5
Estudo de Caso: LuaBREW

Neste capítulo, será apresentado o LuaBREW, uma ferramenta de desenvolvimento baseada na linguagem de *script* Lua[27], onde sua máquina virtual executa sobre o BREW[6], um ambiente orientado a eventos com processamento assíncrono projetado para permitir o desenvolvimento de aplicações para dispositivos móveis. O modelo de escalonador proposto será utilizado para permitir e gerenciar o processamento colaborativo de múltiplas linhas de execução.

5.1
Ambiente BREW

O BREW (*Binary Runtime Environment for Wireless*)[6] surgiu da necessidade de uma plataforma de desenvolvimento padronizada para o emergente mercado de aplicações para dispositivos móveis. No início de 2001, foi proposto pela *Qualcomm* um modelo padronizado de ambientes de desenvolvimento para estes dispositivos visando um baixo grau de dependência dos hardwares, e assim permitir sua integração em praticamente todos os dispositivos de telefonia móvel, incluindo *smartphones* e *PDAs*. O modelo consiste em um ambiente de execução de aplicações situado sobre o software do *chipset*, integrado com um sistema de distribuição de conteúdo.

Com o BREW, permite-se o desenvolvimento de aplicações para dispositivos móveis de maneira independente do sistema operacional individual de cada modelo de dispositivo, fazendo com que toda a complexidade de adaptação e tradução de funcionalidade seja de responsabilidade do ambiente BREW. Apesar da tecnologia de integração do ambiente de desenvolvimento com a plataforma dependente do *chipset* ser fechada e proprietária, as ferramentas necessárias para o desenvolvimento de aplicações para a plataforma BREW, bem como extensiva documentação, estão disponíveis em [6].

A API de programação define todas as interfaces básicas para o ambiente de desenvolvimento. Os serviços são inteiramente desenvolvidos utilizando essa API padrão, que define métodos de acesso aos dispositivos de exibição, sistema de arquivos, sistemas de telefonia (no caso de telefones celulares), bem como outras interfaces específicas desses terminais. O acesso a esses dispositivos é feito através de mecanismos assíncronos, baseados no registro de funções de *callback* como finalizadores responsáveis por recuperar os valores dos métodos invocados.

A interface de programação é definida em C, apesar de algumas características de C++ serem utilizadas, como herança de classes e sobrecarga de métodos e operadores. Entretanto, não existe suporte ao lançamento e captura de exceções C++ . O SDK provê um simulador do ambiente celular para a plataforma *Windows*, que permite que a aplicação seja testada antes de ser carregada no dispositivo final. O código fonte deve ser compilado com um compilador ARM específico para a plataforma BREW, porém também existe a possibilidade de utilizar compiladores GCC tradicionais com a configuração de alguns parâmetros.

A plataforma também permite a criação de extensões utilizando o conceito de bibliotecas dinâmicas. As extensões permitem o desenvolvimento de ferramentas que encapsulam funcionalidades que podem ser carregadas de acordo com a necessidade da aplicação. Utilizando esse suporte, foi desenvolvida uma ferramenta de programação, composta pela máquina virtual Lua adaptada para a realidade de um ambiente móvel, permitindo a execução de *scripts* Lua dentro de um ambiente BREW, associada a um conjunto de componentes gráficos.

5.2 Ferramenta LuaBREW

Apesar de ser uma linguagem de programação altamente difundida, o desenvolvimento de aplicações para dispositivos móveis em C acarreta altos custos de desenvolvimento e depuração, além da necessidade de um período longo de treinamento para adaptar os desenvolvedores às restrições e necessidades de programação para esse ambiente. A necessidade de compilação também impede a carga de trechos de código e atualização em tempo de execução.

Frente a isso, surge a necessidade de um ambiente de desenvolvimento mais flexível, porém sem agregar um custo grande de memória nem perda de desempenho. A linguagem Lua possui um ambiente de execução leve,

flexível e fornece uma API que permite a troca de dados entre a aplicação e o ambiente Lua bem como a extensão das funcionalidades padrão da linguagem. Esse conjunto de fatores levou ao projeto de adaptação da máquina virtual Lua para a plataforma BREW, criando uma ferramenta que poderia ser usada em conjunto com aplicações tradicionais em tarefas onde o uso de *script* se mostre eficiente, como a construção de *parsers* ou no desenvolvimento de componentes com comportamento passível de atualização em tempo de execução.

5.2.1 Análise de Requisitos

Mesmo com a flexibilidade da linguagem Lua, o ambiente de desenvolvimento continua dependente de um sistema operacional assíncrono, onde o valor de retorno de grande parte dos métodos de interação com o dispositivo móvel tem que ser obtido por meio de funções de *callback*, fazendo com que a execução do código siga um fluxo não linear. O modelo de programação assíncrona ainda se mostra hostil para grande parte dos desenvolvedores, gerando a motivação para o desenvolvimento de um ambiente onde se pudesse simular o processamento de múltiplas linhas síncronas de execução.

O processamento é feito de forma assíncrona para permitir que múltiplas fontes de eventos, como interação com o usuário, controle de eventos de rede e telefonia possam ser tratados de forma eficiente, sem que o processamento de um evento impeça o recebimento dos demais. A implementação inicial da máquina virtual Lua para *BREW* utilizava modelos síncronos para o despacho dos eventos para o ambiente Lua. A figura 5.1 ilustra o comportamento desse modelo.

Como tem-se apenas uma linha de execução no ambiente *LuaBREW*, o despacho síncrono dos eventos faz com que o sistema fique impossibilitado de processar novos eventos até que o evento original tenha seu tratamento finalizado. Nesse ambiente, o uso de um escalonador colaborativo possibilita o processamento síncrono de múltiplas linhas de execução, permitindo o tratamento de múltiplos eventos simultaneamente e desacoplando a complexidade do gerenciamento de funções de *callback* das tarefas de desenvolvimento específicas da aplicação.

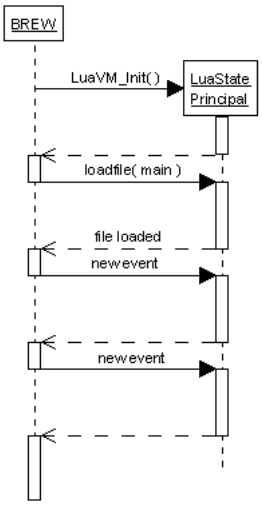

Figura 5.1: Modelo de despacho síncrono para o *LuaBREW*.

5.2.2 Solução

O objetivo principal da adaptação do modelo apresentado no capitulo 3 para a arquitetura *LuaBREW* é prover mecanismos que permitam isolar a complexidade do gerenciamento do fluxo não linear de dados observado no modelo assíncrono de programação tradicional. Esse novo modelo deve permitir a execução síncrona dos métodos, porém mantendo as características não bloqueantes do modelo assíncrono original, respeitando a necessidade de se manter o sistema livre para processar eventos de maior prioridade. A falta de flexibilidade na descrição de dados da linguagem C e as particularidades encontradas no sistema operacional dos dispositivos móveis celulares levaram a adaptações no modelo proposto no capítulo 3. Nesse cenário, observa-se algumas modificações em alguns dos componentes apresentados anteriormente, com algumas funcionalidades oferecidas diretamente pelo sistema operacional, porém mantendo-se a estrutura lógica de componentes proposta.

Por se tratar de um ambiente operacional assíncrono, o demultiplexador de eventos síncrono do *Observador de Eventos* é fornecido pelo sistema através do registro de uma função de tratamento de eventos (*HandleEvent*), que recebe como parâmetros de entrada a fonte de origem do evento e possivelmente outros valores específicos do evento. Na arquitetura de integração com o *LuaBREW*, cabe ao *Observador de Eventos* apenas registrar as fontes definidas pela implementação das *Fontes de Eventos* e demultiplexar os eventos sinalizados pelo sistema.

As *Fontes de Eventos* são os componentes responsáveis por imple-

mentar a camada de integração das interfaces assíncronas BREW com o mecanismo definido para o processamento colaborativo dos eventos. Essa integração com a API BREW requer uma implementação específica para cada componente, porém com uma estrutura bem parecida. Para cada operação assíncrona, deve existir um método para iniciar o processamento, bem como métodos para o registro de tratadores de notificações de mudanças no estado dos objetos.

O ciclo de acesso se inicia com uma chamada explícita ao método assíncrono que por sua vez retorna imediatamente o controle em caso de indisponibilidade dos dados desejados. Para que o processamento seguisse um fluxo linear, nesse momento, o desenvolvedor poderia inserir um ciclo de verificação terminado com a disponibilização dos dados requisitados na operação. Porém, restrições impostas pelo ambiente de desenvolvimento BREW impedem que a aplicação permaneça por grandes intervalos de tempo sem responder aos eventos de alta prioridade do sistema, normalmente ligados à eventos de telefonia e de rede.

A adaptação proposta para a *Fonte de Eventos* utiliza o mecanismo de corrotinas para permitir que uma linha de execução permaneça bloqueada sem que todo o sistema fique bloqueado. O processamento executado pelas *Fontes de Eventos* deve ser encapsulado em objetos *Thread* de Lua[27] permitindo que através da utilização de chamadas ao método `yield` ocorra somente o bloqueio da linha de execução associada ao tratamento do evento, permitindo que o sistema continue respondendo aos eventos críticos.

O *Observador de Eventos* também fornece suporte ao registro de temporizadores, registrados como métodos assíncronos onde os eventos de notificação não estão associados a um fluxo de dados, porém ao tempo.

Quando um programa BREW cria uma instância da máquina virtual Lua, é aberto um `LuaState` responsável pelo carregamento do código a ser executado. O *Escalonador* do sistema deve executar nesse `LuaState` principal e a partir dele cria-se um *Repositório de Threads*, componente equivalente ao *Repositório de Tratadores de Eventos*, onde se armazena um conjunto de objetos *Thread* utilizados para processar os eventos recebidos pelo *Observador de Eventos*.

O *Repositório de Threads*, responsável pelo armazenamento das rotinas, é implementado utilizando uma tabela armazenada no registro de Lua, associando cada objeto *Thread* a um identificador de estado. Devido às restrições de memória, decidiu-se utilizar um conjunto finito e pré-alocado, permitindo que o tamanho desse conjunto seja especificado de acordo com a capacidade do dispositivo e seja modificado em tempo de execução pela

aplicação. Essa política segue a estratégia *Thread Pool*, onde a pré-alocação dos *threads* garante um baixo custo em termos de desempenho e limita a utilização dos recursos disponíveis no sistema[14].

Com um conjunto finito de *Threads* para o processamento, o *Escalonador* permite o registro de um método para receber notificações de eventos não processados, permitindo que o conjunto de *Threads* seja aumentado ao receber a notificação de um evento crítico. Essas notificações são processadas utilizando o `LuaState` principal (e não por um *Thread*) fazendo com que não seja possível executar métodos assíncronos. Um modelo semelhante é utilizado no tratamento de erros gerados pela execução dos métodos em ambiente protegido.

O modelo apresentado no capítulo 3 é aplicado à estrutura da ferramenta LuaBREW fazendo com que ao receber um evento na função registrada como tratador principal no *Observador de Eventos*, o *Escalonador* acesse o *Repositório de Threads* e busque um objeto *Thread* livre para o processamento iniciando o tratamento do evento. Nesse momento o *thread* utilizado é retirado da lista de disponíveis para processamento, garantindo que ele não será utilizado para o processamento de outros eventos até ser liberado. O uso de uma coleção de corrotinas (através dos *Threads*) permite que o despacho dos tratadores seja executado de maneira assíncrona, permitindo o retorno do controle ao escalonador. Na figura 5.2 observa-se o comportamento do modelo de despacho assíncrono dos tratadores de eventos.

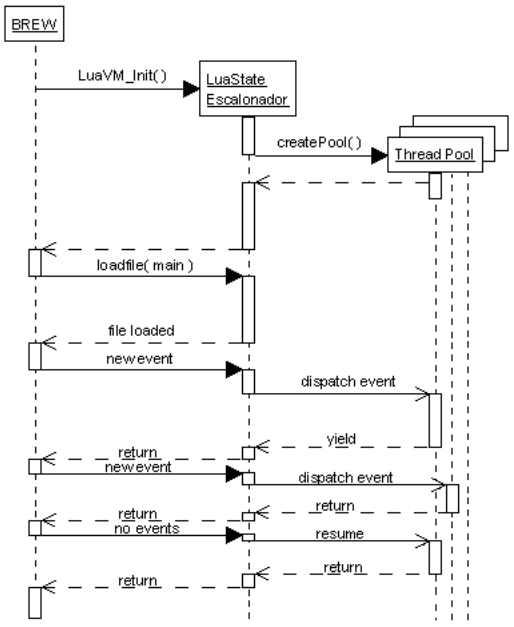

Figura 5.2: Modelo de despacho assíncrono para o *LuaBREW*.

Durante o processamento do evento, o controle normalmente é retornado ao escalonador de maneira transparente para o desenvolvedor da aplicação através da implementação da camada de integração da *Fonte de Eventos*, porém, em um modelo de escalonamento colaborativo não se pode garantir a periodicidade das chamadas à essa interface, tornando fundamental a existência de meios de executar métodos que consomem tempo sem impedir o processamento de outros eventos. Dessa forma, o controle também pode ser retornado ao *Escalonador* de maneira explicita através do método `shell.processevents()`, que bloqueia o processamento corrente e insere a linha de execução no final da *Fila de Prontos*.

5.3 Exemplo de Uso

O principal objetivo da incorporação do modelo de escalonamento colaborativo proposto dentro do ambiente exposto pelo BREW é permitir que o desenvolvimento de serviços se torne mais intuitivo e simples para os usuários dessa ferramenta. Essa necessidade é diferente da que foi apresentada no capítulo 4 pois, apesar do desempenho ser fator fundamental, o escalonamento das tarefas deve proporcionar uma interface de programação que permita um fluxo lógico linear inserido em um ambiente totalmente baseado em processos assíncronos.

Nesse ambiente de teste procurou-se identificar os pontos da API *LuaBREW* que faziam com que o fluxo de processamento dos dados fugisse do modelo de programação seqüencial. Em um ambiente onde grande parte da API é baseada em procedimentos assíncronos com valores retornados através de chamadas de *callback* utilizando métodos estáticos, pode-se observar um cenário onde o grau de complexidade de manutenção de código cresce muito rapidamente com a evolução do serviço e com a adição de funcionalidades.

Até mesmo em serviços simples, observa-se que embora seja viável a compreensão e manutenção do código fonte, essa estrutura representa um ambiente hostil para desenvolvedores sem muita experiência. Nesta seção vamos ilustrar esse caso com um exemplo de serviço simples, mostrando a arquitetura de implementação em um ambiente BREW tradicional e comparando com o que pode ser obtido aplicando-se o modelo proposto de escalonamento embutido nos serviços oferecidos pela adaptação da máquina virtual Lua.

O serviço proposto utiliza uma interface de comunicação assíncrona para recuperar dados armazenados em um servidor externo utilizando *sockets TCP* e armazenando esses dados na memória interna do dispositivo celular. Esses dados são então processados e o resultado é exibido na tela do celular. Esse modelo é muito utilizado como parte de serviços de busca de conteúdo, onde a navegação é feita através de comunicação direta com servidores externos.

Abaixo encontra-se a estrutura do código observado no ambiente tradicional BREW:

```
1  boolean CApp::connect( void )
2  {
3      this->pSock = INETMGR_OpenSocket( m_pINetMgr, AEE_SOCK_STREAM );
4      if ( !this->pSock ) return FALSE;
5
6      // ... Assumindo que o nome já está resolvido no DNS
7      return ISOCKET_Connect( this->pSock, Addr, nPort, sendStart, this );
8  }
9
10 static void sendStart( CApp * pApp )
11 {
12     pApp->buffer = "STRT";
13     int bytesToWrite = 4 - pApp->numBytesWritten;
14
15     // Escreve dados no canal de comunicação.
16     int32 rv = ISOCKET_Write( pApp->pSock,
17                               pApp->buffer[ pSocket->numBytesWritten ],
18                               bytesToWrite );
19
20     if ( rv == AEE_NET_ERROR )  // Falha no envio de dados.
21     {
22         ISHELL_SetTimer( 0, errorFunction, pApp );
23     } else if ( rv == bytesToWrite )  // Transmissão finalizada OK.
24     {
25         // Pode-se executar a callback para leitura de dados.
26         ISHELL_SetTimer( 0, getDataSize, pApp );
27     } else  // O numero de bytes escritos foi menor que o esperado
28     {
29         // Atualiza o valor da variável
30         if( rv > 0 ) pApp->numBytesWritten += rv;
31         // Registra a callback para escrever mais dados
32         ISOCKET_Writeable( sendStart, pApp );
33     }
34 }
```

Listing 5.1: Exemplo de código em BREW.

O protocolo de comunicação definido nesse serviço especifica que o terminal, após o estabelecimento da conexão com o servidor de dados, deve enviar inicialmente 4 *bytes* requisitando o inicio da comunicação enviando a *string* "STRT". A rotina de envio de dados (`sendStart()`) deve ser implementada de maneira a permitir que os dados possam ser enviados em blocos transmitidos em ciclos de processamento não bloqueante. O

procedimento de leitura de dados do canal de comunicação também deve ser implementado de maneira análoga, utilizando um método não bloqueante responsável por receber e acumular os dados disponíveis até o recebimento do número de *bytes* requisitados, como pode ser visto na listagem 5.2.

```
1  static void getDataSize( CApp * pApp )
2  {
3      int bytesToRead = 4 - pApp->numBytesRead;
4
5      // Lê dados do canal de comunicação.
6      int32 rv = ISOCKET_Read( pApp->pSock,
7                               pApp->readBuffer + pApp->numBytesRead,
8                               bytesToRead );
9      }
10
11   if ( rv == AEE_NET_ERROR )  // Falha no recebimento de dados.
12     {
13         ISHELL_SetTimer( 0, errorFunction, pApp );
14     } else if ( rv == bytesToRead ) // Transmissão finalizada OK.
15     {
16         // Pode-se executar a callback para leitura do conteúdo.
17         pApp->contentSize = convertToInt( pApp->buffer );
18         pApp->numBytesRead = 0;
19         ISHELL_SetTimer( 0, getDataContent, pApp );
20   } else  // O numero de bytes recebidos foi menor que o esperado
21     {
22         // Atualiza o valor da variável
23         if( rv > 0 ) pApp->numBytesRead += rv;
24         // Registra a callback para ler mais dados
25         ISOCKET_Readable( getDataSize, pApp );
26     }
27 }
28
29 static void getDataContent( CApp * pApp )
30 {
31   // ... Implementação similar à getDataSize()
32 }
```

Listing 5.2: Exemplo de código em BREW.

O recebimento dos dados se inicia com o método `getDataSize()` responsável por receber 4 *bytes* contendo o tamanho do conteúdo a ser recebido. Nesse exemplo, pode-se observar a existência de um fluxo linear de processamento dos dados, porém a necessidade de criação de múltiplos métodos de *callback* torna o código mais extenso e aumenta o grau de dificuldade de entendimento e, conseqüentemente, de manutenção.

Deve-se também ter cuidado com o processamento dos dados recebidos já que este não deve ser executado em um único ciclo pois dessa maneira o dispositivo poderia ficar um tempo longo sem poder processar eventos recebidos do sistema. Essa particularidade é contornada adicionando-se um temporizador e um contador de progresso para que o procedimento seja executado em passos, permitindo que o sistema operacional tenha seus

eventos tratados entre o processamento de dois blocos. Essa implementação pode ser observada na listagem 5.3.

```
1  static void processData( CApp * pApp )
2  {
3      if( pApp->processStatus < pApp->contentSize )
4      {
5          // ... Executa algum processamento com o bloco de dados.
6          pApp->processStatus += pApp->blockSize;
7          ISHELL_SetTimer( 0, processData, pApp );
8      } else
9      {
10         // Dados processados, cria o Menu.
11         pApp->createMenu( processedData );
12     }
13 }
```

Listing 5.3: Exemplo de código em BREW.

Nesse cenário, a inserção dos métodos de escalonamento colaborativo de rotinas permite simplificar a tarefa de desenvolvimento fazendo com que os fluxos individuais de dados sejam tratados de maneira seqüencial, simulando um ambiente de processamento síncrono porém atendendo aos requisitos do sistema ao evitar o processamento bloqueante. Uma implementação equivalente à apresentada acima está ilustrada no trecho de código da listagem 5.4.

```
1  function processContent()
2    socket.connect( host, port )
3    socket.write( "STRT" )
4    local contentSize = socket.receive( 4 )
5    local contentData = socket.receive( contentSize )
6
7    while not finished do
8      -- Executa algum processamento com o bloco de dados.
9      shell.processevents( )
10   end
11   createMenu( processedData )
12 end
```

Listing 5.4: Exemplo de código em LuaBREW.

Dessa maneira, toda a complexidade do gerenciamento do fluxo de dados através de objetos de *callback* concentra-se na implementação do escalonador, eliminando a necessidade desses elementos no código do serviço desenvolvido. O processamento bloqueante é aliviado através de chamadas explícitas ao método `shell.processevents()` permitindo que o desenvolvedor execute chamadas dentro de um ciclo sem impedir o processamento de eventos de maior prioridade pelo sistema. Além disso, o *Escalonador* encapsula os métodos de gerenciamento do fluxo de informação, permitindo centralizar o tratamento dos eventos de sistema, proporcionando um ambiente menos suscetível a erros.