Openbus SDK-Lua 1.5.2 - Documentação

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

Fevereiro 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 Openbus usa o Oil (ORB in Lua) [6] [7], um ORB de código fonte aberto e desenvolvido na linguagem Lua [5]. O objetivo deste documento é o de descrever a estrutura e dinâmica da API sdk-lua 1.5.2 de acesso ao barramento para serviços desenvolvidos na linguagem Lua. Este documento não descreve o funcionamento interno do Openbus em si, também não descreve em detalhes o modelo de componentes SCS [8] no qual o Openbus foi desenvolvido. Para mais informações sobre como usar a API Lua do SCS ou sobre informações básicas de uso e configuração da API sdk-lua 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.
- Oil.
- Modelo de Componentes SCS v1.2.
- Conceitos básicos do Openbus.
- Linguagem de programação Lua.

Como Lua é uma linguagem interpretada, flexível e não orientada a objetos, existem alguns elementos citados nos diagramas que não possuem uma representação de objeto direto como ilustrado. Porém como não existiria outra forma de representar nos diagramas, eles foram representados conceitualmente afim de deixar a descrição da API didática e evitar inserção de código no manual.

2 Entidades e Relacionamentos

A API é composta pela fachada *openbus.Openbus* e pelos pacotes *authenticators*, *lease*, *interceptors*, *faulttolerance* e *util*. Além disso, a API usa os proxies dos serviços básicos: as facetas do Componente do Serviço de Controle de Acesso, do Serviço de Registro.

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.

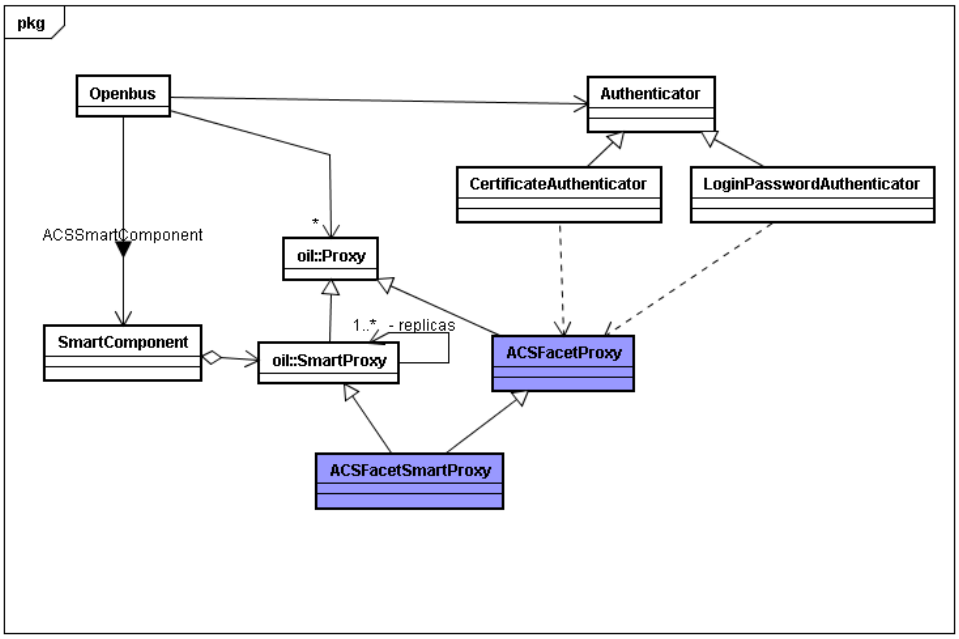

Figura 1: Modelo Conceitual das entidades de conexão com o barramento. As classes em azul são uma representação da implementação, elas não existem de fato.

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. Ambos dependem do proxy da faceta do Serviço de Controle de Acesso que implementa a interface *IAccessControlService*. Tais proxies estão representados pelas classes em azul: *ACSFacetProxy* e *ACSFacetSmartProxy*. A fachada *Openbus* utilizará ou proxies normais ou *smart* proxies. *Smart* proxies são aqueles que são configurados com as referências para todas as réplicas do servant ao qual se refere de forma que, em caso de falha, o proxy possa redi-

recionar a chamada para outra réplica. Eles só serão utilizados na API, caso o mecanismo de Tolerância a Falhas seja habilitado. A fachada *Openbus* também possui uma referência para a classe *SmartComponent* **??** que é composta pelos *smart* proxies de todas as facetas de um componente, no caso, o componente do Serviço de Controle de Acesso.

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

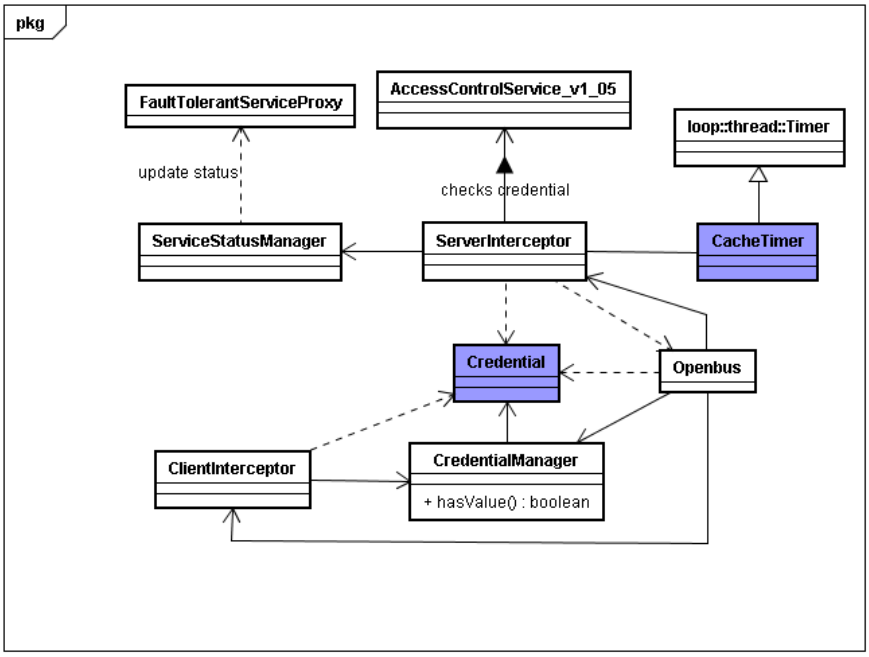

Figura 2: Modelo Conceitual das entidades de interceptação e validação de credenciais. As classes em azul são uma representação da implementação, elas não existem de fato.

Sempre que o barramento é inicializado, a fachada *Openbus* inicializa o ORB, que no caso é o *Oil* com os interceptadores *ClientInterceptor* e *ServerInterceptor*. O interceptador *ClientInterceptor* é responsável por pegar a credencial salva no *CredentialManager* após o login no Serviço de Controle de Acesso, e inserir no contexto da requisição. Do lado do servidor, o interceptador *ServerInterceptor* recupera a credencial do contexto da requisição e verifica na faceta que implementa a interface *IAccessControlService* se a credencial é válida. Na figura 2, é ilustrada a faceta *AccessControlService_v1_05*.

O interceptador *ServerInterceptor* quando inicializado na aplicação servidor também inicializa uma *thread*, representada conceitualmente pela classe *CacheTimer* que implementa o padrão *Template Method* e tem um relógio que de tempos em tempos executa a ação de verificação das credenciais salva na memória interna do interceptador para fins de otimização do processo de validação da credencial. Quando a aplicação cliente inicializa a fachada *Openbus*, ela pode definir a política de validação de credencial que decidirá se este *timer* será ativado.

Finalmente a classe *ServiceStatusManager* é responsável por, durante o envio de resposta da requisição, disparar o mecanismo de atualização de estado das réplicas. Tal mecanismo é ativado somente se existirem réplicas para serviço no qual o interceptador foi instalado.

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.

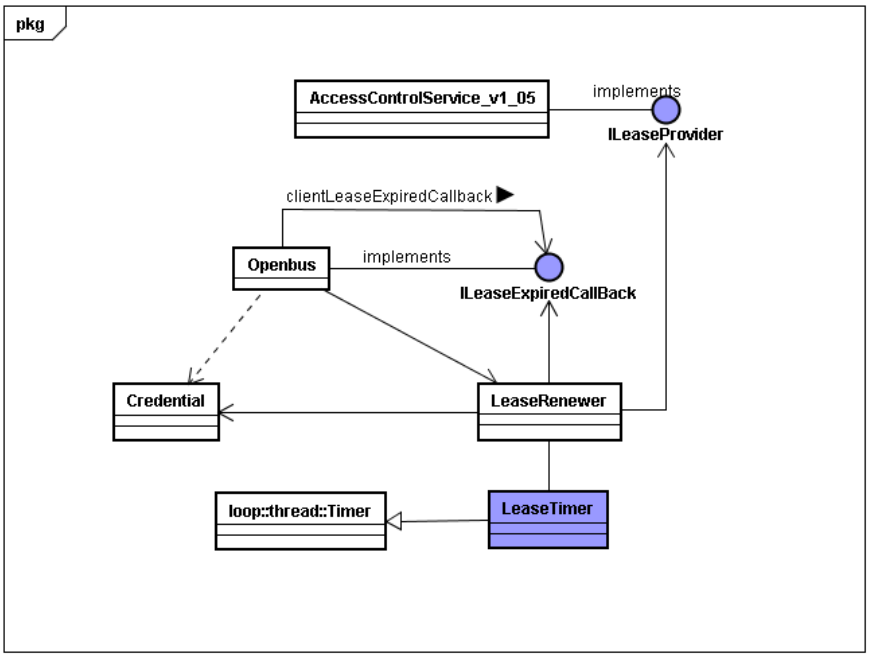

Figura 3: Modelo Conceitual das entidades de renovação da lease da credencial. As classes em azul são uma representação da implementação, elas não existem de fato.

A API sdk-lua-1.5.2 fornece operações na fachada *Openbus* para o cliente especificar a classe que implementa a interface representada conceitualmente por *ILeaseExpiredCallBack*. A classe *LeaseRenewer* chama operações desta interface no momento em que a credencial expira. A fachada *Openbus* por sua vez implementa estas operações funcionando como um delegador.

Além disso o Serviço de Controle de Acesso sendo provedor de *leases* implementa a interface conceitual *ILeaseProvider*, porém a API é flexível para receber outros provedores se desejado.

[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.

Para que a lease seja renovada de tempos em tempos, a API possui uma *thread* ilustrada pela classe *LeaseTimer* que implementa a ação de renovação a partir do provedor.

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. Dois parâmetros são obrigatórios para a inicialização do barramento: o nome da máquina e número da porta onde o Serviço de Controle de Acesso se encontra. Em seguida a classe *Openbus* inicializa o ORB enviando um arquivo de propriedades que define, entre outras informações, se as requisições são interceptáveis [10]; carrega as IDLs das interfaces do barramento; e carrega[2] as classes dos interceptadores cliente e servidor.

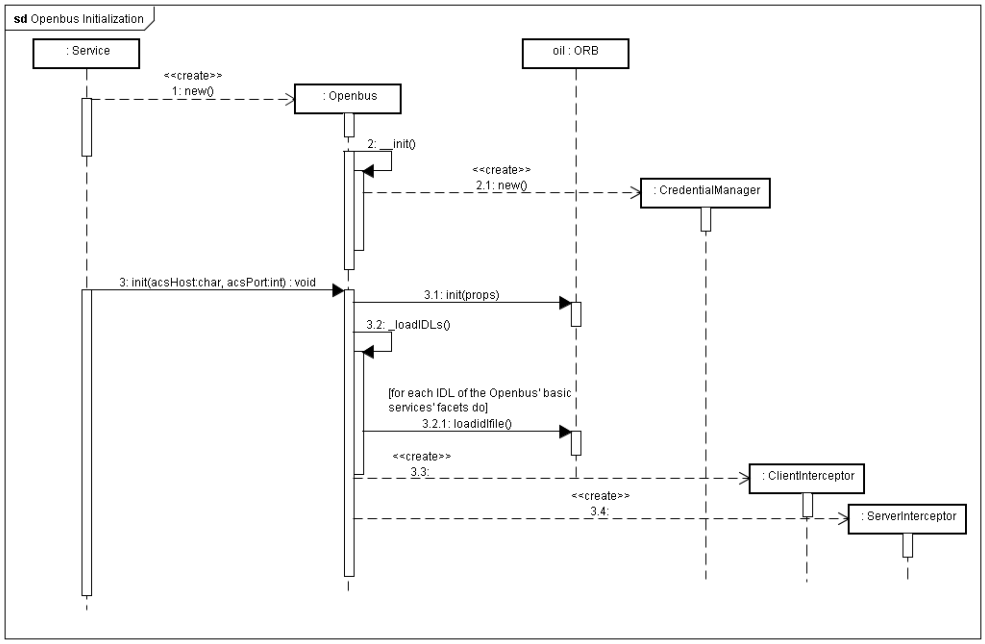

Figura 4: Inicialização do Openbus

[2]Esta última etapa é necessária por uma limitação do uso da fachada *Openbus* como *Singleton*, pois um como é utilizada nos interceptadores, se o *require* for feito antes da classe Openbus tiver sido inicializada (o que aconteceria se o *require* fosse feito no topo do arquivo), ocorreria um erro em tempo de execução alegando que *Openbus* é **nil** dentro das classes *ClientInterceptor* e *ServerInterceptor*.

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 *connectByLoginPassword()* e *connectByCertificate()*, respectivamente.

A figura 5 ilustra o processo de conexão através de *login* e senha[3]. 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 operação *authenticate()* que, por sua vez, irá diferenciar o tipo de acesso ao Serviço de Controle de Acesso propriamente dito.

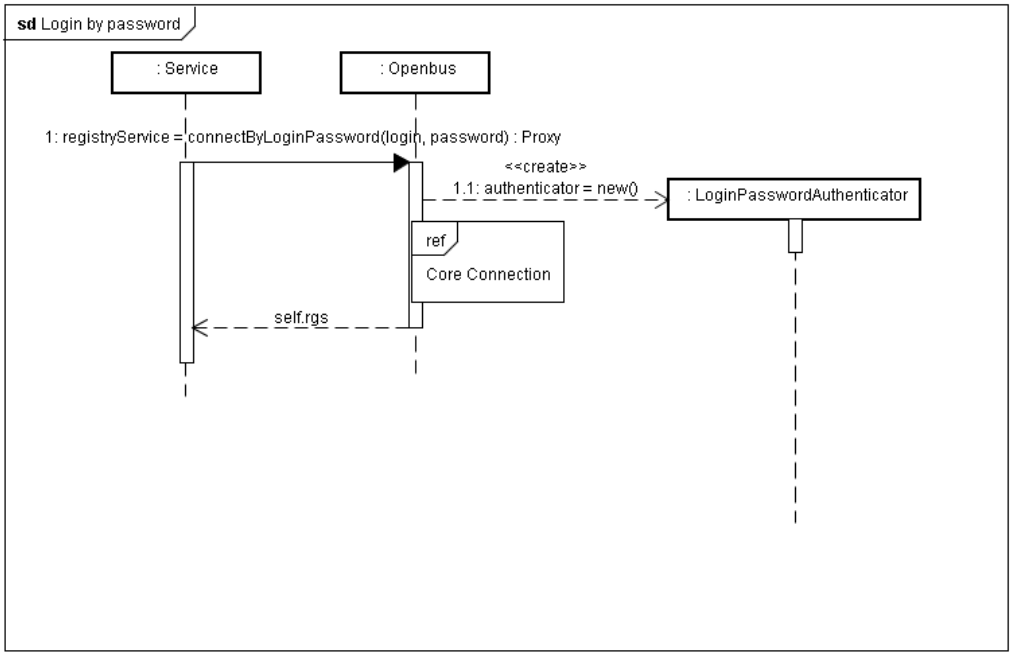

Figura 5: Conexão por senha

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

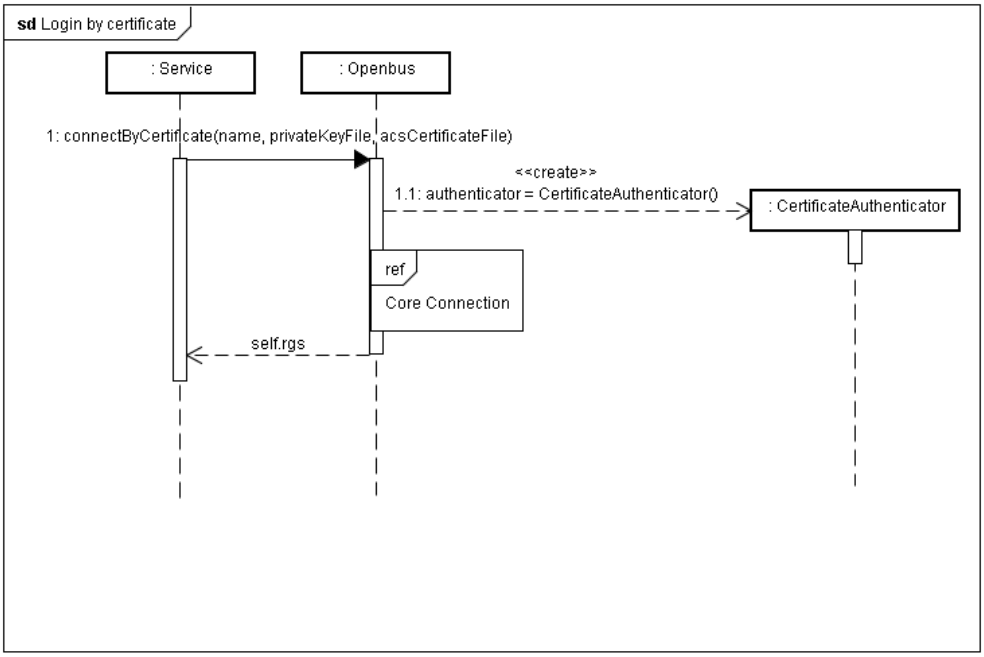

Figura 6: Conexão por certificado

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 o *CredentialManager* possui alguma credencial salva em seu estado a partir da operação *hasValue()* que retorna um *boolean*. 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).

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, neste caso, simplesmente chama a operação *loginbypassword()* no Serviço de Controle de Acesso.

Se o *loginbypassword()* for executado com sucesso, a classe *Openbus* receberá a credencial e a *lease* da credencial. A conexão é finalizada então pela operação privada *_completeConnection()* que:

1. salva a credencial no *CredentialManager*,
2. instancia a classe *LeaseRenewer*,
3. inicia o *timer* de renovação através da mensagem *startRenew()*, e
4. recupera e retorna a referência para o Serviço de Registro.

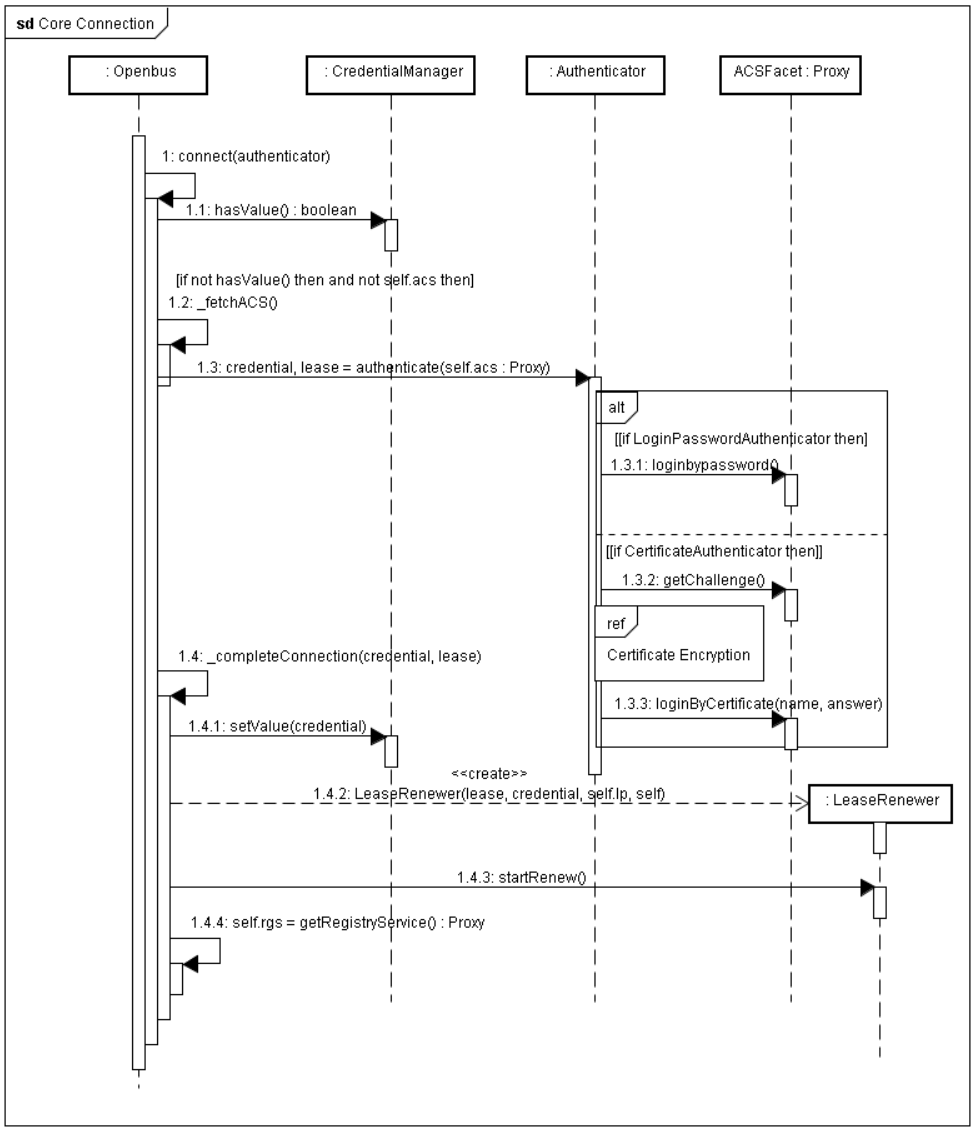

Figura 7: Processo de conexão básico

O processo de autenticação por certificado através da classe *CertificateAuthenticator* por outro lado precisa ler o certificado do cliente antes de efetuar o login no barramento. Este procedimento usa a API externa *LCE - Lua Cryptography Extension* [11] que é uma biblioteca de *binding* da biblioteca *OpenSSL crypto* [10] para a linguagem Lua, e é ilustrado na figura 8.

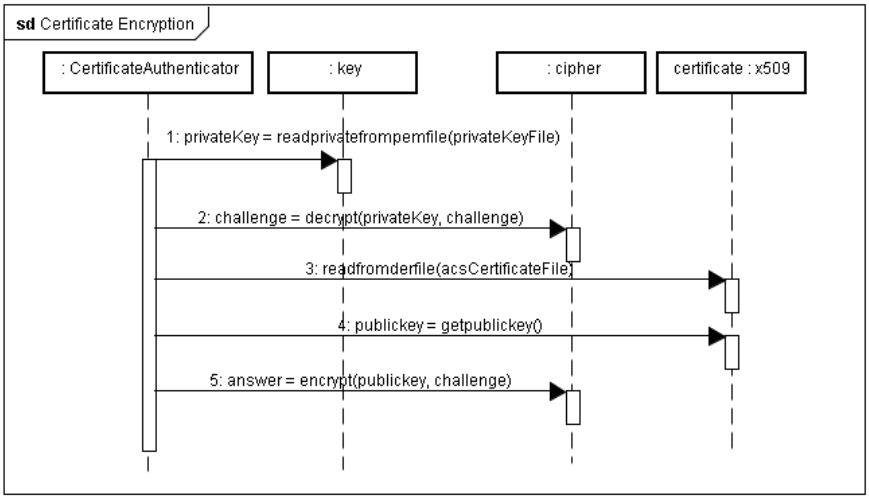

Figura 8: Leitura do Certificado

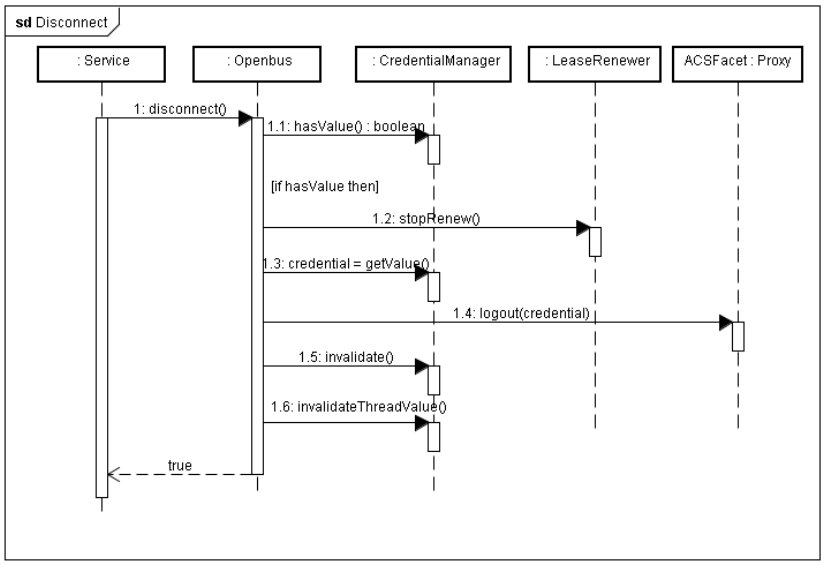

Figura 9: Desconexão

O processo de desconexão é simples (figura 9). Se o *CredentialManager* tiver uma credencial, 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), ela será invalidada e o cliente receberá um retorno de sucesso. Se algo de errado acontecer durante esse processo, o cliente receberá

uma mensagem de erro.

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 iniciado. A figura 10 descreve esse processo a partir do seu início pela chamada *startRenew()*. A classe *LeaseRenewer* verifica se o *timer* já não foi criado antes. Caso não tenha sido, cria sua instância passando o tempo fornecido pelo *lease*. O *timer* representado conceitualmente pela classe *LeaseTimer* executa a operação *_renewLeaseAction()*. Durante sua execução, a operação *renewLease()* do provedor é executada. Nesse caso, o provedor da lease é o Serviço de Controle de Acesso e esta operação faz parte de uma das suas facetas: *ILeaseProvider*.

Durante o processo de renovação da credencial, se a renovação for executada com sucesso, uma nova *lease* é fornecida. O tempo do *LeaseTimer* é atualizado com este valor.

Caso o processo de renovação não seja autorizado, *LeaseRenewer* irá enviar a mensagem *expired()* para a instância da classe que implementa a interface *ILeaseExpiredCallback*. 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* é 'resetada', ou seja, todas as referências para as facetas do Serviço de Controle de Acesso e Registro são invalidadas, assim como o estado de *CredentialManager*.

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).

6 Mecanismo de Interceptação e Validação da Credencial

Como já visto, o Oil é inicializado com interceptação de requisições, e dois interceptadores são instalados: *ClientInterceptor* e *ServerInterceptor*. 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.

A figura 11 ilustra esse processo. Uma aplicação cliente representada pela classe *Service* envia a requisição para o proxy em questão, que pode ser do Serviço de Registro ou de outro serviço cadastrado no barramento, e a requisição é interceptada por *ClientInterceptor* através da mensagem *sendrequest()*. A credencial é então verificada no *CredentialManager* e, caso exista ela é inserida no contexto da requisição (veja mensagem 1.1.1.1.3). A requisição é

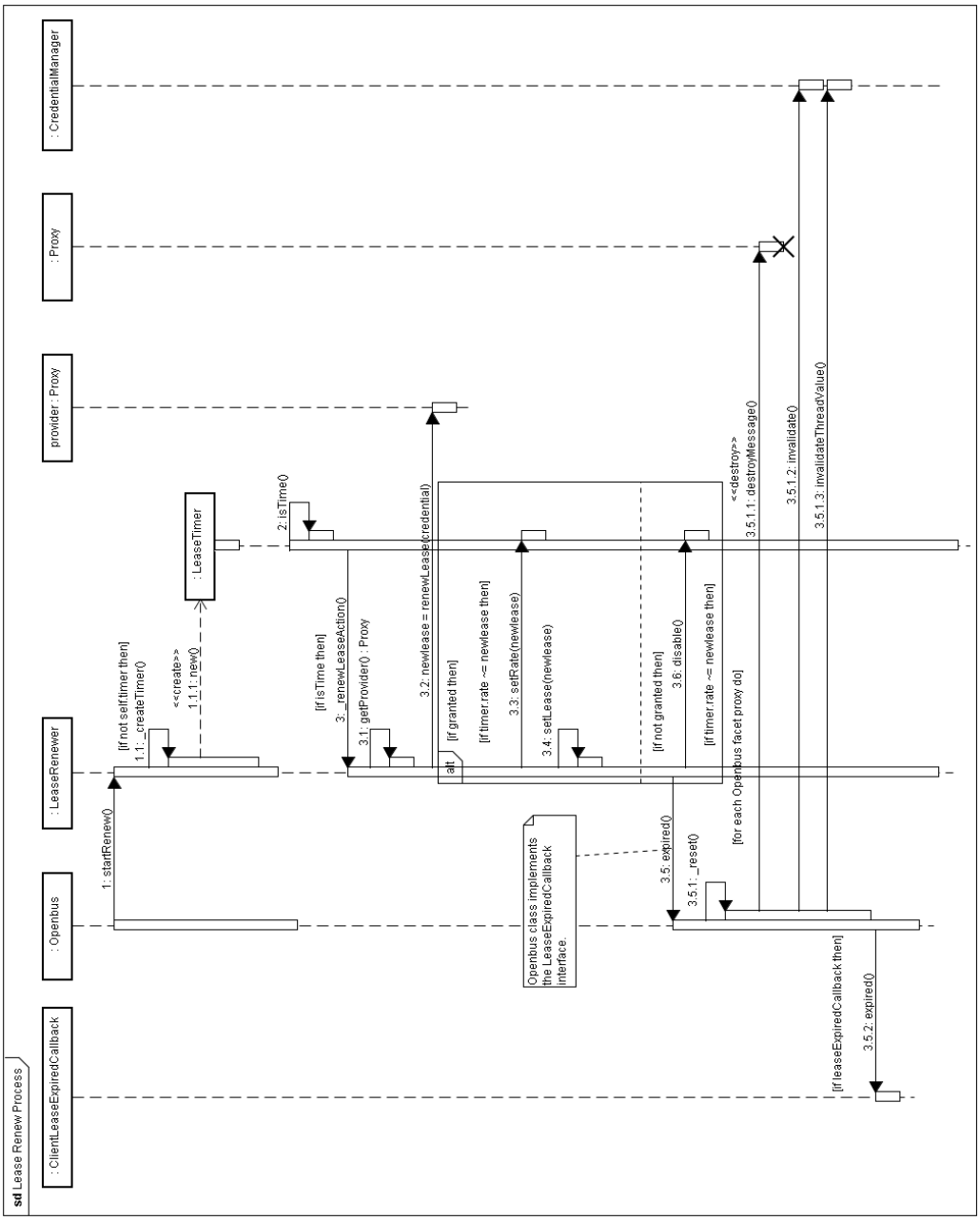

Figura 10: Processo de Renovação de Lease de Credencial

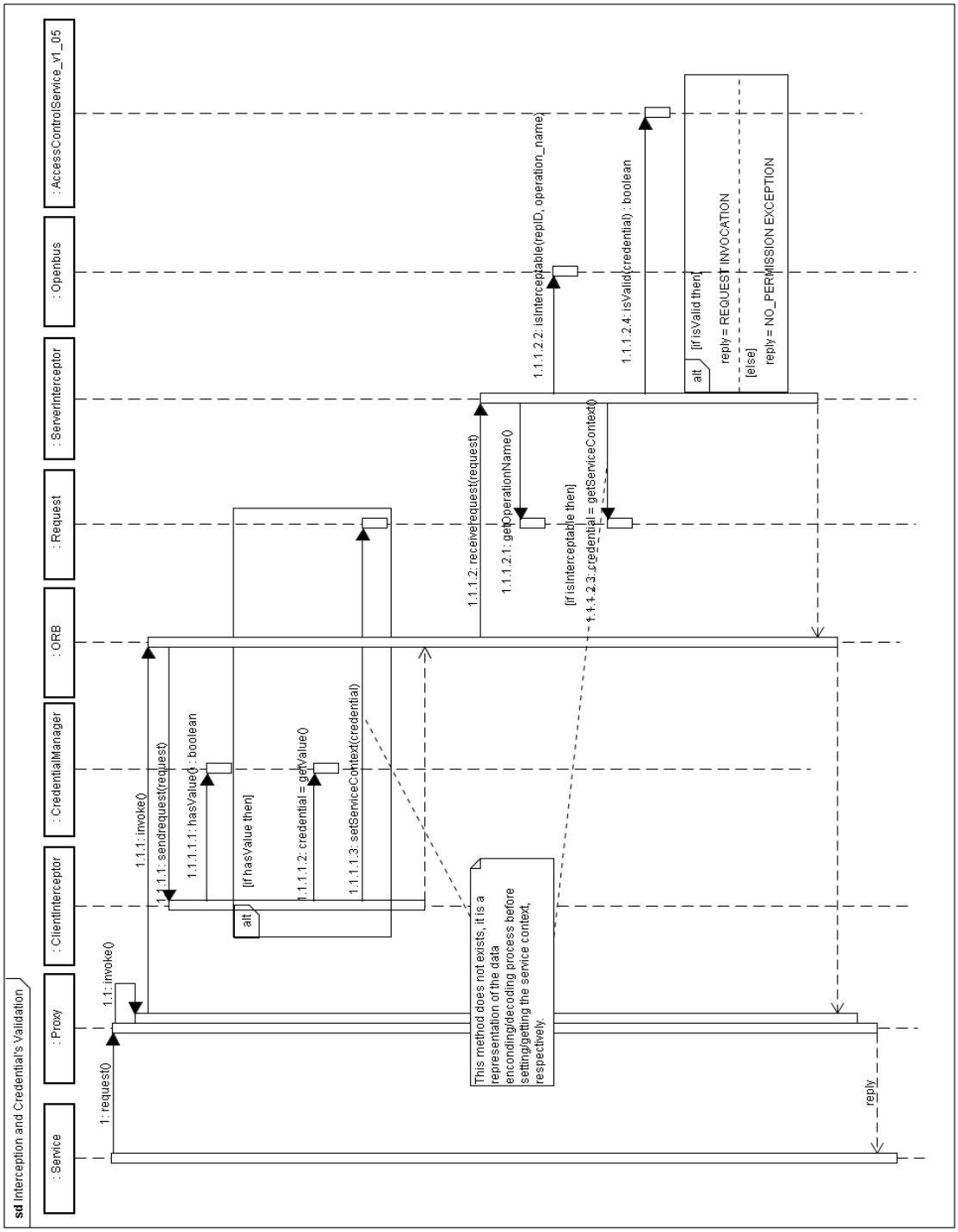

Figura 11: Interceptadores e Validação da Credencial

redirecionada pelo Oil para o interceptador servidor (*ServerInterceptor*) que, pelo nome da requisição, vai verificar se é interceptável ou não (mensagem 1.1.1.2.2). Se interceptável, o interceptador vai tentar recuperar a credencial do contexto e validá-la na faceta *AccessControlService_v1_05*. Se a credencial foi recuperada do contexto e é válida, a requisição é processada normalmente no servidor alvo, senão o interceptador servidor enviará para o cliente a exceção *NO_PERMISSION*.

7 Qualidade do Serviço

A API sdk-lua 1.5.2 fornece dois mecanismos que melhoram a qualidade do serviço do 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 três políticas que podem ser usadas:

NONE Indica que as credenciais interceptadas não serão validadas;

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.

Atualmente a política *NONE* é utilizada somente em casos específicos do barramento, uma vez que o cliente não conseguiria registrar ou consumir nenhum serviço se esta política estiver habilitada.

A figura 12 ilustra a troca de mensagens durante o processo de verificação da política pelo interceptador servidor seguida da validação em si. No momento em que o intercetador servidor é inicializado, a política passada é verificada. Se for *CACHED*, um *timer* será criado (ilustrado conceitualmente pela classe *CacheTimer*). Posteriormente, quando uma requisição do cliente for interceptada, a requisição é bloqueada, a credencial é buscada na cache, se encontrada, o índice é atualizado, a requisição desbloqueada e redirecionada para o servidor alvo.

Se não for encontrada, a requisição também é desbloqueada porém a credencial é verificada no Serviço de Controle de Acesso. Como pode ser que a credencial inserida no contexto seja de uma versão anterior do barramento, é preciso passar as duas credenciais inseridas no contexto (atual e anterior, veja a figura 13). Se a credencial foi validada, tenta-se adicioná-la na *cache*. Ela só não será adicionada se a cache já tiver atingido o seu limite máximo permitido.

Os outros casos, *NONE* e *ALWAYS* são ilustrados nos blocos inferiores.

Já a figura 14 ilustra a troca de mensagens durante o processo de validação das credenciais que estão na *cache*. De tempos em tempos, o *timer* representado pela classe *CacheTimer* executa a ação *credentialValidatorAction()*. Este por sua vez verifica se não está bloqueado (está somente quando uma credencial está sendo verificada na cache no momento de uma requisição), e se não estiver, bloqueia e verifica todas as credenciais de uma só vez no Serviço de Controle de Acesso através da chamada *areValid()* na faceta *AccessControlService_v1_05*. Se alguma delas não for válida, é removida da *cache*.

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 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 no arquivo *"/conf/ACSFaultToleranceConfiguration.lua"*.

Além disso, o barramento define alguns tempos padrão usados durante a detecção e tratamento de falhas em uma determinada requisição. Esses tempos são definidos no arquivo de configuração */conf/FTTimeOutConfiguration.lua*:

reply Tempo de resposta de uma requisição normal;

non_existent Tempo de resposta de uma requisição que testa se o objeto remoto existe;

fetch Tempo de busca por alguma réplica disponível.

Como ilustrado na figura 15, para habilitar o mecanismo de tolerância a falhas é preciso executar o método *enableFaultTolerance()* na fachada *Openbus*. Esta chamada irá criar uma instância do *SmartComponent* do Serviço de Controle de Acesso, que nada mais é que um gerenciador dos *smart* proxies de suas facetas.

Note que o método *enableFaultTolerance()* deve ser chamada logo após a chamada do *init()* ou o mecanismo não estará ativado para quaisquer operações da API Openbus ou requisições ao barramento seguintes a sua inicialização.

O *SmartComponent* por sua vez, no momento em que for definir os *smart* proxies, vai carregar do arquivo de configuração */conf/FTTimeOutConfiguration.lua* o tempo de busca (*fetch*), e para cada réplica definida a partir do arquivo de configuração *"/conf/ACSFaultToleranceConfiguration.lua"*, um proxy será buscado no Oil (veja o diagrama da figura 16). Enquanto uma faceta buscada não estiver disponível e o tempo não tiver se esgotado, o processo de busca continua iterando nas réplicas. Ou seja, o processo só para se o tempo tiver sido antigido ou todas as facetas do componente, no caso o Serviço de Controle de Acesso, estiverem disponíveis e respondendo corretamente. A mensagem 1.5 muda os proxies da facetas encontradas de *'synchronous'* para *'smart'*, tornando-os *smart* proxies.

A figura 17 ilustra como a API trata a exceção através do Smart Proxy para garantir que uma requisição não falhe. Desta forma, suponha que o cliente tenha

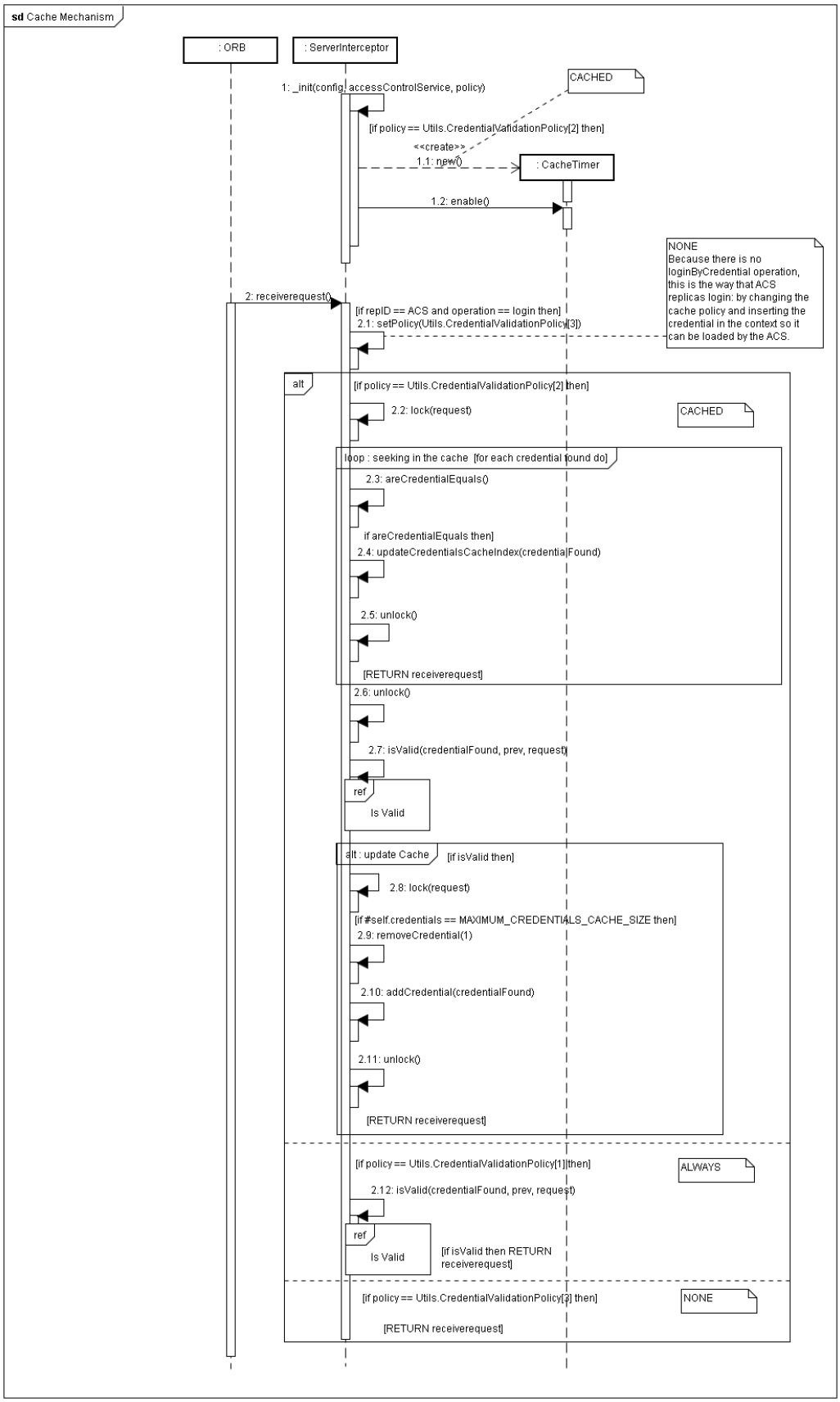

Figura 12: Mecanismo de Cache

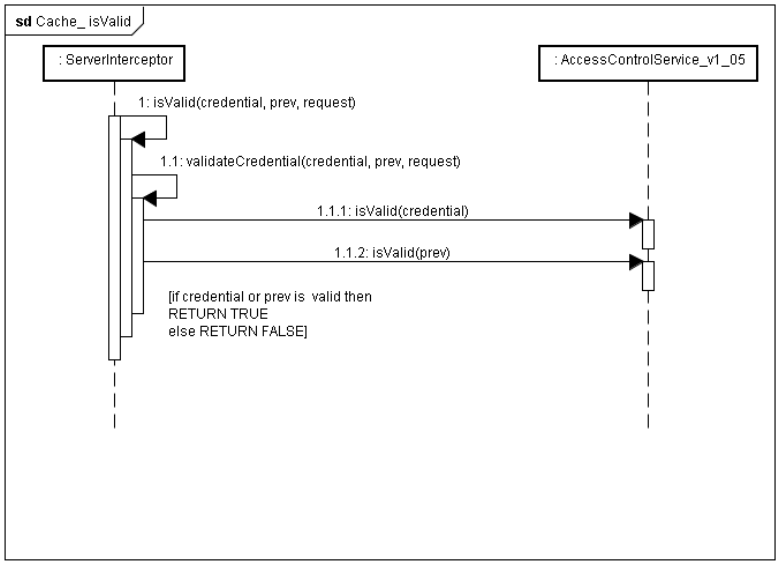

Figura 13: Validação no Serviço de Controle de Acesso

requisitado a operação *logout*, se uma das exceções abaixo for recebida, o Smart Proxy tentará achar outra réplica válida para redirecionar a requisição:

```
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
```

A busca acontecerá enquanto uma réplica sem falhas não for encontrada ou o tempo máximo definido tiver sido atingido. E a operação *smartmethod()* é executada recursivamente[4] no caso de uma réplica sem falhas tiver sido encontrada e enquanto ocorrer um exceção, exceto por TIME_OUT, que é lançada quando nenhuma réplica válida é encontrada ou consegue responder a requisição no tempo especificado pelo cliente.

Note que foi omitida a sequência de mensagens a partir da mensagem *1.1.1 reply, exception = invoke()* pois a descrição sobre como a chamada é realizada, depois transferida para os interceptadores, e a credencial validada antes da requisição ser redirecionada para o objeto remoto alvo, foi ilustrada na figura 11 a partir da mensagem 1.1.1.

[4]Na figura 17 a recursão é representada pelo *loop* com a condição de guarda *while not reply* e *reply* é um atributo público para esta operação, ou seja, uma vez retornada corretamente, o laço termina.

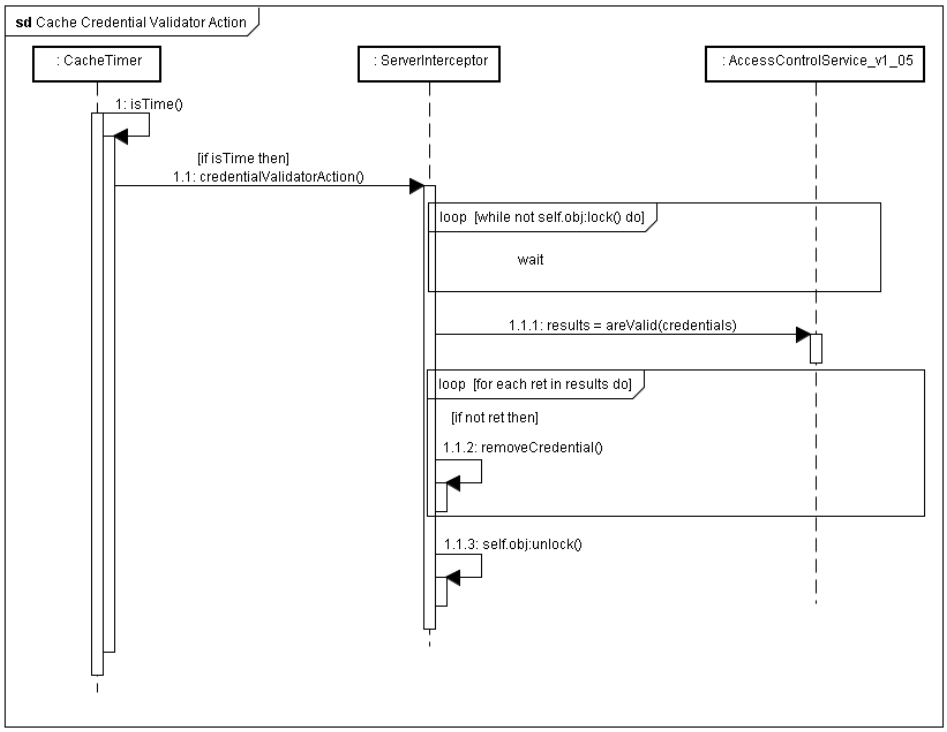

Figura 14: Validação das Credencias em Cache

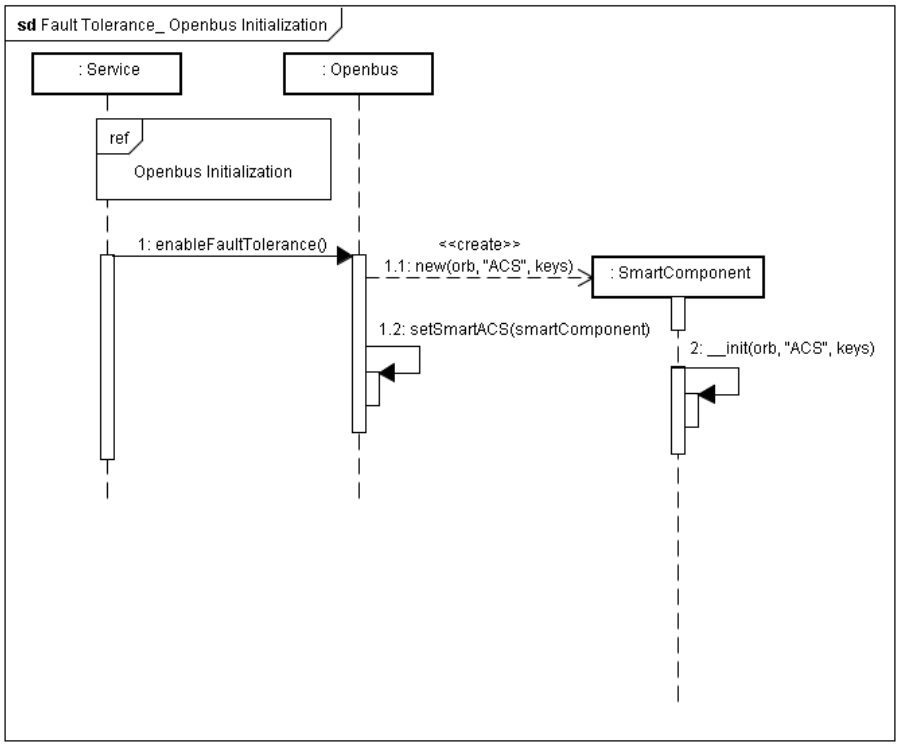

Figura 15: Inicialização do Openbus com Mecanismo de Tolerância a Falhas

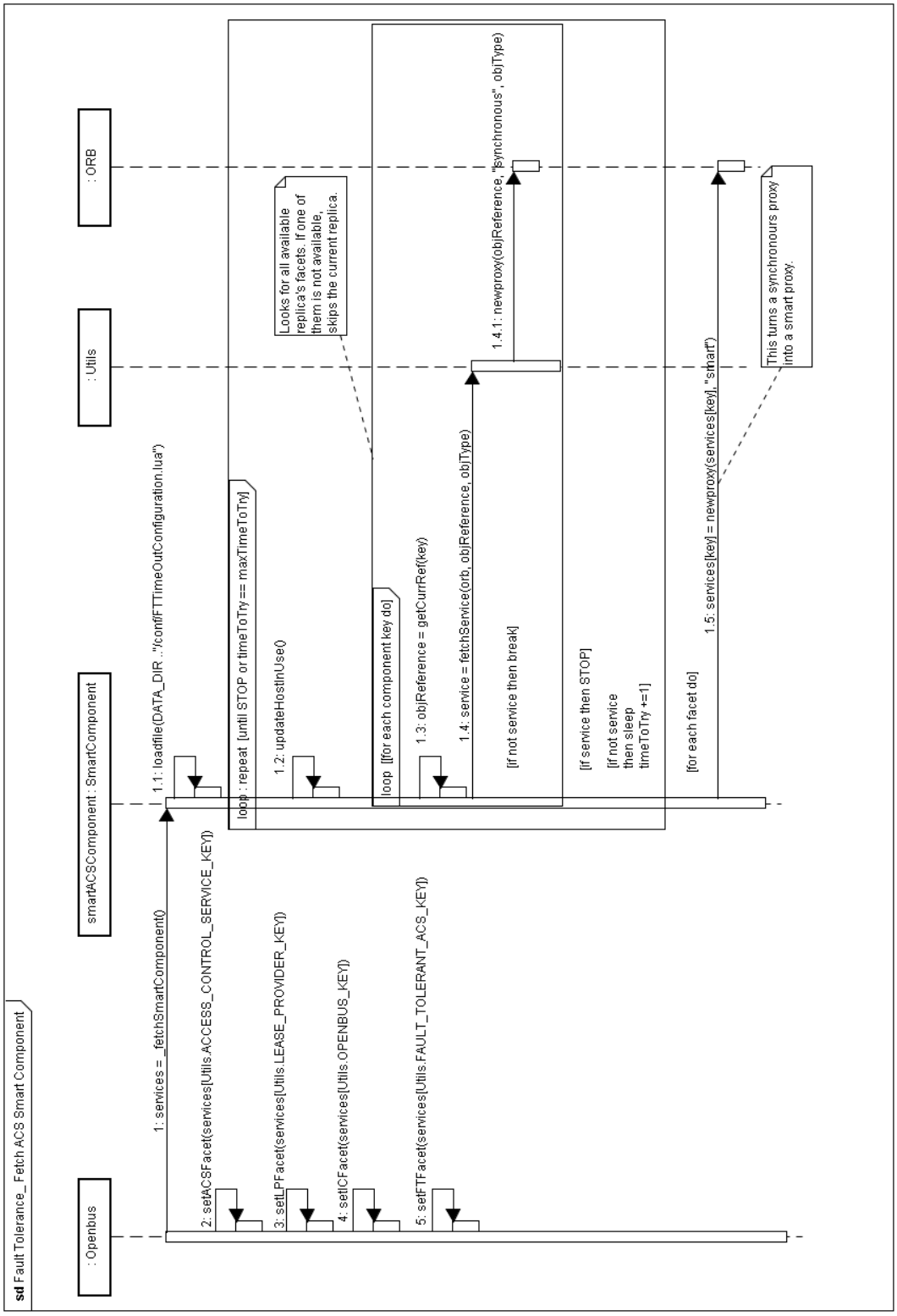

Figura 16: Configuração do Smart Component do Serviço de Controle de Acesso

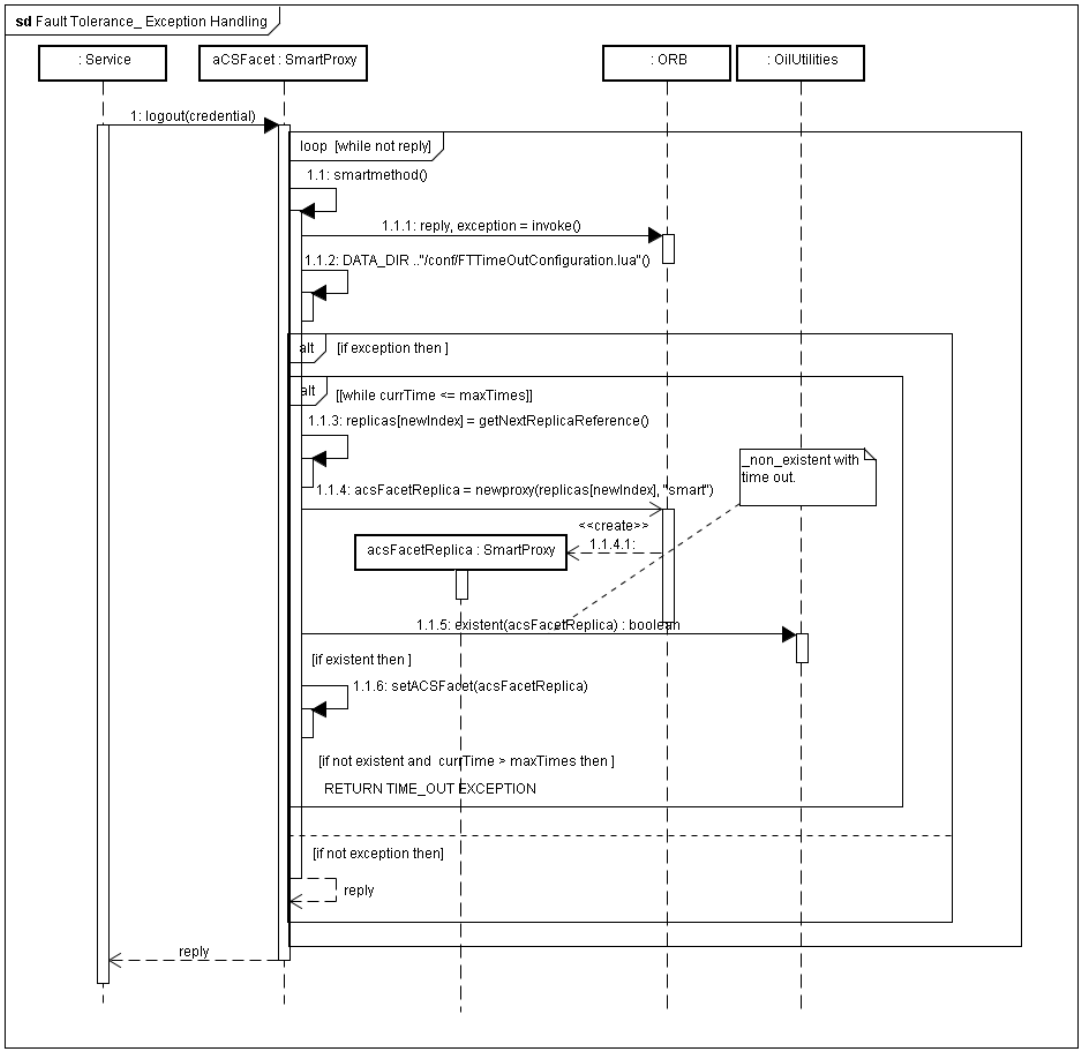

Figura 17: Mecanismo de Tratamento de Exceção do FT

8 Ofertando Serviços

O processo básico de cadastro de oferta de serviços é ilustrado no diagrama da figura 18. O serviço a ser registrado no barramento deve ser descrito através de uma oferta de serviço, que é representada pela estrutura *ServiceOffer*.Para que um servidor possa ofertar serviços no barramento através da API sdk-lua-1.5.2 é preciso:

1. Inicializar o barramento;
2. Conectar no barramento;
3. 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()*;
4. Cadastrar a oferta através da operação *register* chamada no proxy do Serviço de Registro.

Para cadastrar a oferta é preciso que o servidor componente passe a sua faceta *IComponent* e as propriedades que descrevem as ofertas.

Após a chamada da operação *register*, a sequência de mensagens segue como descrito no diagrama de interceptação e validação de credenciais, veja figura 11.

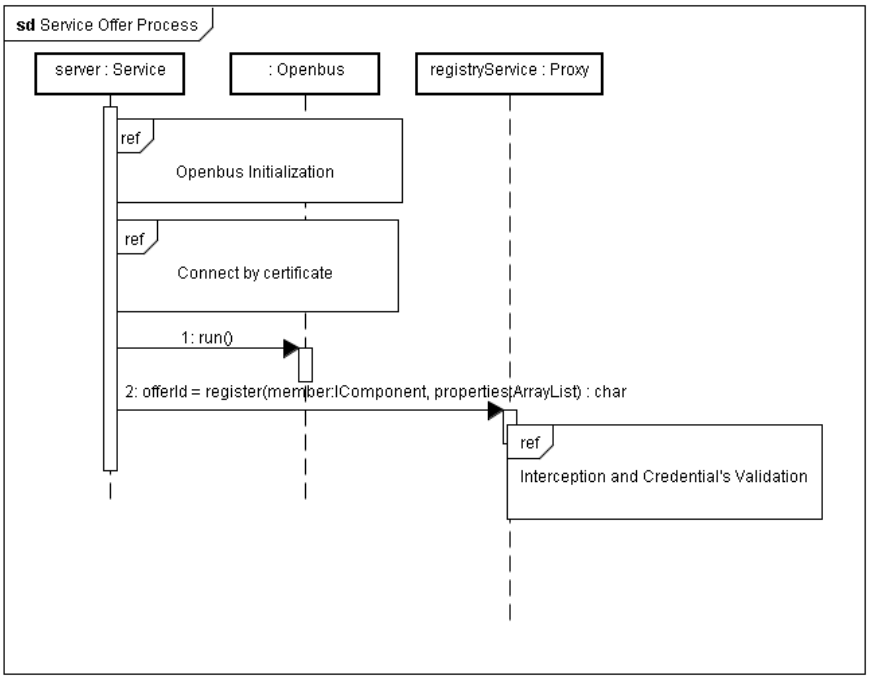

Figura 18: Processo básico de oferta de serviços

9 Consumindo Serviços

O processo básico de busca de ofertas de serviços é ilustrado no diagrama da figura 19. Para que uma aplicação cliente possa consumir serviços registrados no barramento através da API sdk-lua-1.5.2 é preciso:

1. Inicializar o barramento;
2. Conectar no barramento;
3. Buscar o serviço através da mensagem *find* 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 de ofertas que implementam a faceta buscada, na qual deverá iterar e para cada uma delas mapear a interface *IComponent* usando a operação *narrow* da API do SCS;
5. Através da faceta *IComponent*, buscar a faceta do componente desejada, como por exemplo '*IDL:demoidl/hello/IHello:1:0*';
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()*.

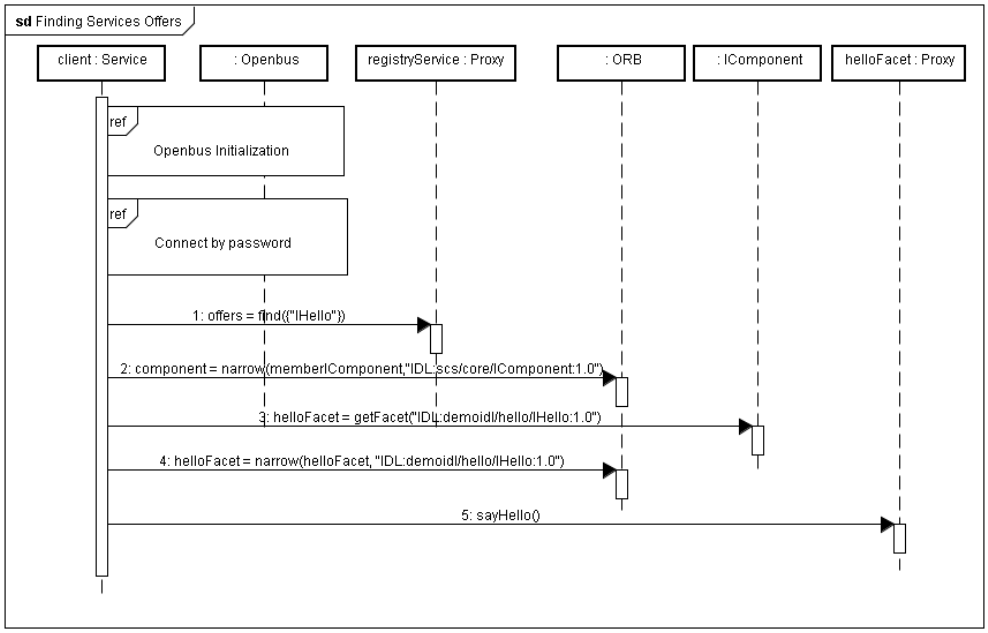

Figura 19: Buscando por serviços no barramento

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] Maia, R., Cerqueira, R. and Kon, F.; A Middleware for Experimentation on Dynamic Adaptation. In Proc. 4th Workshop on Adaptive and Reflective Middleware (ARM2005), co-located with 6th International Middleware Conference, Grenoble, France, November 2005.

[6] The Oil Project: An Object Request Broker in Lua. http://oil.luaforge.net/index.html

[7] Linguagem de Programação Lua. http://www.lua.org/

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

[9] Gatti, M., Cerqueira, R.; Design Experiences and Strategies on Applying Fault-Tolerance in a Component-based Enterprise Service Bus. Technical Report, Tecgraf, PUC-Rio, 2010.

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

[11] LCE - Lua Cryptography Extension. http://luaforge.net/projects/lce/

[10] OpenSSL crypto. http://www.openssl.org/docs/crypto/crypto.html