursos disponíveis em diferentes domínios e instituições. Mas para isso, o desenvolvedor de aplicações paralelas precisa atenuar o foco em desempenho para tratar questões que em contextos mais restritos, como é o caso dos *clusters*, poderiam ser ignoradas. Nas redes geográficas, o efeito do retardo da comunicação e da intermitência das conexões, além de criar contextos mais dinâmicos, requer o uso de mecanismos

PUC-Rio - Certificação Digital Nº 0210641/CA

que favoreçam a sobreposição entre a computação em cada processo e a comunicação entre eles para que os custos com a comunicação sejam reduzidos. Esse cenário cria a necessidade de investigar abstrações de programação distribuída que se adequem ao modelo de programação frequentemente usado na computação paralela.

No capítulo 3, descrevemos a implementação de um ambiente de computação para facilitar o desenvolvimento de aplicações paralelas que aproveitam os recursos disponíveis em máquinas de diferentes domínios (Rodriguez/05). Esse ambiente, denominado LuaRPC, combina comunicação assíncrona com multi-tarefa cooperativa provida através da construção de co-rotinas. Descrevemos o conjunto de operações oferecidas pelo ambiente LuaRPC e as possibilidades distintas de interação entre os processos de uma aplicação providas por essas operações. O programador pode explorar, simultaneamente, o estilo ativo do modelo de chamadas remotas, ou o estilo reativo da notificação de eventos. Exemplificamos como esses estilos de programação favorecem a construção de aplicações com o comportamento mestre/trabalhador — característico das aplicações paralelas — dentro de contextos mais dinâmicos, como é o caso das redes geográficas.

Nas redes de sensores, os sistemas projetados para executar em cada nó da rede precisam ser capazes de atender várias interações simultâneas com o mundo físico, como sensoriamento, comunicação via rádio e roteamento de mensagens. Os nós são acoplados ao mundo físico com a finalidade de medir, processar e transmitir informações sobre fenômenos ambientais, tais como temperatura, umidade e luminosidade. Para permitir esse acoplamento, os nós são dispositivos de tamanho reduzido e apresentam severas restrições de recursos computacionais, como capacidade de processamento e espaço de memória. Em consequência, os sistemas projetados para as redes de sensores devem ser capazes de lidar apropriadamente com essas restrições.

O TinyOS (Hill/00) é um dos sistemas operacionais mais adotados na pesquisa com redes de sensores. A arquitetura dirigida a eventos definida pelo TinyOS prioriza fortemente o tratamento das restrições características do ambiente dos sensores, mas oferece uma interface de programação difícil de usar. Nas aplicações TinyOS, as tarefas que requerem interação com dispositivos externos (incluindo a rede de comunicação) são tipicamente iniciadas através de uma requisição e o seu término é indicado posteriormente com a sinalização de um evento. Essa abordagem de construir operações em duas fases, típica da arquitetura dirigida a eventos, é o fator principal que torna complexo o desenvolvimento das aplicações TinyOS. O modelo de programação *multithreading* (com linhas de execução distintas que podem ser automaticamente suspensas

PUC-Rio - Certificação Digital Nº 0210641/CA

e retomadas) seria uma alternativa para lidar com essa dificuldade, mas ele não é adotado pelo TinyOS em razão do seu custo computacional.

No capítulo 4, mostramos que é possível oferecer um caminho intermediário para o desenvolvimento de aplicações usando o TinyOS — sem o custo excessivo do uso de *threads* e com uma interface de programação mais simples — usando uma infra-estrutura adicional que permite multi-tarefa cooperativa através de co-rotinas (Rossetto/06). Com base nessa infra-estrutura, é possível oferecer ao programador a visão síncrona das tarefas naturalmente sequenciais sem perder o assincronismo das interações. Para avaliar a carga computacional adicionada com o uso de co-rotinas dentro da arquitetura do TinyOS, realizamos simulações que comparam a versão original do sistema com a versão que oferece uma interface de programação mais simples. Os resultados obtidos mostram que o custo computacional adicionado está dentro de um limite razoável, considerando a comodidade oferecida para o programador. Com a nova interface de programação, o programador pode optar pelo estilo de programação mais adequado para cada problema. Tarefas que refletem o estilo reativo da programação dirigida a eventos podem continuar sendo modeladas como na arquitetura original do TinyOS. Nos demais casos, quando é preciso codificar uma tarefa que reflete um comportamento tipicamente sequencial, mas que requer a interação com dispositivos externos, a visão síncrona provida com a manutenção de contextos distintos de execução pode ser explorada.

1.4 Organização do texto

O restante deste texto está estruturado da seguinte forma: Na primeira parte do capítulo 2, discutimos como os modelos baseado em *threads* e dirigido a eventos permitem interações assíncronas entre os processos de uma aplicação, e quais as vantagens e dificuldades encontradas em cada modelo. Na segunda parte do capítulo 2, discutimos nossa proposta para combinar as características principais desses modelos com o objetivo de oferecer uma interface de programação mais apropriada para o programador. No capítulo 3, descrevemos uma implementação dessa proposta através da construção de um conjunto de operações que permitem a interação entre processos distribuídos em uma rede. No capítulo 4, descrevemos a mesma proposta implementada dentro do sistema operacional TinyOS, com a finalidade de prover uma interface de programação mais fácil de usar e ainda compatível com as restrições particulares do ambiente de redes de sensores. No capítulo 5, apresentamos as conclusões do nosso trabalho e apontamos direções de pesquisa geradas por ele.

PUC-Rio - Certificação Digital Nº 0210641/CA