

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 contêineres armazenados na lista **deployer.container**, 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.

```

1  producers = luaorb.narrow(deployer.container[1]:install_home(
2    "luaccm/demo/eventflow/package.lua",
3    "Producer", {}))
4  processorsA = luaorb.narrow(deployer.container[2]:install_home(
5    "luaccm/demo/eventflow/package.lua",
6    "Processor", {}))
7  processorsB = luaorb.narrow(deployer.container[3]:install_home(
8    "luaccm/demo/eventflow/package.lua",
9    "Processor", {}))
10
11 server = loaf.handler(producers:create())
12 client1 = loaf.handler(processorsA:create())
13 client2 = loaf.handler(processorsB:create())
14
15 server.name = "Servidor"
16 client1.name = "Cliente A"
17 client2.name = "Cliente B"
18
19 server.request = client1.done
20 server.request = client2.done
21 client1.raw = server.produced
22 client2.raw = server.produced
23
24 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 regulação 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

```

1  module eventflow {
2
3      module flowsync {
4          interface Rateable {
5              void set_rate(in double rate);
6          };
7          interface Limited {
8              attribute double value;
9          };
10     };
11
12     module distdebug {
13         interface Inspectable {
14             string evaluate(in string expression);
15             string execute(in string code);
16         };
17         interface Pauser {
18             boolean locked();
19         };
20     };
21 };

```

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

de limite, então é necessário ativar o mecanismo de regulação 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 regulação 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 regulação de fluxo através do receptáculo **regulator**. Dessa forma, todos os eventos subsequentes são enviados a todos os processadores na mesma taxa.

Para completar o algoritmo de sincronização, é necessário definir o mecanismo de regulação. 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 papéis definidos anteriormente, é possível inserir o recurso de análise e regulação 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**,

```

1 FlowWatcher = loaf.Role {
2   provides = {
3     limit = {
4       interface = "eventflow::flowsync::Limited",
5       code = [[ { value = 0.05 } ]],
6     },
7   },
8   uses = {
9     regulator = {
10      interface = "eventflow::flowsync::Rateable",
11    },
12  },
13  before = {
14    raw = {
15      code = [[
16        function(self, request)
17          request.start_time = get_time()
18        end
19      ]],
20    },
21  },
22  after = {
23    raw = {
24      code = [[
25        function(self, request)
26          local now = get_time()
27          local time_spent = now - (request.start_time or now)
28
29          local limit = request.context:get_executor("limit")
30          if time_spent > limit.value then
31            local regulator =
32              request.context:get_connection_regulator()
33            if regulator then
34              regulator:set_rate(time_spent)
35            end
36          end
37        end
38      end
39    ]],
40  },
41  },
42 }

```

Figura 6.5: Papel de análise de fluxo de evento

```

1 FlowRegulator = loaf.Role {
2   before = {
3     produced = {
4       code = [[
5         function(self, request)
6           local now = get_time()
7           local last = self.last or now
8           self.last = now
9
10          local time_spent = now - last
11          if request.context.rate then
12            if time_spent < request.context.rate then
13              sleep(request.context.rate - time_spent)
14            end
15          end
16        end
17      ]],
18    },
19  },
20  provides = {
21    rater = {
22      interface = "eventflow::flowsync::Rateable",
23      code = [[{
24        set_session_context = function(self, context)
25          self.context = context
26        end,
27        set_rate = function(self, rate)
28          self.context.rate = rate
29        end,
30      }]],
31    },
32  },
33 }

```

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  Inspectable = loaf.Role {
2    provides = {
3      evaluator = {
4        interface = "eventflow::distdebug::Inspectable",
5        code = [[{
6          set_session_context = function(self, context)
7            self.context = context
8          end,
9          evaluate = function(self, expression)
10           return self:execute("return " .. expression)
11         end,
12         execute = function(self, code)
13           local getter, message = loadstring(
14             "return function(context) " .. code .. " end"
15           )
16           if getter
17             then return tostring(getter()(self.context))
18             else return message
19           end
20         end,
21       }]],
22     },
23   },
24 }

```

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

```

1 BreakPoint = loaf.Role {
2   uses = {
3     pauser = {
4       interface = "eventflow::distdebug::Pauser",
5     },
6   },
7   before = {
8     port = {
9       code = [[
10        function(self, request)
11          local pauser = request.context:get_connection_pauser()
12          if pauser then
13            while pauser:locked() do
14              sleep(1)
15            end
16          end
17        end
18      ]],
19    },
20  },
21 }

```

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 papéis 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

```

1 oo.class.luaccm.demo.debugger.DebugProtocol {
2
3   constructor = function(self, inspected)
4     self.inspectable:assign(inspected)
5     self.set_private("inspected", inspected)
6
7     — Inicializacao de atributos
8   end,
9   private = {
10    inspectable = Inspectable,
11
12    — Demais operacoes privadas
13  },
14  public = {
15    evaluate = function(self, segment, field)
16      return self.inspected.evaluator:evaluate(
17        "context:get_executor('" .. segment .. "')." .. field
18      )
19    end,
20
21    add_break = function(self, point, moment)
22      local id = moment .. point
23      local break_point = loaf.Role {
24        uses = {
25          pauser = { interface = ">::debugger::Pauser" },
26        },
27        [moment] = {
28          [point] = {
29            code = [[
30              function(self, request)
31                local pauser = request.context:get_connection_pauser()
32                if pauser then
33                  while pauser:locked() do
34                    sleep(1)
35                  end
36                end
37              end
38            ]],
39          },
40        },
41      }
42      break_point:assign(self.inspected)
43      self.inspected.pauser = self:create_pauser(id)
44      self.lock[id] = true
45      self.breaks[id] = break_point
46    end,
47
48    continue = function(self, point, moment)
49      self.lock[moment .. point] = false
50    end,
51
52    — Demais operacoes do protocolo
53  },
54
55 }

```

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

```

1  — Criando outro par de produtor–processador
2  client3 = loaf.handler(processorsB:create())
3
4  client3.name = "Cliente C"
5
6  server.request = client3.done
7  client3.raw = server.produced
8
9  — Adaptando todas as instancias
10 container = deployer.container[2]
11 adaptor = container:get_component_adaptor("eventflow::Processor")
12 Replicator:assign(adaptor)
13
14 — Implantando um servidor de replicas
15 replicas = luaorb.narrow(deployer.container[4]:install_home(
16     "luaccm/demo/eventflow/package.lua",
17     "Processor", {}))
18
19 replica1 = loaf.handler(replicas:create())
20 replica2 = loaf.handler(replicas:create())
21
22 replica1.name = "Replica 1"
23 replica2.name = "Replica 2"
24
25 replica1.raw = client2.copied
26 replica2.raw = client3.copied

```

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

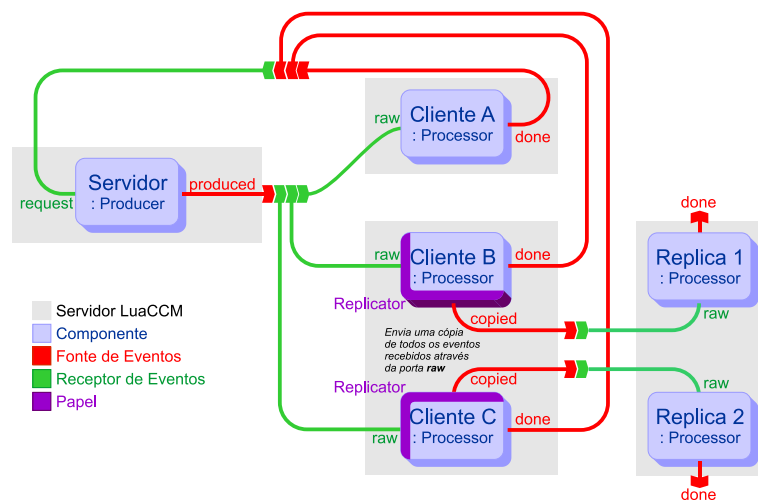


Figura 6.15: Aplicação de fluxo de eventos com replicação passiva aplicada por papéis.

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.