# Secure Authentication in Integrations of Systems with OpenBus

**Renato Maia**[1], **Hugo Roenick**[1]

[1]Instituto Tecgraf – Pontifícia Universidade Católica do Rio de Janeiro (PUC-Rio)
R. Mq. São Vicente 225 – 22453-900 – Rio de Janeiro – RJ – Brasil

{maia,hroenick}@tecgraf.puc-rio.br

***Abstract.*** *OpenBus is a CORBA-based middleware for integration of computational systems and was originally developed to support efficient communication, identification of the origin of requests and service discovery mechanisms in corporative distributed computing environments. With the wider use of OpenBus came the need to adapt the technology to consider environments with malicious participants that can inspect the communications to gather information and tamper with the origin identification mechanisms provided by OpenBus. Such possibility would allow systems to impersonate the identity of other systems or users with privileged access. To avoid such vulnerabilities we developed OpenBus 2.0, which is a redesign of the OpenBus technology where we adopt a new credential model over CORBA for identification of origin and delegation. In this paper, we present this new credential mechanism of OpenBus 2.0 where credentials cannot be forged or reused by malicious parties even though they have complete access to all information sent through the network (sniffers). We also discuss some of the challenges of the solutions adopted in OpenBus 2.0 and some ideas to deal with such challenges that we intend to investigate and experiment in the future.*

## 1. Introduction

The current ubiquitous nature of information technology makes most companies rely on a variety of computational systems to carry out their activities. Such systems are usually specialized in specific tasks, like handling some particular kind of data or enforcing certain business rules. As the systems evolve and embrace more aspects of theirs activities they become more complex and start to overlap with other systems. If such systems are designed and evolve independently from one another, they end up maintaining their own collection of data and implementations of business rules. Ultimately this tends to cause problems such as replication and inconsistency of data and rules, multitude of data representations, non-automatic and error-prone workflows, among others. System integration is about redesigning or adapting independent systems to communicate and cooperate with each other in order to automatically avoid or minimize such problems by sharing data resources and implementations.

In essence, the most basic requirement for system integration is some sort of communication support, so the systems can interact with each other, which usually vary from file transfer (manual or automatized), shared database, message exchange or use of RPC mechanisms [Hohpe and Woolf 2004]. Once systems are able to communicate with other

parties some additional issues arise besides basic communication, specially in the context of an institution where some degree of control and security are required.

OpenBus [Tecgraf/PUC-Rio 2006] is a CORBA-based [OMG 2002a, Bolton and Walshe 2001] middleware designed to aid the integration of systems under the control of a corporation or institution. OpenBus was originated as a project in conjunction with Petrobras, the Brazilian OiL Company, aiming to provide support for integration of the company's systems, which process big volumes of geological data and logistic processing to support activities of oil exploration and production.

The CORBA standard defines a technology for distributed communication in a structured way using the distributed object-oriented paradigm. Additionally the whole CORBA standard is designed to be compatible to a variety of platforms and programming languages. This makes it easier to provide OpenBus support for systems written in different languages and platforms. Moreover CORBA is a open standard and also offers mechanisms to facilitate the introduction of extensions or additional features.

Some of these main issues addressed in OpenBus are:

**Discovery** Before systems are able to communicate and cooperate they must make themselves available, be able to find others systems that are available and choose properly which of these systems shall be used for a particular integration. CORBA already defines support for discovery through standard services like Naming Service or Trading Service. OpenBus provides an alternative solution for discovery designed with specific security and management concerns.

**Authentication** For systems that deal with critical data or processes, some guarantees are required about the systems they interact with, usually through some secure authentication mechanism. Additionally, authentication of systems integrations frequently require support for delegation, where on system can act on behalf or by command of others.

**Governance** Systems are usually developed independently from one another, but their design for integration with other systems is a cooperative effort that usually requires coordination and management that defines standard interfaces, practices and policies. In such cases, it is important that the infrastructure provides mechanisms to control and manage the integrated systems.

Initially the focus of the project was on efficient communication to allow direct interaction between systems. With the wider use of OpenBus came the need to enforce the security of the authentication mechanism, focusing particularly to address security issues like personification and free delegation in the presence of malicious participants in the network environment. In this paper we discuss the support for authentication in OpenBus focusing on the recent redesign to address the mentioned security issues.

This paper is organized as follows: in section 2 we present other solutions described in the literature that are related to OpenBus; an overview of the OpenBus architecture is presented in section 3; section 4 presents a detailed description of the secure authentication support introduced in OpenBus 2.0, and some security considerations about

the adopted approach are presented in section 5. Later, in section 6 we discuss some of the current issues in OpenBus we intend to address in the future. Finally, in section 7 we present some final remarks about the current state of OpenBus.

## 2. Related Work

There are different technologies related to system integrations, addressing issues like security, interoperability, maintainability, etc. In this section we describe some of the technologies that are related to OpenBus's authentication support discussed in this paper or related to OpenBus in general.

### 2.1. Kerberos

Kerberos [Neuman and Ts'o 1994] is a authentication system originally developed by MIT. Unlike OpenBus, Kerberos does not provide or is tied to any particular distributed communication technology. Kerberos focus on providing mutual authentication of communicating parties over a network. In fact Kerberos can be adapted for a variety of distributed systems technologies to provide authenticated access over insecure networks. Although Kerberos does not impose any particular communication technology, it provides features to facilitate the use of encrypted communication since its authentication process result in a encryption key being shared between the authenticated parties.

Kerberos can be used in CORBA. However it usually requires specific support by the ORB implementation to make use of the encryption support provided by Kerberos. This mainly because Kerberos is designed for use at the network transport level by authentication of the end points of a network channel. The standard CORBA programming model (API) hides the underlying communication technology concepts, like network channels. At the application level, CORBA only exposes concepts of object references and operation invocations. Therefore it is only possible to encrypt individual parameters separately, but it would not guarantee the integrity of the whole request, which could be braked apart.

OpenBus on the other hand is designed on top of CORBA programming model. It only provides authentication of the operation caller, and does not provide server-side authentication nor encryption to enforce data privacy or integrity like Kerberos. This makes OpenBus authentication exposed to manipulations of the network traffic by possible malicious parties (*man-in-the-middle* attacks) if no encryption is provided by the CORBA implementation.

Since OpenBus can be implemented over CORBA standard API without modifying the ORB implementation it is easier to add such support to different CORBA implementations.

### 2.2. CORBA Security Service

CORBA Security Service (CORBAsec) [OMG 2002b] is a complementary CORBA specification by OMG that introduces services and common facilities for applications to deal with security issues like authentication, confidentiality, integrity, accountability, etc. However, CORBAsec does not define any security mechanisms. Instead, CORBAsec specifies only architectures and programming interfaces to integrate security in CORBA

systems. Actual implementation of the security features are provided by specific vendors of CORBA implementations.

While OpenBus security support only comprises caller authentication, CORBAsec address issues ranging from authentication, message protection, access control, auditing and non-repudiation. CORBAsec standard carries a lot of complexity due to its design goals of portability, generality and interoperability. CORBAsec design accommodates a variety of security technologies under a single programming model. The specification defines four standard security technologies: GSS-Kerberos, SPKM, CSI-ECMA, and SSL. However few non-commercial CORBA implementations support the full CORBAsec specification.

### 2.3. Enterprise Service Bus

The *Enterprise Service Bus* (ESB) [Chappell 2004] is a concept of a middleware solution that enables interoperability among heterogeneous environments using a service-oriented model, aiming to provide loose coupling and distributed integrations network.

Integration still is an important concept in ESBs, however its main focus switched to service creation and composition. Unfortunately this change cause the term to be redefined, overloaded and diluted to the point that it currently does not have a very precise meaning [Manes 2007]. There are several solutions that provide support for system integrations, and commonly call themselves as ESB, but any two products labeled "ESB" are likely to be very different.

In summary, ESB represents the consolidation of the *Enterprise Application Integration* (EAI) and application server product categories. They commonly provide integration and mediation capabilities (e.g., routing, protocol binding, message transformation) as well as application hosting and life-cycle management capabilities. As a general rule, one of the protocols that an ESB supports is Simple Object Access Protocol (SOAP), but it doesn't require all services to communicate via SOAP.

When it comes to scientific scenarios (for example, the Exploration and Production area of Petrobras - E&P), which generally exchange very large messages and involve a large volume of messaging, performance, scalability and portability are requirements of great relevance. The use of standard ESB does not suit well in this scientific field, as ESB define the use of mediators, centralizing orchestration and message translation. This strategy brings high probability of generating link bottlenecks when contacting the bus, and as all data are usually transmitted in XML format, the cost to translate raw scientific data to a textual representation is relevant [Davis and Parashar 2002, Govindaraju et al. 2000, Bustamante et al. 2000, Kohlhoff and Steele 2003].

The motivation for OpenBus emerged from the need to support authenticated invocations over an heterogeneous environment without the need to develop a complete middleware solution, which would make the costs of the project too high. CORBA already addresses aspects of distribution to several platforms and programming languages and also enables transferring structured data in its binary representation (CDR), which reduces the conversion impact (marshalling) in the construction of messages to be transferred in the network. Moreover, CORBA provides extension mechanisms in the form of CORBA Portable Interceptors (PI), which are well supported by most CORBA implementations available and can be used to implement authentication support without the need to

change the underlying implementation. Therefore, OpenBus can provide authentication support on top of a full-featured middleware solution with little effort.

## 3. Overview

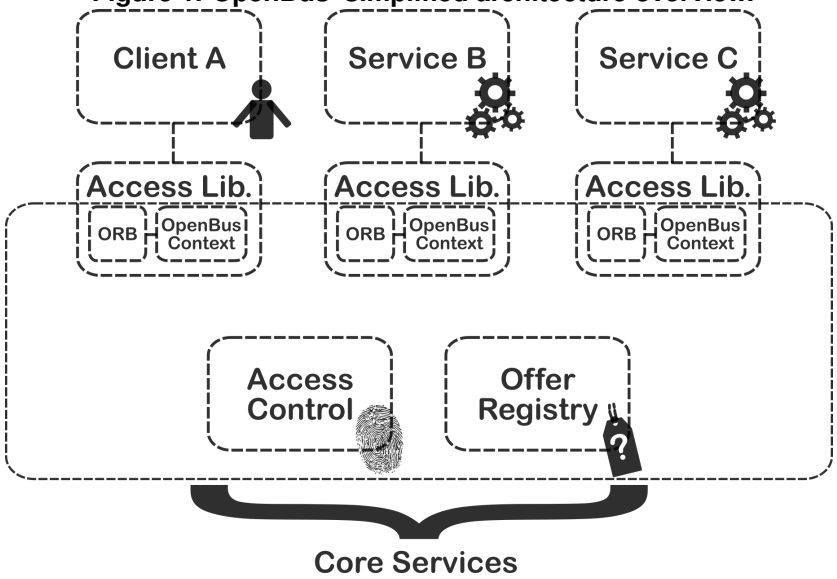
The central concept in OpenBus is the *bus* [Josuttis 2007], which is the medium where all interactions between the systems take place. In essence, the bus is a standard CORBA environment with additional services introduced by OpenBus. These additional services are called *OpenBus Core Services*. The two main core services are:

**Access Control Service – ACS** service used for authentication of systems and to support the validation of the systems identities sent within the calls made through the bus.

**Offer Registry Service – ORS** service responsible for storing references of services offered by authenticated systems that have permission to offer such services.

Moreover, OpenBus defines an additional set of rules about how CORBA invocations must be performed in the bus. These rules define how to use the core services to authenticate systems, how to attach authentication information to the invocations and how to validate such information. This set of rules is collectively called the *OpenBus Protocol*. OpenBus also provides an auxiliary library called *OpenBus Access Library* which implements the *OpenBus Protocol* and hides most of its complexities from the system developers. Figure 1 depicts a simplified overview of the OpenBus architecture.

**Figure 1. OpenBus' simplified architecture overview.**




To perform an integration every interested party must access the same OpenBus infrastructure, or in other words, the same bus instance. To gain access to a particular bus, a system must first be authenticated providing a proof of identity, like a password. Each identity in OpenBus refers a *entity*, which is a unique name that identities something accessing the bus. Entities are usually user names, or names specifying system instances accessing the bus. The entity names are defined by the bus manager. After a successful authentication to the bus, the *OpenBus Protocol* allows the system to make CORBA invocations containing authentication information verifiable by any system accessing the same bus. This authentication information is called *credential*.

The permissions in a OpenBus integration is usually done based on the identity provided in the credentials. Therefore, when a system receives a call it evaluates the caller identity contained in the credential to decide whether accept the call or not. This is the same model used by the ORS to authorize service offers. In the case of the ORS the bus manager first defines the entities that are allowed to offer service interfaces in the bus. This way, whenever a system makes a call to the ORS requesting the publication of a service offer, the ORS inspects the identity in the credential and the interfaces provided by the service to decide whether to accept the service offer or not.

Besides the identity of the caller, the credential can also contain authentication information about other entities indirectly involved in the call, forming a delegation chain. For example, when a system authenticated as entity *A* receives a call with a credential of entity *B*, it can use this credential received to compose a new credential that indicates the caller is *A* but also containing information about *B* as well. This way, when a system receives an invocation it can decide whether to accept or refuse the call not only in regard of the caller's identity, but also the identity of other systems behind the call.

### 3.1. OpenBus Access Library

Currently the *OpenBus Access Library* is implemented in four programming languages supported by OpenBus: Java, C++, C# and Lua. Since the library is implemented over the standard CORBA programming model, it can be used in conjunction with any compliant CORBA implementation. Currently we provide official support for the following CORBA implementations: JacORB [Brose and Cross 2015], Mico [ObjectSecurity 2010], TAO [Schmidt 2013], IIOP.NET [ELCA 2007] and OiL [Maia et al. 2006].

The API provided by the *OpenBus Access Library* defines a set of abstractions to guide the integration development. The main ones are:

- **OpenBus Context:** the component used to configure the authentication information that shall be attached to an outgoing request and also inspect the authentication information attached in an incoming request.
- **Connection:** represents the link to a particular bus and can be authenticated using an entity's proof of identity.
- **Caller Chain:** an object that contains authentication information about every entity involved in a request. The caller chain always contains the identity of most immediate caller, and possibly the identity of other entities involved in the same request through delegation.

A system that wishes to access the bus first need to use the *OpenBus Access Library* to initialize an ORB extended with all OpenBus features as illustrated in listing 1. The initialized ORB provides access to the *OpenBusContext* that is used to control the authentication information embedded in every call manipulated by this same ORB. The *OpenBusContext* also allows systems to create *connections* to a particular bus. Buses are identified by the host name and port number where its core services are running, which must be informed when a connection is created (line 5).

To complete the connection to the bus it is necessary to bind it to some authenticated entity name (line 6). Once a connection is authenticated, it receives a unique identifier representing this link to the bus called *login*. The login is permanently binded

**Listing 1. Set up credential to be used in a call.**

```
1  ORB orb = ORBInitializer.initORB();
2  OpenBusContext context =
3    (OpenBusContext) orb.resolve_initial_references("OpenBusContext");
4
5  Connection connection = context.createConnection(buscorehost, buscoreport);
6  connection.loginByPassword(entity, password);
7
8  context.setCurrentConnection(connection);
9
10 context.getOfferRegistry().registerService(createServiceRef(orb),
11                                             new ServiceProperty[0]);
```

to the entity name used in the authentication and has a validity time that can be renewed. The *OpenBus Access Library* automatically manages the login renew process using an internal thread.

With an authenticated connection in hands the *OpenBusContext* can be used to set which connection must be attached in the outgoing requests. It can be done by operations `setCurrentConnection` that sets the connection to be used by calls in the current context [1].

The *OpenBusContext* can also be used to retrieve information about the origins of the operation currently being dispatched. This is done using operation `getCallerChain` as illustrated in listing 2. The return value of this operation is the structure `CallerChain`, illustrated in listing 3.

**Listing 2. Inspecting the identity of the current dispatched call.**

```
1  void someOperation() throws UnauthorizedOperation {
2    /* 'context' is an attribute initialized with the 'OpenBusContext' */
3    CallerChain chain = context.getCallerChain();
4    LoginInfo login = chain.caller();
5    String entity = login.entity;
6    String loginId = login.id;
7    if (!entity.equals(expectedCallerEntity)) {
8      throw new UnauthorizedOperation();
9    }
10   /* do actual work */
11 }
```

One main goal in the design of the *OpenBus Access Library* API is to be flexible to perform integrations of non trivial scenarios. The most common case is when parties only need to connect to one OpenBus and maintain only one connection. However exists scenarios that a party may need to impersonate several identities, which requires it to manage more than one connection, and in some cases may also needs to connect to different bus instances. This feature is called *connection multiplexing*.

Another important feature in OpenBus is the support for delegation, which occurs in OpenBus when a system performs calls informing not only its own identity but also including the identity of other systems that previously contacted it in preceding call. To support this the `CallerChain` structure can contains not only the identity of the caller, but also the identity of other systems that contacted the caller previously, which are called *originators*.

[1] As defined by the object `Current` of CORBA, which generally represents the local thread context.

**Listing 3. The CallerChain structure.**

```
1  public interface CallerChain {
2    String busid();
3    String target();
4    LoginInfo[] originators();
5    LoginInfo caller();
6  }
```

By default calls are done without any originator, which means that it will only contain the identity of the caller. To include the identity of another system as originator, the operation `joinChain` is used as illustrated in listing 4. Operation `joinChain` receives a `CallerChain`, which contains the identities that will become originators in all following calls performed in the current context. For example, considering the code in listing 4, the call to operation `otherOperation` in line 5 will indicate the caller and any eventual originator contained in the chain obtained in line 3 as originators.

**Listing 4. Performing call with originators.**

```
1  void someOperation() throws UnauthorizedOperation {
2    /* 'context' is an attribute initialized with the 'OpenBusContext' */
3    CallerChain chain = context.getCallerChain();
4    context.joinChain(chain);
5    otherRef.otherOperation();
6  }
```

The *OpenBus Access Library* also provides access to the ORS, which can be used to register and search for services available through the bus. ORS is accessed as any ordinary service in OpenBus. Therefore the system must first be authenticated in order to access the it, and the ORS only accepts offers from systems that have been previously allowed to offer services by the bus manager. This way, every system that searches for services in the bus using the ORS will only find authentic and authorized service offers.

The governance of the bus is done by a set of tools provided for the bus manager. These tools access special management interfaces of the ACS and ORS. These interfaces allows many management actions like listing current active logins, define entities authorized to offer services and which interfaces these services can implement, among other features.

## 4. Authentication Support

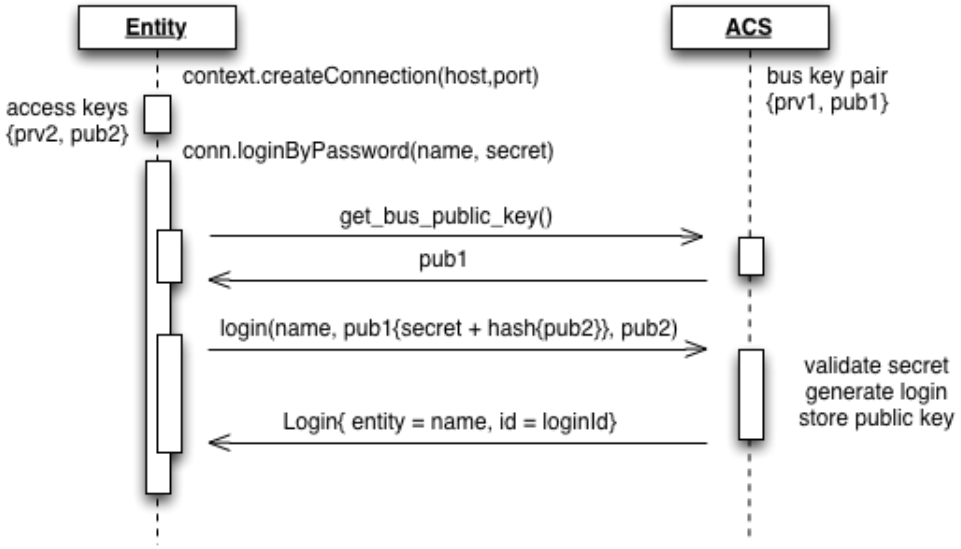
As mentioned in the previous session, to access the bus every system must have a connection to the bus authenticated in the name of an entity. Lets get in more details about what happens when performing an authentication. For security purpose, during the authentication process it is also necessary to provide a pair of asymmetric encryption keys, which will be used later on to certify its identity to other systems connected to the bus. These asymmetric keys are called *access keys* and are usually automatically generated by the *OpenBus Access Library* during the creation of the connection to a bus.

In possession of a pair of *access keys* the authentication is basically done by sending the public *access key* to the ACS combined with an authentication secret. The authentication secret can be a user's password (`loginByPassword`) or an answer to a

challenge [2] (`loginByCertificate`).

To avoid that the authentication secret can be read by someone eavesdropping the network flow, it is sent encrypted with the bus public key provided by the ACS. Additionally, to avoid that a malicious party capture this encrypted secret and reuse it in another authentication, the secret shall be encrypted together with a hash of the public access key provided for the login. This way, someone that captures the encrypted secret can only use it to create another login with the same public access key, which will be useless without the corresponding private key. Figure 2 depicts a diagram that illustrates this process.

**Figure 2. Authenticating in the ACS.**




Once the connection is authenticated, its public *access key* is registered in the ACS and the connection is binded to a newly created *Login*. In possession of this *Login* and the corresponding private *access key*, it is possible to generate *credentials* that provide authentication of the identity of the authenticated entity as will be presented in the following sections.

### 4.1. Caller Credentials

OpenBus uses CORBA's Portable Interceptors (PI) to add support for caller identification on its integrations interactions. CORBA's PI is a standard mechanism to add custom components to the ORB infrastructure that are executed whenever an invocation request is sent to a remote servant or one is received to be dispatched to a local servant. A PI can insert arbitrary data into the request [3] so it is transmitted to the target servant on a remote ORB. Moreover, the PI can also inspect the request, along with any extra data inserted by other PI that manipulated the request locally or remotely.

To extend CORBA invocations with information about the origin of every invocation, the *OpenBus Access Library* installs an special PI during ORB initialization (operation `initORB`). This PI is responsible for a special data structure called *credential*,

[2] A short-lived random generated value encrypted with the public key previously associated to an entity by the bus manager, which can only be correctly decrypted with the corresponding private key.

[3] In the form of a additional Service Contexts as defined in the GIOP protocol of the CORBA standard.

which is used to authenticate the caller in every invocation request sent through the bus. Moreover, the OpenBus PI is also responsible for inspecting each received request to validate the credential information and make it available to the system through the operation `getCallerChain`.

### 4.2. Avoiding Personification

One of the main issues addressed in OpenBus 2.0 was to design a new credential that could not be stolen by malicious parties inspecting the network traffic (sniffers) and reused to personify the identity of other system. The OpenBus 2.0 credential is designed to be used only once and in a particular context. This context is defined by the establishment of a credential session between two communicating parties. The setup of credential sessions is done automatically by the OpenBus PI in both the client and server sides.

Whenever the server-side PI receives a call with an invalid credential, it cancels the dispatch and replies with the standard CORBA exception `CORBA::NO_PERMISSION` containing a special minor code value used to indicate that the associated credential was considered invalid. In this reply the server-side PI also inserts a special data structure called `CredentialReset` that contains information about a new session to be used in future communications. The `CredentialReset` structure is illustrated in listing 5. Field `target` indicates the login of the server where the session was allocated (as a matter of fact, this information cannot be trusted as there is no assurance that the server actually owns this login, as we will discuss in section 4.4). Field `session` is an identification number for the new session allocated in the context of the contacted server. Field `challenge` contains a randomly generated secret for this particular session and encoded with the client's public *access key* that was registered in the ACS during the authentication of the client. The server-side PI keeps a cache of active credential sessions, where each entry contains information like the session ID, the client's *login ID*, the randomly generated secret, etc.

**Listing 5. OpenBus `CredentialReset` Structure in IDL.**

```
1  struct CredentialReset {
2    Identifier target;
3    unsigned long session;
4    EncryptedBlock challenge;
5  };
```

Therefore, whenever a client needs to establish a new credential session with a server to create credentials to be embedded in the calls to its servants, the client-side PI simply makes a call with an invalid credential. This invalid credential contains the client's *login ID* and usually null data on other fields. As a consequence the client-side PI will intercept a reply containing the `CredentialReset`. It decodes the secret in field `challenge` using the client's private *access key* and saves all this information in a cache to be used in future calls to the same reference. Finally the client-side PI reissues the call using CORBA's PI exception `ForwardException`. The reissued call will be intercepted again, but now there will be a established credential session that can be used to generate a valid credential for the call.

Listing 6 illustrates the structure of the OpenBus 2.0 credential. Field `bus` indicates the bus instance the client's login belongs to. This information is used mainly

to allow the use of multiple bus instances by one single system. Field `login` indicates the client's login. Field `session` indicates the session established with the server. Field `ticket` indicates a numeric value that must be different for every credential issued within the session. Every credential generated within a session must use a different ticket value and any credential generated with a ticket that was already used is considered invalid. Field `hash` contains a hash of data composed from the session's secret, credential's ticket and the invoked operation name. This hash value is what guarantees the authenticity of the credential since it contains a secret that can only be recovered using the private *access key* of the login indicated in the credential. Field `chain` is used to implement the support for delegation and shall be discussed in section 4.3.

**Listing 6. OpenBus Credential Structure in IDL.**

```
1  struct CredentialData {
2    Identifier bus;
3    Identifier login;
4    unsigned long session;
5    unsigned long ticket;
6    HashValue hash;
7    SignedData chain;
8  };
```

The tickets used in the credentials are chosen by the client-side PI and the server-side PI only keeps track of the used tickets. Ideally the tickets chosen by the client-side PI should be sequential to enable the server-side to efficiently keep track of the history ticket values already used. However, the server-side PI cannot assume that credentials using sequential tickets will arrive in order due to eventual concurrency issues of client-side invocations. In order to restrict the use of memory resources the server-side PI assumes that only a few credentials will arrive out of order. If one ticket takes too long to be used, the server-side PI considers it invalid anyway and forgets about it. Therefore whenever the credential with that ticket arrives, the server-side PI will consider it invalid. As a consequence, a new session will be allocated and informed to the client in a `CredentialReset` and this new session shall replace the old one on the client-side. The same happens when all the range of tickets of a session is used.

### 4.3. Supporting Delegation

To support delegation the credential must include some authentication information that outlives the context of the session between the server and the client. That is because the server will have to provide this delegation authentication information to others systems that it might need to contact. Additionally, the delegation authentication must also outlive the lifespan of the client because the server might need to create delegated requests even long after the client finishes its execution, for example, to carry out some offline processing in behalf of the client. For this reason the delegation information is provided as a data structure signed by the bus infrastructure itself, so it can remain valid and verifiable even outside the credential or after the session becomes invalid. This signed data is provided in field `chain` of the `CredentialData` structure, as shown in listing 6, and its contents are described by structure `CallChain` illustrated in listing 7.

This signed `CallChain` structure works as a delegation permission that allows the receiver of the credential to perform further calls as part of a call chain composed

**Listing 7. Call Chain structure in IDL.**

```
1  struct LoginInfo {
2    Identifier id;
3    Identifier entity;
4  };
5  typedef sequence<LoginInfo> LoginInfoSeq;
6
7  struct CallChain {
8    Identifier target;
9    LoginInfoSeq originators;
10   LoginInfo caller;
11 };
```

by a whole set of systems. Field `caller` contains information about the login of the most immediate requester, which is the credential issuer and must refer to the same login of field `login` from the `CredentialData` structure. Field `originators` contains information about the logins of other systems that allowed delegation indirectly. When `originators` is an empty list the current call is not delegated, that is, it is not related to any other previous request from other system. Field `target` indicates the *login ID* of the server receiving the credential. The intent of this field is to prevent other parties to use this delegation permission, as we will discuss further in the following section.

### 4.4. Signing Call Chains

Whenever the OpenBus PI needs to create a new signed chain to embed in a credential in a particular session it calls the `signChainFor` operation on the ACS, illustrated in listing 8. Operation `signChainFor` creates a `CallChain` structure signed by the bus infrastructure. Since such data should only be issued to authenticated systems, this operation must be called with a credential that authenticates the caller's identity. To be able to receive requests with proper credentials, the core services have a login of their own and establish credential sessions much like other service-providing systems.

**Listing 8. Signing Chain method.**

```
1  typedef octet EncryptedBlock[256];
2  typedef sequence<octet> OctetSeq;
3
4  struct SignedCallChain {
5    EncryptedBlock signature;
6    OctetSeq encoded;
7  };
8
9  interface AccessControl {
10   // suppressed informations ...
11   SignedCallChain signChainFor(in Identifier target);
12 };
```

However, the credentials for calls of `signChainFor` can have a null value for field `chain`. When this is the case, the returned signed `CallChain` structure contains the login information of the caller in field `caller` and field `originators` is left empty. The value of field `target` of the returned signed `CallChain` is provided by the OpenBus PI as a parameter to operation `signChainFor`. The value provided is the same `target` in the `CredentialReset` structure that describes the current session with the server.

The signed `CallChain` can be sent in a credential as a permission to the server to perform further requests that extends this call chain. The operation `signChainFor` can also be used to extend a received signed chain. In this case this received signed chain is placed in field `chain` of the credential attached to the request for `signChainFor`. The ACS will validate the credential and check if the target of the signed chain matches the identity authenticated in the credential. Therefore, if the caller of `signChainFor` is the target of the signed chain, then the ACS returns a new signed `CallChain` structure with the caller's login information in field `caller` and all logins from the original signed chain in field `originators`.

To exemplify this, let us consider three entities: A, B and C. First, entity A calls an operation from an object provided by B. The PI of A establishes a session with B using the servant reference and then calls `signChainFor` informing the login of B as the target. Then a signed chain with entity A as the caller and entity B as the target is created and sent to B inside the credential for the operation requested to servant provided by B.

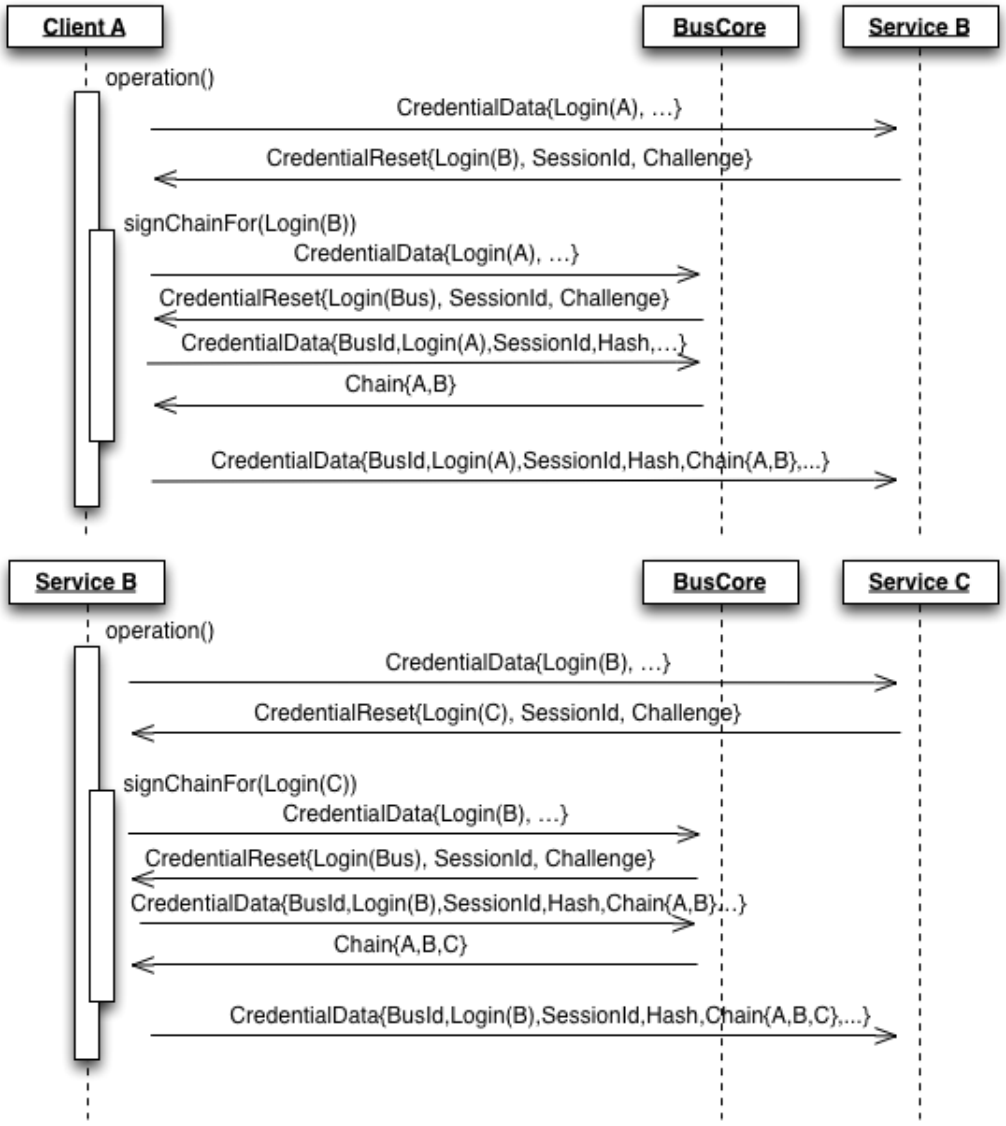
When the PI of B intercepts the request, it extracts the signed chain from the credential and supplies this information for B as an object (`CallerChain` mentioned in section 3.1 as illustrated in listing 3). Entity B can inform the PI that every following call must extend this signed chain (using operation `joinChain` mentioned in section 3.1). When the PI of B intercepts a request to an object of C it calls operation `signChainFor` informing the login of C as the target. However, to extend the original signed chain, this time the credential sent to the ACS contains the original signed call in field `chain`. The ACS inspects the received credential and verifies that the target of the informed chain matches the identity of caller in the credential. Now the ACS returns a new signed chain that contains in the field `originators` the login of A (extracted from the original signed chain). Field `caller` contains the login of B, and field `target` contains the login of C. Such signed chain can now be sent in a credential from B to C, and C shall be able to extend it further. Figure 3 depicts a diagram that illustrates this process.

We mentioned previously that the login informed in field `target` of the `CredentialReset` structure that initiates sessions is not verified. This seems that a malicious process can inform a false login as an attempt to impersonate some entity. However, the `target` of the `CredentialReset` is basically used only to create signed `CallChain` that will be included in credentials to the server. More specifically, this field is only used to restrict which process can use (extend) the signed `CallChain`. This is because the `signChainFor` operation on the ACS certifies that the caller must provide a credential that matches the target login informed in the signed `CallChain`, otherwise it raises a `CORBA::NO_PERMISSION` system exception. Therefore, even if the server informs a login of another process in the `CredentialReset` structure, it will receive a credential containing a call chain that it cannot extend and is useless.

## 5. Security Considerations

In the design of OpenBus 2.0 we ignored the possibility of a malicious party that could not only inspect the network traffic but also interfere with it by modifying data transmitted or its destination, which is generally called as a *man-in-the-middle*. This was mainly due to the fact that to provide a defense against such attacks would require a more basic support from the ORB to provide data integrity to the network stream, which is not possible using

**Figure 3. Signing call chains.**



CORBA's PI.

The main issue with a *man-in-the-middle* attack is that it can modify a valid request to a innocuous operation with a valid credential into a potentially harmful operation. The OpenBus' credential contains the operation name encoded in its verification hash code. Therefore, if the operation name in the request is changed the credential will become invalid. However, the parameters of the request can be modified freely.

For example, consider that a user with privileged access starts a system that logs into the bus using its personal password. Then the system sends a request for operation `deleteFile` providing the path to a temporary file on the remote system. If a *man-in-the-middle* is able to intercept and change the parameter of the request to refer to a important document in the remote system that only the user can delete, the malicious party can cause the deletion of the document and that will be registered as solicited by the user.

A more dangerous scenario is a *man-in-the-middle* that can interfere with requests to the ACS. Since the ACS is responsible for the distribution of public *access keys*, a malicious party can replace the actual public keys in ACS replies with alternative generated keys and effectively personify the identity of any system with a valid login in the bus.

## 6. Future Work

Originally we adopted a simple and minimalistic approach on the design of OpenBus, focusing on satisfying the requirements of some system integrations, but also keeping its usage simple and straightforward. Even though OpenBus is now being successfully used in the integration of a variety of systems, it might be inadequate for use in other scenarios. For instance, some system integrations may require a higher level of control like defining special permissions for entities, restriction on delegation rights or protection of confidential data. In this section we discuss some of the main issues that might have to be addressed in the future of the OpenBus project.

### 6.1. Permission Attributes

The credential only contains information about the identity of the originators of the call. Any other information about the originators, like its permissions, are left for the service to figure it out, usually using a local permission database. However this approach of permission databases scattered across different integrated systems might not be feasible due to the administrative effort implied. Therefore it might be necessary to include in the credential some information about the entity permissions as defined by some central management authority. This way systems can rely also in such additional permission information. Note that such approach should not stop systems to enforce additional local permission restrictions whenever necessary.

### 6.2. Data Protection

Security issues related to a *man-in-the-middle* are essential in many contexts. One option is to make the ORB use some secure transport technology such as TLS [Dierks 2008] or SSL [Freier et al. 2011], which provides both data integrity and privacy. Most ORBs implementations provide support for use of TLS/SSL as defined in the CORBA standard which allows interoperability between implementations. However these implementations

rely on non-standard interfaces to control the use of TLS/SSL and in some cases provide different ranges of functionality.[Alireza et al. 2000]

TLS provides an identity authentication mechanism based on public key certificates. Such mechanism could replace in part the origin identification provided by OpenBus. However, the management of a public key certification infrastructure can be too costly for many scenarios where OpenBus is used. For example, it might not be feasible to provide public key certificates and secure a private key for all human users that can execute programs that access the bus. That is specially cumbersome where the users are habituated to rely on their personal passwords for authentication.

Currently, OpenBus core services do not support TLS as defined in the CORBA standard. This makes difficult for systems to use TLS while contacting other systems, since it would be required to control the use of TLS in a object by object basis. We are currently adapting the core services to support TLS to provide data integrity and privacy in invocations. However, since the TLS support is not integrated with OpenBus features, it is up for the system implementation to guarantee the necessary security requirements. For example, to avoid that a *man-in-the-middle* be able to change the requests, before performing any remote call the system should require the identity of the service using the TLS support and verify that it matches the expected identity of the system that should be providing the service. Otherwise, the target could be someone imitating the correct destination.

In the future, we intend to provide an integrated support of SSL with OpenBus, which could combine the caller authentication of OpenBus with either the certificate-based authentication provided by TLS or some other alternative server-side authentication mechanism.

## 7. Final Remarks

OpenBus is a free and open-source CORBA-based middleware technology designed to fulfill the basic requirements of a infrastructure for integration of systems under the control of a corporation. The adoption of the CORBA standard as the basis for the solution greatly reduced the development effort by the reuse of freely available quality middleware implementations. The use of open-source CORBA implementations also allowed us to identify problems in the use of the technology and apply solutions promptly. The CORBA communication support also showed flexible enough to fulfill the demands of the system integrations done using OpenBus.

On the other hand, the CORBA standard also presents some drawbacks for systems using OpenBus. Mainly this is due to the complexity of CORBA's programming model, which aims to be very general and handle a wide variety of use cases. Some CORBA-based integrations tends to be viewed as unnecessarily complicated when compared to similar solutions focused on specific use scenarios or platforms, like WebServices or JavaRMI. Moreover, the CORBA technology has attracted few attention and investments lately. As result many CORBA implementations are not actively maintained anymore. Eventually this can force the OpenBus team to embrace the maintenance of some of the CORBA implementations that OpenBus relies on. Ultimately this will result in a increase of the costs of the project and might motivate the adoptions of other technologies as an alternative to CORBA. Alternatively, the need to maintain the CORBA

implementations can motivate the deign of new solutions that are not limited to standard CORBA API like the use of Portable Interceptors as used by OpenBus currently. The possibility to introduce new features in the ORB implementations can bring performance and security improvements to the solutions adopted in OpenBus.

Even though OpenBus still exhibits many possibilities of evolution and improvement, it is already being successfully used in different scenarios. OpenBus major usage is in the area of oil extraction and production of Petrobras, the Brazilian OiL Company. Petrobras maintains two main instances of OpenBus in production: one for integration of exploration data management systems and other for integration of oil and derivatives supply chain systems. OpenBus is also being used in SINAPAD, Brazilian National HPC Network, as the middleware that integrates the mc2 portal [Gomes et al. 2015] to CSGrid [de Lima et al. 2005] services. In a similar scenario, the OpenBus and CSGrid are, at the moment, part of the solution adopted in EUBrazilCC [EUBrazilCC 2014] project, aiming at supporting the federation of HPC and cloud resources to other programming frameworks.

## References


Alireza, A., Lang, U., Padelis, M., Schreiner, R., and Schumacher, M. (2000). The Challenges of CORBA Security. In *Sicherheit in Netzen und Medienströmen*, pages 61–72. Springer.

Bolton, F. and Walshe, E. (2001). *Pure CORBA: A Code-Intensive Premium Reference*. Sams Indianapolis, IN, USA.

Brose, G. and Cross, N. (2015). JacORB. `http://www.jacorb.org/`. Accessed on March, 2015.

Bustamante, F., Eisenhauer, G., Schwan, K., and Widener, P. (2000). Efficient wire formats for high performance computing. In *Supercomputing, ACM/IEEE 2000 Conference*, pages 39–39. IEEE.

Chappell, D. (2004). *Enterprise Service Bus*. O'Reilly Media, Inc.

Davis, D. and Parashar, M. (2002). Latency performance of SOAP implementations. In *Cluster Computing and the Grid, 2002. 2nd IEEE/ACM International Symposium on*, pages 407–407. IEEE.

de Lima, M. J., Melcop, T., Cerqueira, R., Cassino, C., Silvestre, B., Nery, M., and Ururahy, C. (2005). CSGrid: um Sistema para Integração de Aplicações em Grades Computacionais. In *Proceedings of SBRC'05 - Salão de Ferramentas*, volume 2, pages 1207–1214, Fortaleza, Brazil. SBC.

Dierks, T. (2008). The transport layer security (TLS) protocol version 1.2.

ELCA (2007). IIOP.NET Overview. `http://iiop-net.sourceforge.net`. Accessed on March, 2015.

EUBrazilCC (2014). EU Brazil Cloud Connect — EU Brazil Cloud Computing for Science. `http://www.eubrazilcloudconnect.eu`. Accessed on March, 2015.

Freier, A., Karlton, P., and Kocher, P. (2011). The secure sockets layer (SSL) protocol version 3.0.

Gomes, A. T. A., Bastos, B. F., Medeiros, V., and Moreira, V. M. (2015). Experiences of the brazilian national high-performance computing network on the rapid prototyping of science gateways. *Concurrency and Computation: Practice and Experience*, 27(2):271–289.

Govindaraju, M., Slominski, A., Choppella, V., Bramley, R., and Gannon, D. (2000). Requirements for and evaluation of RMI protocols for scientific computing. In *Proceedings of the 2000 ACM/IEEE conference on Supercomputing*, page 61. IEEE Computer Society.

Hohpe, G. and Woolf, B. (2004). *Enterprise integration patterns: Designing, building, and deploying messaging solutions*. Addison-Wesley Professional.

Josuttis, N. M. (2007). *SOA in practice: the art of distributed system design*. " O'Reilly Media, Inc.".

Kohlhoff, C. and Steele, R. (2003). Evaluating SOAP for high performance business applications: Real-time trading systems.

Maia, R., de Gusmão Cerqueira, R. F., and Calheiros, R. (2006). OiL: An object request broker in the Lua language. In Batista, T., editor, *Proceedings of SBRC'06 - Salão de Ferramentas*, pages 1439–1446, Porto Alegre, Brazil. SBC.

Manes, A. T. (2007). Enterprise Service Bus: A Definition. *Burton Group*, pages 1–35.

Neuman, B. C. and Ts'o, T. (1994). Kerberos: An authentication service for computer networks. *Communications Magazine, IEEE*, 32(9):33–38.

ObjectSecurity (2010). MICO CORBA. `http://www.mico.org/`. Accessed on March, 2015.

OMG (2002a). *Common Object Request Broker Architecture: Core Specification - Version 3.0*. Object Management Group. document: formal/2002-12-06.

OMG (2002b). *CORBA Security Services Specification*. Object Management Group.

Schmidt, D. C. (2013). Real-time CORBA with TAO (The ACE ORB). `http://www.cs.wustl.edu/~schmidt/TAO.html`. Accessed on March, 2015.

Tecgraf/PUC-Rio (2006). OpenBus - Enterprise Integration Application Middleware. http://www.tecgraf.puc-rio.br/openbus. Accessed on March, 2015.