4
SCS com Suporte a Fluxo de Dados

O capítulo anterior apresentou conceitos importantes sobre o modelo do SCS e sobre Conectores de Fluxo de Dados. De posse desses conceitos e da análise das diferentes abordagens apresentadas no capítulo 2, nos propusemos a desenvolver uma extensão do modelo de componentes do SCS que o fizesse suportar comunicação através de fluxos de dados. Este capítulo apresenta a extensão desenvolvida.

Na seção 4.1 apresentamos um conjunto de requisitos, funcionalidades e diretivas levantados para o suporte de CFDs. Na seção 4.2, apresentamos o modelo de componentes estendido com os novos conectores. Em seguida, na seção 4.3, descrevemos detalhes da implementação do protótipo desenvolvido e, na seção 4.4, ilustramos a implementação de um componente. Por fim, na última seção apresentamos as considerações finais do capítulo, comentando as decisões tomadas.

4.1
Funcionalidades, Requisitos e Diretivas

Como mencionado no capítulo 1, este trabalho se originou a partir de uma demanda encontrada em uma aplicação comercial que utiliza o *framework CSBase*. Essa aplicação possui um editor visual para a execução de fluxos de algoritmos, no qual pode-se planejar um fluxo de algoritmos por qual um conjunto de dados deve ser processado. A representação visual de cada algoritmo é feita através de caixas com entradas e saídas. Através do editor, é possível conectar a saída de um algoritmo à entrada de outros, formando uma estrutura similar a um grafo acíclico. Esse conceito se assemelha ao de filtros de fluxo independentes citado na seção 3.2.1.

Atualmente, após a etapa de planejamento, uma máquina que possui uma instalação prévia dos algoritmos é escolhida para a execução do fluxo. Em seguida, um arquivo de dados inicial é transferido para a máquina e, por fim, o processamento é iniciado. Nessa etapa, os algoritmos, que são executados em processos diferentes, começam a se comunicar através de arquivos ou *named pipes*. Após o término do processamento, o resultado é disponibilizado para o

usuário.

Esse método atual restringe a execução do fluxo a uma máquina, limitando, assim, a escalabilidade do processo. Além disso, existe uma insatisfação dos usuários quanto ao desempenho dessa solução. Logo, um estudo de possíveis alternativas foi iniciado, resultando na demanda por implementar a execução do fluxo de forma distribuída. Para isso, propusemos o encapsulamento de cada algoritmo em componentes SCS distintos, e a utilização da infraestrutura de implantação desenvolvida por *Barbosa* [17] para fazer a implantação dinâmica do fluxo planejado. Com isso, a necessidade de Conectores de Fluxo de Dados foi criada, dando origem a esta pesquisa.

Com esse cenário em mente, uma entrevista com um dos desenvolvedores da aplicação foi realizada para questionar quais funcionalidades seriam interessantes para a solução. As funcionalidades levantadas foram:

- **Suporte a comunicação através de fluxo de dados** - devido à natureza dos dados e dos algoritmos.
- **Suporte a implantação dinâmica** - permitiria uma maior flexibilidade na utilização das máquinas.
- **Suporte a interrupções e resumos** - adicionaria uma maior flexibilidade na execução do fluxo e no tratamento de falhas.
- **Suporte a monitoração dos componentes e conexões** - auxiliaria no tratamento de falhas e na representação visual do progresso realizado.

Além das funcionalidades, o estudo das soluções apresentadas nos trabalhos relacionados auxiliou na identificação de características comuns que, independente das abordagens escolhidas, se mostraram necessárias. Consideramos essas características como requisitos para nossa abordagem. Esses requisitos são:

- **Uma API para conexão dinâmica.** - Devido a natureza do problema, deseja-se conectar e desconectar componentes dinamicamente, inclusive substituí-los.
- **Um sistema de tipos.** - Devido a necessidade de verificar a compatibilidade de conectores e garantir que operações sejam realizadas apenas em valores com os quais essas fazem sentido.
- **Diferentes modelos de execução.** - Fornece flexibilidade e expressividade para desenvolvedores de componentes.

Esse último requisito, somado às expectativas de alto desempenho, desencadeou um estudo sobre arquiteturas de servidores com o intuito de

encontrar formas de viabilizar essas características. Diversos trabalhos já foram publicados sobre o assunto, como Seda [31], μserver [32] e *Capriccio thread Library* [33], assim como diversas estratégias para lidar com múltiplas chamadas de I/O já foram propostas, desde técnicas baseadas em um processo e uma *thread* a múltiplos processos e múltiplas *threads* [34, 31, 32, 33]. Sistemas operacionais modernos oferecem diferentes APIs para lidar com essas estratégias, desde métodos para monitorar sockets e descritores de arquivos como *select, poll* e *kqueue (FreeBSD)*, como métodos de leitura e escrita não bloqueantes e assíncronos [34]. Inclusive já foram realizados estudos sobre incluir parte do código servidor no próprio kernel [35].

A análise desses trabalhos resultou em um conjunto de diretrizes que utilizamos para guiar o design e a implementação da solução. São elas:

– **Evitar cópias de dados onde possível.** - Como mencionado na seção 3.2.1, aplicações desse domínio lidam com grandes quantidades de dados, portanto essa diretiva tem como objetivo minimizar o tempo gasto na cópia de dados.

– **Minimizar a troca de contexto entre *threads*** - Essa diretiva tem como objetivo minimizar o tempo gasto na troca de contexto. Uma das formas de atingi-la é minimizar o número de *threads*.

– **Minimizar a necessidade de acesso exclusivo a dados compartilhados.** - Essa diretiva tem como objetivo minimizar a necessidade de bloquear o processamento de outras *threads* para garantir acesso exclusivo a dados compartilhados.

4.2 O Modelo Estendido

Dados os requisitos, funcionalidades e diretivas levantados na seção anterior, o modelo de componentes SCS foi estendido com dois novos tipos de conectores chamados de *ISource* e *ISink* que representam, respectivamente, conectores produtores e consumidores de fluxo de dados. Além disso, uma faceta chamada *IStreamComponent* também foi adicionada. Essa é responsável por oferecer operações de identificação e introspecção para esses novos conectores. A figura 4.1 apresenta uma representação visual do modelo estendido.

Os novos conectores, *ISource* e *ISink*, são representados em IDL por interfaces de mesmo nome, que definem operações para o estabelecimento de conexões e controle de transmissão. Ambas interfaces herdam operações comuns da interface *IStreamPort*. Os códigos 4.1 e 4.2 apresentam as operações de cada conector.

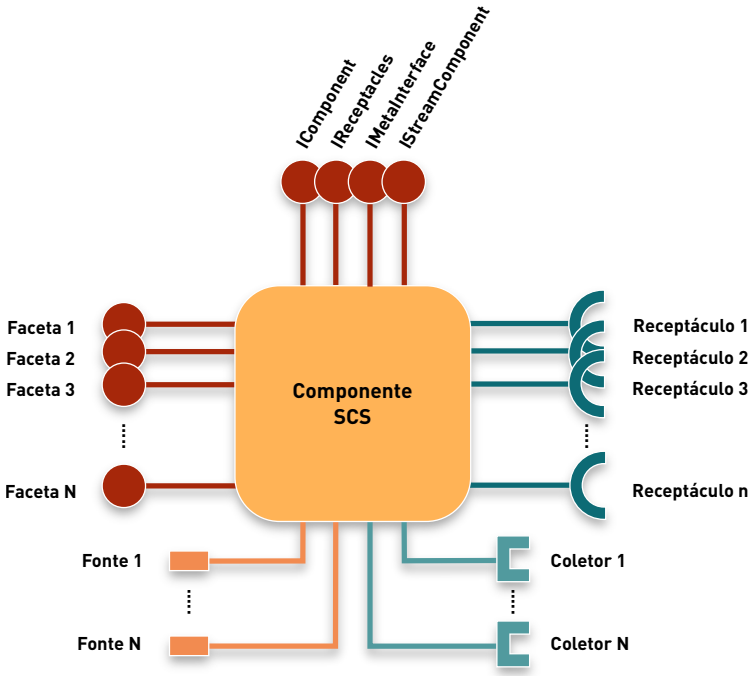

Figura 4.1: Meta-modelo do SCS com conectores de fluxo de dados

As interfaces desses conectores foram modeladas de acordo com as necessidades de prover uma API para conexão dinâmica e o suporte a interrupções e resumos de fluxos de dados. As operações *connect* e *disconnect* servem para estabelecer uma conexão entre dois conectores. Por simplicidade e por analogia a sistemas cliente-servidor, apenas os consumidores (*ISink*) podem iniciar uma conexão. Os métodos *start e stop*, presentes na interface *ISource*, são utilizados para iniciar e interromper uma transmissão após uma conexão estabelecida. O método *update* é utilizado quando se deseja atualizar dinamicamente os parâmetros de transmissão de uma conexão estabelecida. Essa última operação é opcional.

```
1 interface IStreamPort {
2     void disconnect( in ConnectionId id ) /*...*/;
3     void update( in ConnectionId id, in NameValueList params ) /*...*/ ;
4 };
```

Código 4.1: Interface *IStreamPort*

```
1  interface ISink : IStreamPort {
2      readonly attribute SinkDescription description;
3
4      ConnectionDescription connect( in ISource port,
5            in NameValueList parameters ) /*...*/;
6
7      boolean configure( in ConfigureEvent event ,
8            inout NameValueList parameters, in ISource port ) /*...*/;
9
10     ConnectionDescription getConnection();
```

```
11 };
12
13 interface ISource : IStreamPort {
14     readonly attribute SourceDescription description;
15     readonly attribute boolean enabled;
16
17     void start( in ConnectionId id ) /*...*/;
18     void stop( in ConnectionId id ) /*...*/;
19
20     boolean configure( in ConfigureEvent event,
21           inout NameValueList parameters, in ISink port ) /*...*/;
22
23     ConnectionDescriptions getConnections();
24 };
```

Código 4.2: Interfaces dos conectores

O método *configure* presente nas interfaces *ISource* e *ISink* é utilizado para centralizar as operações de reconfiguração de uma conexão entre um consumidor e um produtor. Os eventos de reconfiguração estão definidos na enumeração chamada *ConfigureEvent*. Esse método é chamado apenas em trocas de mensagens entre os próprios conectores. O funcionamento dessa operação é detalhado mais adiante na seção 4.3.3.

Os últimos métodos, *getConnection* e *getConnections*, das interfaces *ISink* e *ISource* respectivamente, servem ao propósito de listar conexões estabelecidas em cada conector. Uma fonte pode prover para vários consumidores, mas um consumidor só pode consumir de uma fonte, o que causa a diferença na assinatura dos métodos. Uma das utilidades dessas operações é o monitoramento de conexões ativas, um dos requisitos ideais levantados na seção 4.1.

A transmissão de dados entre dois componentes pode ser realizada através de diferentes mecanismos de transporte. Esses mecanismos são implementados em módulos separados chamados de *drivers* de conexão. Cada conector, provedor ou consumidor, está associado a um *driver* de conexão.

A figura 4.2 ilustra a disposição de três componentes SCS após o estabelecimento de uma conexão de fluxo de dados. Nessa, é exibida uma visão expandida dos conectores para evidenciar as entidades *Endpoint*, que representam instâncias dos *drivers* de conexão na origem e no destino. Essas entidades são responsáveis pela transmissão dos dados através do mecanismo de transporte.

A nova faceta *IStreamComponent*, como pode ser observado no código 4.3, oferece métodos para a obtenção de descrições e referências dos conectores. As características dessas operações se assemelham bastante com as operações das facetas *IComponent* e *IMetaInterface*. Por isso, inicialmente, consideramos estender essas duas facetas básicas com as operações encontradas na *IStream-*

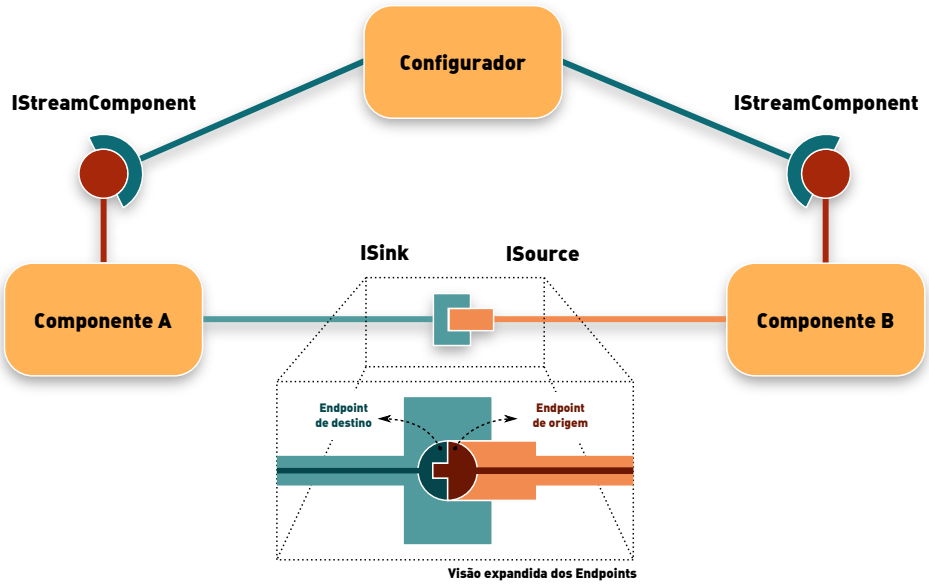

Figura 4.2: Exemplo de componentes SCS conectados

Component. Entretanto, a utilização dessa extensão foi considerada opcional e, portanto, deveria coexistir pacificamente com outras possíveis extensões do SCS. Logo, descartamos a especialização dessas interfaces e optamos pela composição através da adição dessa nova faceta.

```
1 interface IStreamComponent {
2     SinkDescriptions getSinks();
3     SourceDescriptions getSources();
4     ISink getSink(in PortId id);
5     ISink getSinkByName(in string name );
6     ISource getSource(in PortId id);
7     ISource getSourceByName(in string name );
8 };
```

Código 4.3: Interface *IStreamComponent*

Por fim, o código 4.4 apresenta as estruturas que descrevem as características dos conectores. As descrições das fontes e dos consumidores são constituídas por um identificador numérico, um nome, um tipo de conteúdo e o mecanismo de transporte suportado. As fontes possuem uma propriedade extra que indica o suporte a múltiplos clientes. Apesar das descrições dos conectores compartilharem três informações semelhantes, optou-se pela utilização de duas estruturas diferentes porque as *structs* em *CORBA* não podem ser estendidas e a utilização de *valuetypes* adicionaria uma complexidade extra que julgamos não justificável. A IDL consolidada pode ser encontrada no apêndice A.1.

```
1 struct SinkDescription {
```

```
2     PortId id;
3     string name;
4     string content_type;
5     string transport_type;
6 };
7
8 struct SourceDescription {
9     PortId id;
10    string name;
11    string content_type;
12    string transport_type;
13    boolean is_multiplex;
14 };
```

Código 4.4: Estrutura de descrição dos conectores

4.3 A Implementação

Com base nas interfaces descritas na seção anterior, um protótipo foi desenvolvido utilizando a implementação do SCS em *Java*. A esse protótipo foi atribuído o nome de *SCS-Streams*. O objetivo desta seção é apresentar uma visão geral das principais características da arquitetura, comentando algumas decisões do projeto.

4.3.1 Estendendo o Modelo de Componentes SCS

Como mencionado na seção anterior, o suporte a fluxo de dados no modelo de componentes SCS foi considerado uma funcionalidade opcional que deveria coexistir pacificamente com outras extensões do modelo. Um dos primeiros problemas encontrados com essa política de implementação foi a questão prática de como estender o contexto do componente. Por exemplo, no SCS Java, o contexto é representado por uma classe chamada *ComponentContext*. Nessa linguagem em particular, o modelo de objetos utilizada herança simples, no qual uma classe só pode estender uma única classe. Como consequência, uma extensão da classe de contexto por herança simples dificultaria a coexistência de múltiplas extensões do modelo, pois requisitaria diferentes implementações do contexto com diferentes configurações.

Em um trabalho anterior, *Silveira* [36] estendeu o modelo de componentes SCS com interfaces Coletivas, resultando na adição de conectores para sincronização e comunicação paralela. Em sua implementação, *Silveira* evitou estender o contexto mantendo os dados na implementação da faceta. Por outro lado, implementações do *CORBA Componente Model* não esbarram com esse problema, pois geram o código da classe de contexto com os dados necessários a partir da descrição feita em IDL.

Para solucionar esse problema, a implementação original do SCS sofreu pequenas alterações para permitir a extensão do contexto por composição. Com isso, foram adicionados métodos que possibilitam o registro e a remoção de objetos que representam extensões do contexto. Além disso, em certas ocasiões, se mostrou necessária a atualização das informações contidas nessas extensões após os eventos de início e término de um componente. Logo, mais duas operações foram adicionadas para possibilitar o cadastro e remoção de observadores de eventos desse tipo. O código 4.5 ilustra essas novas operações adicionadas ao contexto e o código 4.6 apresenta as interfaces que definem as operações dos objetos de extensão do contexto e observador de eventos.

```
1  package scs.core;
2  /* ... */
3  public class ComponentContext {
4    /* ... */
5    public void addListener(LifeCycleListener listener) {/* ... */}
6    public void removeListener(LifeCycleListener listener) {/* ... */}
7    public void registerExtension(ComponentContextExtension extension)
8      {/* ... */}
9    public ComponentContextExtension getExtension(String name) {/* ... */}
10   public boolean removeExtension(String name) {/* ... */}
11 }
```

Código 4.5: Novos métodos que possibilitam estender o contexto por composição

```
1  public interface LifeCycleListener {
2    void startup(ComponentContext context) throws StartupFailed;
3    void shutdown(ComponentContext context) throws ShutdownFailed;
4  }
5
6  public interface ComponentContextExtension extends LifeCycleListener {
7    void setContext(ComponentContext context) throws SCSException;
8    String getName();
9  }
```

Código 4.6: Interface observadora de eventos de inicio e fim de execução e a interface que define uma extensão do contexto.

Com essas alterações é possível estender o contexto do SCS dinamicamente, no entanto, as novas funcionalidades não estão limitadas a extensões do modelo de componentes, pois também podem ser utilizadas para o registro de objetos de contexto da aplicação/serviço. Uma consequência direta dessas alterações pode ser vista no código responsável por construir componentes, pois também precisa ser modificado para suportar a construção de novas extensões.

4.3.2 A Arquitetura

A arquitetura da extensão baseou-se inicialmente na ideia de generalizar a abordagem da especificação de áudio e vídeo para CORBA [22], e adaptá-la ao paradigma de programação baseada em componentes. A abordagem dessa especificação foi escolhida como ponto de partida por possuir uma arquitetura simples, e por não depender de alterações na sintaxe da IDL e de geração de código. Contudo, algumas características foram inspiradas pela abordagem apresentada na especificação de fluxo de dados para CCM [12].

A figura 4.3 ilustra a interação entre os principais objetos de um componente SCS com suporte a fluxo de dados. Nessa figura, as facetas e receptáculos foram omitidos com objetivo de enfatizar os objetos desenhados. As áreas tracejadas sinalizam objetos que compõem um conector de fluxo de dados. As responsabilidades deles são:

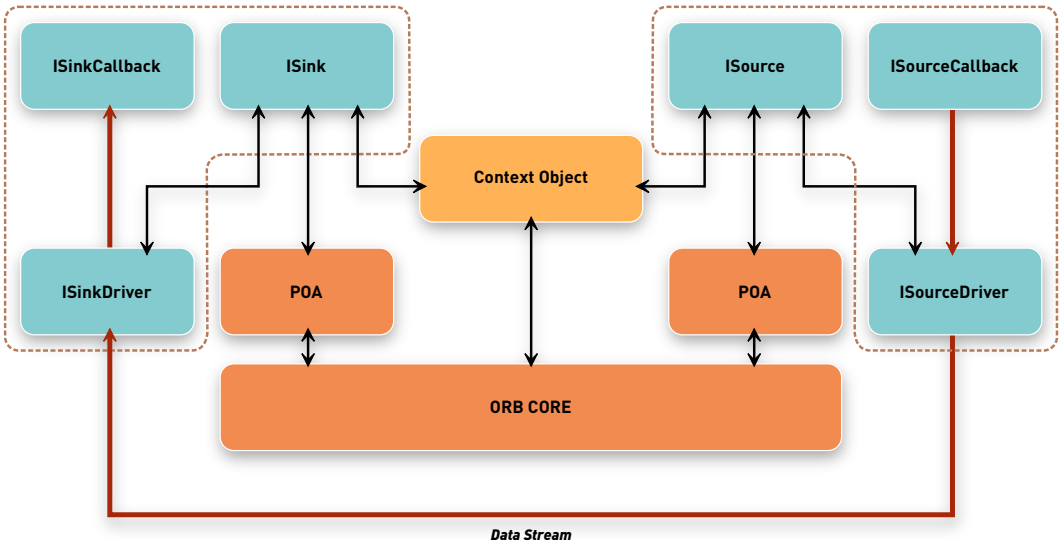

Figura 4.3: Arquitetura do *SCS-Streams*

1. ***ISink* e *ISource*** - Objetos que implementam as interfaces dos conectores de fluxo de dados. Eles são responsáveis por prover operações de estabelecimento de conexões e controle de transmissão. A implementação é provida pelo *SCS-Streams*.

2. ***ISinkDriver* e *ISourceDriver*** - Objetos que implementam os *drivers* de conexão, responsáveis por transmitir e receber dados através de um mecanismo de transporte. O *SCS-Streams* provê algumas implementações, porém desenvolvedores podem prover *drivers* para outros mecanismos de transporte.

3. ***ISinkCallback*** **e** ***ISourceCallback*** - Objetos responsáveis pela produção e consumo de dados. Esses objetos são implementados pelo desenvolvedor do componente.

4.3.3 Estabelecimento de Conexão & Controle de Transmissão

As duas especificações da OMG utilizam chamadas de métodos distintas para notificar os conectores sobre a ocorrência de determinados eventos, como início e fim de fluxo. Por exemplo, na especificação de fluxos de dados para CCM, o método *start_stream* da interface *Sink* indica o início de um fluxo. Para evitar alterações constantes na interface durante as experimentações com o processo de estabelecimento de conexão do protótipo, uma abordagem baseada no conceito de notificações de eventos de configuração/reconfiguração foi utilizada. A consolidação dessa ideia resultou no método *configure* presente nas interfaces *ISink* e *ISource*.

O estabelecimento de conexão ocorre de forma simples. O diagrama apresentado na figura 4.4 demonstra a sequência de passos para o estabelecimento de uma conexão entre dois componentes, iniciada por um terceiro componente configurador. A sequência ilustrada se divide em duas etapas: 1.x - Obtenção das referências dos conectores e 2.x - Pedido de conexão.

A primeira etapa assume que o configurador possui referências para as facetas *IStreamComponent* de cada componente e os nomes dos conectores que deseja conectar. Com essas informações, o componente configurador obtém as referências para os conectores através dos métodos *getSinkByName e getSourceByName* dessas facetas. Após obter as referências, o configurador pode dar início a próxima etapa.

Na segunda etapa do estabelecimento de conexão, o componente configurador executa uma chamada ao método *connect* do conector $VIDEO\_IN$ passando o conector $VIDEO\_OUT$ e os parâmetros de conexão desejados. O conector $VIDEO\_IN$ então chama o método *configure* do $VIDEO\_OUT$ indicando que uma conexão foi requisitada, passando os parâmetros de conexão. Se não houver impedimentos (ex: tipos incompatíveis), o conector $VIDEO\_OUT$ retorna os parâmetros de conexão que serão utilizados e um identificador de conexão (*ConnectionId*). Em seguida o conector $VIDEO\_IN$ requisita que seu *Endpoint* conecte-se ao *Endpoint* do $VIDEO\_OUT$. Se não houver problemas no estabelecimento do mecanismo de transporte, o conector $VIDEO\_IN$ retorna a chamada da operação *connect*, passando um *ConnectionDescription* para o configurador e finalizando o estabelecimento de uma conexão.

Nessa segunda etapa, é interessante notar que a conexão entre o *ISource* e *ISink* é apenas arquitetural, com propósito de Coordenação e Facilitação entre os componentes. Os dados em si são transferidos pelos *Endpoints* (Comunicação). Por isso, essa etapa do estabelecimento de conexão é realizada em duas partes: uma no nível de arquitetura dos componentes utilizando as interfaces *CORBA* dos conectores, e outra, no nível de implementação, utilizando o mecanismo de transporte definido pelo *driver* de conexão.

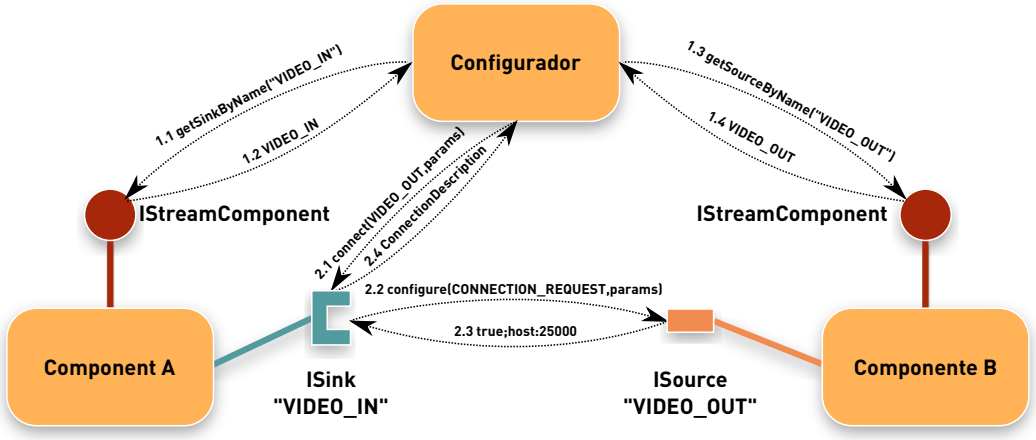

Figura 4.4: Sequência de passos para estabelecimento de uma conexão

A figura 4.5 ilustra os possíveis estados de uma conexão no conector *ISink* e os eventos que proporcionam as transições entre esses estados. Os estados de uma conexão no conector *ISource* são similares, entretanto as transições ocorrem devido a outros eventos lógicos. O diagrama desse último pode ser visto na figura 4.6. Em caso de erros durante uma transmissão em progresso, o fluxo de dados é interrompido e uma exceção é lançada para a aplicação.

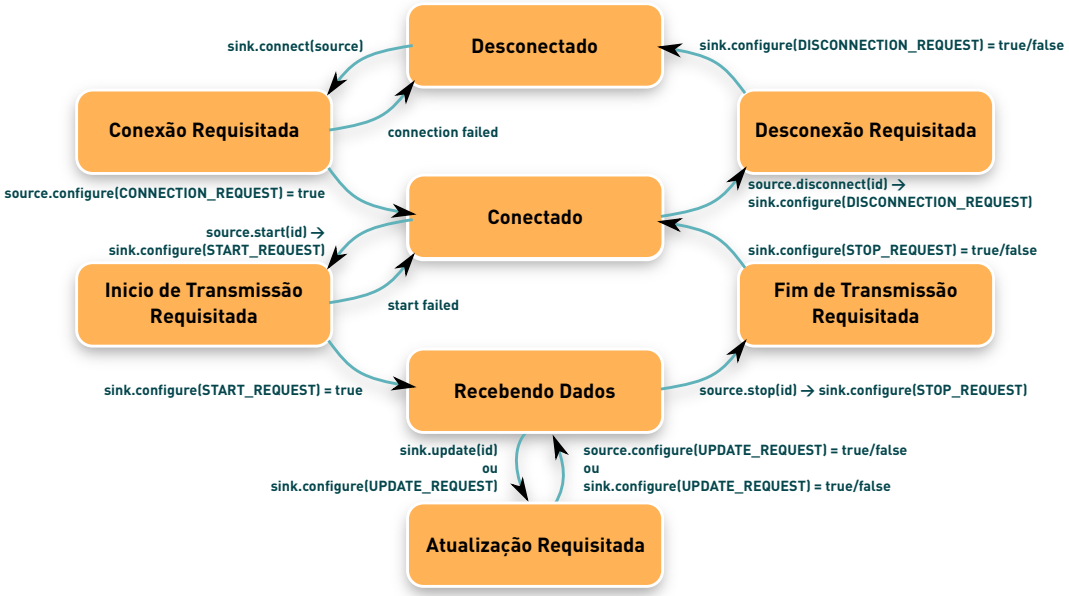

Figura 4.5: Máquina de estados de uma conexão no conector *ISink*.

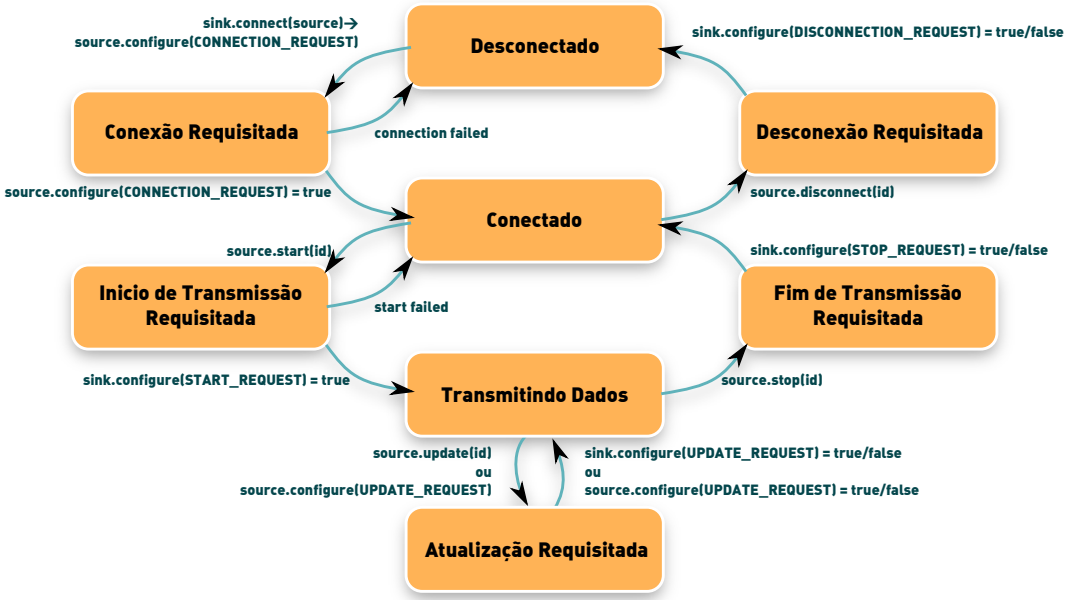

Figura 4.6: Máquina de estados de uma conexão no conector *ISource*.

Após o estabelecimento da conexão entre um *ISink* e um *ISource*, o componente configurador requisita o início da transmissão no conector do componente provedor de dados. A sequência de passos é similar à segunda etapa do processo anterior: o método *start* é invocado no conector *ISource*, resultando na chamada do método *configure* no conector *ISink*, utilizando o

evento que sinaliza início de fluxo. O método *stop* funciona de forma análoga, assim como o *update*. Esse último pode ser utilizado para avançar a posição no fluxo caso o componente provedor suporte essa operação.

4.3.4 Sistema de Tipos

Os trabalhos analisados no capítulo 2 apresentaram três diferentes abordagens para solucionar a compatibilidade entre os conectores provedores e consumidores de fluxo de dados. O estudo dessas soluções fomentou a dúvida sobre qual sistema de tipos seria mais adequado. Seguindo a filosofia de simplicidade presente nas abordagens utilizadas no SCS, procuramos levantar as características necessárias para a compatibilidade de transmissão e recepção de dados entre dois conectores. Basicamente, apenas um acordo sobre qual mecanismo de transporte que será utilizado é necessário. Entretanto, a sinalização do tipo de conteúdo se mostra extremamente útil em ocasiões nas quais deseja-se interpretar o conteúdo do fluxo e escolher quais conexões realizar. Portanto, decidimos por uma abordagem baseada na comparação do mecanismo de transporte e do tipo de conteúdo.

Mais especificamente, no *SCS-Streams*, a compatibilidade entre conectores é verificada durante o estabelecimento de uma conexão, utilizando os atributos *transport_type* e *content_type* presentes nas estruturas de descrição de cada conector (Código 4.4). Quando uma conexão é requisitada, primeiro se verifica a equivalência dos mecanismos de transporte. Esses são definidos pelo *driver* de conexão utilizado. Se essa verificação for bem sucedida, os tipos de conteúdo fornecido e esperado são comparados. Caso esses tipos sejam idênticos, o processo de conexão continua normalmente. Caso contrário, é interrompido. O tipo de conteúdo é definido por uma *string* com valor atribuído pelo desenvolvedor do componente. Porém, sugere-se a utilização de tipos *MIME* [21] para uma melhor padronização desses valores.

Optamos pela adoção de um sistema de tipos simplificado no protótipo pois este atendia as aplicações do cenário original e nos permitia validar as demais decisões do projeto. Entretanto, consideramos que o sistema de tipo pode ser evoluído em um trabalho futuro sem afetar significativamente outras partes do projeto.

4.3.5 Modelo de Execução

No protótipo desenvolvido, um conjunto de métodos e interfaces são oferecidos para o suporte aos dois modelos de execução descritos na seção

3.2. Entretanto, são as implementações dos *drivers* de conexão que decidem quais desses modelos são oferecidos para o desenvolvedor do componente. Desta forma, uma maior flexibilidade é obtida para o suporte a diferentes mecanismos de transporte. A escolha do modelo de execução é realizada por porta e informada durante a construção de um componente. Alternativamente, pode-se utilizar um arquivo de descrição XML para esse objetivo.

A API exportada por esse conjunto de métodos e interfaces foi inspirada inicialmente na especificação de fluxos de dados para CCM, porém, a semântica de algumas primitivas foi alterada. De uma forma geral, caso o *driver* de conexão utilizado suporte os dois modelos de execução, o desenvolvedor do componente se depara com duas opções: implementar métodos de uma interface de *callback* que são invocados durante a notificação de eventos, ou, utilizar um loop de envio/recebimento de dados dentro de uma *thread*. Os trechos de código 4.7, 4.8, 4.9 e 4.10 apresentam os métodos e interfaces que viabilizam essas opções. Partes do código foram omitidas para facilitar o entendimento.

Independentemente do modelo utilizado, o desenvolvedor do componente deve implementar objetos que serão utilizados para notificar eventos ocorridos nos conectores, como por exemplo início de transmissão requisitada. Para cada tipo de conector, *ISource* e *ISink*, é especificada uma interface que esses objetos devem implementar. Para conectores provedores os objetos utilizam a interface *SourcePortCallback* e, para os consumidores, *SinkPortCallback*. Esses objetos só podem estar associados a uma única instância de conector por vez.

Como pode ser observado nos códigos 4.7 e 4.8, essas interfaces definem métodos para inicialização de contexto e notificações de eventos específicos. Exceto pelo método *new_buffer*, que será explicado mais adiante, os nomes escolhidos para os métodos representam claramente o significado deles. Apesar da semelhança atual entre interfaces, optamos por mantê-las separadas, pois durante o desenvolvimento do protótipo, a separação reduzia consideravelmente a quantidade de código afetado como consequência de experimentações.

```
1  package scs.streams;
2  /* ... */
3  public interface SourcePortCallback {
4
5    public void setContext(StreamComponentContext context,
6      SourcePortServant sourcePort);
7
8    public void connection_requested(int connId) throws SCSStreamException;
9
10   public void disconnection_requested(int connId)
11     throws SCSStreamException;
12
13   public void start_requested(int connId) throws SCSStreamException;
14
```

```
15   public void stop_requested(int connId) throws SCSStreamException;
16
17   public void update_requested(int connId,
18     scs.streams.NameValue[] parameters) throws SCSStreamException;
19
20   public void new_buffer(int connId, StreamBuffer buffer);
21 }
```

Código 4.7: Interface *SourcePortCallBack*

```
1 package scs.streams;
2 /* ... */
3 public interface SinkPortCallback {
4
5   public void setContext(StreamComponentContext context,
6     SinkPortServant sinkPort);
7
8   public void disconnection_requested(int connectionId)
9     throws SCSStreamException;
10
11  public void start_requested(int connectionId) throws SCSStreamException;
12
13  public void stop_requested(int connectionId) throws SCSStreamException;
14
15  public void update_requested(int connectionId,
16    scs.streams.NameValue[] parameters) throws SCSStreamException;
17
18  public void new_buffer(int connId, StreamBuffer buffer);
19 }
```

Código 4.8: Interface *SinkPortCallBack*

No modelo de execução baseado em eventos, o método *new_buffer* é utilizado para notificar a chegada de dados em conectores consumidores. O parâmetro *buffer* desse método contém os dados recebidos. Em conectores provedores de dados, esse método possui um outro significado: ele é utilizado para fornecer um *buffer* que deve ser preenchido pela aplicação com os dados a serem transmitidos. Em ambos os casos, a implementação desse método não deve bloquear esperando o preenchimento ou o consumo dos dados. Além disso, *new_buffer* só é invocado após o início da transmissão ser notificado. Caso o modelo de execução bloqueante seja escolhido, a implementação desse método deve ficar vazia e nunca será chamada.

```
1 package scs.streams.ports;
2 /* ... */
3 public class SourcePortServant extends ISourcePOA {
4   /* ... */
5   public StreamBuffer get_buffer(int connId, int min_size)
6     throws NotConnected, InvalidConnection, SCSStreamException {/*...*/}
7
8   public boolean send_buffer(int connId, StreamBuffer buffer, long wait)
9     throws NotConnected, InvalidConnection, SCSStreamException {/*...*/}
10  /* ... */
```

```
11   public SourcePortCallback getCallback() {/*...*/}
12
13   public void setCallback(SourcePortCallback callback) {/*...*/}
14 }
```

Código 4.9: Parte da API do *SourcePortServant*

```
1 package scs.streams.ports;
2 /* ... */
3 public class SinkPortServant extends ISinkPOA {
4   /* ... */
5   public StreamBuffer get_buffer(int connId, int min_size, long wait)
6     throws NotConnected, InvalidConnection, SCSStreamException {/* ... */}
7   /* ... */
8   public SinkPortCallback getCallback() {/* ... */}
9
10  public void setCallback(SinkPortCallback callback) {/* ... */}
11 }
```

Código 4.10: Parte da API do *SinkPortServant*

No modelo de execução bloqueante, a responsabilidade de requisitar e enviar dados é da aplicação. Para atingir esse objetivo em conectores consumidores, o método *get_buffer* deve ser utilizado, bloqueando a execução do código na espera de dados durante o tempo máximo definido pelo parâmetro *wait*. Os dados recebidos são retornados em um *buffer*, representado no código pela classe *StreamBuffer*. Em conectores provedores de dados, o método de mesmo nome serve para obter um *buffer* que deve ser preenchido pela aplicação. Nesse caso o método retorna imediatamente, mesmo na ausência de *buffers*, retornando *null* nessa ocasião. Após preenchido, o *buffer* deve ser enviado através do método *send_buffer* que, por sua vez, coloca o mesmo na fila de envio do *driver* de conexão. Caso a fila esteja cheia, a execução fica bloqueada até uma entrada ficar disponível ou até o tempo máximo definido pelo parâmetro *wait*. Exemplos das implementações são apresentados na próxima seção.

Tipicamente, nesse modelo de execução, o laço de leitura ou fornecimento de dados é iniciado quando o método *start_requested* é chamado. Analogamente, o laço é terminado quando a chamada do método *stop_requested* é realizada. Entretanto, as chamadas desses procedimentos ocorrem dentro de uma *thread* iniciada pelo ORB como resultado da invocação dos métodos definidos nas interfaces dos conectores. Pelo padrão CORBA, espera-se que essa linha de execução não seja bloqueada e, por essa razão, o laço deve estar contido dentro de uma *thread* própria, evitando compartilhamento de dados desnecessários com as outras *threads* quando possível.

O diagrama contido na figura 4.7 ilustra a composição de *threads* nos dois modelos de execução. Em ambos os modelos, as chamadas aos métodos do objeto de *callback* são realizadas por *threads* do ORB/contêiner.

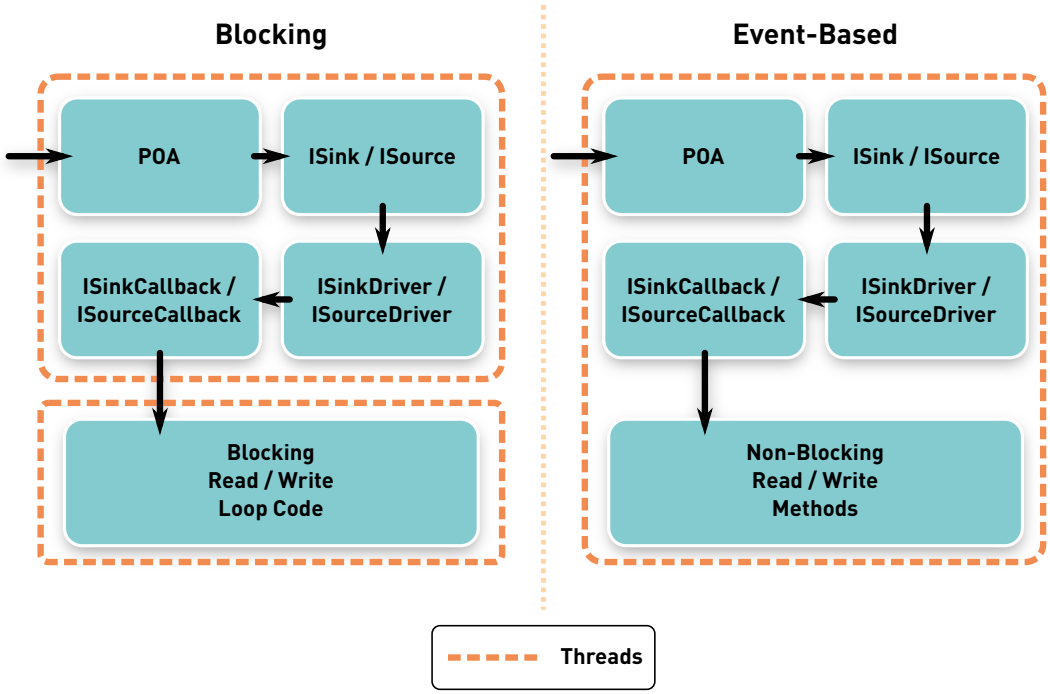

Figura 4.7: Separação das *threads* em cada modelo de execução.

4.4 Exemplo de Uso

A implementação de um componente SCS com suporte a conectores de fluxo de dados não é muito diferente da implementação de componentes sem suporte. Primeiro, o componente deve ser criado e, em seguida, é necessário registrar os objetos que irão tratar as notificações dos eventos dos conectores mencionados na seção anterior. Esses objetos devem possuir uma implementação de acordo com o modelo de execução determinado em sua criação.

Para a construção de um componente através de um arquivo de descrição é necessário utilizar a classe *XMLStreamComponentBuilder*. Como no SCS sem extensão de fluxo de dados, uma instância dessa classe possui um método chamado *build* que recebe os parâmetros de construção. O arquivo utilizado deve descrever os conectores especificando seu nome, tipo e propriedades. Algumas dessas propriedades podem ser parâmetros específicos para o *driver* de conexão utilizado. O código 4.11 apresenta a descrição de um componente com dois conectores, um consumidor (*"input"*) e um produtor (*"output"*). O primeiro utiliza o modelo de execução bloqueante e o segundo, baseado em eventos.

```
1 <?xml version="1.0" encoding="iso-8859-1" ?>
2 <scs:component xmlns:scs="tecgraf.scs.core">
3   <id>
```

```
4     <name>StreamComponent</name>
5     <version>1.0.0</version>
6     <platformSpec>Java 1.6</platformSpec>
7   </id>
8   <stream_sinks>
9     <sink name="input" driver="scs.streams.drivers.tcp.TcpSink">
10      <content>raw</content>
11    </sink>
12  </stream_sinks>
13  <stream_sources>
14    <source name="output"
15            multiplex="false"
16            driver="scs.streams.drivers.tcp.TcpSource">
17      <content>raw</content>
18      <properties>
19        <property name="scs.streams.port.execution_model"
20                  type="string">event_driven</property>
21        <property name="scs.streams.drivers.tcp.address"
22                  type="string">127.0.0.1</property>
23        <property name="scs.streams.drivers.tcp.port"
24                  type="integer">23330</property>
25      </properties>
26    </source>
27  </stream_sources>
28 </scs:component>
```

Código 4.11: Descrição XML do componente com suporte a fluxo de dados.

Após a criação do componente e de posse de seu contexto, o desenvolvedor deve obter as referências dos conectores para em seguida registrar os objetos de *callback*. Essas referências podem ser obtidas através dos métodos *getSinkByName* e *getSourceByName* utilizando os nomes dados no arquivo de descrição. O código 4.12 ilustra esse processo.

```
1  // ORB initialization
2  ORB orb = /*...*/
3  POA poa = /*...*/
4  poa.the_POAManager().activate();
5
6  // Criando o componente a partir de sua descrição
7  File descriptor = new File("StreamComponent.xml");
8  XMLStreamComponentBuilder xmlBuilder = new XMLStreamComponentBuilder();
9  ComponentContext context;
10 try {
11   context = xmlBuilder.build(orb, poa, descriptor);
12 }
13 catch (SCSException e) {
14   /* ... */
15   return;
16 }
17
18 StreamComponentContext streamContext =
19   (StreamComponentContext) context.getExtension("StreamComponentContext");
20
21 // Registrando os objetos de callback
22 SinkStreamComponentCallback cb1 = new SinkStreamComponentCallback();
```

```
23 SinkPortServant sink = streamContext.getSinkByName("input");
24 sink.setCallback(cb1);
25
26 SourceStreamComponentCallback cb2 = new SourceStreamComponentCallback();
27 SourcePortServant source = streamContext.getSourceByName("output");
28 source.setCallback(cb2);
29
30 // Esperando por requisições
31 orb.run();
```

Código 4.12: Criação do componente com suporte a fluxo de dados.

Quando o modelo de execução não for definido no descritor, como no caso do conector consumidor *"input"* (código 4.11), o modelo bloqueante é utilizado por padrão. Como explicado na seção anterior, nesse modelo as leituras e escritas são bloqueantes e, por esta razão, a utilização de uma *thread* se faz necessária. Logo, o objeto *cb1* exibido no código 4.12 implementa a interface *SinkPortCallback* de maneira que, ao receber a notificação de início de fluxo, uma *thread* contendo o laço de leitura é iniciada. Quando a notificação de término de fluxo for recebida a *thread* deve ser interrompida.

O código 4.13 ilustra uma possível implementação de uma *thread* que lê dados de um conector e os escreve em um arquivo de saída. Nessa, podemos observar um laço que termina apenas quando a variável *stopWorking* for definida como verdadeira, ou quando o *buffer* recebido indicar ser o último do fluxo. No protótipo a classe *StreamBuffer* é utilizada para encapsular uma implementação de *buffer* nativa de Java, porque não seria prático estendê-la com novas operações. A operação *release* da classe *StreamBuffer* libera o *buffer* para reutilização pelo middleware.

```
1  package demo.blocking.utils;
2  /* ... */
3  public class InputDataCopyWorker extends Thread {
4    /* ... */
5    private SinkPortServant sink;
6    private FileChannel fchannel;
7    private boolean stopWorking;
8    private int connectionId;
9    /* ... */
10   public InputDataCopyWorker(SinkPortServant sink, int connId,
11     FileOutputStream out) { /* ... */ }
12
13   public void setStopWorking(boolean stopWorking) {/* ... */}
14
15   public void run() {
16     System.out.println("DataCopyWorker started");
17     try {
18       while (!stopWorking) {
19         StreamBuffer sBuffer =
20             sink.get_buffer(connectionId, BUFFER_SIZE, WAIT_TIME);
21
```

```
22          if (sBuffer.hasData()) {
23            while (sBuffer.buffer().hasRemaining())
24              fchannel.write(sBuffer.buffer());
25          }
26
27          boolean last = sBuffer.last();
28          sBuffer.release();
29
30          if (last)
31            break;
32        }
33      }
34      catch (Exception e) {
35        /* ... */
36      }
37      System.out.println("DataCopyWorker stopped");
38    }
39  }
```

Código 4.13: Loop de consumo dos dados.

No arquivo de descrição XML apresentado no código 4.11, o conector *"output"* é definido como provedor e optante pelo modelo de execução baseado em eventos. Nesse caso, o objeto *cb2* do código 4.12 deve implementar a interface *SourcePortCallback*, provendo os dados na implementação do método *new_buffer*. Como no caso anterior, nenhum método deve bloquear a linha de execução da *thread* que o invocou. Entretanto, não é necessária a utilização de *threads*.

O código 4.14 apresenta uma possível implementação dessa classe, que lê dados de um arquivo e os envia para o conector. Quando dados são lidos do arquivo para o *buffer*, a implementação deve sinalizar que o *buffer* foi preenchido com dados através da chamada ao método *hasData(true)*. Caso a leitura atinja o final do arquivo, a implementação deve sinalizar que não há mais dados nesse fluxo, invocando o método *last(true)* do *buffer*.

```
1  package demo.event_driven.tcp;
2  /*...*/
3  public class SourceStreamComponentCallback implements SourcePortCallback {
4    /*...*/
5    private FileChannel channel = /*...*/;
6    /*...*/
7    public void new_buffer(int connectionId, StreamBuffer sbuffer) {
8      try {
9        int bytesRead = channel.read(sbuffer.buffer());
10
11       if (bytesRead > 0)
12         sbuffer.hasData(true);
13
14       if (bytesRead < 0)
15         sbuffer.last(true);
16     }
17     catch (Exception e) {
```

```
18          /* ... */
19        }
20      }
21  }
```

Código 4.14: Implementação do método *new_buffer*.

4.5 Considerações Finais

Este capítulo apresentou um conjunto de funcionalidades, requisitos e diretivas que guiaram o desenvolvimento de uma extensão que adiciona o suporte a conectores de fluxo de dados no modelo de componentes SCS. Com isso, um modelo estendido foi proposto e suas interfaces descritas. Em seguida, as principais características da arquitetura do protótipo que implementa essa extensão foram apresentadas, finalizando com um exemplo de implementação de componente.

De acordo com as variações de conectores de fluxo de dados levantados por Metha et al [8], revistas no capítulo 3, o protótipo desenvolvido pode ser analisado da seguinte forma:

- ***Delivery*** - Variações sobre a política de entrega dos dados são suportadas através da utilização de diferentes mecanismos de transporte. Esses mecanismos são implementados por *drivers* de conexão.
- ***Bounds*** - Os dados podem ser transmitidos com ou sem limites. No protótipo, um fluxo pode ser transmitido continuamente sem limites, contudo, se necessário, o limite de fim de fluxo é implementado através de operações que notificam os conectores sobre esse evento.
- ***Buffering*** - Essa dimensão não varia, pois a utilização de *buffers* é obrigatória.
- ***Throughput*** - A unidade de medida utilizada é atômica (bytes/segundo), mas desenvolvedores podem utilizar outras unidades de acordo com o domínio da aplicação desenvolvida.
- ***State*** - Os conectores propostos possuem um estado, independente do mecanismo de transporte utilizado.
- ***Identity*** - Os conectores possuem um nome representado por uma cadeia de caracteres e o fluxo de dados é representado por um identificador numérico.
- ***Locality*** - Do ponto de vista de um componente, os fluxos podem ser transmitidos localmente (mesmo componente) ou remotamente (outro componente).

- ***Synchronicity*** - A sincronia do fluxo é determinada pelo *driver* de conexão.
- ***Format*** - O protótipo não força a utilização de dados estruturados, entretanto, o mecanismo de transporte e a aplicação podem utilizar dados estruturados se forem necessários.
- ***Cardinality*** - No protótipo, essa dimensão é determinada pela implementação do *driver* de conexão. O fluxo pode ser realizado ponto a ponto ou com um servidor e múltiplos clientes.

As funcionalidades levantadas foram todas implementadas. O suporte a comunicação através de fluxo de dados é possível através dos novos conectores e seus *drivers* de conexão. O suporte a interrupções e resumos de fluxos é implementado nas operações *start* e *stop* apresentadas no código 4.1. O monitoramento das conexões ativas é possível através dos métodos *getConnections* e *getConnection*, mas o monitoramento dos componentes deve ser implementado pelo próprio desenvolvedor do componente. A implantação dinâmica foi implementada e será abordada no próximo capítulo.

Na seção 4.3.3 foram apresentados os processos de estabelecimento de conexão e controle de transmissão. Esses processos são controlados através das operações fornecidas nas interfaces dos próprios conectores. Essas interfaces compõem uma API para conexão em tempo de execução, atendendo ao primeiro requisito listado na seção 4.1.

O sistema de tipos desenvolvido buscou uma implementação simples, porém funcional, baseada na equivalência do mecanismo de transporte e do tipo de conteúdo. Nos dois primeiros trabalhos relacionados analisados, tipos MIME são utilizados para padronizar esse tipo de conteúdo. A mesma padronização pode ser aplicada no sistema implementado. Diferentemente da especificação de fluxos de dados para CCM, o sistema de tipos apresentado não suporta tipos hierárquicos. Entretanto, futuramente consideramos a hipótese de adicionar metadados de tipo nos conectores, de forma similar ao terceiro trabalho relacionado, para obter mais flexibilidade na escolha de conexões a serem realizadas.

O último requisito diz respeito ao suporte de diferentes modelos de execução. Conforme explicado na seção 4.3.5, duas possibilidades de modelo de execução são oferecidas para o desenvolvedor do componente, no entanto, a implementação dos dois modelos não é mandatória para todos os *drivers* de conexão. Na extensão do CCM, é oferecido um terceiro modelo baseado em eventos de um nível mais alto. Deixamos esse modelo para evoluções futuras da extensão, caso a necessidade seja comprovada.

Finalmente, as diretivas levantadas com a análise de trabalhos sobre arquitetura de servidores auxiliaram no desenvolvimento da arquitetura. Entretanto, a utilização de *threads* em certas ocasiões não pôde ser evitada, como no laço de leitura utilizando o modelo de execução bloqueante. Nessas situações, recomendamos uma certa cautela ao compartilhar dados entre as *threads* para minimizar a necessidade de acessos exclusivos. A utilização de diversos conectores no modelo de execução bloqueante pode afetar o desempenho da aplicação devido às excessivas trocas de contexto. Por fim, o número de cópias realizadas foi minimizado, utilizando um mecanismo interno de gerência de *buffers*.