6

Exemplos de Uso

Para ilustrar o uso das ferramentas do LOAF na adaptação dinâmica de sistemas, utilizaremos uma aplicação composta por três componentes, um componente produtor de dados e dois componentes que processam os dados produzidos, que são recebidos através de eventos. Após o processamento de cada evento, o componente processador envia um evento ao servidor solicitando o evento seguinte. Os eventos são numerados de forma que o servidor só envia eventos na primeira solicitação de cada evento, ignorando as demais solicitações de eventos já enviados. Isso impede que duplicadas de eventos sejam processadas pelos clientes. A figura 6.1 ilustra a estrutura da aplicação de exemplo utilizada. Cada componente é instalado e instanciado num servidor de componentes LuaCCM diferente. Além dos componentes da aplicação, um outro processo é iniciado com um console LuaOrb através do qual os componentes são instalados, instanciados, configurados e manipulados através das ferramentas do LOAF. Esse console interativo de comandos poderia ser substituído por um ou mais processos autônomos (agentes), responsáveis por comandar adaptações no sistema com base em políticas e regras que norteassem o comportamento geral do sistema ou de componentes individualmente.

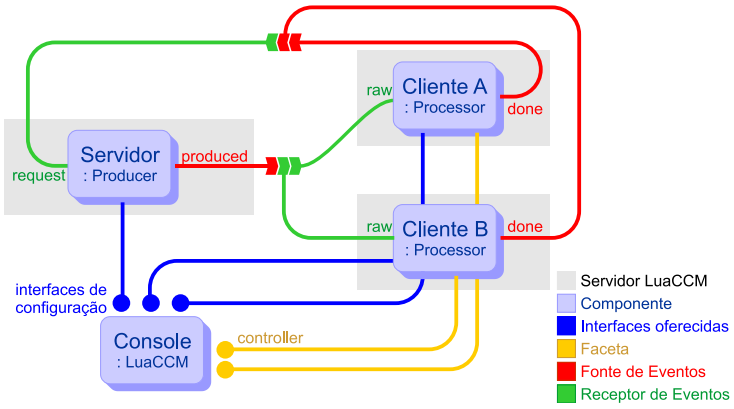

Figura 6.1: Aplicação de fluxo de eventos

A definição dos componentes da aplicação é ilustrada na figura 6.2. Cada componente Producer envia dados produzidos através do publicador de eventos produced e recebe solicitações de novos dados através do receptor de eventos request. Além disso, os componentes Producer definem a faceta controller que é utilizada para iniciar, encerrar e suspender a produção dos dados. Os dados são

processados através de componentes Processor, que recebem dados através do receptor de eventos raw e sempre que terminam o processamento de um evento enviam a solicitação de novos dados através do emissor de eventos done. Ambos os componentes são instrumentados através da interface oferecida Instrumented, que permite definir a velocidade de cada componente (*i.e.*, velocidade de produção ou processamento de dados), assim como um nome de identificação de cada componente.

```
1  module eventflow {
2    interface Instrumented {
3      attribute string name;
4      attribute float speed;
5    };
6
7    interface Controlable {
8      void start();
9      void stop();
10     void pause();
11     void continue();
12   };
13
14   eventtype SerialEvent {
15     public long seq_no;
16   };
17
18   component Producer supports Instrumented {
19     provides Controlable controller;
20     consumes SerialEvent request;
21     publishes SerialEvent produced;
22   };
23   home ProducerHome manages Producer {};
24
25   component Processor supports Instrumented {
26     consumes SerialEvent raw;
27     emits SerialEvent done;
28   };
29   home ProcessorHome manages Processor {};
30 };
```

Figura 6.2: Definição dos componentes da aplicação de fluxo de eventos em IDL 3.0

A figura 6.3 ilustra o roteiro de implantação da aplicação de exemplo. Inicialmente o pacote de componentes é instalado nos contêiners armazenados na lista deployer.containers, que são previamente criados através da arquitetura de implantação do modelo CCM. A operação install_home do contêiner recebe o identificador do pacote (no LuaCCM o identificador é o endereço do arquivo Lua que implementa o pacote) e o ponto de entrada para a implementação do componente a ser instalado (no LuaCCM o ponto de entrada é uma variável global que contém a implementação e definição do *home* do componente). Em seguida, os componentes são criados juntamente com os manipuladores do LOAF, que são utilizados na definição de atributos e no estabelecimento das conexões entre suas portas. Por fim, a aplicação é iniciada com a chamada da operação start da faceta controller do componente produtor.

```
producers = luaorb.narrow(deployer.container[1]:install_home(
  "luaccm/demo/eventflow/package.lua",
  "Producer", {}))
processorsA = luaorb.narrow(deployer.container[2]:install_home(
  "luaccm/demo/eventflow/package.lua",
  "Processor", {}))
processorsB = luaorb.narrow(deployer.container[3]:install_home(
  "luaccm/demo/eventflow/package.lua",
  "Processor", {}))

server = loaf.handler(producers:create())
client1 = loaf.handler(processorsA:create())
client2 = loaf.handler(processorsB:create())

server.name  = "Servidor"
client1.name = "Cliente A"
client2.name = "Cliente B"

server.request = client1.done
server.request = client2.done
client1.raw = server.produced
client2.raw = server.produced

server.controller:start()
```

Figura 6.3: Roteiro de implantação dos componentes da aplicação de exemplo

Com base na aplicação ilustrada acima, os recursos do LOAF serão utilizados para adaptar dinamicamente a aplicação, de forma a inserir três novos recursos: sincronização de fluxo, depuração distribuída e replicação passiva. Tanto a aplicação de exemplo como as adaptações apresentadas são baseadas nos exemplos de uso do *middleware* Comet apresentado em [14].

Cada recurso é definido através de um protocolo que utiliza papéis para aplicar alterações nos componentes. Além disso são utilizados manipuladores no estabelecimento de novas conexões através das quais novas interações são feitas. As interfaces das portas adicionadas pelos papéis definidos neste capítulo são apresentadas na figura 6.4 A seguir, são apresentados os detalhes de cada uma das adaptações realizadas.

6.1

Sincronização de Fluxo

Neste exemplo, será definido um protocolo cujo objetivo é sincronizar o fluxo de eventos entre os componentes, de forma que nenhum processador de dados seja capaz de processar uma quantidade de eventos maior que os demais. Para tanto, serão definidos mecanismos de análise e regulagem de fluxo que serão adicionados ao sistema através de adaptações dinâmicas. O critério utilizado para definir a vazão dos eventos é o tempo de processamento de um evento. Quando a vazão dos eventos ultrapassa um determinado valor

```
module eventflow {

  module flowsync {
    interface Rateable {
      void set_rate(in double rate);
    };
    interface Limited {
      attribute double value;
    };
  };

  module distdebug {
    interface Inspectable {
      string evaluate(in string expression);
      string execute(in string code);
    };
    interface Pauser {
      boolean locked();
    };
  };
};
```

Figura 6.4: Interfaces dos papéis de sincronização de fluxo e depuração distribuída.

de limite, então é necessário ativar o mecanismo de regulagem do fluxo.

A figura 6.5 mostra a definição de um papel que permite medir a vazão dos eventos através de um interceptador associado ao receptor de eventos raw de um componente Processor. O papel também define um receptáculo onde a interface do mecanismo de regulagem de fluxo deve ser acoplada e uma faceta utilizada para definir o valor de vazão limite adotado. Sempre que um evento é recebido, a hora atual do sistema é obtida e armazenada na tabela que identifica a requisição. Após o término do processamento do evento, a vazão é calculada com base na hora atual. Se o tempo calculado for maior que o valor de limite atual, então uma nova taxa de vazão é definida no mecanismo de regulagem de fluxo através do receptáculo regulator. Dessa forma, todos os eventos subseqüentes são enviados a todos os processadores na mesma taxa.

Para completar o algoritmo de sincronização, é necessário definir o mecanismo de regulagem. A figura 6.6 mostra a definição de um papel utilizado para controlar a vazão de saída do servidor. Esse papel faz com que o envio de cada evento seja interceptado antes do evento ser repassado aos processadores. Nesse momento, a taxa de envio de eventos é calculada de acordo com a hora de envio do último evento. Se os eventos estão sendo enviados numa taxa maior do que a definida através da faceta rater, então um atraso é inserido de forma que o evento não seja enviado tão rapidamente.

Note que através dos papeis definidos anteriormente, é possível inserir o recurso de análise e regulagem de fluxo nos componentes da aplicação. Contudo, também é necessário estabelecer as conexões adequadas. Para tanto é definido um protocolo através de um objeto Lua, como ilustrado na figura 6.7. O protocolo de sincronização é composto apenas da operação sync,

```
FlowWatcher = loaf.Role {
  provides = {
    limit = {
      interface = "eventflow::flowsync::Limited",
      code = [[ { value = 0.05 } ]],
    },
  },
  uses = {
    regulator = {
      interface = "eventflow::flowsync::Rateable",
    },
  },
  before = {
    raw = {
      code = [[
        function(self, request)
          request.start_time = get_time()
        end
      ]],
    },
  },
  after = {
    raw = {
      code = [[
        function(self, request)
          local now = get_time()
          local time_spent = now - (request.start_time or now)

          local limit = request.context:get_executor("limit")
          if time_spent > limit.value then
            local regulator =
              request.context:get_connection_regulator()
            if regulator then
              regulator:set_rate(time_spent)
              end
            end
          end
        end
      ]],
    },
  },
}
```

Figura 6.5: Papel de análise de fluxo de evento

```
FlowRegulator = loaf.Role {
  before = {
    produced = {
      code = [[
        function(self, request)
            local now = get_time()
            local last = self.last or now
            self.last = now

            local time_spent = now - last
            if request.context.rate then
              if time_spent < request.context.rate then
                sleep(request.context.rate - time_spent)
              end
            end
        end
      ]],
    },
  },
  provides = {
    rater = {
      interface = "eventflow::flowsync::Rateable",
      code = [[{
        set_session_context = function(self, context)
          self.context = context
        end,
        set_rate = function(self, rate)
          self.context.rate = rate
        end,
      }]],
    },
  },
}
```

Figura 6.6: Papel de regulagem de fluxo de evento

que recebe o servidor e o cliente que devem ser sincronizados. Para tanto, o papel de regulador é aplicado ao servidor responsável por produzir os eventos e o papel de analisador é aplicado ao cliente. Além disso, uma conexão é estabelecida entre a faceta de controle do mecanismo de regulagem do servidor e o receptáculo regulator do cliente. Por exemplo, a chamada FlowSyncProtocol:sync(server, client2) ativa automaticamente o mecanismo de sincronização de fluxos entre o Cliente B e o Servidor da aplicação de fluxo de eventos, resultando na estrutura ilustrada na figura 6.8.

6.2

Depuração Distribuída

Além de adicionar um novo comportamento aos componentes, um papel pode interagir com a implementação do componente através das operações get_executor e get_interceptor do objeto de contexto. Isso permite que o pa-

```
1  FlowSyncProtocol = {
2    sync = function(self, server, client)
3      FlowRegulator:assign(server)
4      FlowWatcher:assign(client)
5      client.regulator = server.rater
6    end,
7  }
```

Figura 6.7: Protocolo de sincronização de fluxo de evento

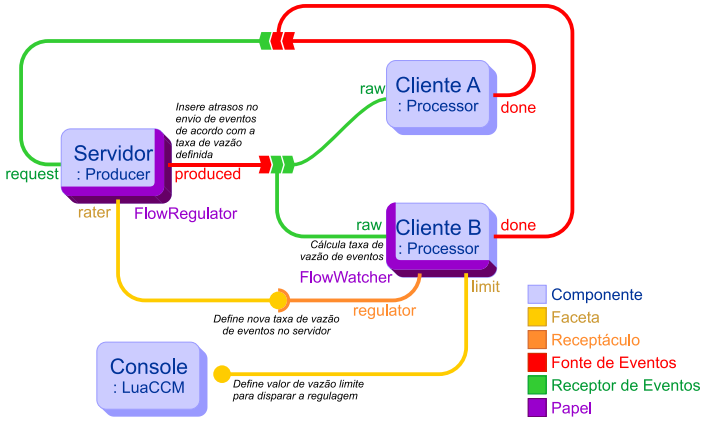

Figura 6.8: Aplicação de fluxo de eventos com sincronização aplicada por papéis.

pel possa consultar ou até mesmo alterar o estado interno do componente. Para exemplificar esse recurso, será descrito um protocolo para depuração de aplicações LuaCCM através de mecanismos que permitam inspecionar o estado interno do componente, assim como alterar a implementação do componente através de trechos de código Lua que são executados dentro do componente. O protocolo de depuração distribuída, também define recursos para inserir paradas de execução (*break points*) antes ou depois de interações nas portas do componente.

Inicialmente é definido um papel que permite inspecionar o estado interno da implementação do componente através de expressões escritas em Lua que são executadas dentro do componente. A figura 6.9 mostra a definição de um papel que adiciona a faceta evaluator, que oferece a operação evaluate e execute. Essas operações recebem um trecho de código Lua que é executado dentro do componente e o seu resultado é enviado de volta como uma string. O objeto de contexto do componente é disponibilizado através da variável local context. Através desse objeto, é possível inspecionar e alterar todos os segmentos do executor do componente, inclusive os interceptadores associados às portas.

Outro recurso importante na depuração de sistemas, é a inserção de paradas que permitam interromper temporariamente a execução do sistema em pontos definidos. Este recurso pode ser adicionado através de interceptação. A figura 6.10 exemplifica a definição de um papel que pode ser aplicado para inserir uma parada de execução antes de uma interação na porta port de um componente. Para tanto, é definido o receptáculo pauser e um interceptador,

```
Inspectable = loaf.Role {
  provides = {
    evaluator = {
      interface = "eventflow::distdebug::Inspectable",
      code = [[{
        set_session_context = function(self, context)
          self.context = context
        end,
        evaluate = function(self, expression)
          return self:execute("return " .. expression)
        end,
        execute = function(self, code)
          local getter, message = loadstring(
            "return function(context) " .. code .. " end"
          )
          if getter
            then return tostring(getter()(self.context))
            else return message
          end
        end,
      }]],
    },
  },
}
```

Figura 6.9: Papel de inspeção de componentes

que sempre que é acionado, mantém a execução parada até que o objeto acoplado ao receptáculo pauser informe que a execução deve prosseguir. Note porém, que a definição do papel varia ligeiramente de acordo com a porta e o momento em que a execução deva ser interrompida. Por essa razão, um papel diferente deve ser criado especificamente para cada ponto de parada que se deseje aplicar a um componente.

Para utilizar os recursos de depuração distribuída, é definido um protocolo através de uma classe LOOP, como ilustrado na figura 6.11. Cada instância do protocolo é criada para inspecionar um único componente que é definido no parâmetro do construtor. No próprio construtor do protocolo, o papel de inspeção é aplicado ao componente, permitindo assim inspecionar o estado interno do componente através da faceta evaluator. A operação evaluate do protocolo devolve uma string contendo o valor do campo dado pelo parâmetro field no objeto que implementa o segmento especificado pelo parâmetro segment.

Já o recurso de pontos de parada são utilizados através da operação add_break, que cria um papel de parada de acordo com a porta e o momento em que a execução deva ser suspensa, resultando num papel similar ao da figura 6.10. Em seguida, o papel é aplicado ao componente sendo inspecionado, fazendo com que este se adapte às novas interações. Só então, é que uma conexão é estabelecida entre o receptáculo do componente adaptado e um objeto criado pelo próprio protocolo, que é capaz de informar ao componente se a trava da parada de execução ainda está ativa. Quando a trava for removida

```
BreakPoint = loaf.Role {
  uses = {
    pauser = {
      interface = "eventflow::distdebug::Pauser",
    },
  },
  before = {
    port = {
      code = [[
        function(self, request)
          local pauser = request.context:get_connection_pauser()
          if pauser then
            while pauser:locked() do
              sleep(1)
            end
          end
        end
      ]],
    },
  },
}
```

Figura 6.10: Exemplo de papel de inserção de ponto de parada

através da operação continue, a execução continuará normalmente. A figura 6.12 mostra a estrutura da aplicação de fluxo de eventos após aplicação do protocolo para inserir um ponto de parada na fonte de eventos done do componente Cliente B, além de inspecioná-lo juntamente com o componente Servidor.

Uma das limitações do protocolo de depuração distribuída apresentado é devido ao console do LuaOrb utilizado não permitir a execução de duas operações concorrentemente, a menos que uma esteja aguardando o resultado de uma chamada remota. Por esta razão, a implementação do ponto de parada não pode fazer com que o processo durma até que uma operação seja executada, pois isso travaria o tratamento de operações pelo servidor indefinidamente.

6.3 Replicação Passiva

Além das alterações numa única instância feita através de protocolos, os papeis do LOAF também permitem realizar adaptações mais abrangentes através do recurso de alteração em níveis do LuaCCM. Por exemplo, suponha que haja num servidor diversos componentes processadores que recebem eventos de um componente produtor. Suponha também que se deseje introduzir um mecanismo de replicação nos componentes processadores, fazendo com que cada evento recebido por um componente, seja retransmitido através de um publicador de eventos adicional para um componente de réplica. Neste caso, um papel como o ilustrado na figura 6.13 pode ser aplicado a todas as instâncias de um componente, fazendo com que todos os componentes se

```
oo.class.luaccm.demo.debugger.DebugProtocol {

  constructor = function(self, inspected)
    self.inspectable:assign(inspected)
    self.set_private("inspected", inspected)

    -- Inicializacao de atributos
  end,
  private = {
    inspectable = Inspectable,

    -- Demais operacoes privadas
  },
  public = {
    evaluate = function(self, segment, field)
      return self.inspected.evaluator:evaluate(
        "context:get_executor('" .. segment .. "')." .. field
      )
    end,

    add_break = function(self, point, moment)
      local id = moment .. point
      local break_point = loaf.Role {
        uses = {
          pauser = { interface = "::debugger::Pauser" },
        },
        [moment] = {
          [point] = {
            code = [[
              function(self, request)
                local pauser = request.context:get_connection_pauser()
                if pauser then
                  while pauser:locked() do
                    sleep(1)
                  end
                end
              end
            ]],
          },
        },
      }
      break_point:assign(self.inspected)
      self.inspected.pauser = self:create_pauser(id)
      self.lock[id] = true
      self.breaks[id] = break_point
    end,

    continue = function(self, point, moment)
      self.lock[moment .. point] = false
    end,

    -- Demais operacoes do protocolo
  },

}
```

Figura 6.11: Parte da implementação do protocolo de depuração distribuída.

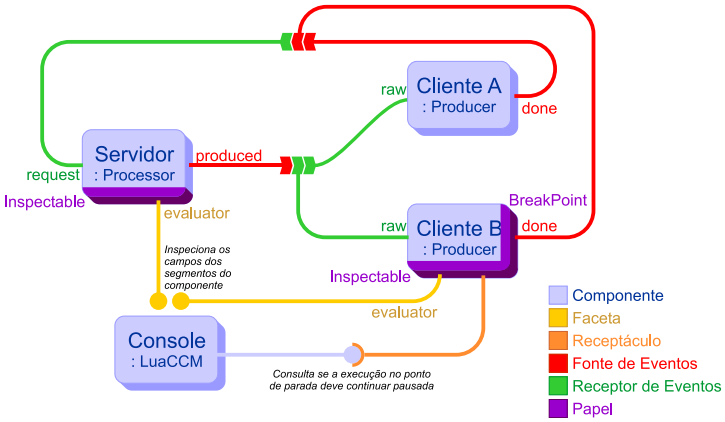

Figura 6.12: Aplicação de fluxo de eventos com depuração distribuída aplicada por papéis.

tornem automaticamente capazes de se acoplarem a uma réplica que receberá todos os eventos enviados ao componente original.

```
1  Replicator = loaf.Role {
2    publishes = {
3      copied = {
4        event = "eventflow::SerialEvent",
5      },
6    },
7    before = {
8      raw = {
9        code = [[
10         function(self, request)
11           request.context:push_copied(request.params[1])
12         end
13       ]]
14     },
15   },
16 }
```

Figura 6.13: Papel de replicação passiva.

Para ilustrar essa situação, considere o código da figura 6.14, onde é criado um novo componente processador, que é conectado e ao componente produtor da aplicação. Agora, há três clientes recebendo os eventos de um servidor. Supondo agora, que se deseje replicar todos os componentes de um servidor, por exemplo os componentes Cliente A e Cliente C. Para permitir a replicação destes componentes, o papel da figura 6.13 é aplicado sobre o adaptador de componente obtido através da interface AdaptableContainer do contêiner onde eles foram criados, fazendo com que a adaptação seja feita no nível da definição do componente e todos os componentes recebam a alteração. Agora, componentes de réplica podem ser implantados em um outro servidor e conectados aos componentes originais através da porta copied adicionada pelo papel, fazendo com que a replicação seja feita adequadamente. A figura 6.15 ilustra a estrutura da aplicação resultante.

```
-- Criando outro par de produtor-processador
client3 = loaf.handler(processorsB:create())

client3.name  = "Cliente C"

server.request = client3.done
client3.raw = server.produced

-- Adaptando todas as instancias
container = deployer.container[2]
adaptor = container:get_component_adaptor("eventflow::Processor")
Replicator:assign(adaptor)

-- Implantando um servidor de replicas
replicas = luaorb.narrow(deployer.container[4]:install_home(
  "luaccm/demo/eventflow/package.lua",
  "Processor", {}))

replica1 = loaf.handler(replicas:create())
replica2 = loaf.handler(replicas:create())

replica1.name = "Replica 1"
replica2.name = "Replica 2"

replica1.raw = client2.copied
replica2.raw = client3.copied
```

Figura 6.14: Exemplo de utilização do papel de replicação passiva.

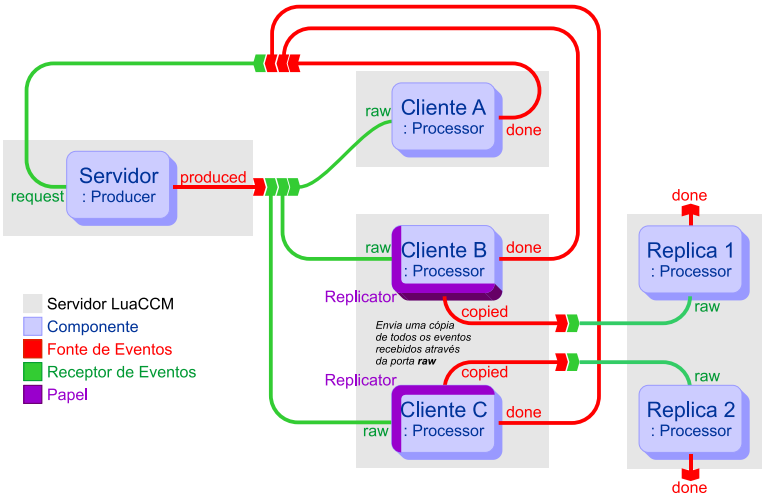

Figura 6.15: Aplicação de fluxo de eventos com replicação passiva aplicada por papéis.

A partir dos exemplos apresentados neste capítulo, é possível verificar que a introdução do conceito de papéis no modelo CCM da forma proposta apresenta níveis de complexidade compatíveis com a proposta apresentada em [14], onde esse conceito é aplicado no *middleware* Comet, que utiliza um modelo de componentes mais simples que o modelo CCM. Isso funciona como uma validação de que a proposta de papéis e protocolos é adequada como mecanismo de adaptação em modelos mais complexos.