

# Tutorial do SDK Java 2.0.0 do OPENBUS

Tecgraf

10 de fevereiro de 2014

## Sumário

<b>1</b>	<b>Introdução</b>	<b>1</b>
<b>2</b>	<b>Escopo</b>	<b>2</b>
2.1	Aplicações do ASSISTANT . . . . .	2
2.2	O que o ASSISTANT não faz . . . . .	2
2.3	Vantagens de se utilizar o ASSISTANT . . . . .	2
<b>3</b>	<b>Preparação do Ambiente</b>	<b>3</b>
3.1	Teste . . . . .	3
<b>4</b>	<b>Desenvolvimento</b>	<b>3</b>
4.1	API . . . . .	3
4.2	Primeiros Passos . . . . .	4
4.2.1	Autenticação em um Barramento . . . . .	4
4.2.2	Registro de Ofertas . . . . .	5
4.2.3	Processo Cliente . . . . .	6
4.2.4	Acessando a cadeia de chamadas para identificar o cliente . . . . .	7
4.3	Configurações avançadas . . . . .	9
4.3.1	Definindo os níveis de log do JACORB . . . . .	9
4.3.2	Definindo os níveis de log da API . . . . .	9
4.3.3	Definindo as propriedades opcionais do Assistente . . . . .	10

## 1 Introdução

Este documento é destinado aos usuários desenvolvedores de aplicações Java que desejem utilizar o OPENBUS [4]. O processo de *deployment* do barramento não é coberto neste tutorial. Considera-se como pré-requisito para um bom entendimento deste documento o conhecimento básico dos seguintes assuntos:

- Linguagem de programação Java.
- CORBA [3].
- Modelo de Componentes SCS [1].
- Conceitos básicos do OPENBUS [5].

A proposta do manual é capacitar o desenvolvedor, que atenda aos pré-requisitos do documento, a utilizar e desenvolver serviços para o barramento OPENBUS dentro do escopo dessa API.

## 2 Escopo

Assistente, ou ASSISTANT, é o nome que damos a uma abstração da biblioteca de acesso do OPENBUS, também conhecida como API básica, cujo objetivo é atuar como um utilitário que atende os cenários de uso mais comuns. Para atingir esse objetivo, o ASSISTANT torna-se menos flexível que o uso direto da API básica, mas atende a maioria dos casos de uso envolvendo o barramento OPENBUS.

### 2.1 Aplicações do Assistant

As aplicações mais comuns, e as que são melhor implementadas usando o ASSISTANT, têm o seguinte em comum:

- Comunica-se com um único barramento.
- Possui apenas uma autenticação no barramento.

Geralmente um aplicativo que se conecta ao barramento é um de dois tipos, ou um híbrido:

- Servidor.
- Cliente.
- Híbrido do dois.

Um servidor adiciona ofertas ao barramento, disponibilizando um ou mais serviços através destas ofertas. Um cliente, por outro lado, consulta o barramento por ofertas de serviços que deseja consumir.

### 2.2 O que o Assistant não faz

Por ser uma abstração mais simples e focada em um grupo de casos de uso, naturalmente o ASSISTANT não consegue resolver todos os problemas que a API básica (em cima da qual ele é implementado), consegue.

Portanto, os seguintes casos só conseguem ser implementados com o uso da API básica:

- A aplicação se comunica com mais de um barramento.
- São necessárias múltiplas autenticações com um ou mais barramentos.

O ASSISTANT recebe as informações de autenticação (como usuário, senha, chave privada, etc) em sua construção, assim como outras informações necessárias para o uso de CORBA. Com essas informações ele se autentica ao barramento de forma assíncrona e mantém válida essa autenticação, renovando-a quando estiver para expirar. Também é refaz a autenticação se a mesma ficar inválida por qualquer motivo, como por exemplo erros de rede temporários.

A construção do ASSISTANT não implica que a autenticação já tenha ocorrido com sucesso. Isso significa que métodos síncronos como *findServices* podem falhar com a exceção *NO\_PERMISSION* com código de erro *NoLogin* pelo fato de a aplicação não estar ainda autenticada no barramento. No entanto, alguns mecanismos são oferecidos para facilitar o tratamento desses erros, como veremos na próxima seção.

### 2.3 Vantagens de se utilizar o Assistant

Para as aplicações que estão no escopo do ASSISTANT, o mesmo possui várias vantagens, dentre as principais:

- Autenticação simplificada com o barramento.
- Tratamento automático de erros.
- Ofertas de serviços são mantidas, sendo refeitas automaticamente caso a autenticação mude.

Como foi dito anteriormente, a autenticação ocorre de forma assíncrona. O mesmo ocorre com o registro de ofertas. Essa diferença entre o ASSISTANT e a API básica é crucial para entender as vantagens de se usar o mesmo.

O registro de ofertas ocorre de forma assíncrona e, se por qualquer motivo o registro for invalidado pelo barramento o ASSISTANT automaticamente faz um novo registro da oferta. Assim, o usuário não precisa se preocupar com os vários casos de erro que possam acontecer num sistema distribuído para garantir que suas ofertas estarão sempre visíveis a outros serviços.

Métodos síncronos da API do ASSISTANT, como por exemplo a busca de ofertas, em geral contam com um parâmetro que indica quantas tentativas devem ser feitas, já para evitar que erros como *NO\_PERMISSION* com código de erro *NoLogin* e outros precisem ser capturados. É possível ainda definir um parâmetro opcional do ASSISTANT que informa o tempo de espera entre as tentativas. Essas e outras funcionalidades serão explicadas com mais detalhes neste documento em seções posteriores.

Utilizando o ASSISTANT precisamos nos preocupar apenas com o conceito da autenticação ao barramento, já que apenas um barramento é utilizado. Caso a API básica fosse utilizada, teríamos de nos preocupar ainda com o conceito de conexão ao barramento (para mais detalhes, consultar a documentação da API básica).

## 3 Preparação do Ambiente

O OPENBUS utiliza o mecanismo de interceptação de CORBA no processo de controle de acesso aos serviços. Após a chamada de requisição de *login* no barramento, todas as chamadas seguintes devem estar com a credencial de acesso anexada. Sendo assim, faz-se necessário a habilitação dos interceptadores frente ao ORB que será utilizado. No SDK Java a comunicação será feita utilizando o JACORB [2], e o SDK se encarregará de registrar os interceptadores em sua inicialização, não sendo necessária nenhuma ação por parte do usuário nesse aspecto.

### 3.1 Teste

Antes de começar a programação, é aconselhável que o usuário coloque em funcionamento a demo Hello que acompanha o pacote do ASSISTANT. O correto funcionamento da demo elimina uma série de possíveis problemas que o desenvolvedor possa enfrentar durante o seu projeto, como por exemplo: problemas de comunicação com o barramento e ausência de bibliotecas externas.

A demo Hello Java do ASSISTANT possui um *jar* com a classe servidor e cliente, e pode ser executada de acordo com as instruções do arquivo README.txt que a acompanha. Importante lembrar que, de acordo com a documentação do JACORB, é necessário utilizar o mecanismo de sobrescrita de padrão do Java (*Java Endorsed Standards Override Mechanism*). Isso é feito utilizando a opção *-Djava.endorsed.dirs* da JVM e passando o diretório onde se encontra os *jars* do JACORB. Assim garantimos que a implementação do JACORB e as classes disponibilizadas da OMG são encontradas em preferência a qualquer outra classe já incluída na JVM.

Após a execução do cliente, no terminal do processo servidor a seguinte mensagem deve ser exibida: “Hello <nome-da-entidade-cliente>!”

## 4 Desenvolvimento

### 4.1 API

Prover ou buscar serviços em um barramento OPENBUS usando o ASSISTANT são tarefas bem simples. Para isso, precisamos construir uma instância da classe ASSISTANT que nos proverá as funções necessárias para ofertar ou buscar serviços no barramento. Em Java instancia-se a classe *tecgraf.openbus.assistant.Assistant*.

No caso de aplicações do tipo servidor, que recebam chamadas, pode ser necessário o uso de mais uma classe: *tecgraf.openbus.OpenBusContext*. Em Java essa classe é única para cada instância de ORB. Através dela, será possível obter a identidade do cliente ou cadeia de clientes de uma chamada. Para obter a instância

do ORB como da classe *OpenBusContext*, devemos utilizar a classe estática *tecgraf.openbus.core.ORBInitializer*. Essa classe é responsável por instanciar e configurar o ORB e o *OpenBusContext*.

Nas próximas seções veremos o uso dessas classes em mais detalhes.

## 4.2 Primeiros Passos

Basicamente há dois tipos de aplicações distribuídas: uma que oferta serviços e outra que consome serviços, sendo possível e natural uma aplicação híbrida que tanto oferte quanto consuma. Para ambos os tipos, a API do assistente tenta facilitar toda funcionalidade comum.

O uso do Assistente será demonstrado com base na demo Hello do ASSISTANT. Propositalmente esta é uma demo muito simplória, que é composta de um processo cliente e outro servidor. O servidor oferta um serviço do tipo “Hello” num determinado barramento, enquanto que o cliente procura por este serviço no barramento e o utiliza. Quando isso acontece, o servidor imprime na saída padrão a frase “Hello World!”. Posteriormente, alteraremos o exemplo para que imprima a frase “Hello <nome-da-entidade-cliente>!”.

### 4.2.1 Autenticação em um Barramento

O primeiro passo de qualquer aplicação que utilize um barramento OPENBUS através do ASSISTANT é se autenticar. Atualmente existem três modos de autenticação:

- Par usuário e senha.
- Certificado digital.
- Utilização de autenticação compartilhada.

O primeiro é destinado normalmente a clientes que estejam acessando o barramento à procura de um serviço. O segundo é mais adequado a processos servidor que registrarão um determinado serviço no barramento. Já o terceiro é útil para aplicações que dependam de terceiros para realizar uma autenticação, ou seja, não tenham acesso a um par usuário/senha nem a um certificado, mas sim a um terceiro elemento que se responsabilize pela autenticação dessa entidade.

No caso da autenticação via certificado, o responsável pelo serviço deve previamente encaminhar ao administrador do barramento o certificado do serviço, ou seja, um arquivo especial que contenha a chave pública do serviço.

Um exemplo do código necessário para a autenticação em um barramento pode ser visto no Código 1.

#### Código 1: Autenticação em um Barramento

```
1 String host = args[0];
2 int port = Integer.parseInt(args[1]);
3 String entity = args[2];
4 String privateKeyFile = args[3];
5 PrivateKey privateKey =
6     OpenBusPrivateKey.createPrivateKeyFromFile(privateKeyFile);
7
8 final Assistant assist =
9     Assistant.createWithPrivateKey(host, port, entity, privateKey);
```

Note que estamos utilizando um método estático de fábrica para instanciar o assistente, onde são necessários quatro parâmetros obrigatórios:

**host** Endereço do barramento.

**port** Porta do barramento.

**entity** Nome da entidade a ser autenticada ao barramento. Esse nome deve ser o nome que foi associado previamente ao certificado digital, junto ao administrador do barramento.

**privateKey** Chave privada associada ao certificado digital cadastrado no barramento.

Uma instância que implemente a interface *tecgraf.openbus.PrivateKey* deve ser construída através da classe utilitária *tecgraf.openbus.core.OpenBusPrivateKey*, que contém métodos para se obter uma chave privada através de uma sequência de *bytes* ou de um arquivo.

Como vimos anteriormente, após a construção de *tecgraf.openbus.assistant.Assistant*, a autenticação não necessariamente já terá ocorrido, pois a mesma é feita assincronamente e pode se repetir indefinidamente, sem intervenção do usuário. O ASSISTANT assume que a aplicação pretende se manter autenticada por todo o tempo em que o ASSISTANT se mantiver vivo. Por isso, qualquer erro que implique na perda de autenticação será tratado pelo ASSISTANT, que tentará se autenticar novamente até conseguir. Essa é uma das principais vantagens de se usar o ASSISTANT.

#### 4.2.2 Registro de Ofertas

O propósito principal do ASSISTANT para um servidor é o de autenticar a um barramento OPENBUS e ofertar os serviços no Registro de Ofertas.

Um serviço pode ser ofertado no barramento através de uma lista de propriedades e de uma referência a um componente SCS. A lista de propriedades pode ser utilizada para adicionar características para o serviço, a fim de facilitar a identificação por parte dos clientes.

A arquitetura do Openbus é baseada no modelo de componentes SCS. Os serviços são de fato representados por componentes, que podem apresentar distintas facetas (interfaces). Sendo assim, o primeiro passo para o registro de um serviço é criar um componente que represente esse serviço. Maiores informações sobre a criação de componentes SCS podem ser obtidas nos tutoriais desse projeto. Um exemplo de criação de componente pode ser visto no Código 2.

Código 2: Exemplo de criação de um componente SCS

```
1 // criando o serviço a ser ofertado
2 // - ativando o POA
3 POA poa = POAHelper.narrow(orb.resolve_initial_references("RootPOA"));
4 poa.the_POAManager().activate();
5 // - construindo o componente
6 ComponentId id =
7     new ComponentId("Hello", (byte) 1, (byte) 0, (byte) 0, "java");
8 ComponentContext component = new ComponentContext(orb, poa, id);
9 // Cria a faceta Hello para o componente.
10 // A classe HelloImpl deve ser implementada de acordo com a especificação do serviço Hello.
11 component.addFacet("Hello", HelloHelper.id(), new HelloImpl());
```

A lista de propriedades referentes ao serviço deve ser descrita numa lista de estruturas do tipo *ServiceProperty*. Um exemplo de registro de uma oferta pode ser conferido no Código 3.

Código 3: Registro de uma Oferta de Serviço

```
1 // registrando serviço no barramento
2 ServiceProperty[] serviceProperties =
3     new ServiceProperty[] { new ServiceProperty("offer.domain", "Demo Hello") };
4 // usa o assistente do OpenBus para registrar a oferta.
5 assist.registerService(component.getComponent(), serviceProperties);
```

Para finalizar a implementação do nosso servidor, falta apenas implementarmos o serviço *Hello* de fato, representado pela classe *HelloImpl* em nosso exemplo. Começaremos com uma implementação que apenas imprime localmente “Hello World!” para qualquer chamada recebida. O código pode ser visto no Código 4.

É interessante (mas não obrigatório) garantir o término correto das conexões do ORB. Para isso, aconselhamos a inclusão de uma thread de finalização no Runtime da JVM. Um exemplo pode ser conferido no Código 5.

Código 4: Implementação da faceta Hello

```

1 public final class HelloImpl extends HelloPOA {
2     public HelloImpl() {
3     }
4     @Override
5     public void sayHello() {
6         System.out.println("Hello World!");
7     }
8 }

```

Código 5: Exemplo de uma thread de finalização do ORB.

```

1 final ORB orb = assist.orb();
2 // - criando thread para parar e destruir o ORB ao fim da execução do processo
3 Thread shutdown = new Thread() {
4     @Override
5     public void run() {
6         assist.shutdown();
7         orb.shutdown(true);
8         orb.destroy();
9     }
10 };
11 Runtime.getRuntime().addShutdownHook(shutdown);

```

Até este ponto, o servidor basicamente se autenticou ao barramento e registrou seu serviço “Hello”. O último passo necessário para finalizar o código do servidor é habilitar o processo a escutar as requisições CORBA que serão direcionadas para o serviço ofertado, o que pode ser feito disparando o método *run* do ORB em uma thread separada, como demonstrado pelo código 6.

Código 6: Exemplo de thread para disparar o método *run* do ORB.

```

1 // - disparando a thread para que o ORB atenda requisições
2 Thread run = new Thread() {
3     @Override
4     public void run() {
5         orb.run();
6     }
7 };
8 run.start();

```

### 4.2.3 Processo Cliente

Com o servidor pronto, é necessário agora construir um consumidor deste serviço. Alguém que irá buscar o serviço no Registro de Ofertas e chamar *sayHello()*, que é o que o nosso serviço atualmente faz.

Para tal precisamos, assim como no servidor, autenticarmo-nos ao barramento. Porém, as coincidências acabam aí. No cliente não criaremos *servants* nem criaremos ofertas, apenas faremos uma busca e, de posse da oferta, a utilizaremos.

Vejamos novamente a autenticação, mas desta vez de nosso cliente, no Código 7.

Como vimos, a instanciação da classe *tecgraf.openbus.assistant.Assistant* coloca o ASSISTANT em responsabilidade por nos autenticar ao barramento de forma assíncrona. Agora, nos resta fazer uso do Registro de Ofertas através dele. Para fazermos uma busca no barramento por serviços, utilizamos o método *findServices*.

Para podermos fazer a busca, precisamos saber o que buscar. Sabemos que nosso serviço possui uma faceta chamada “Hello”, que o mesmo se autenticou no barramento com um nome de entidade específico, e que foi especificado o valor “Demo Hello” para a propriedade “offer.domain”. As duas primeiras informações correspondem aos valores das propriedades automáticas “openbus.offer.entity” e “openbus.component.facet” respectivamente. De posse dessa lista de propriedades, podemos realizar a busca. Um exemplo pode ser conferido no Código 8.

### Código 7: Autenticação da Aplicação Cliente

```
1 String host = args[0];
2 int port = Integer.parseInt(args[1]);
3 String entity = args[2];
4 String password = args[3];
5
6 final Assistant assist =
7     Assistant.createWithPassword(host, port, entity, password.getBytes());
```

### Código 8: Busca por Ofertas de Serviço

```
1 String serverEntity = "nome-da-entidade-servidor";
2 ServiceProperty[] properties = new ServiceProperty[3];
3 properties[0] = new ServiceProperty("offer.offer.entity", serverEntity);
4 properties[1] = new ServiceProperty("offer.component.facet", "Hello");
5 properties[2] = new ServiceProperty("offer.domain", "Demo Hello");
6 ServiceOfferDesc[] services = assist.findServices(properties, -1);
```

O método *findServices* de busca de serviços recebe dois parâmetros: uma lista de propriedades e um inteiro que representa o número de re-tentativas que a função deve fazer em caso de erros. Esse número pode ser “-1”, que indica que a função deve tentar indefinidamente; pode ser o número “0”, que indica que a função deve retornar uma exceção se qualquer falha ocorrer e nenhuma nova tentativa se a mesma falhar; e qualquer número maior que zero, que indica o número de novas tentativas em caso de falhas. Caso o número de re-tentativas se esgote, a última exceção recebida será lançada.

Como visto anteriormente, a oferta de serviço contém uma referência para um componente SCS ou, mais especificamente, para sua faceta “IComponent”. Como esse componente pode apresentar diversas facetas, o usuário deve obter a faceta que deseja utilizar. No exemplo há somente uma faceta além das facetas básicas do modelo SCS, chamada “Hello”. A faceta recebida na oferta, “IComponent”, oferece os métodos (*getFacet* e *getFacetByName*) para que o usuário possa obter um objeto CORBA que represente a faceta desejada. De posse do objeto CORBA, o usuário deve efetuar um *narrow* para mapeá-lo para a interface desejada e, daí em diante, o programador estará apto a utilizar essa faceta/serviço.

É importante notar que, apesar do ASSISTANT ajudar no tratamento de erros das funcionalidades mais comuns, a aplicação deve tratar por conta própria os erros que podem ocorrer ao realizar chamadas CORBA diretamente a um objeto remoto. As chamadas *getFacet* (ou *getFacetByName*) e *sayHello* se enquadram nesse caso, portanto precisam do tratamento adequado. Como a operação precisa coordenar programas diferentes, rodando em arquiteturas e localidades diferentes, inúmeras falhas são possíveis, sendo as mais comuns uma falha de comunicação física (por exemplo, o cabo de rede desconectado). O serviço pode também não estar mais funcionando corretamente, ou ter sido finalizado.

Portanto, toda chamada remota deve estar protegida, de algum modo, por tratamentos de erros. O tratamento de erro em CORBA é feito através de exceções. A exceção *CORBA::TRANSIENT* significa que o serviço não conseguiu se comunicar com o outro ponto; *CORBA::COMM\_FAILURE* significa a mesma coisa, mas a conexão não conseguiu ser iniciada por algum motivo; *CORBA::OBJECT\_NOT\_EXIST* significa que o serviço que deveria prover esse objeto CORBA não possui esse objeto em sua memória, e a sua referência deve ser descartada. Outras exceções herdam de *CORBA::SystemException*. Um exemplo do tratamento de erros adequado pode ser visto no Código 9.

Por fim, podemos finalizar o ASSISTANT para que seja feito o *logout* no barramento, como exemplificado no Código 10.

Nesse ponto, podemos executar o servidor e em seguida o cliente. Assim que o cliente realizar a chamada *sayHello*, o servidor imprimirá na tela a mensagem “Hello World!”.

#### 4.2.4 Acessando a cadeia de chamadas para identificar o cliente

A maior evolução no barramento OPENBUS na versão 2.0 foi a reformulação do sistema de segurança. Com este novo sistema, é possível para um serviço saber qual a entidade que fez a chamada, e também é possível

### Código 9: Acesso à Faceta Desejada

```
1 // analisa as ofertas encontradas
2 for (ServiceOfferDesc offerDesc : services) {
3     try {
4         org.omg.CORBA.Object helloObj =
5             offerDesc.service_ref.getFacet(HelloHelper.id());
6         if (helloObj == null) {
7             System.out
8                 .println("o serviço encontrado não provê a faceta ofertada");
9             continue;
10        }
11        Hello hello = HelloHelper.narrow(helloObj);
12        hello.sayHello();
13    }
14    catch (TRANSIENT e) {
15        System.err.println("o serviço encontrado encontra-se indisponível");
16    }
17    catch (COMM_FAILURE e) {
18        System.err.println("falha de comunicação com o serviço encontrado");
19    }
20    catch (NO_PERMISSION e) {
21        switch (e.minor) {
22            case NoLoginCode.value:
23                System.err.println(String.format(
24                    "não há um login de '%s' válido no momento", entity));
25                break;
26            case UnknownBusCode.value:
27                System.err
28                    .println("o serviço encontrado não está mais logado ao barramento");
29                break;
30            case UnverifiedLoginCode.value:
31                System.err
32                    .println("o serviço encontrado não foi capaz de validar a chamada");
33                break;
34            case InvalidRemoteCode.value:
35                System.err
36                    .println("integração do serviço encontrado com o barramento está incorreta");
37                break;
38        }
39    }
40 }
```

### Código 10: Finalizando o ASSISTANT

```
1 assist.shutdown();
```

para algum serviço intermediário passar para um serviço do qual ele é cliente a identidade de quem está usando esse intermediário. Isso cria uma cadeia de entidades que pode ser lida pelo serviço final para autorizar ou não a chamada.

Veremos o caso mais simples, modificando o serviço *Hello* que implementamos anteriormente para imprimir a mensagem “Hello <nome-da-entidade-cliente>!” ao invés de “Hello World!”. Uma aplicação mais complexa poderia usar essa informação para escolher de qual banco de dados pegaria os dados, ou como faria a autenticação para obter os dados ou simplesmente recusar a operação para determinadas entidades.

Relembrando, nosso *servant* “Hello” foi implementado como descrito no Código 4.

Os métodos relativos a cadeias ficam na classe *OpenBusContext*, como mencionado anteriormente. São eles: *getCallerChain*, *joinChain*, *exitChain* e *getJoinedChain*.

Faremos uso do método *getCallerChain*, que retorna a cadeia de chamadas relativa à chamada atual. Como comentado anteriormente na subseção 4.1, cada instância de ORB possui uma instância de *OpenBusContext* associado. O Código 11 ilustra como recupera-se essa instância a partir do ORB.

Em posse do *OpenBusContext*, podemos fazer a chamada a *getCallerChain* em *sayHello*. Um exemplo pode ser visto no Código 12

Desta forma implementamos um serviço que imprime o nome da entidade que fez a chamada remota para nosso serviço. O código completo pode ser conferido na demo Hello do ASSISTANT.

Código 11: Acessando o *OpenBusContext*.

```
1 ORB orb = assist.orb();
2 // recuperando o gerente de contexto de chamadas à barramentos
3 OpenBusContext context =
4   (OpenBusContext) orb.resolve_initial_references("OpenBusContext");
```

Código 12: Implementação da faceta Hello acessando a cadeia relativa a uma chamada

```
1 public final class HelloImpl extends HelloPOA {
2   private OpenBusContext context;
3   public HelloImpl(OpenBusContext context) {
4     this.context = context;
5   }
6   @Override
7   public void sayHello() {
8     CallerChain callerChain = context.getCallerChain();
9     LoginInfo caller = callerChain.caller();
10    String hello = String.format("Hello %s!", caller.entity);
11    System.out.println(hello);
12  }
13 }
```

## 4.3 Configurações avançadas

### 4.3.1 Definindo os níveis de log do JacORB

o JACORB utiliza o SLF4J para realizar seu mecanismo de logging. O SLF4J é uma fachada de logging que pode ser utilizada em conjunto com outras soluções de logging arbitrárias, como Log4J, JCL, ou JDK. Para alterar para uma solução de logging diferente, basta incluir a biblioteca no *classpath*. O sistema de logging padrão adotado na distribuição do JACORB é o JDK.

As semânticas associadas aos níveis de log utilizados pelo JACORB são:

**error** Eventos que sugerem um erro no JACORB ou no código do usuário. Isto inclui, mas não restringe, erros fatais que ocasionarão o término do programa.

**warn** Eventos que demandam atenção mas são tratados corretamente de acordo com a especificação CORBA. Por exemplo, término inesperado de uma conexão, falta de recurso (fila cheia), entre outros.

**info** Inicialização e término de subsistemas, realização e finalização de conexões, registro de objetos no POA.

**debug** Informações que podem ser necessárias para a identificação de erros no JACORB ou código do usuário.

**trace** Não é utilizado pelo JACORB, e é desencorajado pela equipe do SLF4J.

A propriedade *jacorb.log.default.verbosity* especifica o nível em que as mensagens são registrados. O valor é um número de 0 a 4, onde 0 significa que não há registro, 1 significa apenas mensagens de erro, 2 significa mensagens de aviso (*warning*), 3 significa mensagens de informação, e 4 significa mensagens de depuração. Os níveis mais elevados incluem também níveis mais baixos.

A propriedade *jacorb.logfile* define o arquivo a ser utilizado para registrar as mensagens de log. Caso a propriedade não seja configurada, as mensagens de log são direcionadas para a saída padrão.

Maiores informações e opções sobre o mecanismo de log do JACORB podem ser encontradas em sua documentação. [2]

### 4.3.2 Definindo os níveis de log da API

A API utiliza o log do Java, que possui os níveis de log descritos na classe *java.util.logging.Level*. Para alterar programaticamente o nível de log da API deve-se chamar o método *setLevel()* enviando o nível desejado como

parâmetro. Um exemplo de configuração do log é apresentado no Código 13. Note que recuperamos o logger através do pacote “tecgraf.openbus”, que é um pacote comum a todas as classes do SDK.

Código 13: Configurando o log da API.

```
1 Logger logger = Logger.getLogger("tecgraf.openbus");
2 logger.setLevel(level);
3 logger.setUseParentHandlers(false);
4 ConsoleHandler handler = new ConsoleHandler();
5 handler.setLevel(level);
6 logger.addHandler(handler);
```

### 4.3.3 Definindo as propriedades opcionais do Assistente

Ao instanciar o ASSISTANT podemos definir um conjunto de propriedades, através da classe *tecgraf.openbus.assistant.Assistant* para configurar o assistente. Essa classe fornece uma série de campos opcionais que podem ser configurados antes da instânciação do ASSISTANT. São eles:

**interval** Tempo em segundos indicando o tempo mínimo de espera antes de cada nova tentativa após uma falha na execução de uma tarefa.

**orb** O ORB a ser utilizado pelo assistente para realizar suas tarefas.

**connprops** Propriedades opcionais da conexão utilizada internamente. Para mais detalhes, consulte a documentação da API básica.

**callback** Objeto de callback que recebe notificações de falhas das tarefas realizadas pelo assistente.

## Referências

- [1] C. Augusto, E. Fonseca, L. Marques, S. Correa, and R. Cerqueira. SCS: Software component system. <http://www.tecgraf.puc-rio.br/scs>, May 2006.
- [2] JacORB. Jacorb. <http://www.jacorb.org>, 2009.
- [3] Object Management Group. *The Common Object Request Broker Architecture (CORBA) Specification - Version 3.1*, January 2008. document: formal/2008-01-04.
- [4] TecGraf. OpenBus - Enterprise Integration Application Middleware. <http://www.tecgraf.puc-rio.br/openbus>, 2006.
- [5] TecGraf. *Manual do OpenBus 2.0.0*. TecGraf, 2012.