5
Exemplo de Aplicação

Para demonstrar os recursos oferecidos pelo LuaProxy para a adaptação dinâmica, esta dissertação apresenta o *LuaEmpire*, um jogo de estratégia espacial bem simples, cujo objetivo é conquistar os planetas de todos os adversários.

Os participantes usam o LuaProxy para repassarem os comandos dos jogadores (por exemplo, movimentação de frotas) a um servidor compartilhado. O servidor armazena em uma grande tabela o estado global da galáxia, que se modifica com as ações dos jogadores e pela passagem tempo (como o progresso da construção de uma nave espacial). O servidor transmite as mudanças no seu estado a todos os jogadores periodicamente.

A IDL do jogo se encontra na Figura 5.1. Para jogar, a pessoa deve criar um proxy para uma instância da interface *Server* (como mostrado nas Figuras 5.2 e 5.3), e chamar o método *login*, que requer o nome/apelido do jogador (*nick*) e um objeto de *callback* para receber as atualizações. O objeto de *callback* deve implementar a interface *Client*.

O método *login* retorna uma string contendo o estado inicial do jogo (posição dos planetas, frotas, etc.). O cliente converte a string em uma tabela (função *loadstring* de Lua), e a usa para armazenar as modificações enviadas do servidor pelo método *update* do objeto de *callback*. Os métodos *createShip*, *moveFleet* e *attack* são auto-explicativos.

5.1
Procedimento de Adaptação

Como mostra a Figura 5.2, os parâmetros de QoS usados são a largura de banda (*bandwidth*) e o retardo (*delay*). Nos experimentos executados, os

```
module LuaEmpire
{
   interface Client
   {
      typedef sequence<octet> OctetSeq;
      oneway void update (in OctetSeq changes);
   };

interface MigratingClient : Client
   {
      oneway void migrate (in string offerId);
   };

   interface Server
   {
      string login (in string nick, in Client callback);

      long createShip (in long planetId, inout long fleetId);
      void moveFleet (in long id, in double x, in double y);
      void attack (in long id, in long targetId);

      void loadState (in string state);
      void clock ();
   };
};
```

Figura 5.1: IDL do LuaEmpire

```
p = LuaProxy ("LuaEmpire::Server", "delay < 40, high",
   "bandwidth > 80", impl)
univ = p:login ("Joao", callback)
```

Figura 5.2: Uso do LuaProxy

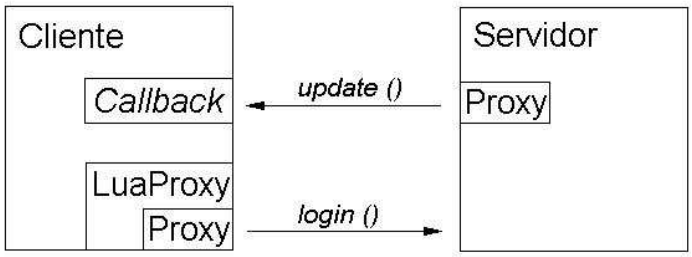

Figura 5.3: Comunicação cliente-servidor no LuaEmpire.

```
if event == "bandwidth_decrease" then
   if not self.callback.CompressUpdates then
      -- Instrui cliente e servidor a compactarem dados
      self._proxy:setUpdateCompression (self.nick, true)
      self.callback.CompressUpdates = true

      -- Relaxa atributo do QoS descriptor para 40
      return true, 40
   else
      return false
   end
end
```

Figura 5.4: Adaptação ao evento *bandwidth_decrease*

```
function callback:update (changes)
   if self.CompressUpdates then
      changes = lzo.decompress (changes)
   end

   self:oldUpdate (changes)
end
```

Figura 5.5: Adicionando suporte à compactação

servidores disponibilizam dstubs que interceptam o método *login*, de modo a capturarem o objeto de *callback* do jogador. Antes de chamar o método *login* do objeto remoto, o dstub acrescenta o método *migrate* ao objeto de *callback*, fazendo-o implementar a interface *MigratingClient*. Esse método é usado na troca de servidor (discutido mais adiante).

Para se adaptar à falta de largura de banda, a implementação de QoS (parâmetro *impl* do construtor) transfere um ustub ao servidor para estendê-lo com o método *setUpdateCompression*. Esse método encapsula o proxy para o objeto de callback no lado do servidor, de modo a ligar/desligar a compactação das atualizações. O método *update*, pelo qual se recebem essas atualizações no lado do cliente, é encapsulado para que aceite dados normais ou compactados. A nova versão desse método (Figura 5.5) usa o campo especial *CompressUpdates* para determinar se a compactação de dados está ativa. Em caso afirmativo eles são descompactados antes de serem passados adiante, o que é feito de maneira transparente ao cliente. A arquitetura modificada da aplicação é mostrada na Figura 5.6.

A implementação de QoS reage ao retardo diminuindo a freqüência com

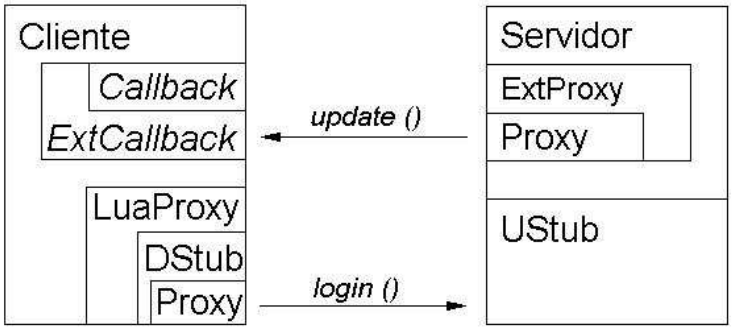

Figura 5.6: Arquitetura modificada do LuaEmpire.

```
function delay_increase:adapt (luaproxy)
   if client.Callback.FrameSkip == 1 then
      client.Callback.FrameSkip = 2

      -- Requer retardo de 60. Sistema pode ser considerado
      -- recuperado quando retardo < 30.
      return true, 60, 30
   else
      return false
   end
end
```

Figura 5.7: Adaptação ao evento *delay_increase*

que o cliente redesenha sua interface gráfica. Para tanto, a rotina de adaptação simplesmente ativa um *flag* que instrui o objeto de callback a atualizar seu estado interno normalmente, mas redesenhar a interface com o usuário somente a cada duas atualizações. Essa interação entre a rotina de adaptação e o objeto de *callback* é possível devido ao fato de ambos terem sido desenvolvidos pelo cliente para trabalharem conjuntamente.

Caso a adaptação não seja possível, o LuaProxy usa seu método *_select* para escolher outro servidor. Nesta aplicação, o dstub também redefine esse método, de modo que o estado do jogo seja automaticamente transferido para o novo servidor (método *loadState*). Um dos jogadores (o que tiver o menor id) é eleito coordenador do processo de migração. Ele é responsável por encontrar o novo servidor no *trader*, transferir o estado do jogo para ele e difundir o novo servidor aos demais jogadores.

Isso é possível porque o dstub estende os objetos de *callback* com o método *migrate*, definido na interface *MigratingClient*. Ele recebe e armazena

```
function callback:update (changes)
   local t = assert (loadstring ("return " .. changes)) ()

   -- Atualiza universo.
   table.foreach (t, function (id, object)
      self.Universe [id] = object
   end)

   -- Flag FrameSkip controla o redesenho alternado de quadros.
   if not self.FrameSkip or math.mod (self.FrameId, 2) == 0 then
      -- Redesenha tela do jogador.
   end

   self.FrameId = self.FrameId + 1
end
```

Figura 5.8: Método *update* do objeto de callback.

o id da nova oferta de serviço, fazendo com que o jogador migre para o servidor especificado.

5.2 Roteiro de Testes

Foram criados dois clientes e dois servidores, sendo estes configurados com diferentes níveis de recursos disponíveis inicialmente. A seguir, os clientes usaram o LuaProxy para se logarem no servidor, e, depois de alguns segundos os níveis de retardo e largura de banda foram reduzidos manualmente, de modo a simular esses eventos.

Ao usarem o LuaProxy pela primeira vez, ambos os clientes selecionaram a mesma oferta de serviço, pois um dos servidores inicialmente dispunha de mais recursos. Ao criar uma referência ao servidor, o dstub foi automaticamente baixado e mesclado ao proxy. Interceptando o método *login*, o dstub capturou o objeto de *callback*, acrescentou o método *migrate* e encapsulou o *update*, conforme descrito na seção anterior.

Ao ser notificado do evento *bandwidth_decrease*, o dstub ativou o procedimento de compactação de dados, usando a biblioteca LuaLZO [16]. Como o volume de dados nesse exemplo foi muito pequeno, a compactação não diminuiu o volume de dados transferido, mas comprovou que a estratégia de adaptação entrou em vigor. Em um jogo com vários sistemas solares e

jogadores, as strings com as atualizações poderiam ser grandes o bastante para que o uso da compactação fizesse diferença.

O evento *delay_increase* ativou como o esperado a alternância de quadros. Como a interface gráfica ainda não está disponível, o método que o objeto de *callback* usa para redesenhar a tela do usuário simplesmente imprime as informações atualizadas na forma de texto.

Finalmente, para demonstrar a troca de servidores, a largura de banda do que estava em uso foi reduzida a zero, forçando o LuaProxy a disparar o processo de migração. O primeiro jogador a logar selecionou o outro servidor e usou o método *migrate* para transmitir o *id* da nova oferta de serviço ao outro participante. Seu objeto de *callback* armazenou esse id, e a versão redefinida pelo dstub do método *_adapt* migrou para o novo servidor.

É interessante ressaltar que o único propósito dessa aplicação é o de testar o comportamento do LuaProxy em uma situação *simulada*. Por conta disso, e por falta de tempo para desenvolver um exemplo mais completo, alguns detalhes não foram tratados. Por exemplo, quando a rotina de adaptação ativa a compressão de dados, não é possível saber ao certo se a próxima atualização do servidor virá compactada ou não. Isso depende de qual método chega ao servidor primeiro: o *setUpdateCompression* ou o próximo *clock*. Em aplicações comerciais seria necessário examinar a string com a atualização para detectar se ela contém texto ou caracteres binários.