6
Aplicações

Implementamos dois casos de testes que exercitam as características do modelo proposto.

Como primeiro caso de uso, escolhemos adaptar o servidor web Xavante (Xavante Web Server, 2004) para acomodar a utilização de processos Lua no tratamento das requisições web. Esse servidor tem como característica atender simultaneamente diversas requisições por conteúdo estático ou dinâmico utilizando multithreading cooperativa (co-rotinas). Nosso objetivo em utilizar processos Lua no atendimento das requisições é poder contar com a preempção e até o paralelismo em vários processadores para melhorar o atendimento simultâneo de vários clientes.

No segundo caso de uso, selecionamos um executor de fluxos de automação empregado na Petrobras. Os fluxos descrevem procedimentos para gerenciar um conjunto de equipamentos. Implementamos uma nova versão do executor onde os fluxos são executados por processos Lua, sendo processados concorrentemente. Além disso, alguns dos fluxos lidam com o mesmo conjunto de dados e equipamentos, podendo ocorrer interferência e inconsistência. Mostraremos como as abstrações descritas que apresentamos podem ser empregadas para coordenar a execução dos fluxos.

6.1
Servidor Web

O servidor web Xavante, além de fornecer a transferência de conteúdo estático, ou seja, transferência de conteúdo armazenado em um arquivo, também suporta a geração de conteúdo dinâmico através de scripts Lua. Na implementação original, o Xavante cria uma nova co-rotina para atender cada requisição HTTP. As co-rotinas são responsáveis por receber os pedidos dos clientes e despachar o arquivo desejado ou por executar o script, enviando a saída deste como resultado.

Adaptamos a arquitetura para um modelo de bolsa de tarefas (*bag of tasks*), tendo um processo Lua mestre, que é responsável por esperar novas requisições HTTP, e vários processos Lua trabalhadores, que irão atender as requisições, fazendo o papel das co-rotinas na arquitetura original. O servidor será executado dentro de um processo do sistema operacional e teremos um número limitado de

threads para atender os processos Lua. A figura 6.1 mostra um diagrama de como os elementos foram distribuídos no nosso modelo proposto.

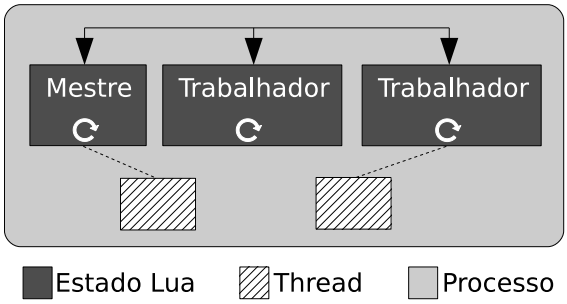

Figura 6.1: Adaptação da arquitetura do Xavante para nosso modelo.

Esta implementação pode se beneficiar de multithreading com o mecanismo de suspensão dos processos Lua. A arquitetura original do servidor Xavante é baseada em co-rotinas que são suspensas nos *timeouts* nas leituras e escritas nos sockets. Um escalonador é responsável por verificar ativamente os sockets, por meio da primitiva `select`, e colocar as co-rotinas de volta em execução quando os sockets estiverem novamente disponíveis para leitura ou escrita. Porém, em nosso sistema, se um processo Lua ficar verificando ativamente a disponibilidade do socket com a primitiva `select`, não teremos uma oportunidade de liberar a thread que executa esse processo para que ela vá executar outros processos Lua.

Desenvolvemos então um módulo adicional que fica responsável por verificar a disponibilidade dos sockets, tanto para leitura como para escrita. Os processos Lua registram quais sockets eles desejam monitorar e o módulo envia um evento de volta assim que for detectada a disponibilidade. Esse módulo é único por processo do sistema operacional e pode ser utilizado por todos os processos Lua lá residentes. Ele executa em uma thread em separado e serviu para validar a possibilidade de módulos adicionais gerarem eventos para os processos Lua.

Com esse novo módulo, os processos Lua podem suspender sua execução e retornar ao loop de evento esperando a notificação de disponibilidade. Por conveniência, as requisições dentro do processo Lua são tratadas dentro de uma co-rotina, assim, podemos suspender o processamento e retornar ao loop de eventos, mantendo o estado da computação. O tratamento do evento de disponibilidade é responsável por retomar a execução da co-rotina suspensa. A figura 6.2 apresenta a interação das partes do processo Lua com o monitor.

`Process`, `Ready` e `Task` são eventos que indicam, respectivamente, o início do tratamento de uma conexão `s`, a disponibilidade de um socket para leitura (`r`) ou escrita (`w`) e o pedido de uma nova tarefa ao programa principal. `register` é uma função que interage diretamente com o monitor para registrar o interesse na disponibilidade da conexão que está sendo tratada.

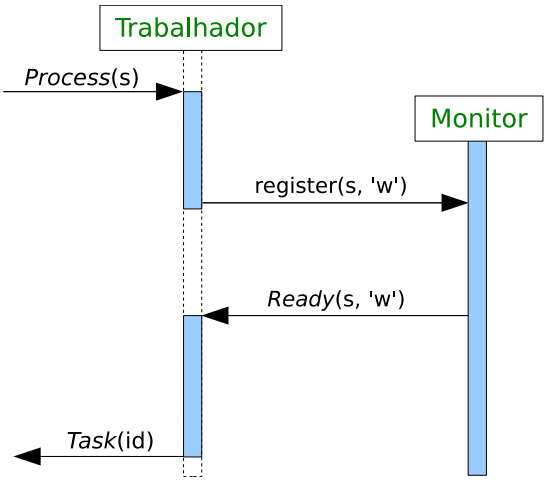

Figura 6.2: Interação entre o loop de eventos e o monitor de socket no Xavante.

Encapsulamos as chamadas de envio e recebimento do socket em novas funções que realizam todo o controle de suspensão, registro no monitor e retomada do processamento. Por exemplo, a nova função de envio, ao detectar a impossibilidade de transmissão de resposta momentânea no socket, registra a co-rotina como pendente em um variável e suspende a execução da mesma, retornando ao loop. Este por sua vez suspende todo o processo Lua, pois não há mais eventos. Pela implementação, não há a ocorrência de outro evento `Process` a menos que o processo indique ao processo principal que ele está livre para uma nova requisição. O evento `Ready` do monitor faz com que a execução da co-rotina seja retomada.

Realizamos testes para determinar o comportamento da nova implementação, comparado com a implementação original. Os testes trataram três situações:

– Provisão de conteúdo estático
– Provisão de conteúdo dinâmico
– Provisão de conteúdo estático e dinâmico

Utilizamos como servidor uma máquina equipada com processador Pentium Core 2 Quad com quatro núcleos, cada um com 2,66 GHz, 4 GB de memória RAM, Ethernet 100Mb/s, executando Linux 32 bits, kernel 2.6.26. Como clientes, utilizamos 10 máquinas com Pentium Core 2 Duo, dois núcleos de 2,66 GHz, 1 GB de memória, Ethernet 100 Mb/s, Linux 32 bits, kernel 2.6.26. Como aplicação cliente, utilizamos a ferramenta ApacheBench (versão 2.3) que realiza diversas requisições concorrentemente e exibe um relatório final sobre tempos e erros ocorridos.

Selecionamos 1.000 como valor máximo de requisições simultâneas para os testes, pois com valores maiores tanto a implementação original quanto a nossa começavam a apresentar falhas no atendimento das requisições na maioria dos testes

realizados. Essas falhas impactavam no cálculo dos tempos, porque, na maioria dos casos, o ApacheBench esperava o estouro de tempo de resposta (30 segundos) para então reportar a falha de não atendimento.

A tabela 6.1 apresenta os tempos médios (em segundos) de um conjunto de requisições concorrentes feitas pelos clientes por um conteúdo estático de 200 kilobytes. Ela mostra a versão original e nossa implementação utilizando uma variação de 2, 4, 8, 16 e 32 threads. O número total de requisições é dividido entre as 10 máquinas cliente. A nossa implementação utiliza um conjunto de 400 processos Lua para o atendimento das requisições.

Requisições	**Original**	**2 Ths**	**4 Ths**	**8 Ths**	**16 Ths**	**32 Ths**
100	1,6098	1,5579	1,6193	1,7503	1,7939	1,8491
400	6,6652	6,5067	6,6484	6,6720	6,7580	6,9926
700	11,3610	11,5742	11,3344	11,6103	11,4747	12,1173
1.000	16,4768	15,7405	16,0221	15,9913	15,6806	17,5887

Tabela 6.1: Tempos médios, em segundos, para a distribuição de conteúdo estático no Xavante.

Podemos ver pela tabela que não houve uma mudança significativa entre a implementação original do Xavante, baseada em co-rotinas, e a nossa implementação, mesmo variando o número de threads para atender os processos Lua. Aparentemente, o limite imposto pelo sistema operacional no acesso ao sistema de arquivos cria um gargalo para as threads. Tentamos requisitar arquivos diferentes, mas o efeito foi o mesmo. Durante as execuções, notamos através da ferramenta *top* que aproximadamente 15% da carga dos processadores era gasta com o Xavante, sendo o restante utilizado pelo sistema operacional.

O segundo teste, de conteúdo dinâmico, consiste de um script Lua que cria uma página HTML de 78 kilobytes com uma seqüência de números de 1 até 5.000, gerada por um laço `for`. Ele serve para verificar o comportamento em uma situação com maior demanda de processamento. A tabela 6.2 apresenta os tempos médios do conjunto de requisições concorrentes com a variação do número de threads.

Requisições	**Original**	**2 Ths**	**4 Ths**	**8 Ths**	**16 Ths**	**32 Ths**
100	24,9155	9,3144	4,4477	4,9089	4,7474	4,9599
400	103,8155	32,7209	20,3039	20,8098	20,5875	20,5223
700	175,5127	82,4459	33,5908	34,5334	35,1575	35,3173
1.000	246,0844	89,6011	52,5368	53,9090	53,5391	53,7620

Tabela 6.2: Tempos médios, em segundos, para a distribuição de conteúdo dinâmico no Xavante.

Neste novo teste, a carga de utilização dos processadores se inverteu, com a maior parte sendo destinada ao Xavante. Com a geração de conteúdo dinâmico e

o tamanho menor da resposta, não temos pontos oportunistas de suspensão nesta situação. A nossa implementação obteve melhoras de 2,68 e 5,16 vezes nos tempos com 2 e 4 threads, respectivamente, comparando com a implementação original. Os tempos permaneceram praticamente estáveis para 8, 16 e 32 threads. Isso faz sentido dado o número de núcleos disponíveis (quatro).

Na terceira avaliação, realizamos requisições de conteúdo estático e dinâmico simultaneamente. Dividimos igualmente o número de requisições para o conteúdo estático e dinâmico. A tabela 6.3 apresenta os tempos médios e os desvios padrão (σ) em segundos. Dado que nos testes anteriores a variação do número de threads não afetou o resultado, apresentamos os valores apenas para 4 threads.

Requisições	**Original**			**4 Threads**		
	Média	σ	Falhas	Média	σ	Falhas
100	6,8678	5,2096	0%	2,5931	0,2132	0%
400	32,2831	19,1890	0%	10,6314	1,8431	0%
700	49,1622	31,2716	13,8%	19,4118	3,0183	0%
1.000	54,6935	36,2738	59,4%	27,7981	3,5755	0%
2.000	—	—	—	32,4385	4,4057	0%

Tabela 6.3: Tempos médios, em segundos, para a distribuição de conteúdo dinâmico e estático no Xavante.

A versão original conseguiu atender sem erros 100 e 400 requisições disparadas simultaneamente. Mas as amostragens de 700 e 1.000 apresentaram, respectivamente, 13,8% e 59,4% de falha no atendimento. A provável fonte das falhas foi a demora causada pela geração do conteúdo dinâmico. Com multithreading, pudemos dividir a carga de processamento nos processos Lua, mas no caso da versão original baseada em co-rotina, isso não é possível. O nosso modelo teve um limite de 2.000 requisições sem falhas.

Como citamos anteriormente, os testes foram feitos utilizando um número fixo de 400 processos Lua para atender as requisições. Essa quantidade de processos foi determinada como o número mínimo aproximado de processos necessários para cumprir todos os testes. Realizamos testes com o número de 100, 200 e 300 processos Lua, e não foi possível atender todas as requisições do conjunto utilizado acima. No entanto, verificamos que a variação do número de processos de 400, 700 e 1.000 não afetou os tempos.

6.2 MPA – Módulo de Procedimentos Automatizados

O MPA é um sistema de controle de automação que executa aplicações para gerenciamento de equipamentos. Ele é desenvolvido pelo laboratório Tecgraf da PUC-Rio juntamente com a Petrobras, a qual o utiliza em suas plataformas

e refinarias. O MPA possui uma ferramenta gráfica que é usada para a criação de fluxos de controle e um servidor é responsável por executá-los. Por meio da ferramenta, geramos um conjunto de estruturas de dados e ações que descreve os fluxos. Esse conjunto será enviado ao servidor de gerenciamento de fluxos que o executará. A figura 6.3 mostra um exemplo de fluxo que tem por finalidade controlar válvulas de um compressor para regular a pressão desejada.

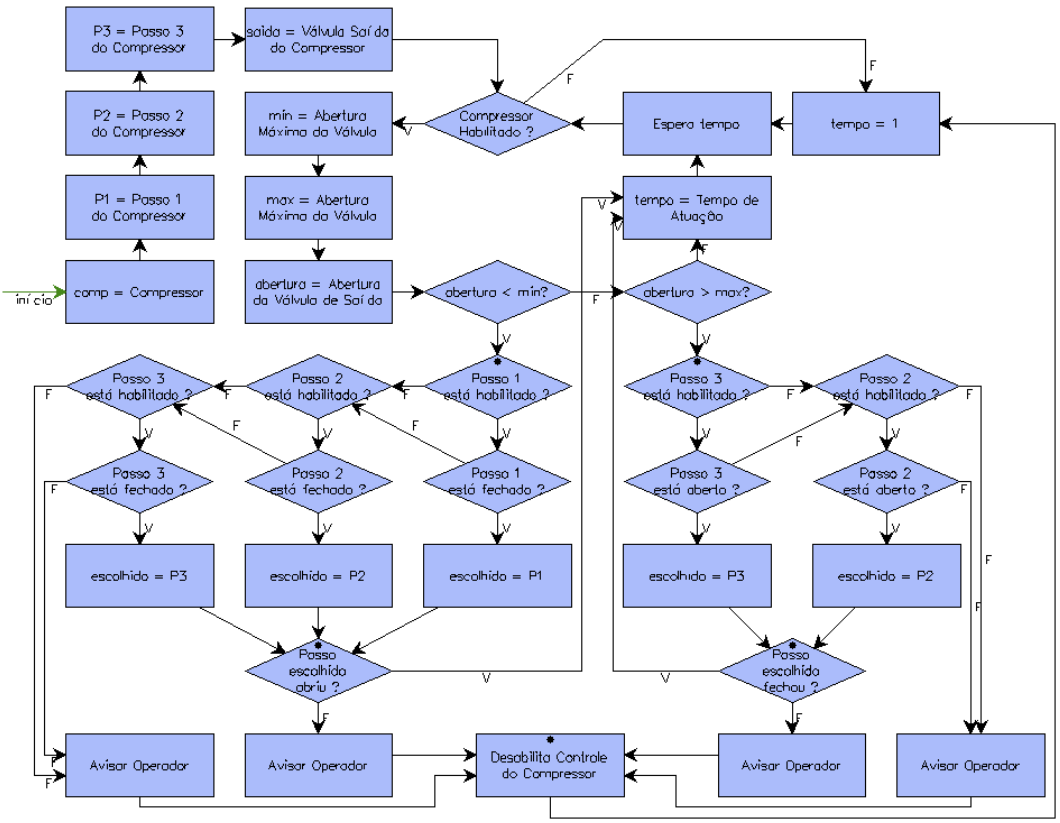

Figura 6.3: Exemplo de um fluxo do MPA.

Como segundo caso de uso, reimplementamos o servidor de execução de fluxos utilizando o modelo de eventos proposto. As aplicações do MPA possuem um conjunto de fluxos em execução simultânea. Esses fluxos avaliam expressões e condições, lêem e alteram variáveis, e interagem com equipamentos. Por exemplo, o fluxo da figura 6.3 lê a abertura de uma válvula principal de saída, armazena o valor na variável `abertura` e depois compara com os valores mínimo e máximo aceitáveis. De acordo com o resultado do teste, o fluxo abre ou fecha válvulas auxiliares (passos).

A interação do MPA com os equipamentos se dá através de *pontos*. Esses pontos se comportam como variáveis que são utilizadas para realizar ações, indicar algum estado do equipamento, ou simplesmente armazenar um valor. Por exemplo, atribuir um valor a um ponto de uma válvula pode fazer que ela se abra. Ler o valor de outro ponto pode indicar se a válvula está aberta ou fechada. Como o

equipamento é compartilhado pelas aplicações, elas podem utilizar os pontos para troca de informação entre si.

Para estruturar melhor as aplicações, o MPA utiliza *objetos*, os quais podem conter pontos, funções auxiliares e atributos. Os atributos podem ser valores como número ou string, bem como referência a outros objetos.

O conjunto de definições de pontos, objetos e funções auxiliares é chamado de *pré-configuração*, e a instanciação desses componentes dá origem a uma *planta*, sobre a qual os fluxos atuam. A planta é compartilhada por todos os fluxos dentro de uma aplicação, ou seja, os fluxos podem estar interagindo com os mesmos objetos ao mesmo tempo.

Em nossa implementação, um conjunto de fluxos que compõe uma aplicação do MPA é disparada em um processo do sistema operacional. Isso nos permite, em caso de problema, finalizar e reiniciar uma aplicação sem afetar as demais. A figura 6.4 ilustra a arquitetura geral de nossa implementação.

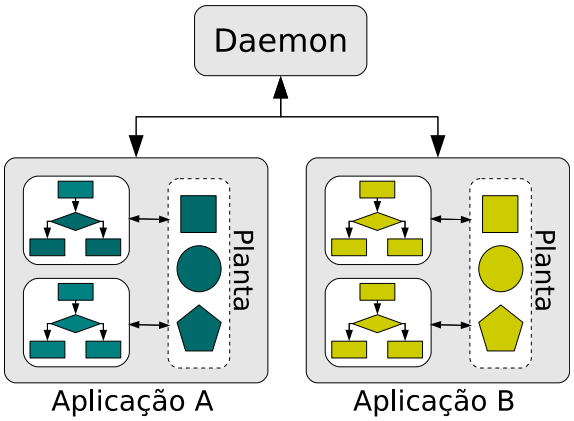

Figura 6.4: Arquitetura geral das aplicações em nossa implementação do MPA.

Ao mapear os elementos do MPA para nosso modelo, adotamos uma granularidade pequena, onde cada um dos fluxos e objetos é executado em um processo Lua separado, como mostrado na figura 6.5. Esse mapeamento tem a vantagem de que os elementos podem executar independente um dos outros. Uma outra abordagem seria agrupar elementos em um mesmo processo Lua, o que reduziria o número de processos Lua envolvidos. Mas, ao compartilhar um processo, apenas um elemento por vez trata eventos. Assim, a forma como o mapeamento é feito pode influenciar na concorrência e no uso de recursos de uma aplicação. Nesta implementação, decidimos manter cada elemento em um processo Lua.

Os objetos da planta são instanciados como processos Lua e ficam esperando eventos vindos dos fluxos ou outros objetos. A figura 6.6 apresenta um exemplo de código da definição de um objeto *sensor de ruído* e criação de duas instâncias do mesmo. O sensor possui um ponto para acesso direto ao dispositivo (`noise`), um

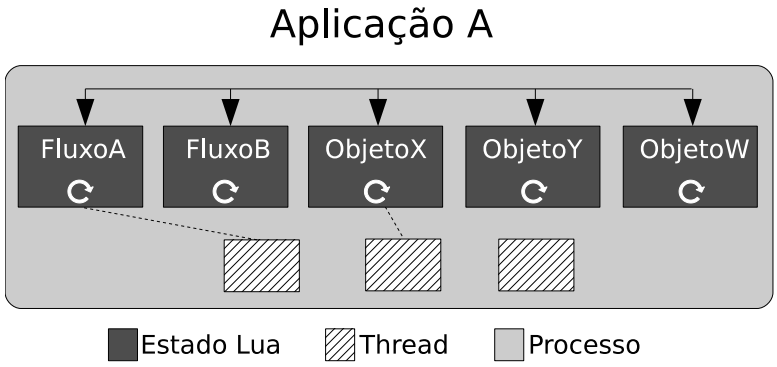

Figura 6.5: Mapeamento dos elementos do MPA para o nosso modelo.

atributo de limiar para indicar a presença (`threshold`) e uma função `presence` que testa se a presença foi detectada.

```
-- Pré-configuração: definição
SoundSensor = function(args)
  local obj = object.create{ id = args.id }
  obj.noise = point.create{ id = args.noise }
  obj.threshold = args.threshold
  function obj.presence()
    return obj.noise.read() > obj.threshold
  end
end

-- Planta: instanciação
Object {
  type = "SoundSensor",
  id = "LivingRoomSensor",
  noise = "livingroom.sensor",
  threshold = 30,
}

Object {
  type = "SoundSensor",
  id = "BedroomSensor",
  noise = "bedroom.sensor",
  threshold = 20,
}
```

Figura 6.6: Exemplo de pré-configuração e planta no MPA.

A criação dos fluxos consiste em definir cada uma das operações do fluxo (os retângulos e losangos da figura 6.3) e interligá-las por meio da interface gráfica. O MPA fornece um conjunto pré-definido de operações para lidar com expressões, teste, execução de outros fluxos, etc. A figura 6.7 apresenta o exemplo simplificado de um fluxo que é responsável por acender uma lâmpada quando verificado que o valor do sensor de ruído ultrapassou um certo limiar.

De modo geral, as operações `GetThreshold` e `GetNoise` interagem com um objeto, armazenado na variável `sensor`, para buscar o valor do limiar (atributo

```
-- Definição das ações do fluxo
GetThreshold = ObjectEval{
  object = "sensor",
  member = "threshold",
  variables = {"t"},
}

GetNoise = ObjectEval{
  object = "sensor",
  member = "noise",
  operation = "read",
  variables = {"n"},
}

Presence = ExpressionTest{
  value = function() return n > t end,
}

TurnOnLamp = ObjectEval{
  object = "lamp",
  member = "turnon",
  operation  = "write",
  parameters = function() return true end,
}

-- Interligação entre as ações
GetThreshold.next = GetNoise
GetNois.next      = Presence
Presence.success  = TurnOnLamp
Presence.failure  = GetThreshold

-- Instanciação do fluxo
FlowPresence = Flow(GetThreshold, {sensor = "BedrooomSensor",
                                   lamp   = "BedroomLamp"})
```

Figura 6.7: Exemplo de definição de um fluxo no MPA.

`threshold`) e o valor atual reportado pelo sensor (ponto `noise`). Ambos valores são salvos em variáveis que depois são utilizadas pelo teste `Presence` que verifica se há algum movimento. Caso seja detectado, a operação `TurnOnLamp` interage com o objeto da variável `lamp`, alterando o valor de um ponto que faz com que a lâmpada seja acesa.

Depois que as ações são definidas, elas são interligadas para definir o fluxo da execução. O objeto que representa o fluxo é então criado (`FlowPresence`), tendo como primeiro argumento a ação inicial do fluxo. O segundo argumento é o estado inicial contendo variáveis usadas pelas ações. Dadas as definições, a figura 6.8 apresenta o código que inicia uma aplicação em nossa implementação do MPA. O módulo `mpa`, construído sobre o ALua, implementa a infra-estrutura do MPA e exporta algumas funções auxiliares. `mpa.run` inicia a comunicação com o daemon e depois retoma a computação via callback (`main`). `mpa.create` instancia uma nova

aplicação, retornando o identificador pelo qual podemos disparar os fluxos. A função `mpa.create` recebe uma tabela de configuração com o nome dos módulos que contêm a pré-configuração, a planta e os fluxos. Após a carga da aplicação, iniciamos a execução do fluxo, no exemplo, `Flowpresence`. Utilizamos a abstração de RPC apresentada no capítulo 3 como forma de comunicação entre fluxos e objetos.

```
local config = { modules = {"preconfig", "plant", "flow"} }
local daemoncfg = { addr = "127.0.0.1", port = 8888 }

function main()
  local id = mpa.create(config)
  rpc.async(id, "FlowPresence.start")()
end

mpa.run(daemoncfg, main)
```

Figura 6.8: Criação de uma nova aplicação MPA.

Em princípio, necessitamos de um número de threads igual ao número de fluxos em execução. Com a comunicação síncrona, quando o fluxo faz uma requisição a um objeto, ele é suspenso e sua thread é liberada. Essa thread pode, por exemplo, executar o processo Lua referente ao objeto requisitado e atender ao pedido do fluxo. Ao finalizar a requisição, o objeto volta ao loop de eventos e libera a thread, que provavelmente voltará a executar o fluxo. Há uma transferência constante de controle de execução entre fluxos e objetos, mas isso não significa que as threads estarão ligadas sempre aos mesmos elementos.

Os fluxos lidam com equipamentos e estes comumente necessitam de um tempo de atuação. Por exemplo, a abertura de uma válvula é um processo mecânico e não é feita instantaneamente. Dependendo do equipamento, são necessários alguns segundos para ter certeza de que a operação foi completada. Nas aplicações atuais do MPA, essas esperas são freqüentes, então, enquanto o fluxo aguarda um equipamento estabilizar, poderíamos liberar a thread para executar outro fluxo ou objeto. Com isso, poderíamos utilizar menos threads dentro de uma aplicação, se os fluxos apresentarem pausas constantes devido aos equipamentos.

Assim como no exemplo do Xavante, implementamos um módulo extra de alarme que os fluxos e objetos utilizam para serem notificados via eventos após decorrer um certo tempo. Os fluxos e objetos podem então retornar ao loop de eventos e liberar a thread durante o tempo de atuação do equipamento. Ao receberem um evento de alarme, eles retomam a execução.

Se por um lado essa abordagem pode diminuir o número de threads, reduzindo o consumo de recursos, por outro, nós abrimos espaço para que outros eventos sejam processados nos objetos. Por exemplo, no caso do compressor citado anteriormente,

podemos definir o objeto válvula como na figura 6.9. O objeto possui os pontos para abrir e fechar a válvula (`PtOpen` e `PtClose`, respectivamente) e as funções utilizadas pelos fluxos para executar essas tarefas, que já lidam com o tempo de atuação da válvula. A função `WaitTime` é responsável por registrar o alarme e suspender e retomar a execução.

```
Valve = function(args)
  local obj   = object.create(args)
  obj.PtOpen  = point.create{ id = args.PtOpen  }
  obj.PtClose = point.create{ id = args.PtClose }
  obj.Time    = args.Time
  function obj.Open()
     obj.PtOpen.write(true)
     WaitTime(obj.Time)
  end
  function obj.Close()
     obj.PtClose.write(true)
     WaitTime(obj.Time)
  end
end
```

Figura 6.9: Definição de um objeto válvula no MPA.

Durante o processo de fechamento da válvula, outro fluxo pode identificar que é necessário reabrí-la. Nesse caso, como os objetos são compartilhados por todos os fluxos da aplicação, um segundo fluxo pode iniciar o processo de fechar a válvula, pois o objeto está no loop de eventos para esperar pelo evento de alarme e nada impede que ele trate outros eventos. Essa mesma condição foi discutida no capítulo 3.

No exemplo anterior, para garantir a utilização correta das funções de abertura e fechamento da válvula, podemos recorrer ao mecanismo de monitor apresentado no capítulo 3. A figura 6.10 apresenta a nova implementação do objeto válvula, onde as funções `Open` e `Close` são versões protegida exportadas para os fluxos.

A implementação original do MPA emprega somente multithreading cooperativa das co-rotinas para a execução dos fluxos, recaindo nos tempos de atuação para a troca de contexto entre os fluxos. Se houver um fluxo que não tenha necessidade de esperar pela atuação de um equipamento, ele irá monopolizar a execução, pois não haverá a troca de contexto com os outros fluxos. Neste caso, o criador do fluxo terá que forçar uma espera. Com o nosso modelo, a restrição do fluxo sem pausa pode ser contornado com as threads.

6.3 Considerações Finais

A reimplementação do Xavante permitiu que testássemos o mecanismo de desacoplamento de threads com os processos Lua. Ao adotarmos o modelo de

```
Valve = function(args)
  local obj    = object.create(args)
  obj.mnt      = monitor.create()
  obj.PtOpen   = point.create{ id = args.PtOpen  }
  obj.PtClose  = point.create{ id = args.PtClose }
  obj.Time     = args.Time
  local function open()
     obj.PtOpen.write(true)
     WaitTime(obj.Time)
  end
  local function close()
     obj.PtClose.write(true)
     WaitTime(obj.Time)
  end
  obj.Open  = monitor.doWhenFree(obj.mnt, open)
  obj.Close = monitor.doWhenFree(obj.mnt, close)
end
```

Figura 6.10: Utilização de um monitor para proteção do objeto válvula.

bolsa de tarefas, temos os processos Lua requisitando trabalho concorrentemente ao programa Lua principal. Vale destacar também o efeito do modelo orientado a eventos. Sem o uso do monitor de socket na provisão de conteúdo estático, não poderíamos suspender a thread. Além disso, na provisão de conteúdo dinâmico, o tempo de processamento era curto, liberando as threads. Caso contrário, teríamos o monopólio de processamento de alguns processos Lua. Esse efeito foi descrito no capítulo 2, com o uso de co-rotinas e eventos. No entanto, com o novo modelo proposto, temos um certo grau de flexibilidade, pois podemos alterar o número de threads que atenderão os processos Lua e, conseqüentemente, eventos.

O caso de uso do MPA permitiu que explorássemos a utilização de processos e threads em sua arquitetura, bem como a abstração de monitor definida anteriormente. As aplicações atuais do MPA utilizam os pontos compartilhados dos equipamentos para promover a coordenação, seja interna ou externa, não havendo outra forma de comunicação entre elas. Em nosso caso, poderíamos utilizar a abstração do sincronizador, para a coordenação entre aplicações.