Openbus SDK-Java 1.5.0 - Documentação

Maíra Gatti
Pontifícia Universidade Católica do Rio de Janeiro (PUC-Rio)
openbus-dev@tecgraf.puc-rio.br

Março 2011

1 Visão Geral

O OpenBus é um barramento de integração de serviços orientado [1] a componentes [2] e baseado em CORBA [3][4]. O objetivo deste documento é o de descrever a estrutura e dinâmica da API sdk-java 1.5.0 de acesso ao barramento para serviços desenvolvidos na linguagem Java. A API sdk-java 1.5.0 utiliza JacORB [5], uma implementação do padrão ORB de CORBA em Java. Este documento não descreve o funcionamento interno do Openbus em si, também não descreve em detalhes o JacORB ou o modelo de componentes SCS [6] no qual o Openbus foi desenvolvido. Para mais informações sobre como usar a API Java do SCS ou sobre informações básicas de uso e configuração da API sdk-java para usuários iniciantes, veja o tutorial. Enquanto o tutorial descreve como **usar** a API, este documento descreve como **funciona** a API, ou seja, sua dinâmica interna.

Desta forma, considera-se como pré-requisito para um bom entendimento deste documento o conhecimento básico dos seguintes assuntos:

- CORBA.
- JacORB.
- Modelo de Componentes SCS v1.2.
- Conceitos básicos do Openbus.
- Linguagem de programação Java.
- Orientação a Objetos

2 Entidades e Relacionamentos

A API é composta pela fachada *openbus.Openbus*, pela classe *stub IRegistryService*, referente ao Serviço de Registro, pela classe *openbus.FaultToleranceManager* e pelos pacotes *authenticators*, *lease*, *interceptors*, *util* e *exception*.

Para um melhor aproveitamento da API, o desenvolvedor deverá usar o máximo possível as operações oferecidas pela fachada *openbus.Openbus*, uma

vez que ela encapsula os mecanismos de validação e renovação de credencial, obrigatórios para o acesso ao barramento, e encapsula os mecanismos opcionais de cache e de tolerância a falhas que melhoram a qualidade do serviço do barramento. Por outro lado, é importante entender como usar as operações do Serviço de Registro, uma vez que a fachada *openbus.Openbus* não disponibiliza facilitadores em seu uso.

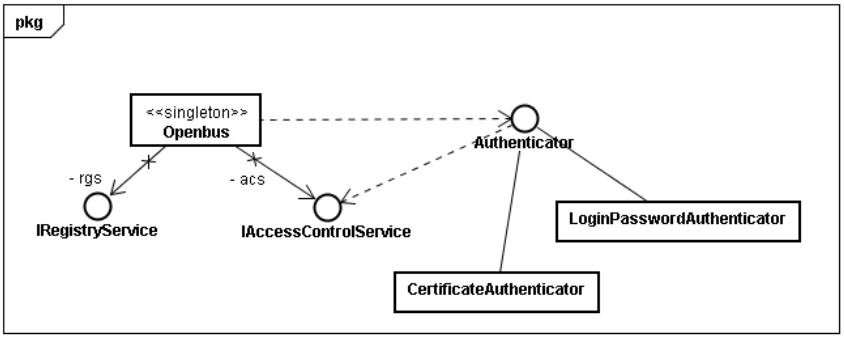

Figura 1: Entidades de conexão com o barramento.

Na figura 1 são ilustradas as entidades utilizadas no processo de conexão com o barramento. Dependendo do tipo de conexão solicitada pelo cliente, a fachada *Openbus* utilizará o autenticador por login e senha, ou o autenticador por certificado, cujas classes implementam a interface *Authenticator*. Ambas dependem do proxy da faceta do Serviço de Controle de Acesso que implementa a interface *IAccessControlService*. O mesmo para a faceta do Serviço de Registro, *IRegistryService*.

A figura 2 ilustra as entidades envolvidas no processo de interceptação de uma requisição, e de validação de credencial do cliente.

Sempre que o barramento é inicializado, a fachada *Openbus* inicializa o ORB, que no caso é o *JacORB* com os interceptadores *ClientInterceptor* e um dos interceptadores de servidor representados pelas classes *ServerInterceptor*, *CredentialValidatorServerInterceptor* e *CachedCredentialValidatorServerInterceptor*, que é escolhido de acordo com a política de validação de credencial.

O interceptador *ClientInterceptor* é responsável por pegar a credencial salva na fachada *Openbus* após o login no Serviço de Controle de Acesso, e inserir no contexto da requisição. Do lado do servidor, o interceptador recupera a credencial do contexto da requisição e verifica se a credencial é válida.

O interceptador *CachedCredentialValidatorServerInterceptor* quando inicializado na aplicação servidor cria uma *task*, *CredentialValidatorTask* que de tempos em tempos executa a ação de verificação das credenciais salvas e encapsuladas pela classe *CredentialWrapper*. Quando a aplicação cliente inicializa a fachada *Openbus*, ela pode definir a política de validação de credencial que decidirá se esta *task* será ativada.

Finalmente a classe *FTClientInterceptor* é responsável por tratar requisições que retornaram com exceção e, dependendo da exceção, acionar o gerenciador de tolerância a falhas, *FaultToleranceManager*, que contém a lista de endereços

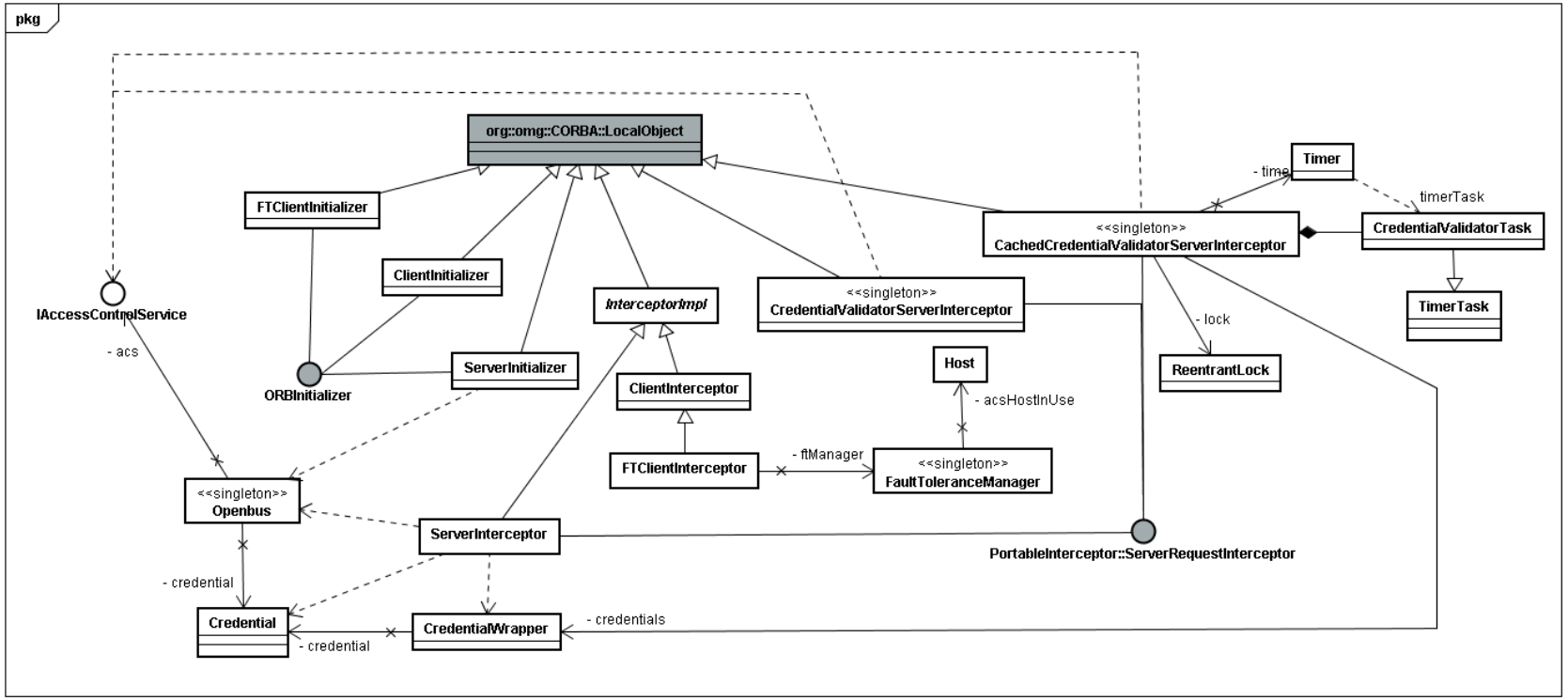

Figura 2: Entidades relacionadas à interceptação e validação de credenciais.

para as réplicas do Serviço de Controle de Acesso. Tal mecanismo é ativado somente se os endereços das réplicas forem corretamente configurados.

A figura 3 por sua vez ilustra as entidades relacionadas com o processo de renovação da *lease*[1] de uma credencial. Se a credencial adquirida durante a conexão com o barramento não for renovada pela classe *LeaseRenewer* em um tempo menor que o de sua expiração, ela não será mais válida no barramento e consequentemente a aplicação cliente do barramento não poderá nem ofertar nem consumir ofertas de serviços. Além disso, todas as ofertas já cadastradas no Serviço de Registro serão removidas.

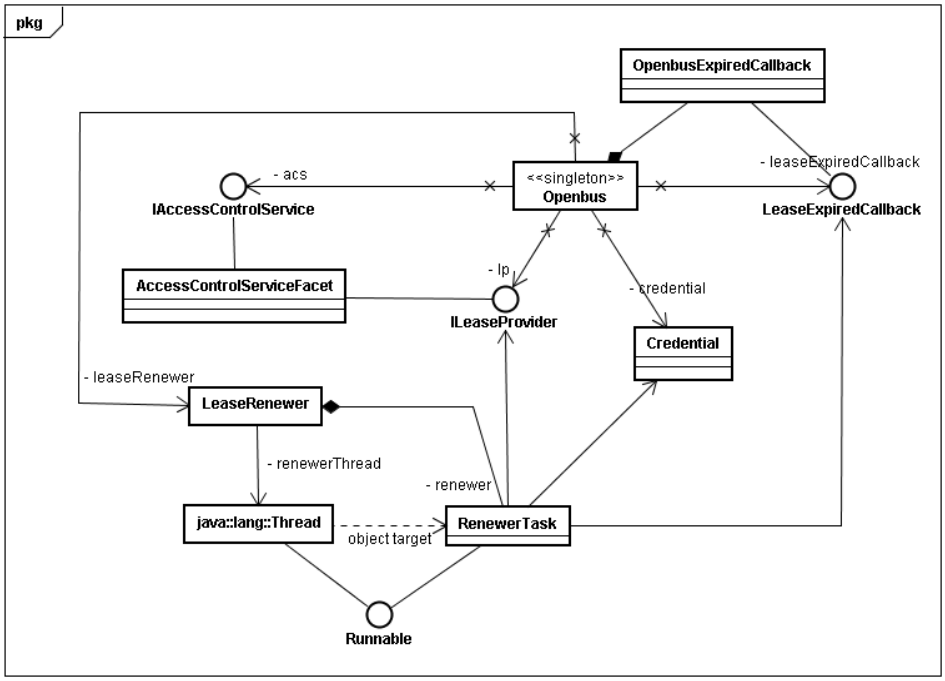

Figura 3: Entidades relacionadas à renovação da lease da credencial. *RenewerTask* é *inner class* da classe *LeaseRenewer* e implementa um *Runnable* para que, de tempos em tepos, possa verificar junto ao provedor do *lease* (*ILeaseProvider*) se a credencial é válida.

A API sdk-java-1.5.0 fornece operações na fachada *Openbus* para o cliente especificar a classe que implementa a interface *LeaseExpiredCallBack*. A classe *LeaseRenewer* chama operações desta interface no momento em que a credencial expira. A fachada *Openbus* por sua vez define a *inner class OpenbusExpiredCallback* que implementa estas operações funcionando como um delegador.

Além disso o Serviço de Controle de Acesso sendo provedor de *leases* implementa a interface *ILeaseProvider*.

[1] *Lease* é um termo originalmente usado para indicar uma locação. No contexto de credenciais, *lease* significa o contrato de uso da credencial por um determinado período, período este definido pelo barramento.

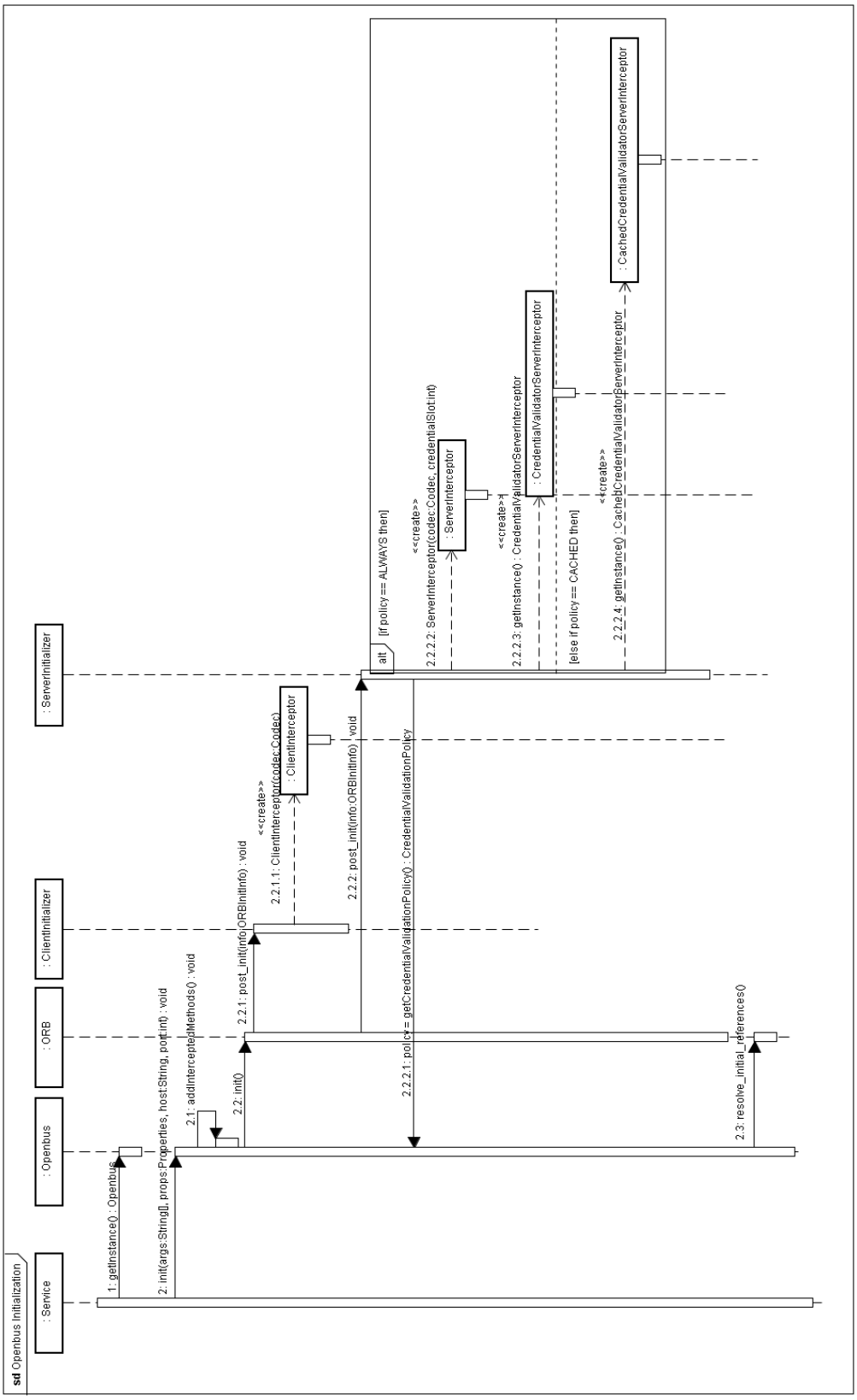

Figura 4: Inicialização do Openbus

3 Inicialização do Openbus

A figura 4 descreve o processo de inicialização do barramento a partir da instanciação da classe *Openbus* seguida da chamada do método *init* pela aplicação cliente. Em seguida a classe *Openbus* define quais requisições são interceptáveis [7] e inicializa o ORB enviando um arquivo de propriedades com os responsáveis por inicializar os interceptadores que serão por sua vez instanciados pelo ORB e criarão os interceptadores de fato. Se o cliente tiver definido a política de validação de credencial, ela será verificada pelo *ServerInitializer* para saber qual interceptador será escolhido (além do *ServerInterceptor*): *CredentialValidatorServerInterceptor* ou *CachedCredentialValidatorServerInterceptor*.

4 Mecanismo de Conexão e Desconexão

Após inicializar o barramento, a aplicação cliente pode se conectar. A conexão pode ser feita de duas formas: por *login* e senha ou por certificado através das operações *connect(user, password)* e *connect(name, privateKey, acsCertificate)*, respectivamente.

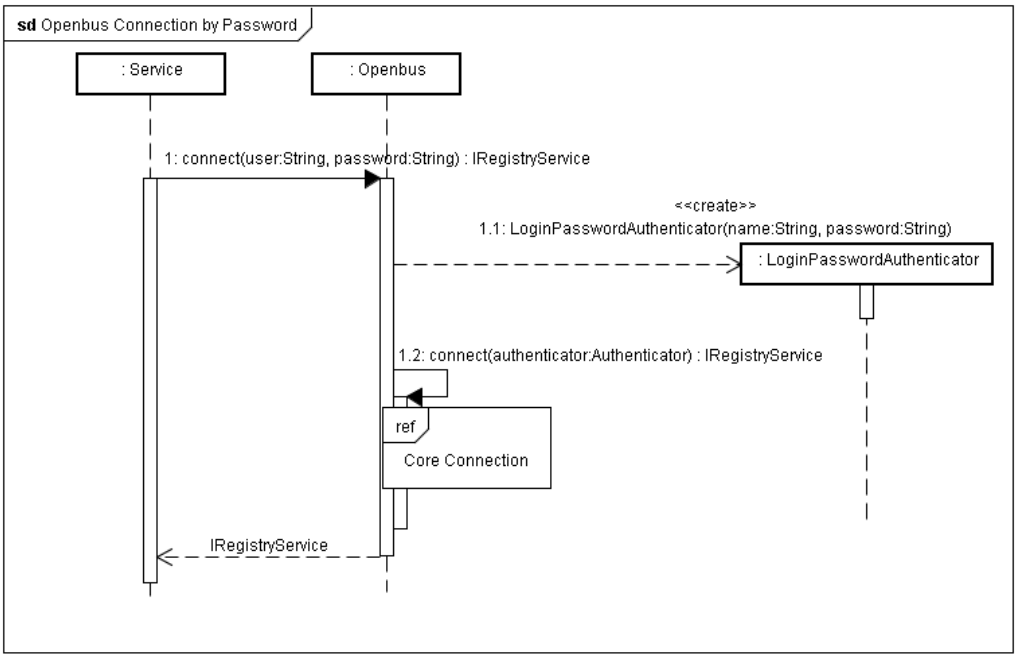

Figura 5: Conexão por senha

A figura 5 ilustra o processo de conexão através de *login* e senha[2]. Durante este processo, o autenticador *LoginPasswordAuthenticator* é criado com as informações de *login* e senha. De posse da instância do autenticador, a classe *Openbus* pode então realizar o procedimento de conexão básico descrito no diagrama *Core Connection* (figura 7) que, em um dado momento, irá chamar a

[2]Veja como é análogo o processo por certificado na figura 6.

operação *authenticate()* que, por sua vez, irá diferenciar o tipo de acesso ao Serviço de Controle de Acesso propriamente dito.

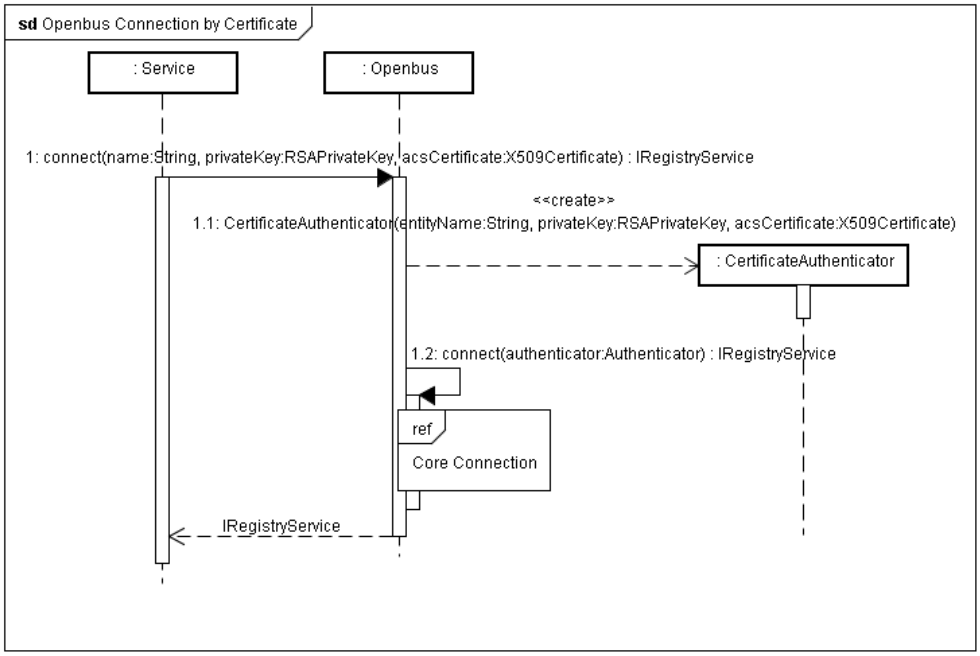

Figura 6: Conexão por certificado

O primeiro passo do processo de conexão básico (veja figura 7) é verificar se o cliente já não está conectado. Para isto, a classe *Openbus* verifica se possui alguma credencial salva em seu estado. Não está ilustrado no diagrama, mas se já estiver conectado, o barramento não deixa se conectar novamente e simplesmente retorna falso. Se nenhuma credencial foi encontrada e não existe uma referência para o proxy do Serviço de Controle de Acesso, a classe *Openbus* primeiramente tenta encontrá-la. Isso sempre acontecerá na primeira conexão uma vez que esta referência é postergada até este momento. Depois a referência é salva e, se ela se tornar inválida ou inacessível, a classe *Openbus* tentará obter uma nova (no caso de existirem réplicas do Serviço de Controle de Acesso e elas tiverem sido configuradas corretamente pelo cliente).

De posse da referência para o proxy do Serviço de Controle de Acesso, a classe *Openbus* chama a operação *authenticate()* na instância do autenticador que, no caso do autenticador *LoginPasswordAuthenticator*, simplesmente chama a operação *loginbypassword()* no Serviço de Controle de Acesso. Para o caso do autenticador *CertificateAuthenticator*, é preciso ler o certificado passado, entre outras operações, e esta interação está descrita no diagrama *Certificate Authenticator Process*, figura 8.

Se a operação *loginbypassword()* for executada com sucesso, a classe *Openbus* receberá a credencial e a *lease* da credencial. A conexão é finalizada com a instanciação da *call back* responsável por ser notificada se a *lease* expirar, que é então passada para a instância da classe *LeaseRenewer* criada nesse momento que, por sua vez, inicia a *task* de renovação através da mensagem *start()* em

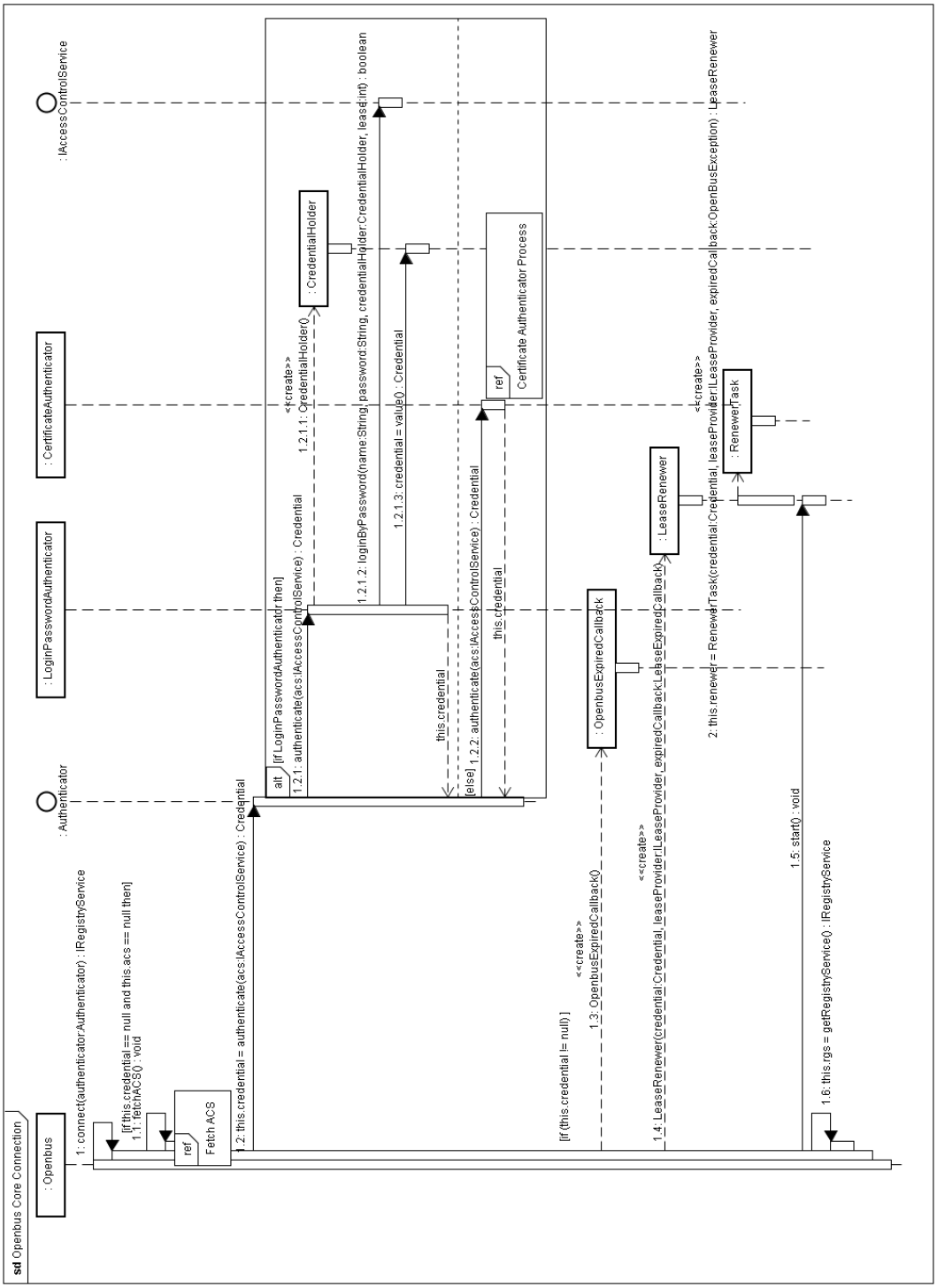

Figura 7: Processo de conexão básico

RenewerTask. Por fim, a referência para o Serviço de Registro é obtida a partir do Serviço de Controle de Acesso e retornada para o cliente.

O processo de autenticação por certificado através da classe *CertificateAuthenticator* por outro lado precisa ler o certificado por criptografia antes de efetuar o login no barramento. Este procedimento é feito a partir da classe fornecida *CriptoUtils* e é ilustrado na figura 8.

Primeiro é obtido um desafio junto ao Serviço de Controle de Acesso, depois uma resposta é gerada a partir do desafio obtido e da chave privada criptografada (chave esta correspondente à chave pública utilizada). Com esta resposta, é possível chamar a operação *loginByCertificate* no Serviço de Controle de Acesso, que, se bem sucedida, retorna a credencial e a *lease*.

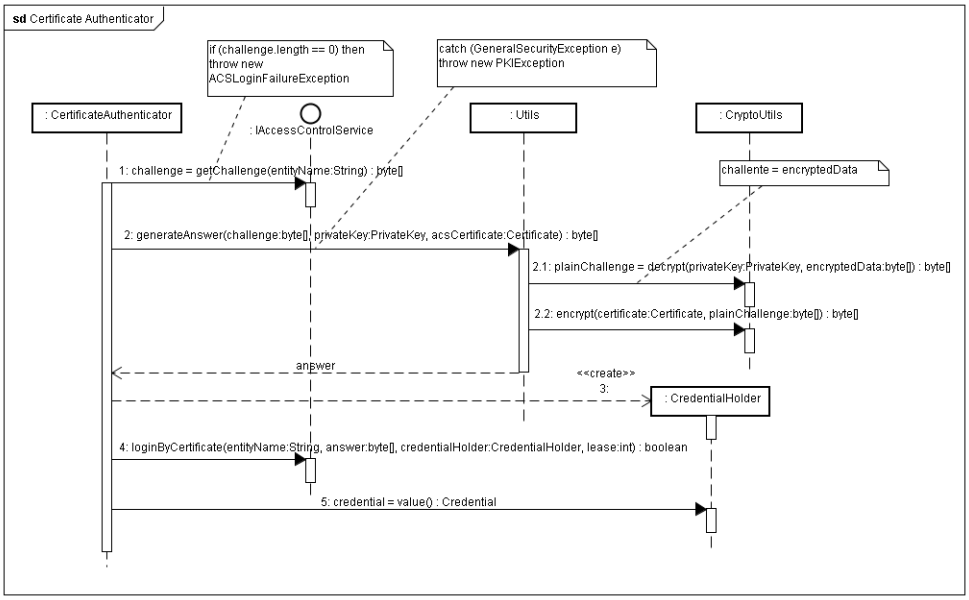

Figura 8: Leitura do Certificado

O processo de desconexão é simples (figura 9). Uma vez acionado pelo cliente, se existir credencial no estado da classe *Openbus*, seu processo de renovação será parado, a credencial será deslogada do Serviço de Controle de Acesso (que por sua vez remove todas as ofertas associadas a ela), o estado do barramento será zerado, e o cliente receberá um retorno de sucesso. Se algo de errado acontecer durante esse processo, o cliente receberá uma mensagem de erro.

Para fins de registro, a figura 10 ilustra como a referência para as facetas do Serviço de Controle de Acesso é recuperada a partir da API do SCS e dos esqueletos gerados pelo JacORB.

5 Mecanismo de Lease e Renovação da Credencial

Durante o processo de conexão, como descrito na seção anterior, após o *login* no barramento, seja por senha, seja por certificado, a API recebe a *lease* da credencial que é a sua validade. O processo então de renovação da credencial é

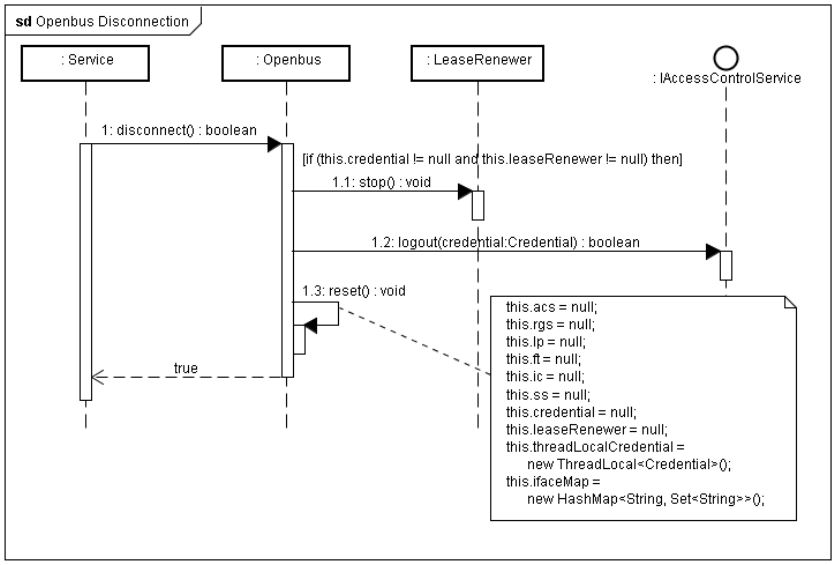

Figura 9: Desconexão

iniciado com a inicialização de *LeaseRenewer* através da operação *start()* (vide figura 11). Note que durante a inicialização, existe um controle em relação a *RenewrTask* de forma que, se *LeaseRenewer* tiver uma referência para a sua *thread*, ou seja, ela já existia previamente, ela precisa antes ser parada corretamente.

A figura 12 ilustra o que acontece sempre que *RenewerTask* é executada após seu início. Ela executa a operação *renewLease(credential, lease)* no provedor da *lease*, ou seja, faceta *ILeaseProvider* implementada pelo Serviço de Controle de Acesso. Se a renovação for executada com sucesso, uma nova *lease* é fornecida. O tempo da *thread* da *task* é atualizado com este valor.

Caso o processo de renovação não seja autorizado, *RenewerTask* irá enviar a mensagem *expired()* para a instância da classe que implementa a interface *LeaseExpiredCallback*, no caso fornecida pela API, a classe *OpenbusExpiredCallback*. Como já explicado anteriormente, a API foi desenhada para que a fachada *Openbus* implemente esta interface sendo um delegador para o objeto criado pela aplicação cliente. Antes de delegar, a fachada *Openbus* tem seu estado reiniciado, ou seja, todas as referências para as facetas do Serviço de Controle de Acesso e Registro são invalidadas, assim como a credencial que estava sendo utilizada.

É importante notar que a API não fornece uma implementação padrão de reconexão da *callback* uma vez que poderia induzir o desenvolvedor de um servidor cliente do barramento a usá-la e não considerar que as ofertas registradas não foram registradas novamente com a re-conexão (e, lembrando, elas são removidas quando uma credencial expira ou é deslogada manualmente).

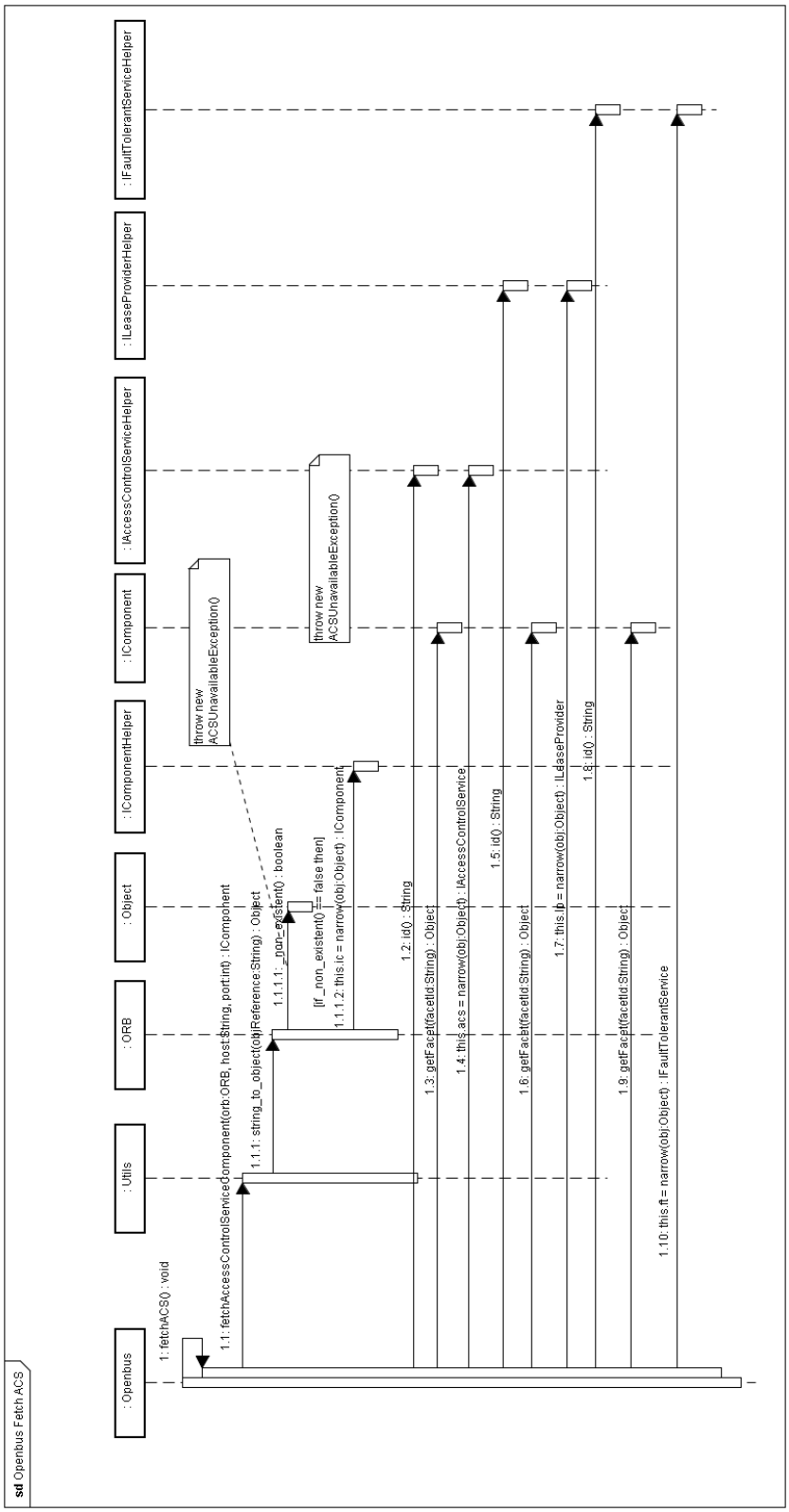

Figura 10: Obtenção das referências das facetas do Serviço de Controle de Acesso

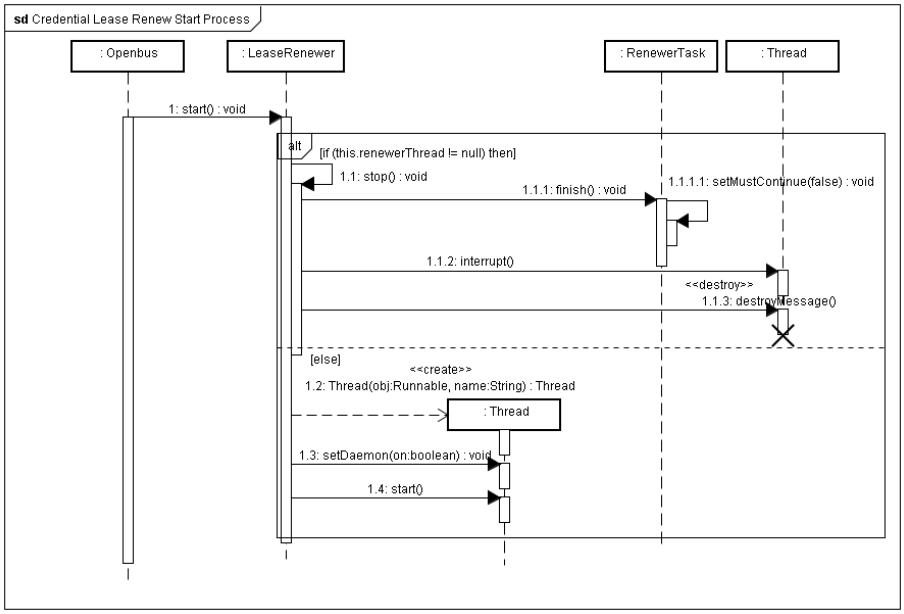

Figura 11: Inicialização do Processo de Renovação de Lease de Credencial

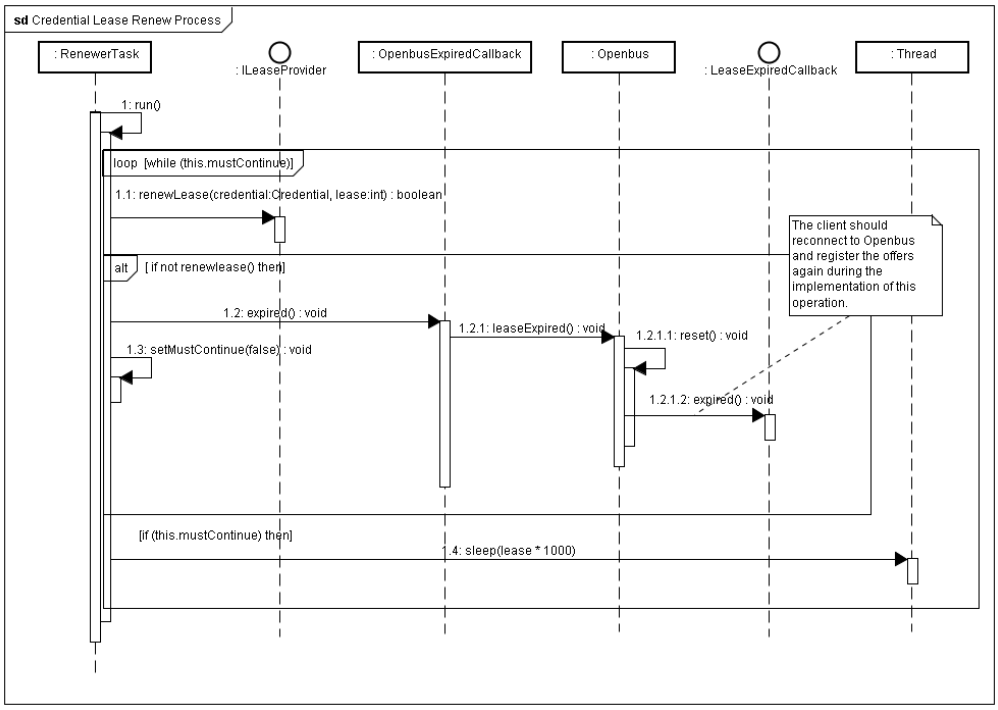

Figura 12: Processo de Renovação de Lease de Credencial

6 Mecanismo de Interceptação e Validação da Credencial

Como já visto, o JacORB é inicializado com interceptação de requisições. Desta forma, sempre que uma requisição para os serviços básicos for realizada, tal como pedir para cadastrar uma oferta, ou removê-la, ou uma requisição para um serviço registrado no barramento, tais requisições serão interceptadas para verificar se possuem credencial e se a credencial é válida no Serviço de Controle de Acesso.

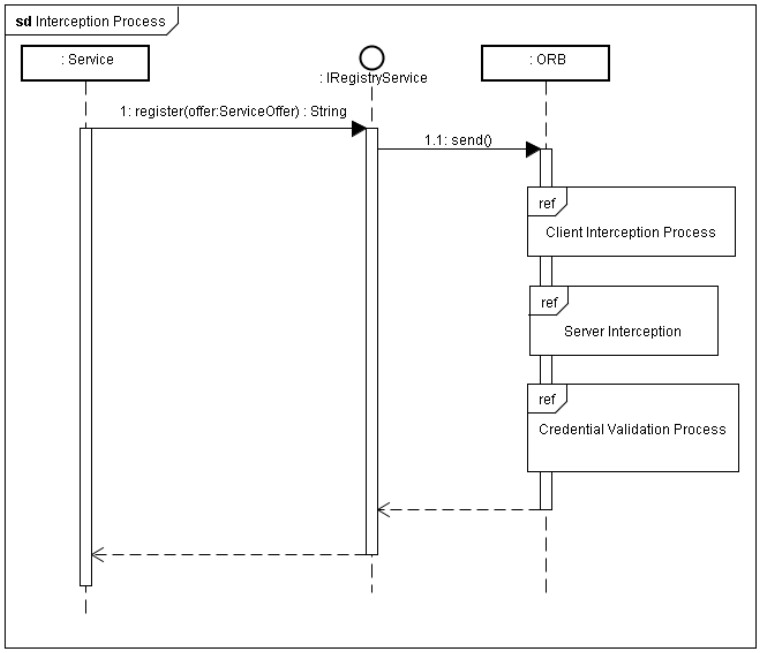

Figura 13: Processo básico de interceptação

A figura 13 ilustra as interações envolvidas desde o momento em que o cliente envia uma mensagem para o servidor alvo, que no caso é está sendo ilustrada pela mensagem *register(offer)* para o Serviço de Registro até a sua resposta. Neste caso específico, como o Serviço de Registro foi desenvolvido na linguagem Lua, o mecanismo de interceptação utilizado está descrito no tutorial da API sdk-lua 1.5.2. O uso dos interceptadores servidores descritos neste tutorial será de servidores registrados no barramento e que foram desenvolvidos em Java.

Independente da política de validação de credencial, o interceptador *ServerInterceptor* sempre será instalado e executado uma vez que ele é responsável por recuperar a credencial do contexto da requisição e salvá-la na fachada *Openbus* para ser utilizada posteriormente pelo interceptador específico de validação. Nesta seção descrevemos o interceptador *CredentialValidatorServerInterceptor*, uma vez que ele é o interceptador padrão utilizado caso o cliente não defina nenhuma política.

A figura 14 ilustra a interceptação no lado do cliente a partir do redirecionamente do ORB para o interceptador após a chamada de um operação em

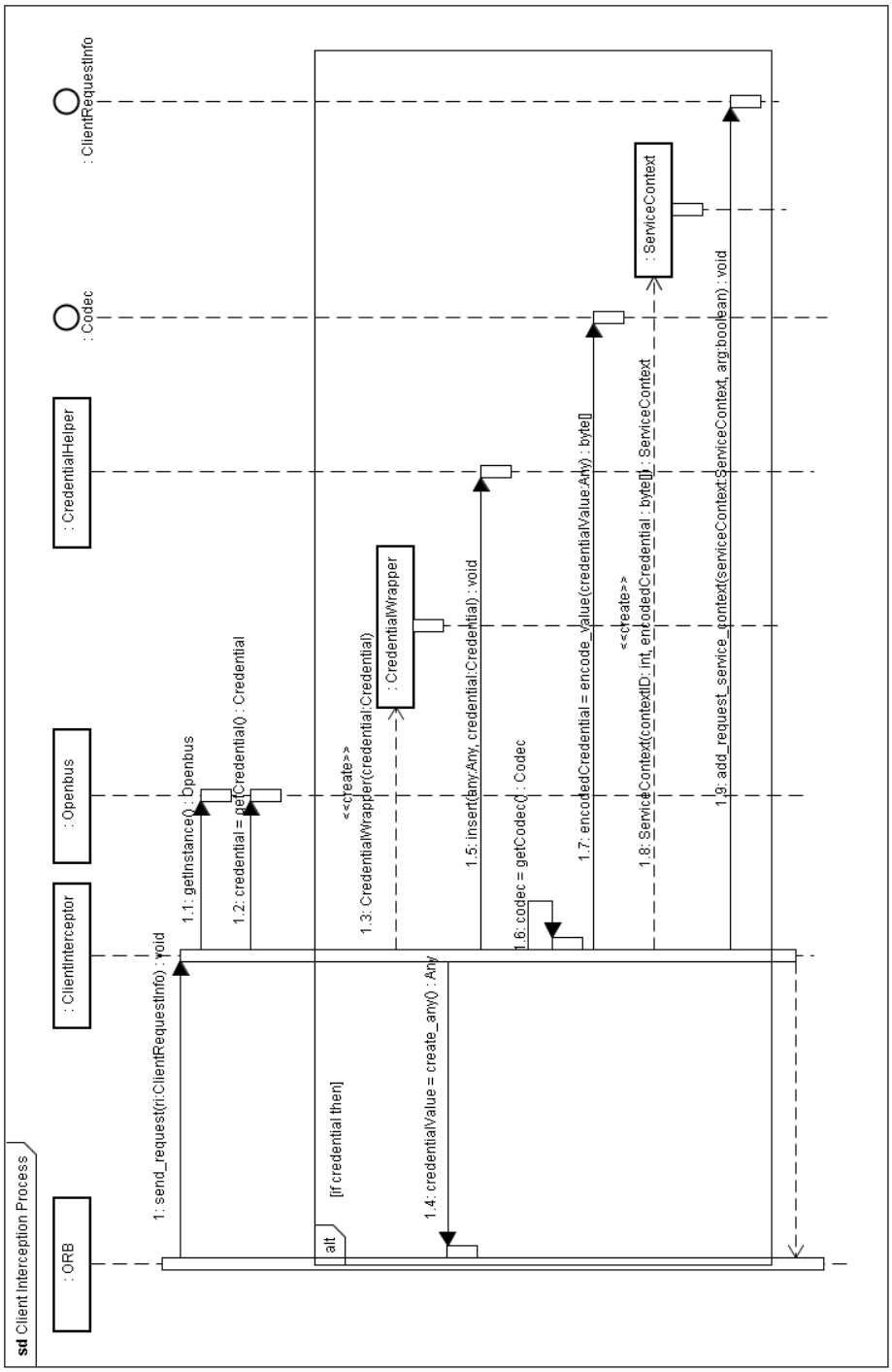

Figura 14: Interceptação no Cliente

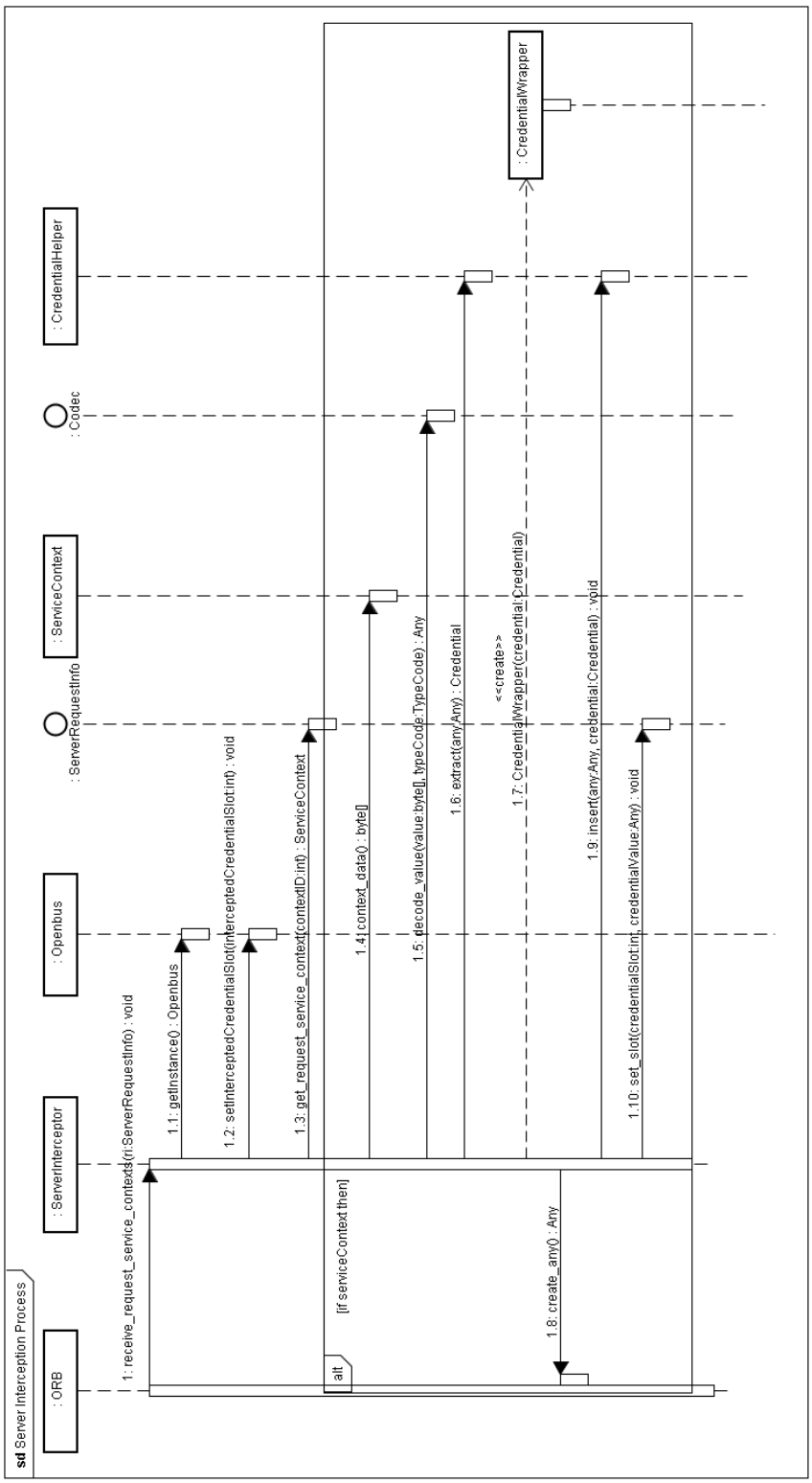

Figura 15: Interceptação no Servidor: Recuperação da Credencial

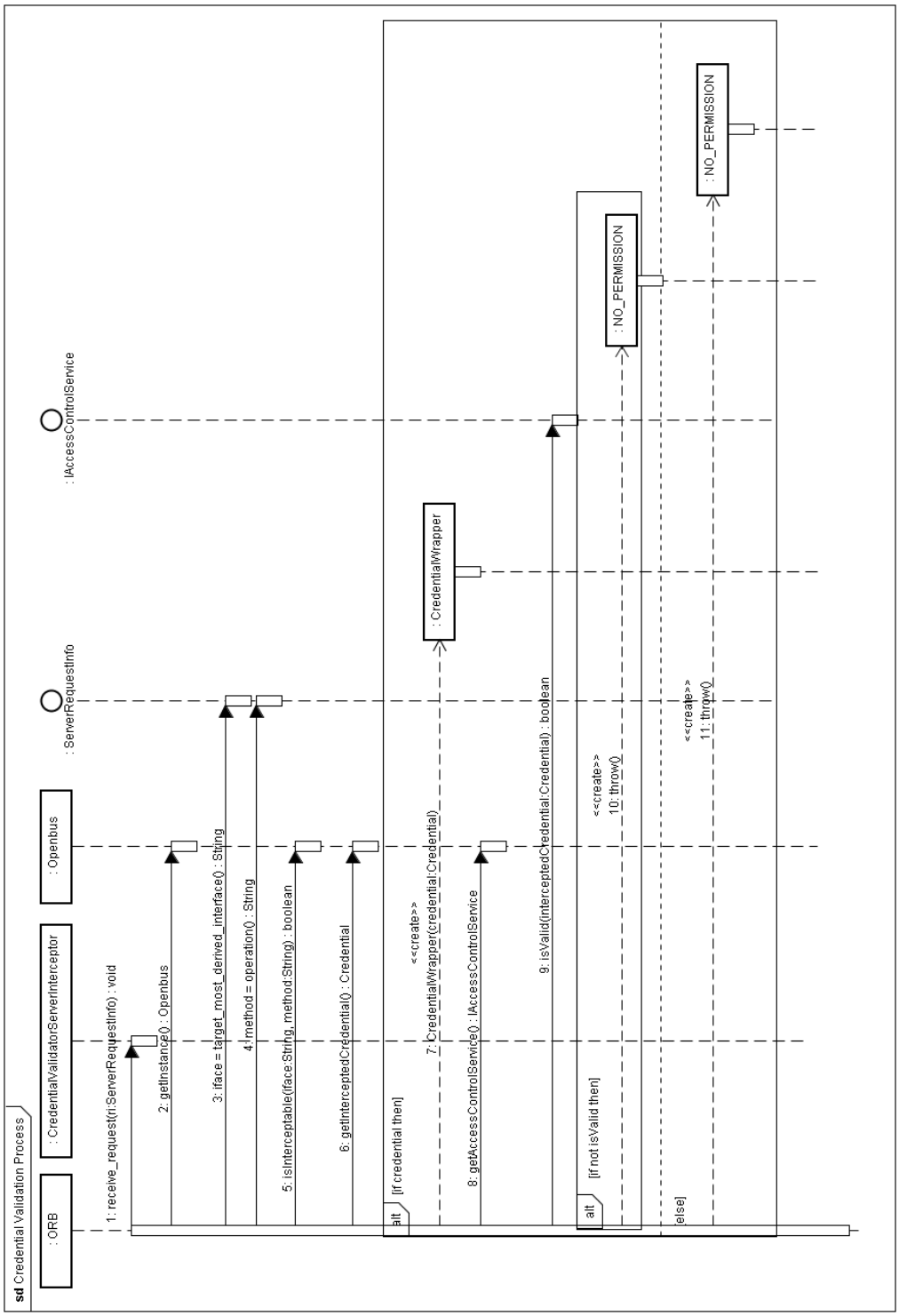

Figura 16: Interceptação no Servidor: Validação da Credencial

um *stub*, que pode ser do Serviço de Registro ou de outro serviço cadastrado no barramento, e a requisição é interceptada por *ClientInterceptor* através da mensagem *send_request()*. A credencial é então verificada e, caso exista, um invólucro é criado através da instanciação da classe *CredentialWrapper*. O invólucro é inserido no contexto da requisição após codificação necessária através da operação *add_request_service_context()* em *ClientRequestInfo*.

Durante a interceptação servidor iniciada com a mensagem *receive_request_service_contexts()* em *ServerInterceptor*, a credencial é recuperada do contexto através da troca de mensagens descrita na figura 15. A figura 16 por sua vez, descreve como o interceptador *CredentialValidatorServerInterceptor* valida a credencial interceptada, se for o caso, ou seja, se for uma operação interceptável, na faceta *IAccessControlService*. Se a credencial foi recuperada do contexto e é válida, a requisição é processada normalmente no servidor alvo, senão o interceptador servidor lançará a exceção *NO_PERMISSION*.

7 Qualidade do Serviço

A API sdk-java 1.5.0 fornece dois mecanismos que melhoram a qualidade do serviço de acesso ao barramento: mecanismo de cache e mecanismo de tolerância a falhas. As seções a seguir descrevem como eles funcionam.

7.1 Mecanismo de Cache

Mecanismos de Cache são os relativos as diferentes políticas de validação de credencial. Atualmente, existem duas políticas que podem ser usadas:

ALWAYS Indica que as credenciais interceptadas serão sempre validadas.

CACHED Indica que as credenciais interceptadas serão validadas e armazenadas em uma *cache*;

O padrão é *ALWAYS*, ou seja, sempre validadas. Porém o cliente pode inicializar o barramento com a política *CACHED* para otimizar as requisições, uma vez que se a credencial estiver armazenada na *cache* não será preciso enviar uma requisição de validação pela rede para o Serviço de Controle de Acesso.

A figura 17 ilustra a troca de mensagens durante o processo de validação de credencial usando o padrão *CACHED*. Quando a requisição do cliente for interceptada, a *cache* é bloqueada, a credencial é buscada na mesma através da tentativa de sua remoção, se encontrada e removida, ela é inserida novamente (porém no final da *cache*, uma vez que é uma lista ordenada por último acesso decrescentemente), a *cache* é então desbloqueada e redirecionada para o servidor alvo.

Se não for encontrada, a *cache* também é desbloqueada porém a credencial é verificada no Serviço de Controle de Acesso. Se a credencial foi validada, tenta-se adicioná-la na *cache*. Se a *cache* já tiver atingido o seu limite máximo permitido, a primeira credencial da lista é removida e a validada é adicionada por último. Se a credencial não for válida, o interceptador lança um exceção *NO_PERMISSION*.

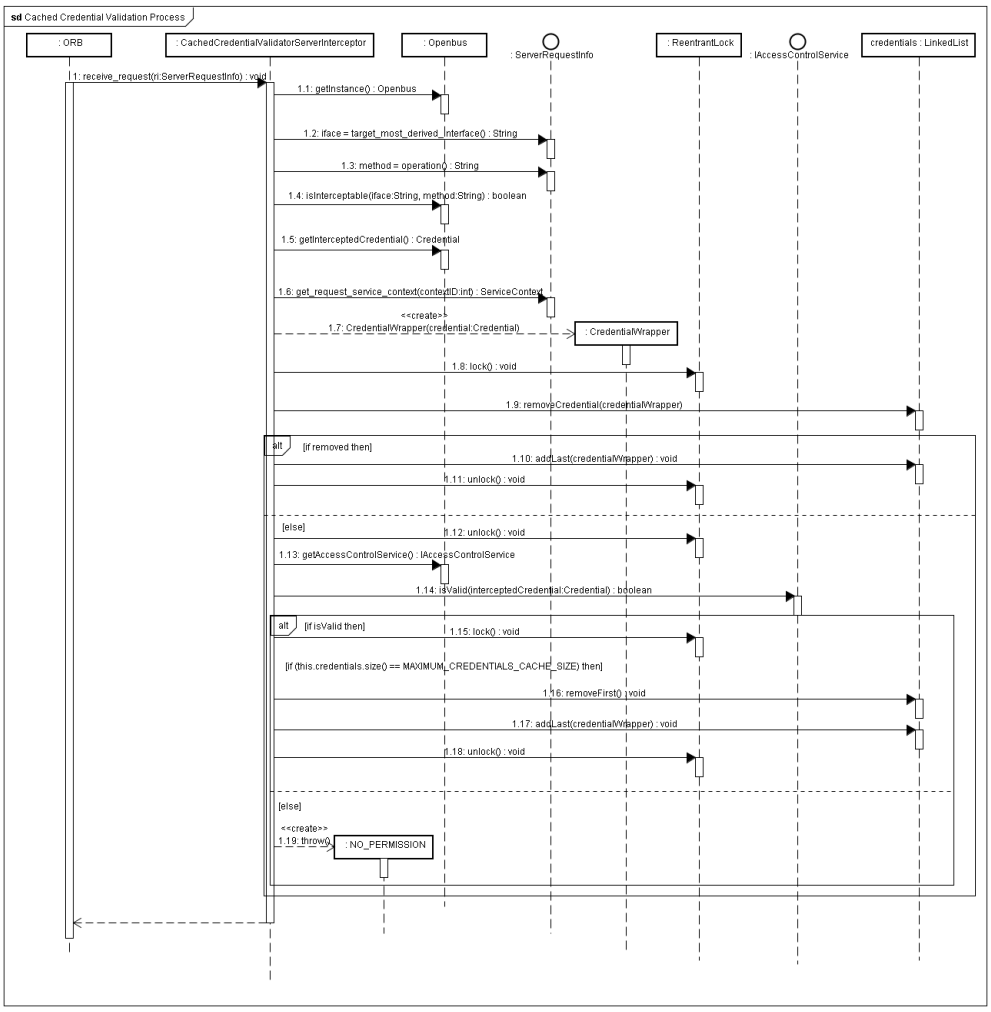

Figura 17: Mecanismo de Cache

Já a figura 18 ilustra a troca de mensagens relativa a criação da *task* responsável pelo processo de validação das credenciais que estão na *cache*, e ilustra o processo em si. Durante a criação, uma lista encadeada é criada, que conterá a lista de credenciais a serem armazenadas por ordem de acesso de forma que a última da lista foi a última a ser validada. O processo começa quando o *Timer*, após o tempo definido, envia a mensagem *run()* para *CredentialValidatorTask*. Este por sua vez bloqueia a *cache* e verifica todas as credenciais de uma só vez no Serviço de Controle de Acesso através da chamada *areValid()* na faceta *IAccessControlService*. Se alguma delas não for válida, é removida da *cache*. A *cache* é então desbloqueada.

7.2 Mecanismo de Tolerância a Falhas

O barramento possui um mecanismo de tolerância a falhas baseado em replicação. Ou seja, ele assume que pode existir mais de uma réplica do componente do

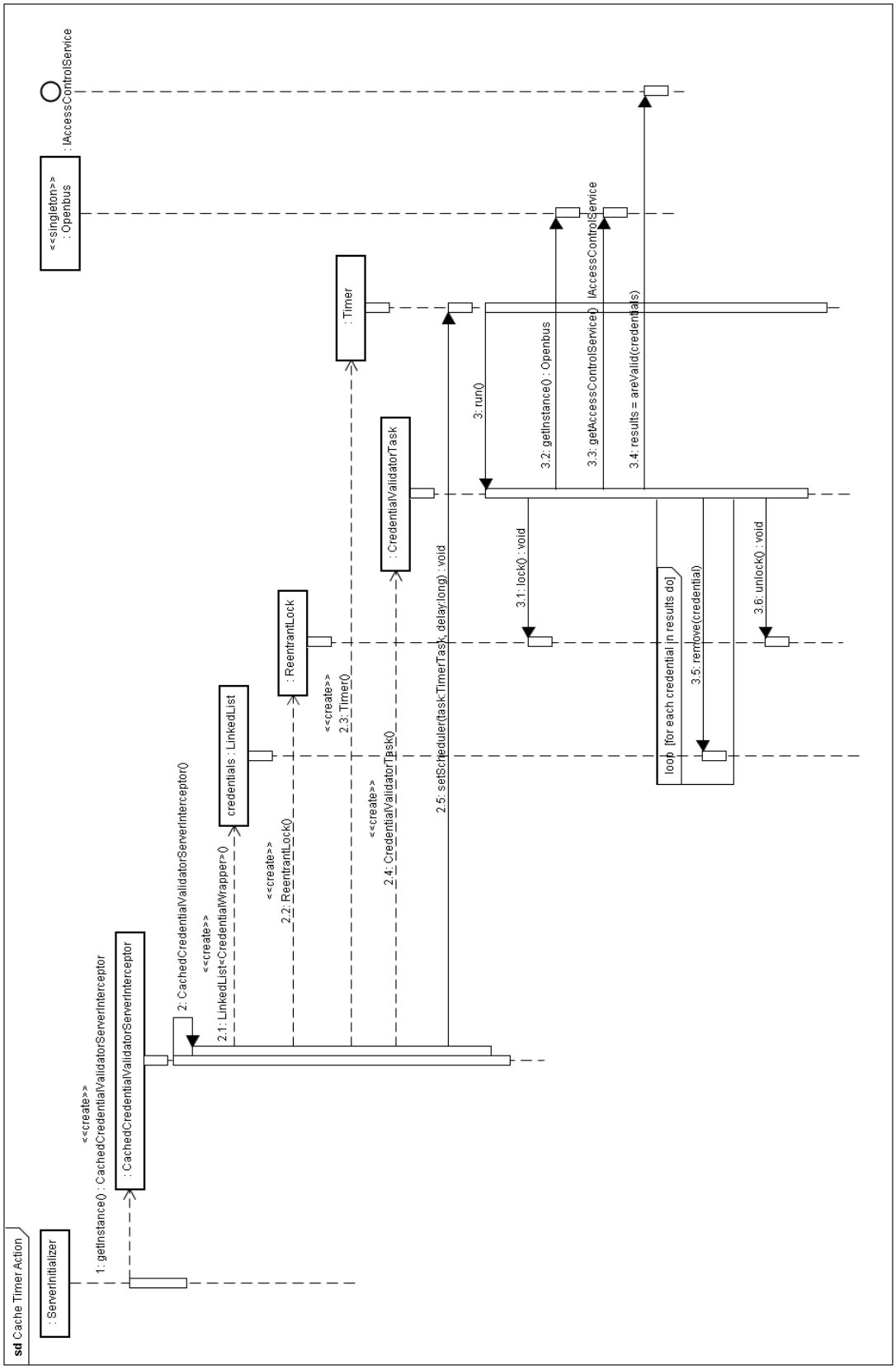

Figura 18: Validação no Serviço de Controle de Acesso

Serviço de Controle de Acesso ativa em um dado momento. Para que este mecanismo possa funcionar corretamente, é preciso configurar os endereços das réplicas do Serviço de Controle de Acesso acessíveis na propriedades *hosts* no arquivo *"/resources/FaultToleranceConfiguration.properties"*.

Além disso, neste mesmo arquivo de configuração é preciso definir a propriedade *trials*, que indica a quantidade de vezes que o gerenciador de tolerência a falhas *FaultToleranceManager* vai iterar na lista de réplicas enquanto não encontra uma disponível. O objetivo é que a busca seja finita.

Para habilitar o mecanismo de tolerância a falhas é preciso inicializar o barramento com a mensagem *initWithFaultTolerance()* ao invés de *init()* na fachada *Openbus* (para analogia, vide figura 4). Neste momento, ao invés de instalar o interceptador *ClientInterceptor*, será instalado o interceptador *FTClientInterceptor* pelo *FTClientInitializer*. Quando este interceptador é instânciado, uma única intância para o gerenciador de tolerância a falhas *FaultToleranceManager* é criada.

A figura 19 ilustra como a API trata a exceção através do interceptador *FTClientInterceptor* e do gerenciador *FaultToleranceManager*. Desta forma, suponha que o cliente tenha requisitado a operação *logout*, se uma das exceções abaixo for recebida, o interceptador *FTClientInterceptor* lançará um *ForwardRequest* enquanto o gerenciador *FaultToleranceManager* iterar na lista de réplicas configuradas e a quantidade de iterações definida na propriedade *trials* não for alcançada:

```
IDL:omg.org/CORBA/NO_RESPONSE:1.0
IDL:omg.org/CORBA/COMM_FAILURE:1.0
IDL:omg.org/CORBA/OBJECT_NOT_EXIST:1.0
IDL:omg.org/CORBA/TRANSIENT:1.0
IDL:omg.org/CORBA/TIMEOUT:1.0
IDL:omg.org/CORBA/NO_RESOURCES:1.0
IDL:omg.org/CORBA/FREE_MEM:1.0
IDL:omg.org/CORBA/NO_MEMORY:1.0
IDL:omg.org/CORBA/INTERNAL:1.0
```

Antes da busca acontecer, o interceptador primeiro recupera a chave do objeto remoto através da classe facilitadora *CorbaLoc* do JacORB. Isso é necessário uma vez que o interceptador é instalado por ORB, ou seja, todos os clientes compartilham do mesmo interceptador independente do objeto remoto alvo. Assim é preciso identificar tal objeto antes de tratar a exceção. Se for para uma das facetas do Serviço de Controle de Acesso [3], pede-se para o gerenciador para atualizar a referência para a réplica, se atualizada com sucesso, busca através da mensagem *1.2.1.2. fetchACS()* na fachada *Openbus*. A referência obtida é então passada para o *ForwardRequest*. Se a referência não tiver sido obtida e tiver acontecido alguma exceção, a busca é continuada.

[3] Para simplificar, o diagrama só ilustra a chave da faceta *IAccessControlService*, logo abaixo da mensagem 1.2.4, antes de entrar no *loop* com a condição de guarda *fetch* que, na primeira exceção será sempre verdadeira

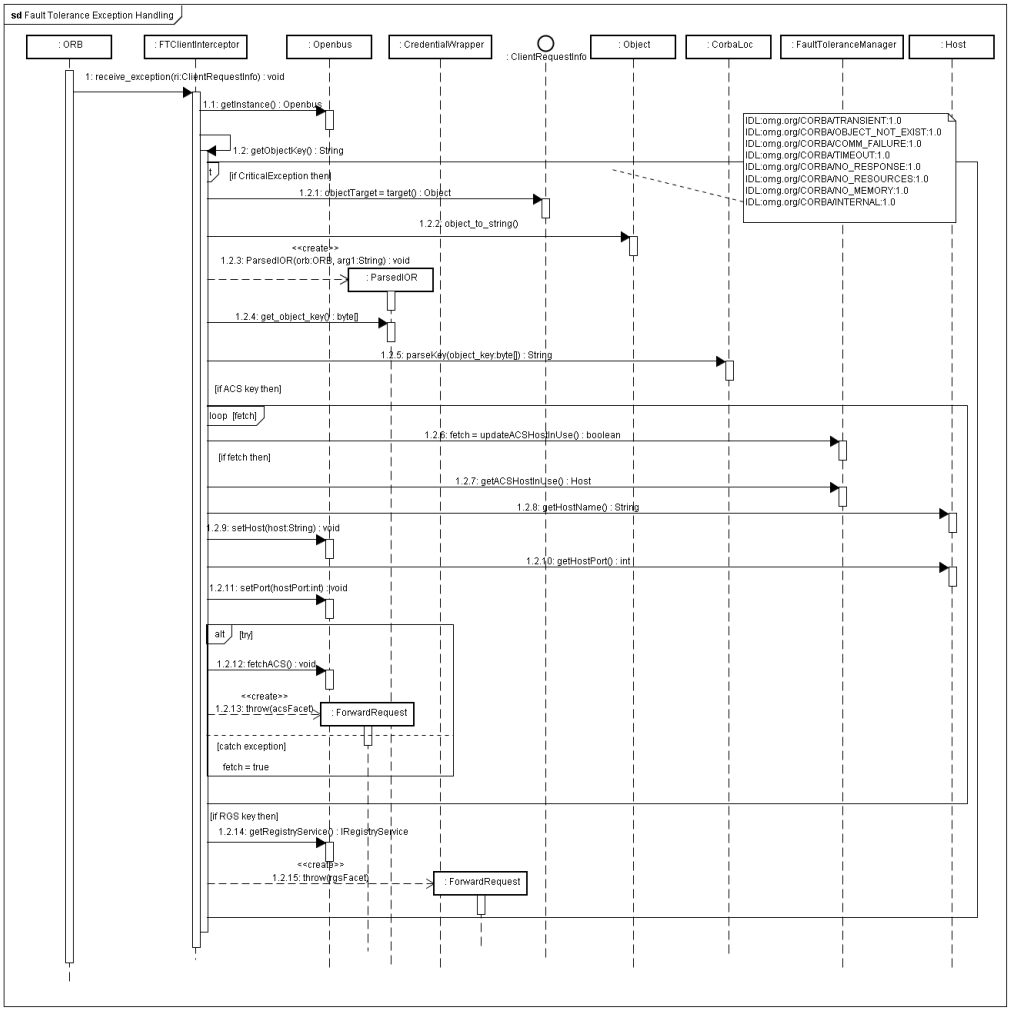

Figura 19: Mecanismo de Tratamento de Exceção do FT

A mensagem *1.2.1.4. getRegistryService()* na fachada *Openbus* só é executada se a chave obtida for a do Serviço de Registro, ou seja, a exceção foi pega após uma requisição na faceta *IRegistryService*. Nesse caso, como quem gerencia as réplicas do Serviço de Registro é o Serviço de Controle de Acesso, elas não são configuradas pelo cliente e é preciso buscá-las com esta indireção.

8 Ofertando Serviços

O processo básico de cadastro de oferta de serviços é ilustrado no diagrama da figura 20. O serviço a ser registrado no barramento deve ser descrito através de uma oferta de serviço, que é representada pela estrutura *ServiceOffer*.

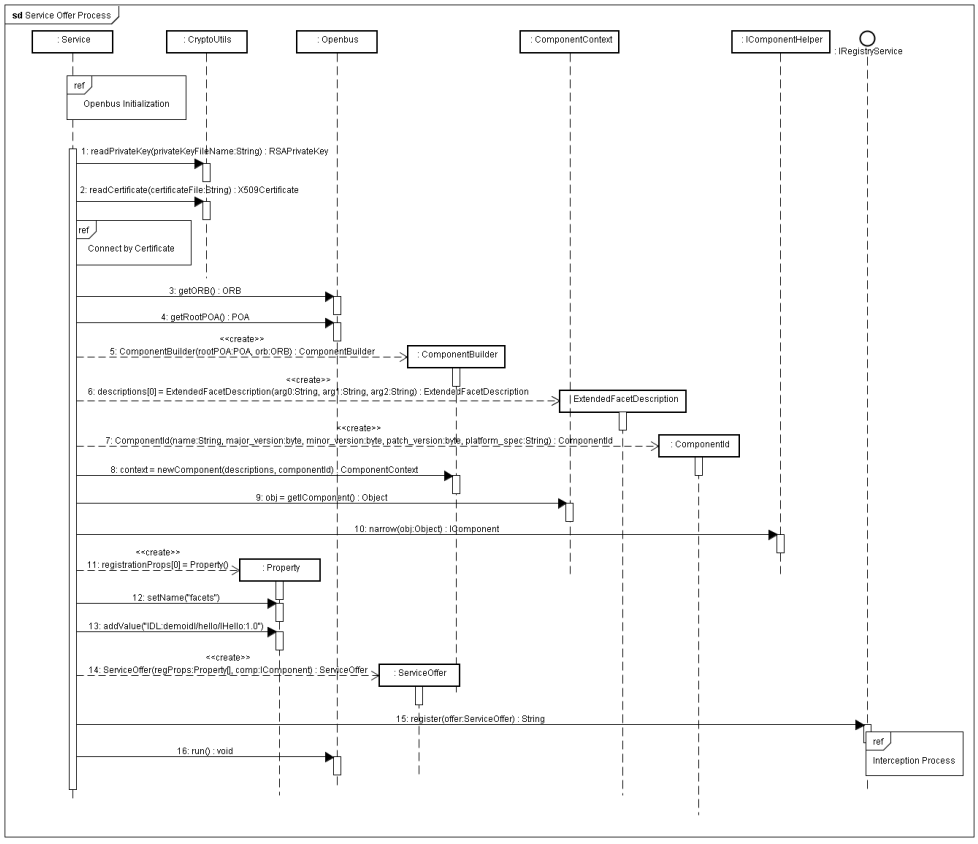

Figura 20: Processo básico de oferta de serviços

Para que um servidor possa ofertar serviços no barramento através da API sdk-java v1.5.2 é preciso:

1. Inicializar o barramento;
2. Carregar a chave privada e o certificado do ACS, caso a conexão seja por certificado (mensagens 1 e 2);
3. Conectar no barramento;

4. Utilizar a API do SCS para: (i) especificar a faceta a ser registrada através da classe *ExtendedFacetDescription*, (ii) seguida da criação do componente com esta descrição;

5. Instanciar a classe *Property* que contém informações da oferta a ser registrada, no caso, nome igual a *facets* e valor igual a *"IDL:demoidl/hello/IHello:1:0"* que representa o nome completo da interface da faceta;

6. Instanciar a classe *ServiceOffer* com a propriedade definida e a faceta *IComponent* do membro criado;

7. Cadastrar a oferta através da mensagem *register(serviceOffer)* enviada para a faceta *IRegistryService* do Serviço de Registro. Este, por sua vez, retorna o identificador da oferta registrada;

8. Habilitar o processo servidor a escutar as requisições CORBA que serão direcionadas para o serviço ofertado, o que pode ser feito através da mensagem *run()*;

Após a chamada da operação *register()*, a sequência de mensagens segue como descrito no diagrama de interceptação e validação de credenciais, veja figura 13.

9 Consumindo Serviços

O processo básico de busca de ofertas de serviços é ilustrado no diagrama da figura 21.

Para que uma aplicação cliente possa consumir serviços registrados no barramento através da API sdk-java v1.5.0 é preciso:

1. Inicializar o barramento;

2. Conectar no barramento;

3. Buscar o serviço através da mensagem *find(ifaceName)* e passando alguma informação sobre o serviço, tal como uma parte do nome da interface, como por exemplo *IHello*;

4. O cliente receberá uma lista com uma ou mais ofertas que implementam a faceta buscada, na qual deverá iterar e para cada uma delas recuperar a interface *IComponent* no campo *member* e mapeá-la usando a operação *narrow* da API do SCS;

5. Buscar a faceta do componente desejada, como por exemplo '*IDL:demoidl-/hello/IHello:1:0*' através da mensagem *3. getFacetByName(name)*;

6. Mapear a faceta retornada para a interface desejada usando a operação *narrow* da API do SCS; e finalmente

7. Executar o serviço desejado, como por exemplo *sayHello()*.

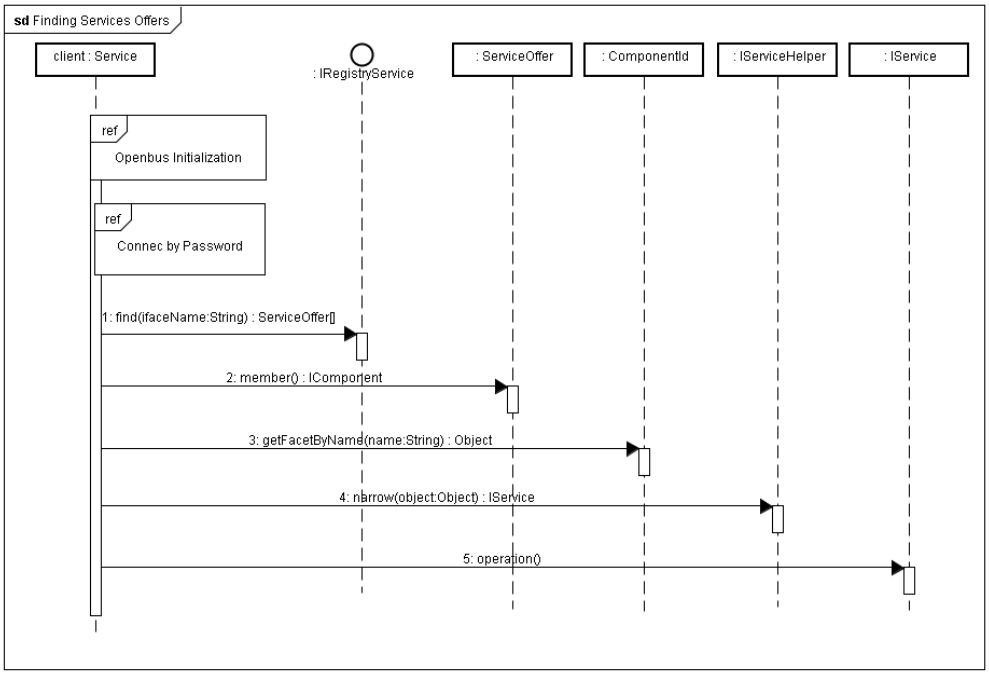

Figura 21: Buscando por serviços no barramento

Referências

[1] Chappell, D. 2004 Enterprise Service Bus. O'Reilly Media, Inc.

[2] Szyperski, C. Component Software: Beyond Object-Oriented Programming. ACM Press : Addison-Wesley Publishing Co. 1998.

[3] OMG. CORBA Components. OMG Document formal/04-03-01 (CORBA, v3.0.3). 2004. http://www.omg.org

[4] Bolton, F.; *Pure* CORBA. Sams Publishing, 2002.

[5] JacORB. http://www.jacorb.org. 2004 - 2009.

[6] The SCS Project. http://www.tecgraf.puc-rio.br/ scorrea/scs/

[7] OMG. CORBA Interceptors. OMG Document formal/04-03-01 (CORBA, v3.0.3). 2004. http://www.omg.org