2
Interação assíncrona entre processos

A interação assíncrona entre processos de uma aplicação tem sido adotada para lidar com a possível variação e imprevisibilidade do tempo necessário para completar uma interação com um processo ou dispositivo remoto. O tempo de resposta de uma interação pode depender de fatores distintos, entre eles: (i) *a disponibilidade dos processos e dispositivos*: os processos podem não estar simultaneamente ativos e prontos para tratar uma requisição; (ii) *a infra-estrutura de comunicação*: no caso típico dos processos comunicarem-se através de uma rede de computadores, o tempo de comunicação é normalmente difícil de prever e pode variar de uma interação para outra; (iii) *o tipo de tarefa requisitada*: o tempo de processamento necessário para a execução da tarefa requisitada varia de acordo com o tipo da tarefa.

Esses fatores, isoladamente ou somados, podem ocasionar um tempo de espera impraticável para determinadas aplicações por duas razões principais: (i) os recursos computacionais alocados para o processo são sub-utilizados durante o tempo de espera (em contextos com limitações de recursos essa é uma restrição crucial); (ii) o processo deixa de atender requisições externas que poderiam ser tratadas independentemente. O estilo de *interação assíncrona* entre processos permite tratar essas dificuldades. Enquanto um processo espera por informações de outro processo ele pode otimizar o uso dos recursos computacionais disponíveis sobrepondo a computação local com a comunicação e a computação remota dos demais processos. Neste capítulo, aprofundamos a discussão sobre a importância de privilegiar esse estilo de interação entre processos de uma aplicação distribuída e, em especial, como implementá-lo sem sobrecarregar a tarefa do programador.

2.1
Arquiteturas que permitem interação assíncrona entre processos

De modo geral, os modelos de programação permitem interação assíncrona entre processos de duas formas básicas: através de uma infra-estrutura multi-tarefa capaz de manter linhas de execução distintas dentro de um mesmo processo; ou através de uma infra-estrutura mono-tarefa que

reflete diretamente o assincronismo da troca de mensagens não-bloqueante. Nessa seção discutimos as vantagens e dificuldades encontradas nessas abordagens e na seção seguinte apresentamos o modelo que propomos para combinar as características principais de cada abordagem de forma a oferecer uma interface de programação mais conveniente para o programador.

2.1.1 Multi-tarefa com gerência preemptiva de tarefas

A arquitetura multi-tarefa surgiu no contexto dos sistemas operacionais e foi posteriormente incorporada no escopo das linguagens de programação. Nos sistemas operacionais, o assincronismo foi introduzido por dispositivos físicos independentes. Originalmente, para executar operações de entrada e saída (I/O) um processo enviava uma requisição para o dispositivo e então permanecia em espera ocupada, testando continuamente a variável que deveria ser modificada pelo dispositivo quando a operação estivesse concluída. Para aproveitar o tempo do processador durante essa espera, desenvolveu-se a idéia de sobrepor as operações de I/O com a computação executada no processador. Para isso, as operações de I/O passaram a ser dirigidas a interrupção, eliminando a necessidade de espera ocupada, e vários processos passaram a residir na memória ao mesmo tempo, permitindo o reaproveitamento do processador durante a espera pelas operações de I/O.

Com a nova abordagem, sempre que um processo faz uma requisição de I/O, o controle é transferido para o sistema operacional que envia a requisição para o dispositivo apropriado. O processo que estava executando permanece suspenso e o sistema operacional elege outro processo para executar. Quando o dispositivo termina a execução da operação requisitada, uma interrupção é gerada e então o tratador dessa interrupção é executado. Quando o tratador termina, o controle volta para o sistema operacional que novamente elege algum dos processos "prontos" para transferir o controle.

Como as interrupções geradas pelos dispositivos físicos podem ocorrer a qualquer tempo, inclusive quando o próprio sistema operacional está usando o processador, os tratadores de interrupção e o algoritmo principal do sistema operacional se transformaram em linhas de controle distintas. Uma consequência imediata, ao permitir contextos de execução distintos que acessam recursos comuns, foi a necessidade de *sincronização* (Andrews/83). Se uma interrupção ocorre enquanto o sistema operacional está modificando uma estrutura de dados que também é acessada pelo tratador da interrupção, essa estrutura poderá ser vista em um estado inconsistente pelo tratador. Esse é um exemplo de um problema conhecido como *condição de corrida*:

o comportamento do sistema depende de qual processo acessa primeiro uma determinada variável. Para garantir um comportamento correto, foi preciso incluir no código do sistema operacional mecanismos para sincronizar o acesso a todas as estruturas compartilhadas.

Com o desenvolvimento dos servidores e das aplicações interativas, surgiu a necessidade de permitir uma relação mais interativa entre esses tipos de programas e os seus usuários. Isso fez os sistemas operacionais introduzirem a noção de *preempção*. Ao invés de transferir o controle entre os programas apenas quando eles eram bloqueados esperando por operações de I/O, a transferência passou a ser marcada para acontecer em intervalos de tempo pré-definidos. Dessa forma, todos os processos podiam avançar, evitando que computações longas adiassem a execução de outras computações por um tempo indeterminado.

Com a introdução de mecanismos para o compartilhamento de dados ou outras formas de comunicação entre processos, permitiu-se que programas distintos cooperassem para a execução de uma atividade global, fazendo a concorrência entre linhas de execução distintas chegar no nível de desenvolvimento de aplicações. Dividir uma aplicação em linhas de execução distintas, deixando a troca do controle entre elas a cargo do sistema operacional, é uma alternativa para permitir o assincronismo no nível do desenvolvimento de aplicações. Sempre que uma linha de execução precisa ser bloqueada para esperar por um processamento ou informação externa, outra linha de execução é automaticamente escalonada para o processador, garantindo o aproveitamento eficiente dos recursos disponíveis. Com o mecanismo de preempção, a distribuição automática do tempo do processador garante que todas as linhas de execução possam evoluir, um aspecto particularmente importante em sistemas interativos.

Entretanto, para manter diferentes contextos de execução dentro de uma aplicação é preciso lidar com algumas dificuldades, como por exemplo, a sincronização do acesso aos recursos compartilhados. Enquanto no nível do sistema operacional o tratamento de questões como essa pode ser encapsulado no código do sistema, no nível do desenvolvimento de aplicações cabe ao programador lidar com essas questões diretamente. Devido à possibilidade de problemas como *condições de corrida*, a depuração das aplicações torna-se mais complexa. Em alguns ambientes, a estrutura necessária para manter diferentes contextos de execução dentro do mesmo sistema é excessivamente custosa (Hill/00). Ousterhout (Ousterhout/96) defende o argumento de que a programação *multithreading* (multi-tarefa com gerência preemptiva) é uma alternativa apropriada apenas quando a concorrência real é possível, ou seja,

em máquinas com múltiplos processadores.

2.1.2 Mono-tarefa com gerência serial de tarefas

Uma alternativa para permitir interações assíncronas entre processos sem a necessidade de lidar com as dificuldades da programação *multithreading* foi introduzida através da arquitetura mono-tarefa com tarefas não-bloqueantes, normalmente referenciada como arquitetura *dirigida a eventos*: a recepção de uma mensagem (ou evento) dispara a execução de um *tratador de evento* que é executado até terminar (Schmidt/95). Nesse caso, as tarefas da aplicação são executadas de forma serial, uma após a outra. Sem a possibilidade de preempção das tarefas, não é preciso usar mecanismos de proteção dos recursos compartilhados. Por outro lado, para garantir o assincronismo, as tarefas não podem ser bloqueantes: quando um processo precisa esperar pela resposta de outro processo a tarefa corrente deve ser particionada em duas tarefas distintas: a primeira executa até a requisição; e a segunda continua a execução a partir da resposta recebida e é disparada quando a mensagem (ou evento) de resposta torna-se disponível (Pyarali/97).

A principal dificuldade com a arquitetura dirigida a eventos é a necessidade de particionar a implementação de tarefas que requerem a interação com outros processos. Isso significa que mesmo as tarefas naturalmente sequenciais não podem ser implementadas dentro de um único procedimento da linguagem, se elas precisam esperar por uma informação de outro processo. De modo geral, sempre que isso ocorre, cabe ao programador se ocupar diretamente da manutenção do fluxo de execução para que a tarefa seja devidamente processada (Kasten/05). Tipicamente, as tarefas fazem requisições e registram uma *continuação* para ser executada posteriormente (Welsh/01). Esse processo, referenciado por Adya et al. como *stack ripping* (Adya/02), é uma das principais dificuldades no desenvolvimento de aplicações usando a arquitetura dirigida a eventos (Behren/03). A programação puramente reativa, como resposta à ocorrência de eventos, resulta em programas estruturados como máquinas de estados, normalmente mais complexos de entender e manter do que os programas sequenciais.

De acordo com Vinoski (Vinoski/05), a complexidade adicionada com as interações assíncronas é consequência do fato de misturar o estilo de comunicação no nível do tratamento de mensagens com a semântica da comunicação no nível da aplicação. As abstrações de programação devem ocultar questões de baixo nível, como detalhes da rede e da troca de mensagens, por trás de um idioma de programação mais familiar para o programador. Para

o desenvolvedor de aplicações, sempre que é necessário obter uma informação antes de decidir qual deve ser o próximo passo na execução de uma tarefa, a visão síncrona da interação é imprescindível. Lea et al. (Lea/06) enfatizam que todos os sistemas de computação são essencialmente assíncronos e que a noção de operações síncronas representa puramente um estilo de programação mais conveniente, que deve ser construído sobre uma base assíncrona de comunicação. A dificuldade está em como oferecer apropriadamente essa aparência de sincronismo.

2.2 Combinando multi-tarefa cooperativa com comunicação baseada em eventos

Diferentes trabalhos propõem combinar as vantagens da arquitetura multi-tarefa: manutenção de contextos distintos de execução que podem ser suspensos e retomados apropriadamente; com as vantagens da arquitetura dirigida a eventos: possibilidade de definir tratadores de eventos que são implicitamente acionados quando uma informação esperada torna-se disponível, ou uma requisição externa é recebida. Um exemplo particular é o modelo adotado pelos sistemas operacionais tradicionais para tratar a interação com os dispositivos de entrada e saída (Schimidt/96). Neste trabalho aprofundamos a discussão sobre como implementar essa combinação de forma a garantir interação assíncrona entre processos oferecendo uma interface de programação familiar para o programador, sem um alto custo computacional.

A idéia central consiste em permitir que as aplicações sejam modularizadas usando os procedimentos da linguagem que serão executados em contextos distintos de execução — estendendo a noção de modularidade de uma aplicação convencional dividida em procedimentos. Como em um programa sequencial, apenas uma tarefa permanece ativa em cada instante de tempo, preservando o aspecto de cooperação entre elas. A independência entre as tarefas favorece a otimização do uso dos recursos computacionais: quando a execução de uma tarefa é suspensa, outra tarefa ("pronta para executar") pode receber o controle do processamento.

A transferência de controle entre as diferentes linhas de execução é gerenciada dentro da própria aplicação, baseado em um modelo de **gerência cooperativa de tarefas**. Isso reduz o custo computacional associado com a gerência preemptiva (típica do modelo *multithreading*) uma vez que o acesso aos recursos compartilhados é protegido sem a necessidade de mecanismos adicionais de sincronização. Além disso, é possível evitar trocas de contextos desnecessárias quando a aplicação não exige rotatividade entre as tarefas.

Cada tarefa engloba instruções que são executadas sequencialmente, i.e., cada instrução só é executada quando a instrução anterior foi concluída. Quando uma instrução envolve a interação com outros processos e a espera por informações externas, a tarefa corrente é implicitamente suspensa, permitindo que os recursos computacionais sejam usados por outras tarefas ou economizados. Quando a mensagem ou evento esperado por uma tarefa suspensa é recebido, o tratador da mensagem (implicitamente gerado quando a tarefa foi suspensa) retoma a execução dessa tarefa que continua a partir do ponto de suspensão. Com essa infra-estrutura, tarefas naturalmente sequenciais podem ser inteiramente codificadas dentro de um único procedimento da linguagem (tratador da mensagem), mesmo quando a tarefa envolve chamadas bloqueantes que requerem a interação com outros processos. A tarefa principal da aplicação codifica um laço responsável pelo tratamento de mensagens ou eventos recebidos de outros processos, como na arquitetura dirigida a eventos. Os tratadores das mensagens são executados em contextos distintos de execução, garantindo assim a possibilidade de serem suspensos e retomados como as demais tarefas da aplicação.

A arquitetura que propomos baseia-se, então, em uma arquitetura multitarefa com gerência cooperativa combinada com um mecanismo de comunicação assíncrona; e constitui-se de um **escalonador de tarefas** — responsável por receber os eventos de comunicação e transferir o controle da tarefa principal para as demais tarefas da aplicação; e de um conjunto de **operações de interação** entre processos — responsáveis por devolver o controle para a tarefa principal quando uma chamada bloqueante é executada, oferecendo uma interface de programação mais conveniente para o programador. Nas próximas sub-seções descrevemos com detalhe cada um desses elementos.

2.2.1 Gerência cooperativa de tarefas

A gerência cooperativa entre linhas de controle distintas oferece uma alternativa às dificuldades encontradas tanto com a gerência preemptiva, da programação *multithreading*, como com a gerência serial, da programação dirigida a eventos (Adya/02, Behren/03). A aplicação pode manter diferentes linhas de execução e a transferência de controle entre elas é explicitamente embutida dentro do próprio algoritmo da aplicação. Dessa forma, o esforço de programação pode ser simplificado: as transferências de controle ocorrerão em pontos definidos pela aplicação, eliminando a necessidade de usar mecanismos adicionais para tratar o acesso aos recursos compartilhados; e as tarefas sequencias podem ser codificadas dentro de um único procedimento da lin-

guagem, uma vez que é possível suspender e retomar a execução dessas tarefas sempre que uma operação bloqueante precisa ser executada.

A abordagem cooperativa é adequada quando o controle do processador precisa ser alternado para evitar a espera ocupada por informações de outros processos — garantindo interações assíncronas — e o paralelismo real não é crucial para um bom desempenho da aplicação. Em máquinas com um único processador, a gerência cooperativa pode trazer ganhos reais no tempo de processamento total de uma aplicação quando comparada com a gerência preemptiva. Isso se deve ao fato de que as transferências de controle ocorrerão apenas nos pontos necessários para garantir que a aplicação não fique bloqueada, ao contrário da gerência preemptiva, que adicionalmente estabelece intervalos de tempo para a transferência de controle entre as linhas de execução.

Estrutura de controle para multi-tarefa cooperativa

Para permitir gerência cooperativa é necessário dispor de uma estrutura de controle que permita manter, no escopo da aplicação, contextos distintos de execução que podem ser suspensos e retomados apropriadamente. Uma alternativa é o uso de *continuações*. Uma *continuação* (Hieb/90) é uma abstração de programação que captura o contexto no qual a execução deve continuar posteriormente, ou seja, representa o "resto de uma computação". Em seu trabalho de tese, Lima (Lima/01) experimentou o uso de continuações, implementadas como valores de primeira classe, para sincronizar chamadas remotas em um sistema distribuído dirigido a eventos. Uma dificuldade com continuações, comprovada no trabalho de Lima, é a complexidade de implementá-las de forma eficiente devido ao suporte necessário para permitir múltiplas invocações de uma mesma continuação. Quando uma continuação é restaurada, o estado do programa anterior é perdido, a menos que uma nova continuação seja explicitamente criada para armazená-lo.

Outra alternativa para permitir gerência cooperativa de tarefas são as *co-rotinas*. Co-rotina é uma construção de programação que permite manter linhas de execução distintas dentro de um mesmo processo e especificar a troca de controle entre elas dentro da aplicação, através de chamadas de funções (Knuth/97, Andrews/83). A transferência de controle entre co-rotinas é mais rápida do que a transferência de controle entre processos ou *threads*, uma vez que não é necessário envolver chamadas ao sistema operacional nessa operação. Por isso co-rotinas são uma opção para permitir multi-tarefa com custo computacional reduzido.

Uma co-rotina pode ter vários pontos de suspensão, controlados pela própria co-rotina, e é capaz de manter informações sobre seu próprio estado

entre cada ativação. A principal diferença entre co-rotinas e continuações é que uma continuação é uma constante — não se modifica depois de criada — enquanto uma co-rotina se modifica sempre que é executada (Scott/06). Da mesma forma que uma *thread* (usada na arquitetura multi-tarefa com preempção), cada co-rotina tem sua própria pilha, variáveis locais e um ponteiro de instrução, e também compartilha variáveis globais com outras co-rotinas. A principal diferença é que *threads* podem executar simultaneamente enquanto co-rotinas são colaborativas. Um programa com co-rotinas executa apenas uma co-rotina em cada momento, mesmo em um sistema multi-processador, e a execução de uma co-rotina só é suspensa quando a própria co-rotina faz essa solicitação.

O conceito de co-rotinas apareceu como uma construção de linguagem de programação nos anos 60 com Simula (Birtwistle/75), mas não é uma construção comum nas linguagens de programação. Como discutido em (Moura/04), o motivo para essa ausência foi a falta de uma visão uniforme do conceito, e a complexidade das primeiras implementações. Recentemente, entretanto, o interesse por co-rotinas tem reaparecido nas aplicações multi-tarefa (Behren/03), em linguagens de *scripting* (Conway/00, Moura/04), e também no contexto de adaptação dinâmica de sistemas distribuídos (Maia/05). Na adaptação dinâmica, a gerência cooperativa de tarefas implementada por meio de co-rotinas garante a atomicidade das operações de adaptação sem a necessidade de mecanismos adicionais de sincronização.

Co-rotinas podem ser classificadas de acordo com o mecanismo de transferência de controle como *simétricas* ou *assimétricas*. Co-rotinas simétricas implementam uma única operação de transferência de controle que ativa ou reativa a co-rotina passada como argumento. Co-rotinas assimétricas, ao contrário, implementam uma operação para invocar uma co-rotina e outra para suspendê-la, devolvendo o controle implicitamente para a co-rotina anterior. Co-rotinas podem ainda ser implementadas como valores de primeira classe, i.e., podem ser atribuídas a variáveis, passadas como argumento para funções ou devolvidas como valores de saída. Além disso, diferentes implementações de co-rotinas podem ou não permitir que uma co-rotina seja suspensa e retomada dentro de métodos aninhados. O termo *stackfull* é usado para caracterizar implementações de co-rotinas que oferecem a possibilidade de transferência de controle dentro de métodos aninhados (Moura/04). Na sua pesquisa de tese, Moura (Moura/04) mostra que co-rotinas completas (implementadas como valores de primeira classe e *stackfull*) e assimétricas são capazes de substituir *continuações* e *threads* e então oferecer uma alternativa aos modelos tradicionais de programação distribuída.

```
void do_something() {
    /* loop com ponto flutuante */
          ...
}

int main() {
    /* armazena o tempo inicial */
    /* invoca o procedimento auxiliar */
    do_something();
    /* loop com ponto flutuante */
            ...
    /* armazena o tempo final */
}
```

Figura 2.1: Aplicação usando chamada de função.

Para avaliar o desempenho de co-rotinas como estrutura de controle para multi-tarefa realizamos alguns experimentos que comparam o desempenho de co-rotinas e de *threads*, ambas implementadas na linguagem C. Projetamos uma aplicação simples que executa duas sequências de operações idênticas: a primeira dentro do procedimento principal da aplicação e a segunda como um procedimento auxiliar. Essa sequência de operações consiste de um laço com operaç oes de ponto flutuante. Implementamos três versões distintas da aplicação: (1) na versão base, o procedimento auxiliar é invocado através de uma chamada de função normal da linguagem (estrutura mono-tarefa); (2) na versão com *thread*, o procedimento auxiliar é invocado como uma nova *thread* de execução; (3) na versão com co-rotina, o procedimento auxiliar é invocado como uma nova co-rotina.

O objetivo principal do experimento foi avaliar o tempo necessário para criar e transferir o controle entre as linhas de execução dentro de uma mesma aplicação. Uma vez que a computação executada em cada procedimento não varia entre as versões, a diferença no tempo total gasto em cada execução representa o custo associado com a criação e as trocas de contexto. A figura 2.1 mostra a estrutura principal da versão base da aplicação.

Na versão que usa *threads* adotamos o mecanismo de *joining* para garantir a transferência de controle imediata do procedimento principal da aplicação para o procedimento auxiliar. Nessa versão temos uma transferência de controle do programa principal para a *thread* e outra da *thread* para o programa principal. Usamos a biblioteca *Pthread* — uma implementação tradicional de *threads* disponível no sistema operacional Linux. A figura 2.2 mostra a estrutura principal da aplicação usando *threads*.

Como a linguagem C não implementa co-rotinas, usamos a biblioteca PCL (*Portable Coroutine Library*) (Libenzi/05) — uma implementação de co-

```
void *do_something(void *data) {
    /* loop com ponto flutuante */
            ...
    pthread_exit(NULL);
}

int main() {
    pthread_t tid;
    pthread_attr_t attr;
    pthread_attr_init(&attr);
    pthread_attr_setdetachstate(&attr, PTHREAD_CREATE_JOINABLE);

    /* armazena o tempo inicial */
    /* cria a thread para executar o procedimento auxiliar */
    pthread_create(&tid, &attr, do_something, NULL);
    /* loop com ponto flutuante */
            ...
    /* espera o final da thread */
    pthread_join(tid, NULL);
    /* armazena o tempo final */
    /* finaliza */
    pthread_attr_destroy(&attr);
    pthread_exit(NULL);
}
```

Figura 2.2: Aplicação usando *threads*.

rotinas assimétricas em C, baseada na biblioteca GNU-Pth (*GNU Portable Threads*) (Engelschall/05). PCL usa as funções `setjmp` e `longjmp`, definidas pelo padrão ANSI-C, para implementar as operações de co-rotinas. A função `setjmp` salva o contexto de execução corrente, incluindo o ponteiro da pilha, em uma estrutura de dados pré-definida que será depois usada como argumento para a função `longjmp`. A função `longjmp`, por sua vez, retoma o contexto de execução previamente salvo pela função `setjmp` (Engelschall/00). As principais operações implementadas pela biblioteca PCL incluem:

- `coroutine_t co_create(void *func, void *data, void *stack, int stack-size)`: Cria uma nova co-rotina. `func` é o ponteiro de entrada para a função que será chamada com o argumento `data`. A base para a pilha da co-rotina é definida pelo parâmetro `stack` e o tamanho da pilha pela parâmetro `stack-size`. Quando a função `func` termina, a co-rotina é eliminada.
- `void co_call(coroutine_t co)`: Transfere a execução para a co-rotina especificada como argumento. Na primeira vez que a co-rotina é executada, o ponteiro de entrada da sua função é chamado, e o parâmetro `data` (usado na criação da co-rotina) é passado para a função. A co-rotina corrente é suspensa até que outra co-rotina devolva o controle pra ela.

```
void do_something() {
    /* loop com ponto flutuante */
          ...
    co_resume(); // << co_resume(); co_resume(); co_resume(); >>
}

int main() {
    coroutine_t coro;

    /* armazena o tempo inicial */
    /* cria a co-rotina para executar o procedimento auxiliar */
    coro = co_create(do_something, NULL, NULL, STACK_SIZE);
    /* loop com ponto flutuante */
            ...
    /* transfere o controle para a co-rotina */
    co_call(coro);  // << co_call(coro); co_call(coro); co_call(coro); >>
    /* finaliza */
    co_delete(coro);
    /* armazena o tempo final */
}
```

Figura 2.3: Aplicação usando co-rotinas.

– `void co_resume(void)`: Devolve o controle da execução para a co-rotina que transferiu o controle para a co-rotina corrente.

Na versão que usa co-rotinas experimentamos com mais de duas transferências de controle: dentro do procedimento principal e do procedimento auxiliar variamos o número de transferências de uma a quatro. A figura 2.3 mostra a estrutura principal da aplicação usando co-rotinas.

Para cada versão da aplicação repetimos a execução 50 vezes. Usamos uma máquina Pentium-IV/1.70GHz configurada com a versão 2.4.20 do Linux e a versão 3.2.2 do GCC. O gráfico apresentado na figura 2.4 confirma o bom desempenho da troca de contexto entre co-rotinas comparado ao desempenho usando *threads*. A versão base da aplicação gastou um tempo médio de $34,3$ μsegundos. A versão com co-rotinas com duas trocas de contexto (uma chamada *call* e uma chamada *resume*) gastou um tempo médio de $38,1$ μsegundos, enquanto a versão com *threads*, também com duas trocas de contexto, gastou um tempo médio de 45.3 μsegundos. Nos demais experimentos, aumentamos o número de trocas de contexto entre as co-rotinas. Com três transferências (*call/resume*), co-rotinas ainda apresentam um desempenho melhor que *threads*, com um tempo médio de $44,5$ μsegundos.

Na arquitetura que queremos experimentar, usamos co-rotinas para permitir interação assíncrona entre os processos de uma aplicação permitindo multi-tarefa com gerência cooperativa de tarefas. Usando co-rotinas reduzimos o custo computacional associado ao modelo tradicional de multi-tarefa com

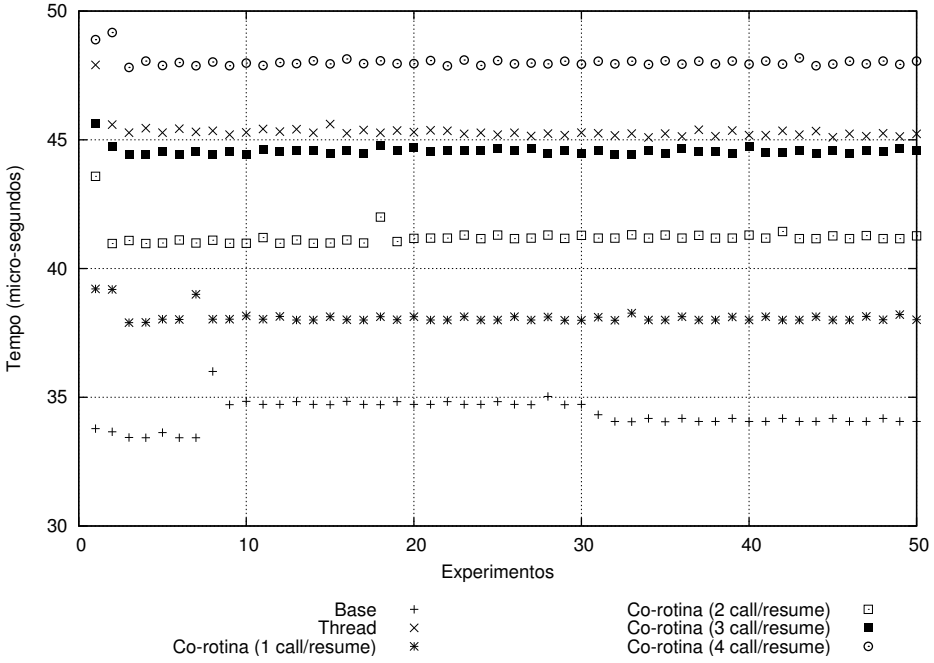

Figura 2.4: Comparando o desempenho entre co-rotinas e *threads*.

gerência preemptiva (*multithreading*). A aplicação é implementada em uma co-rotina principal e as tarefas da aplicação são executadas como co-rotinas secundárias. Com essa arquitetura é possível manter linhas de execução distintas dentro da aplicação e controlar a suspensão e retomada dessas linhas de execução de acordo com informações da própria aplicação.

Uma dificuldade, entretando, é que usando co-rotinas cabe ao programador da aplicação codificar diretamente a transferência de controle entre elas. Na arquitetura com gerência preemptiva, ao contrário, a transferência de controle entre as *threads* é feita de forma automática (pelo sistema operacional ou por uma biblioteca auxiliar) em intervalos de tempo pré-definos ou sempre que uma operação I/O é requisitada, ou seja, o programador não precisa codificá-la diretamente dentro das *threads*.

Para tratar a transferência de controle entre as co-rotinas de uma aplicação sem impor essa tarefa ao programador, combinamos a arquitetura de multi-tarefa com a arquitetura dirigida a eventos e encapsulamos as operações de transferência de controle dentro de dois elementos do modelo: o *escalonador de tarefas* e o conjunto de *operações de interação*. O escalonador de tarefas é responsável por transferir o controle da co-rotina principal para as co-rotinas secundárias, executando automaticamente os tratadores das mensagens (ou eventos) em contextos distintos de execução; e as operações de interação encapsulam a devolução do controle das co-rotinas secundárias para a co-rotina principal quando uma interação remota que requer a espera por resultados é invocada. Na próxima seção descrevemos esses dois elementos.

Escalonador de tarefas e operações de interação

A função do escalonador de tarefas é similar à função do *escalonador de eventos* dos modelos que implementam a arquitetura dirigida a eventos. O escalonador de tarefas é codificado dentro da tarefa (ou co-rotina) principal da aplicação e espera por eventos de comunicação. Consideramos um evento de comunicação uma mensagem recebida de um processo remoto. O conteúdo da mensagem encapsula: (i) a tarefa que está sendo requisitada e os argumentos para a sua execução; ou (ii) a resposta de uma requisição feita previamente ao processo remoto. Quando um evento de comunicação — requisição de uma tarefa ou resposta de uma requisição anterior — é recebido, ou uma nova co-rotina é criada para tratar a requisição recebida, ou a co-rotina que havia sido suspensa para esperar pela informação contida no evento torna-se "pronta" para ser retomada. O escalonador de tarefas é responsável por transferir o controle da execução da co-rotina principal para as co-rotinas criadas para executar as tarefas da aplicação.

As *operações de interação* são responsáveis pela interação com processos remotos e podem ser de dois tipos básicos:

- **Puramente assíncrona**: Quando os resultados da interação podem ser tratados como uma nova tarefa, a mensagem para o processo remoto é enviada e a tarefa corrente continua normalmente executando a próxima instrução. O tratador da mensagem de retorno será uma nova tarefa.
- **Síncrona**: Quando é necessário esperar pelos resultados da interação para executar a próxima instrução, a tarefa corrente deve ser suspensa e o controle devolvido para o escalonador de tarefas para que outras tarefas da aplicação possam ser tratadas. O tratador da mensagem de retorno para essa interação é implicitamente definido e sua finalidade é retomar a co-rotina suspensa.

A Figura 2.5 ilustra o processo de transferência de controle entre o escalonador de tarefas (co-rotina principal) e as linhas de controle criadas para tratar os eventos de comunicação (co-rotinas secundárias). O escalonador de tarefas é executado dentro da linha de controle principal. Quando o escalonador transfere o controle para uma tarefa (ou co-rotina), ela executa até terminar (similar a uma chamada de procedimento). Se o código da tarefa inclui uma interação síncrona com outro processo, a tarefa é suspensa por meio da *operação de interação síncrona* para ser reativada quando a chamada solicitada for completada. Nesse ponto o controle volta para o escalonador de tarefas que pode escolher outra tarefa para executar.

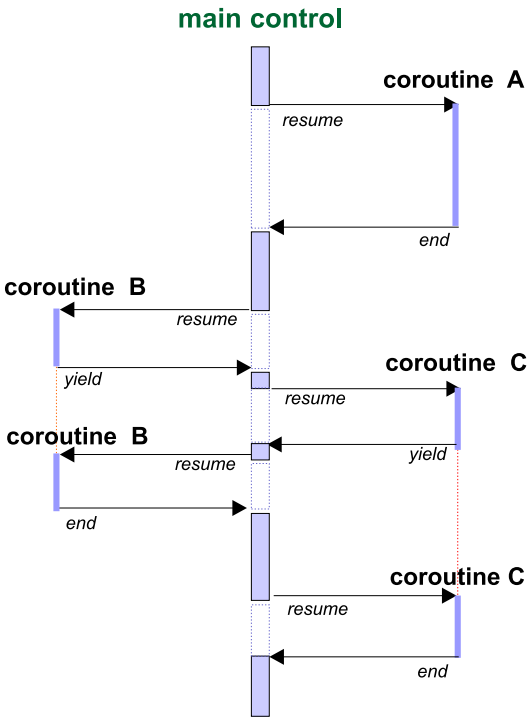

Figura 2.5: Transferindo controle entre co-rotinas.

Na Figura 2.5, quando o primeiro evento de comunicação é recebido, a co-rotina *A* é criada e o controle é transferido para ela, que executa até terminar. Quando outro evento de comunicação é recebido, a co-rotina *B* é criada. Essa co-rotina faz uma chamada a uma operação síncrona e, por isso, o controle é devolvido para o escalonador antes da co-rotina terminar sua execução. O terceiro evento de comunicação é tratado pela co-rotina *C*, que também faz uma chamada síncrona e por isso é suspensa. Quando o resultado da operação síncrona invocada pela co-rotina *B* é recebido, a co-rotina é retomada e a sua execução é concluída. O mesmo ocorre com a co-rotina *C*, quando o evento de comunicação com a resposta da chamada síncrona é recebido.

Os tratadores das mensagens podem ser armazenados em uma estrutura da linguagem (um tabela ou um vetor) e a informação associada à localização do tratador dentro dessa estrutura é embutida diretamente nas mensagens trocadas entre os processos, estendendo a noção de *mensagens ativas* (Eicken/92). Essa abordagem permite simplificar a implementação do escalonador que, ao receber uma mensagem pode tratá-la diretamente, apenas com a informação contida nos próprios campos da mensagem.

2.3 Trabalhos relacionados

Sistemas operacionais como o UNIX, Mach, Windows NT e VMS usam um padrão arquitetural caracterizado por Schimidt e Cranor como *Half-*

Sync/Half-Async (Schimidt/96). Esse padrão integra modelos de entrada e saída (I/O) síncronos e assíncronos para simplificar a programação e garantir a eficiência da execução. As tarefas implementadas no nível do usuário (como as chamadas `read` e `write`) usam um modelo de I/O síncrono, enquanto as tarefas implementadas pelo *kernel* usam o modelo de I/O assíncrono, baseado em interrupções de hardware. Os eventos externos são recebidos e processados pelas tarefas assíncronas e depositados em filas de mensagens para serem recuperados pelas tarefas síncronas, e vice-versa. Pyarali et. al. discutem como estruturar as aplicações para que as mesmas utilizem efetivamente os mecanismos assíncronos implementados pelos sistemas operacionais. Os autores descrevem o padrão arquitetural *Proactor* (Pyarali/97). Quando um processo faz uma requisição para o sistema operacional, ele registra um *despachador* que irá notificá-lo quando a operação for concluída. O sistema operacional trata essa invocação de forma independente da aplicação, que dessa forma pode ter várias operações executando simultaneamente sem precisar de um número igual de *threads*. Quando a operação termina, o sistema operacional armazena os resultados. O despachador é responsável por recuperar esses resultados e disparar a execução da função de retorno definida pelo processo. No modelo que adotamos, a *operação de interação síncrona* permite preservar o estado local de execução de uma requisição e retomá-lo quando os resultados da interação tornam-se disponíveis, sem a necessidade de lidar com a dificuldade de manter várias *threads* de execução.

No modelo de concorrência *Leader_Follower* (Schmidt/01), cada *thread* transita entre três possíveis estados: *leader, processing, follower*. Apenas uma *thread* pode estar no estado *leader* a cada instante de tempo e é essa *thread* que espera por novas conexões. Quando uma requisição é recebida, uma nova *thread* vai para o estado *leader* e a *thread* que recebeu a requisição executa o seu processamento mudando seu estado para *processing*. As demais *threads* permanecem no estado *follower*. Esse modelo permite implementar um tipo de "cooperação" entre um conjunto de *threads* simulando o modelo de gerência cooperativa de tarefas: é possível manter várias linhas de processamento, mas apenas uma delas assume o papel principal a cada instante. Entretanto, se mais de uma *thread* permanece no estado *processing* simultaneamente, torna-se necessário controlar o acesso aos recursos compartilhados. A principal diferença para a abordagem que propomos está na opção pelo uso direto de co-rotinas que permite apenas multi-tarefa cooperativa, i.e., em qualquer instante apenas uma linha de controle assume o processamento, reduzindo o custo computacional requerido.

Fair Threads (Serrano/04) são implementadas dentro da linguagem

Scheme para combinar escalonamento cooperativo e preemptivo de tarefas com o objetivo de explorar as vantagens de cada um, nesse caso, simplicidade de programação e possibilidade de explorar o paralelismo dos processadores. O modelo proposto combina *threads do usuário* que executam usando uma estratégia cooperativa; e *threads de serviços* que executam de forma preemptiva, ou seja, podem executar concorrentemente e em paralelo, sempre que o hardware permitir. O escalonador define *instants* durante os quais todas as *threads* do usuário executam até um ponto de sincronização. As *threads* cooperam devolvendo o controle para o escalonador (através de um função *yield*) ou esperando por *sinais*. Os sinais permitem a comunicação entre as *threads* e são usados para fazer uma ou mais *thread* esperar por uma condição, evitando a espera ocupada pela ocorrência de um determinado evento. As *threads* de serviço são controladas pelo sistema operacional. Do ponto de vista do programador, executar uma *thread* de serviço é equivalente a esperar por um sinal: a *thread* do usuário faz uma requisição a um serviço e espera pela sua conclusão da mesma forma que espera por outros sinais. Uma dificuldade com *FairThreads* é que a codificação dos pontos de cooperação deve ser explicitamente realizada pelo programador. Na nossa abordagem, os pontos de cooperação são encapsulados dentro do escalonador de tarefa e da operação de chamada síncrona e reproduzem a sequência natural de transferência de controle entre as partes de um programa. Por outro lado, nossa abordagem é inteiramente cooperativa, e por isso o paralelismo de hardware (quando existe) não é automaticamente explorado.

2.4 Discussão

Um aspecto importante no nosso trabalho é a opção pelo uso de co-rotinas como estrutura de controle para a gerência cooperativa de tarefas. Nosso objetivo é prover abstrações de programação mais convenientes para o programador que precisa lidar com contextos assíncronos de comunicação, usando uma infra-estrutura de multi-tarefa mais leve que o modelo tradicional da programação *multithreading*. Usando co-rotinas, uma aplicação pode manter vários fluxos de controle, mas apenas um deles pode estar ativo a cada instante do processamento, e a troca entre eles fica explícita no código da aplicação. Essa característica garante maior disponibilidade para atender eventos de comunicação sem o peso computacional comum das soluções com *multithreading*. Entretanto, a transferência de controle entre os fluxos de execução deixa de ser automática e precisa ser tratada no nível da aplicação. Nossa proposta é exatamente encapsular a transferência de controle entre as co-

rotinas dentro das operações de interação entre os processos. Assim, os pontos de troca de controle aparecem de forma explícita no código da aplicação, mas o programador não precisa lidar diretamente com a gerência das co-rotinas.

Em algumas classes de aplicações o tempo de resposta é crucial e precisa ser limitado para cada linha de execução. Nesses casos, a programação *multi-threading* pode ser a solução ótima. Entretanto, em outras classes de aplicações, como as aplicações paralelas, a estrutura de multi-tarefa é usada para garantir a otimização do uso do processador. Não existem normalmente restrições de escalonamento que exijam o compartilhamento de tempo. Uma vez que a troca de contexto consome parte do tempo de processamento, a natureza preemptiva da programação *multithreading* pode impor uma redução desnecessária no desempenho dessas aplicações. Nesse caso, a estrutura de multi-tarefa cooperativa constitui uma alternativa com menor custo computacional e capaz de oferecer uma estrutura de programação mais simples.

As vantagens do modelo que estudamos incluem, particularmente: (a) a simplicidade de lidar com a programação distribuída usando a mesma estrutura básica da programação convencional; (b) a possibilidade de usar abstrações de programação ativas, como chamadas remotas, ou reativas, como programação dirigida a eventos, ambas sobre uma base assíncrona de comunicação, construindo um estilo de programação mais conveniente para o programador; e (c) a ausência de dificuldades comuns da programação multi-tarefa com preempção, entre elas, a necessidade de incluir primitivas adicionais de sincronização e de tratar a possibilidade de condições de corrida.

No próximo capítulo descrevemos uma implementação desse modelo construindo um conjunto de operações para facilitar o desenvolvimento de aplicações que requerem a interação entre processos localizados em máquinas distintas, como aplicações paralelas que usam a estrutura das redes geográficas. No capítulo 4, descrevemos a implementação desse modelo dentro da arquitetura TinyOS (Hill/00) usando a estrutura de comunicação baseada em eventos provida pelo próprio sistema. O objetivo particular, nesse caso, é construir uma interface de programação mais fácil de usar e ainda compatível com as restrições particulares do ambiente de redes de sensores.