4
Binding entre Charm++ e Lua

Após o estudo do primeiro sistema, sobre a integração de Charm++ e Lua através do framework de troca de mensagens, e analisando os resultados e deficiências do binding, planejamos a construção da segunda fase de implementação. Definimos que neste segundo sistema o uso da linguagem de script deveria estar mais integrado a Charm++ e à construção de chares. Para isso é necessário que os chares possam ser implementados em Lua, que a partir de Lua seja possível fazer chamadas assíncronas a métodos de outros chares, e que outros chares possam chamar assincronamente os métodos implementados em Lua sem a necessidade do programador escrever código em C++.

Como os chares poderão ser implementados em Lua, será possível a inclusão de uma funcionalidade na linguagem para manter o dinamismo de carga de código presente no sistema anterior. Lua possibilita a criação de diversas instâncias da máquina virtual, que são chamados de *estados*.

Na primeira integração, os estados podem ser criados e novos trechos de código carregados e executados a qualquer momento. Nosso objetivo é que esse dinamismo seja mantido e expandido para a implementação de chares, aumentando a flexibilidade do sistema.

PUC-Rio - Certificação Digital Nº 0611940/CA

4.1
Descrição

LuaCharm é uma extensão do Charm++ que oferece a possibilidade de que a implementação de um chare seja toda escrita em Lua, sem a necessidade de implementação de classes e métodos em C++. Também é oferecido um mecanismo para a carga dinâmica de código Lua e a execução deste código dentro dos chares. Essa extensão foi feita através de alterações no parser do arquivo de interface e no gerador de código.

Como Charm++ é um sistema estável e maduro, poucas mudanças são feitas em sua linguagem e no gerador de código. Isso é importante para que em futuras versões do sistema, o custo de migração de LuaCharm seja pequeno.

LuaCharm explora a provisão de estados Lua para que cada chare implementado em Lua, quando for instanciado, tenha um ambiente de execução

inalterado e independente de qualquer outro.

4.2 Arquivo de Interface

Para permitir que chares sejam implementados em Lua, e para que código Lua seja carregado e executado nos chares dinamicamente, foram feitas mudanças na linguagem de interface e no parser, descritas a seguir:

A principal mudança na linguagem de interface foi a inclusão do atributo *lua* para ser usado na declaração de chares, de modo semelhante ao que foi feito no binding anterior. Ao atributo deve se seguir o nome do arquivo lua que implementa o chare. Por exemplo:

```
chare [lua "luaChare.lua"] luaChare{
  entry luaChare();
  entry someLuaMethod();
}
```

O chare deve ter o atributo `[lua]` para que seja gerado o código do *façade* responsável pela execução de seus respectivos métodos em Lua.

Um façade (15) é um objeto que provê uma interface simplificada para algum código externo a ele. Esse padrão pode ser utilizado em várias situações, como para oferecer uma interface simples para o uso de uma biblioteca muito complexa ou para oferecer uma interface mais portátil, concisa e de fácil manutenção para um outro objeto.

O façade utilizado aqui é a implementação das classes dos chares em C++, e é essencial para a integração do sistema de execução de Charm++ com a nossa implementação do chare em Lua. Assim, para cada método do chare é gerado pelo parser um método para executar o seu correspondente em Lua, e para cada chare no módulo, é criada uma função de instanciação e exportação do mesmo para Lua.

Uma mudança feita no gerador de código foi a inclusão automática de um entry method, o *updateChare*, que é sempre criado e é usado para possibilitar a execução de um trecho de código Lua no chare. Esse código Lua pode servir para auxiliar na tomada de decisões no fluxo da aplicação, ou até para a mudança completa da implementação do chare em tempo de execução. Esse entry method não precisa ser declarado, e tem a seguinte assinatura:

```
entry void updateChare(int n, char code[n], int newState);
```

onde `n` é o tamanho da string de código (incluindo '\0'), `code` é a string que contém o código Lua a ser executado no estado e `newState` é um booleano

PUC-Rio - Certificação Digital Nº 0611940/CA

que indica se um novo estado Lua deve ser criado para substituir o estado que está sendo utilizado ou se o código deve ser executado no estado já existente. Se o código for executado no estado existente, ele terá acesso a todos os métodos e variáveis já presentes no estado Lua, e poderá fazer uma atualização ou modificação no conteúdo do mesmo. Na criação de um novo estado, o antigo é destruído e tudo que estiver nele é perdido. O código é executado em um estado novo e esse passa a ser a implementação do chare. Neste segundo caso, é importante que o código executado contenha a implementação de todos os métodos do chare, caso contrário haverá uma inconsistência na implementação do chare que podem ocasionar erros de execução.

Deve ser ressaltado que a possibilidade de serem feitas alterações na implementação em tempo de execução, ao mesmo tempo que é uma funcionalidade poderosa, é também perigosa, podendo colocar em risco o funcionamento da aplicação. Além disso, podem ocorrer problemas de segurança devido a essa operação, mas não estamos tratando destes possíveis problemas neste trabalho.

PUC-Rio - Certificação Digital Nº 0611940/CA

4.3 Arquivo Lua

O arquivo Lua que implementa o chare é definido na interface, como visto anteriormente.

Para a implementação do chare, é importante que o código Lua correspondente exporte uma tabela em uma variável global com o mesmo nome do chare. Esta tabela deve conter uma função chamada *ckNew*, que é chamada na inicialização do chare, e também para cada entry method declarado no arquivo de interface deve existir uma função Lua de mesmo nome dentro da tabela. Por exemplo, um arquivo de implementação do chare *luaChare* descrito acima, deve ter uma implementação com o seguinte esqueleto:

```
luaChare = {
  ckNew = function()
      -- lua code
    end,
  someLuaMethod = function()
      -- lua code
    end
}
```

Como no sistema anterior, duas bibliotecas são exportadas para Lua antes da execução do construtor. A primeira biblioteca é a *ck*, que contém as funções *printstr, mype* e *numpes*, de funcionalidade idêntica à descrita no capítulo 3. A

segunda biblioteca se chama *charm*, e contém funções construtoras para todos os chares presentes no módulo.

Para que um chare seja instanciado a partir de Lua, é feita uma chamada para a função `charm.chName()`, onde `chName` é o nome do chare que se deseja instanciar. Essa chamada retorna um proxy para o chare, que pode ser usado para fazer chamadas assíncronas dos métodos do chare.

Como descrito no capítulo 2, um dos tipos de objeto presentes em Charm++ é o de variáveis *readonly*. Essas são variáveis globais inicializadas quando um mainchare é criado e que não podem ter seus valores alterados durante a execução. Uma das utilidades das variáveis globais é criar uma referência para um chare.

O sistema atual de integração com Lua limita a instanciação de variáveis globais, visto que elas devem ser inicializadas através de código C++. Por isso, somente uma variável global é permitida, que é uma referência para o mainchare. Esse handle deve ser declarado no arquivo de interface, dentro do mainchare. Por exemplo:

```
mainchare[lua "main.lua"] main {
  entry main();
  readonly CkChareID mainHandle;
}
```

CkChareID é o tipo usado para declarar um handle para um chare. No código acima um handle global para o mainchare é criado com o nome *mainHandle* e pode ser acessado de qualquer chare implementado em Lua através da variável `main.mainHandle`.

PUC-Rio - Certificação Digital Nº 0611940/CA

4.4 Parser - Implementação

O parser do arquivo de interface é implementado utilizando as ferramentas *flex* (16) e *bison* (17).

Flex é uma ferramenta utilizada para a geração de um analisador de padrões léxicos em texto. Ele lê um arquivo de descrição dos padrões a serem reconhecidos e gera um código em C que faz o processamento léxico do texto.

Bison é um gerador de parser que converte uma gramática livre de contexto em um parser LALR(1) ou GLR. Bison é responsável pela análise sintática e por montar a árvore sintática abstrata (AST), que será usada pelo gerador de código.

O parser é composto de um arquivo de descrição léxica, um arquivo de descrição sintática e quatro arquivos contendo a implementação do gerador de

código, totalizando aproximadamente 5900 linhas de código. Após as mudanças do primeiro binding, as alterações provocaram um aumento de 250 linhas no parser e 1050 linhas do binding. Na implementação do LuaCharm, foram adicionadas ao parser aproximadamente 1400 linhas de código. Como no LuaCharm a gerência do interpretador do script foi embutida dentro do código gerado pelo parser, não foi necessária a inclusão de código externo ao parser, como no primeiro sistema.

4.4.1 Façade

Como afirmado anteriormente, o façade gerado pelo parser é a declaração e implementação das classes em C++ dos chares com o atributo lua no arquivo de interface. Além da implementação das classes, são gerados métodos para a criação de chares e para a indexação de arrays e grupos.

Lua disponibiliza uma API em C com um conjunto de funções para que o código C possa interagir com Lua e vice-versa. Ela contém funções para lidar com variáveis em Lua, chamar funções, executar techos de código e registrar funções C em Lua, entre outras. Uma função C pode ser registrada na máquina virtual de Lua e ser chamada pelo script de forma transparente, da mesma maneira que uma função implementada em Lua.

Um componente muito importante na comunicação entre Lua e C é a pilha virtual. Quase todas as operações da API operam em valores presentes na pilha, e toda a troca de dados entre as duas partes é feita por essa pilha. Algumas das principais funções para manusear dados da pilha são as funções do tipo lua_toXXX, lua_isXXX e lua_pushXXX, onde XXX é o nome de um dos tipos de dados presentes em Lua.

No apêndice B apresentamos um exemplo de dois chares simples, e o código gerado pelo parser para o binding.

4.5 Balanceamento de Carga

Arrays em Charm++ são fortemente integrados com o balanceador de carga, com todos os seus elementos podendo ser migrados de processador quando um algoritmo de balanceamento de carga está ativo na aplicação. Para que seja possível que um chare implementado em Lua seja migrado, são necessárias duas coisas. Primeiramente, é necessário que, após a migração, o estado Lua seja recriado, as variáveis e bibliotecas do sistema sejam novamente exportadas e o seu script recarregado. Para isso, os chares migráveis dispõem

PUC-Rio - Certificação Digital Nº 0611940/CA

de um construtor especial que é usado para criar o objeto no evento de uma migração.

A segunda necessidade é que variáveis do ambiente Lua sejam recuperadas após a migração. Para isso, é necessário que o usuário exporte seus dados em dois métodos que são chamados durante o processo de migração, o *pack* e o *unpack*.

O framework PUP, conforme descrito brevemente no capítulo 2, é um framework existente em Charm++ que permite que o usuário mantenha dados do objeto após a sua migração. Para a exportação de dados em Lua, quando o balanceador de carga executa o método *pup* do chare, ele chama o método pack da implementação do chare no script, e este deve retornar uma string. Essa string será enviada para o novo objeto já migrado e quando chegar a ele, após o estado ser reconstruído, o método do script unpack é chamado, recebendo como parâmetro a mesma string, que será usada para recuperar os dados.

Lua possui sete tipos básicos, sendo que todos os objetos em Lua são valores de primeira classe. Os tipos em Lua são: string, número, tabela, booleanos, userdata, nil, co-rotina e função. Tabelas, strings, nil e números podem facilmente ser serializados, conforme descrito em (20). Para a serialização dos outros tipos, é possível utilizar a extensão da linguagem Lua descrita em (21), na qual os valores em Lua são descritos em tabelas, que por sua vez podem ser transformados em strings. Para funções implementadas em C, é necessário que elas sejam carregadas novamente e sua referência seja feita através de variáveis, que deverão estar presentes no novo ambiente de execução.

No próximo capítulo demonstramos como um array pode ser implementado para utilizar o mecanismo de balanceamento de carga oferecido por Charm++ e integrado no binding.

PUC-Rio - Certificação Digital Nº 0611940/CA

4.6 Geração de código genérico

Uma opção pensada para a integração entre Lua e Charm++ é o uso de apenas um método que é capaz de entender a função que será chamada em Lua. A vantagem deste modelo é o maior uso do dinamismo de Lua, já que os métodos do script que serão chamados não ficariam restritos aos cadastrados na interface. A desvantagem encontrada nesta abordagem é que, por usar apenas um método, o uso do script ficaria menos transparente, sendo mais difícil migrar entre uma implementação em Lua e uma em C++.

4.7 Exemplos

4.7.1 Exemplo de aplicação utilizando chares implementados em Lua

A seguir apresentamos um exemplo de uma aplicação simples, que utiliza chares implementados em Lua. Esta aplicação somente cria um chare principal, que por sua vez cria um novo chare e chama um método seu, e este segundo chare retorna uma mensagem para o primeiro com o resultado de uma soma.

```
//////////////////////////////////////
// Arquivo de interface: prog.ci
// Este arquivo implementa a interface do programa Charm++
mainmodule Prog {
  // mainchare implementado pelo arquivo main.lua
  mainchare[lua "main.lua"] main {
    entry main();
    entry void setResult(int res);
    readonly CkChareID mainHandle;
  };
  // chare implementado pelo arquivo calc.lua
  chare [lua "calc.lua"] calc {
    entry calc();
    entry void putWork(int i1, int i2);
  };
};
//////////////////////////////////////
// Arquivo de implementação: main.lua
// Este arquivo implementa o chare main,
// descrito no arquivo de interface acima
main = {
  ckNew = function()
      calc = charm.Calc();
      calc.putWork(2, 3);
    end,
  setResult = function(res)
      ck.printstr("End of execution");
    end
//////////////////////////////////////
// Arquivo de implementação: calc.lua
// Este arquivo implementa o chare calc,
// descrito no arquivo de interface acima
calc = {
```

PUC-Rio - Certificação Digital Nº 0611940/CA

```
  ckNew = function()
      -- vazio
    end,
  putWork = function(i1, i2)
      main.mainHandle.setResult(i1 + i2);
    end
```

4.7.2 Exemplo de carga dinâmica de código

A seguir mais um exemplo de uma aplicação simples, executando código Lua dinamicamente em um estado antigo e em um estado novo.

```
//////////////////////////////////////
// Arquivo de interface: pgm.ci
mainmodule Prog {
  mainchare[lua "main.lua"] main {
    entry main();
  };
  chare [lua "chare2.lua"] chare2 {
    entry calc();
  };
};
//////////////////////////////////////
// Arquivo de implementação: main.lua
// Este chare instancia um chare do tipo calc e chama o seu
// método updateChare, passando strings contendo código em
// Lua que serão executados
main = {
  ckNew = function()
      calc = charm.Calc();
      calc.updateChare("value = 5", 0);
      calc.updateChare([[
        if value then
          ck.printstr("Estado antigo. Valor:"..tostring(value));
        else
          ck.printstr("Estado novo");
        end]], 0);
      calc.updateChare([[
        if value then
```

PUC-Rio - Certificação Digital Nº 0611940/CA

```
          ck.printstr("Estado antigo. Valor:"..tostring(value));
        else
          ck.printstr("Estado novo");
        end]], 1);
    end,
}
////////////////////////////////////
// Arquivo de implementação: calc.lua
calc = {
  ckNew = function()
      -- vazio
    end
```

PUC-Rio - Certificação Digital Nº 0611940/CA