Wrapping SRS with CORBA: from textual data to distributed objects

Thierry Coupaye*
IMAG LSR, University of Grenoble, Actinart, Bat. 8, Avenue de Vignate, 38610 Gieres, France

Received on October 5, 1998; revised on December 18, 1998; accepted on December 21, 1998

Abstract

**Motivation:** Biological data come in very different shapes. Databanks are maintained and used by distinct organizations. Text is the de facto standard exchange format. The SRS system can integrate heterogeneous textual databanks but it was lacking a way to structure the extracted data.

**Results:** This paper presents a CORBA interface to the SRS system which manages databanks in a flat file format. SRS Object Servers are CORBA wrappers for SRS. They allow client applications (visualisation tools, data mining tools, etc.) to access and query SRS servers remotely through an Object Request Broker (ORB). They provide loader objects that contain the information extracted from the databanks by SRS. Loader objects are not hand-coded but generated in a flexible way by using loader specifications which allow SRS administrators to package data coming from distinct databanks.

**Availability:** The prototype may be available for beta-testing. Please contact the SRS group (http://srs.ebi.ac.uk).

**Contact:** Thierry.Coupaye@imag.fr, Thure.Etzold@ebi.ac.uk

Introduction

The SRS system (Etzold et al., 1996; Carter et al., 1998) was initially developed as a Sequence Retrieval System but has expanded far beyond these limits today. Originally devised at the European Molecular Biology Laboratory (EMBL) in 1990, SRS’s progress has continued at the European Bioinformatics Institute (EBI) and then at LION Bioscience Ltd. One fundamental property underlies most biological databanks: their availability in ASCII text format. SRS exploits this universal medium to create a homogeneous collection of databanks. SRS provides simultaneous access to different semistructured databanks and can create linked data. The kind of data SRS deals with are referred to as semistructured because they are implicitly structured, i.e. they are intrinsically structured but this structure is not explicit and therefore not exposed to the users or applications. The data is self-described, there is no data schema. Also data can be incomplete, redundant, irregular or even erroneous. SRS utilizes a parsing and indexing mechanism to cross-reference entries of different databanks. The system provides several interfaces to access data both for humans and programs or applications. The command line interface is a set of programs that allows parsing, indexing and querying databanks. The World Wide Web (Web) interface (SRSWWW) enhances the command line interface with hypertext links, views, etc.; and provides a remote access to SRS servers. The C-API (Application Programming Interface) interface provides structured information as C structures that can be used in C programs.

The Web interface provides a remote access to data but returns only textual information — while the C-API provides (low) structured data but cannot be accessed remotely. Some applications however need both remote access to SRS servers and objects (instead of text) so that they can achieve complex processing of data that cannot be done within SRS such as visualizing data, chaining operations on data (DNA or protein sequence alignments, etc.), or storing data in relational or object-oriented DBMS so that they can be reused later on. This can be achieved through SRS Object Servers. SRS Object Servers provide remote access to semistructured textual databanks managed by SRS and present query results as distributed objects in a flexible way. They provide CORBA objects (servers) that can be accessed by applications (clients) through an Object Request Broker (ORB) (OMG, 1995a). There are two kinds of objects (the architecture is depicted by Figure 1):

- **Loader Objects** actually contain the information. They extract data from SRS and convert them according to ‘conversion schemes’. These objects are not hand coded but generated based on **Loader specifications**.
Loader specifications

Loader specifications are provided by SRS administrators. They are specific to each SRS Object Server and represent ‘views’ on SRS databanks (some kind of ‘virtual databanks’). The power, genericity and flexibility of our approach comes from the fact that the largest part of an SRS Object Server, the loader objects, are not hand-coded but generated. The SRS Loader Generator is a tool that uses loader specifications plus meta-information about databanks and applications provided by SRS to generate both the IDL definition of loader objects and their implementation. We focus here on loader specifications. Loader definitions (IDL) and implementations are described later.

Loader specifications express the mapping of SRS entries to CORBA loader objects. They express how an SRS entry is converted into a CORBA object, i.e. where to find the data and how to package it. The main features of loader specifications are the following. Several loader specifications can be provided for one databank, providing different views on this databank. For each databank, it is possible to declare a loader as its default loader. Furthermore, the system provides an absolute default loader, the Basic Loader, which can be used to load any entry of any databank. Loaders can have input from several databanks: a general loader can be used for several databanks that are related (for example the sequence databanks) providing a unified view on databanks that are ‘close’ to one another. Loaders can inherit from one another. All loaders inherit from the absolute default loader — but one can also define more specific loaders by using inheritance. Loaders can define foreign attributes: a loader wraps one databank but it can get some information from one or several other linked databanks. It is possible to define if the attribute is single or multi valued. Loaders can define composition links (relationships): a loader object can have a link (relation) to another loader object. This enables the creation of object graphs. It is again possible to define the cardinality of this link (single or many). Loaders can compute data: a function can be associated to a loader attribute so that its value is computed (instead of getting it from the databank).

The language used for expressing loader specifications is Icarus. Icarus refers both to the language used to extract the data by parsing the databank entries and the parser of this language. Icarus is an essential part of SRS: the more flexible and powerful the parser is, the more information can be extracted from the databanks. Both loader specifications and meta-loader specifications (which express what can be done in a loader specification) are expressed in Icarus. A loader definition consists of different attributes such as the name of the loader, the list of databanks that the loader wraps and the inheritance clause. Only the name of the loader is mandatory. After this, a list of loading attribute definitions is the central part of the loader definition. Each loading attribute can be seen as a pair < attribute specification, loading specification > in which:

- attribute specification is the definition of the exported attribute. It consists of the name of the attribute, its type (string, int, real, object, etc.) and its cardinality (single or multi valued).
- loading specification defines how to get the data from SRS in order to make it available as an attribute value.
SRS is based on a parsing and indexing mechanism to cross-reference entries of different semistructured databanks and provides as an output tokens which represent pieces of information. These pieces are not isolated however. SRS provides simultaneous access to different databanks and can create linked data: tokens are linked to other tokens from other databanks. A loading specification defines mainly the token name and optionally the name of the databank, or group of databanks (if several tokens with the same name exist for several databanks); if a link operation has to be performed to obtain the entry containing the requested token (if the token comes from another databank); or an instruction (Icarus code) to compute the value of the attribute.

We give hereafter an example of loader specification. This example is not intended to be very realistic but to show most of the capabilities of loaders.

```plaintext
BioSeq_Class:$LoadClass:[BioSeq
attrs:
$LoadAttr:[AccNumber type:string
load:{
 $Tok:[acc from:@EMBL_DB]
 $Tok:[acnc from:@SWISSPROT_DB]]
}$LoadAttr:[Sequence type:string
isSeq:@PROTSEQ_DATA]
$LoadAttr:[MedlineCit type:string
load:{
 $Tok:[cit link:medline]]
}$LoadAttr:[Taxa type:object
class:@Taxonomy_Class]
$LoadAttr:[SeqType type:string
load:{
 $Tok:[value:'DNA'
from:@EMBL_DB]
 $Tok:[value:'Protein'
from:@SWISSPROT_DB]]]
}
]
```

The Icarus object `BioSeq_Class` defines a loader called `BioSeq`. This loader specification defines 5 attributes: `AccNumber`, `Sequence`, `MedlineCit`, `Taxa` and `SeqType`. The loader may have input from multiple databanks; it can be used to load entries of the EMBL (Rodriguez-Tomé et al., 1996) or SWISSPROT (Bairoch and Apweiler, 1997) databanks. If the actual entry comes from EMBL, the value of the attribute `AccNumber` (AccessionNumber) will be set by getting the token `acc` from SRS. If the entry comes from SWISSPROT, the token `acnc` will be used. After this the attribute `Sequence` is defined. The attribute will eventually provide the actual biological sequence as a string. The sequence is accessed through a predefined token in SRS. This is why no load clause is needed for sequence attributes. The loader specification also defines a foreign attribute: `MedlineCit`. The value of `MedlineCit` will be a collection of bibliographic citations from the Medline databank (cf. http://medline.cos.com/) related to the sequence. Citations are obtained by getting the token `cit` from Medline using a link to the Medline databank. The attribute `Taxa` defines a composition link. The value of the attribute will be a set of entries from the Taxonomy databank as defined by the loader object `Taxonomy_Class`. `SeqType` is a computed attribute. No token is extracted from the databanks. Instead, a value ‘DNA’ is assigned if the entry comes from EMBL which is a databank of DNA sequences; or ‘Protein’ if the entry comes from SWISSPROT which is a databank of protein sequences.

### Loader objects

Loaders specifications are mapped to IDL interfaces, and actual entries are instantiated as objects of these interfaces: the loader objects. Interfaces form a hierarchy. The root of the hierarchy is the interface `Loader` referred to as the **Basic Loader** (the absolute default loader).

#### The basic loader

The Basic Loader `Loader` is the most generic loader. It is very simple. It does not provide a lot of information but can be used to load entries of any databank. The `Loader` interface IDL definition is the following:

```plaintext
interface Loader {
    readonly attribute string source;
    readonly attribute string loader;
    readonly attribute string ID;
    Collection link( in string target_source,
                    in string loader);
}
```

The attribute `source` represents the actual source databank from which the entry has been loaded using the loader `Loader`. `ID` is the entry identifier. Identifiers can be used to compare a loader object with an SRS entry retrieved by another way (command line, Web interface or C-API). IDs can also be used to compare objects: two objects may be instances of different loader interfaces but may actually represent the same databank entry loaded with two different loaders.

Loader objects exhibit a linking capability. The operation `link` retrieves entries from `target_source` that are linked with the current entry in the current databank and used the loader `loader` to load them. Note that the two databanks may not be linked directly. An SRS system can be seen as a
network of databanks. SRS is able to link any databank to any other databank (except databanks that are not connected in any way to any other databank) by computing the shortest path between the two databanks.

Loader generation

When using a CORBA SRS Object Server, most objects (the loader objects) are in fact generated by the Loader Generator. They belong to interfaces that inherit (directly or indirectly) from the Loader interface. They have particular attributes and methods that match attributes defined in the loader specifications according to a mapping not completely exposed here. Roughly, ‘local’ loader attributes (i.e. attributes whose corresponding tokens come from the wrapped databank) are mapped into CORBA attributes except sequence attributes (that provide actual DNA or protein sequences) that are mapped into CORBA operations in order to include a parameter format which defines in which format the sequence should be provided (many databanks or applications require their own format for sequences). Foreign and composition attributes are mapped into operations also. Some other operations facilitate the launch of applications (sequence alignment, restriction maps, etc.) on entries. These operations can be generated because the Loader Generator knows which operation can be launched on entries of each databank by using meta-information provided by SRS. As an example, for the loader \texttt{BioSeq} defined earlier, the Loader Generator would generate the following IDL interface:

```idl
interface BioSeq:Loader {
    readonly attribute string AccNumber;
    string Sequence (in string format );
    readonly attribute string MedlineCit;
    readonly attribute string SeqType;
    TaxonomySeq Taxa ()
        raises (InternalCommFailure);
    Collection blast (  
        in BlastOptions Options,  
        in string LoaderName )
        raises (InternalCommFailure);
    Collection fasta (  
        in FastaOptions Options,  
        in string LoaderName )
        raises (InternalCommFailure);
    ...
};
```

The attribute \texttt{AccNumber} represents the accession number of the sequence (DNA, RNA, protein). The operation \texttt{Sequence} returns the sequence. It is not defined as an attribute since sequences can be expressed in different formats (needed by different applications for instance). The format is a parameter of the operation. If no format is specified, the operation chooses one according to the default format. Each databank has its own default format. Attributes \texttt{MedlineCit} and \texttt{SeqType} are mapped into attributes while the composition link \texttt{Taxa} is mapped onto an operation. Sequence analysis applications such as the NCBI’s Blast and Bill Pearson’s Fasta are widely used in the Bioinformatics community. They are provided by distinct organizations, define their own sequence formats and support different options. Moreover they return results in very different formats. SRS Object Servers are very useful in this context since they enable the retrieval of sequences in various formats, launching of external applications (operations blast and fasta) and choosing the format of the resulting objects through the use of loaders.

The Loader Generator will not stop here. It will not only generate the IDL definition but also the actual C++ implementation of the loader objects that contains all runtime communications with SRS for dynamically extracting the data from the databanks and launching of external applications. It should be noted that the C++ code generated is quite repetitive and not very pleasant to write for it mainly consists of low level C-API calls to SRS. This shows that even for small servers, it is much faster and user-friendly to generate the loaders than to hand-code them. They are also easier to update.

SRS object servers

Loader objects contain the data extracted from databanks by SRS but they are not standalone. They need other objects and services to access and query them. These objects and services are referred to as \textit{Common Objects} because they are generic and shared by all SRS Objects Servers (cf. Figure 1). CORBA front-ends to SRS Object Servers are instances of the interface \texttt{SRSOS}. These objects are registered to a CORBA Naming Service (OMG, 1995b). Clients can get a reference to these objects through the Naming Service and send them requests to get the four \textit{Services} described hereafter. The SRSOS interface is defined as follows:

```idl
interface SRSOS {
    QueryEvaluator get_QueryEvaluator();
    DatabankFactory get_DatabankFactory(  
        in string Name);
    LoaderFactory get_LoaderFactory();
};
```

Querying databanks

\texttt{QueryEvaluator} objects provide the means to submit and remove queries. Queries are expressed against SRS databanks — not against loaders. Loaders are not views. Their only purpose is to convert entries retrieved by querying databanks. \texttt{QueryEvaluator} objects represent the most common use of SRS Object Servers.

```idl
interface QueryEvaluator {
    readonly attribute QLTypeSeq types;
};
```
readonly attribute QLType default_ql_type;
Collection evaluate {
    in string query,
    in QLType ql_type,
    in string queryName,
    in ConversionSchemeSeq conversions);
    short remove (in string queryName);
};

The attributes types and default_ql_type describe the available query languages and default query language respectively. For the time being, only the SRS Query Language is available. The operation evaluate submits the query to the SRS system, gets the set of entries satisfying the query, builds and finally returns the corresponding collection of loader objects. While waiting for ORB vendors to provide Collection Services, SRS Object Servers provide a simple generic Collection interface that facilitates the manipulation of objects: adding an element, suppressing an element, iterating over elements, retrieving an element by position (if the collection is sorted), etc. The returned objects are built according to conversion schemes given in the parameter conversions. A conversion scheme is described by an IDL structure that associates a loader to a databank (it specifies the packaging of the entries):

struct ConversionScheme {
    string DatabankName;
    string LoaderName;
};
typedef sequence <ConversionScheme>
    ConversionSchemeSeq;

Parameters ql_type, queryname and conversions are optional. During the evaluation, if a conversion scheme is not specified for a given databank, the operation evaluates uses the default conversion scheme (default Loader). If this default loader does not exist at all for a given databank, evaluate uses the basic loader (called Loader).

Query names are useful for combining query results (logical operators, etc.). Inside an SRS session associated with a client, it is not possible to define a query with a query name that already exists. The operation remove can be used to remove a query so that its name can be re-used later in the session.

Converting a complete databank

DatabankFactory objects create all objects that can be created using a given loader. They represent the highest abstract level service provided by SRS Object Servers for the user does not have to know anything about SRS or the SRS Query Language. It can be useful, for instance, for converting all at once a complete databank and storing it in a conventional DBMS. DatabankFactory is defined as follows:

interface DatabankFactory {
    Collection load (in string loaderName);
};

Converting an entry

In contrast, LoaderFactory objects represent the lowest abstract level service of SRS Object Servers. A LoaderFactory object converts one SRS entry into a loader object according to a specified loader. LoaderFactory is defined as follows:

interface LoaderFactory {
    Loader load (
        in Any entry,
        in string loaderName);
};

The parameter entry of the operation load represents the SRS entry that will be loaded. This entry has to be retrieved using the SRS C-API. The operation load can thus be used directly by a client application using the C-API. It is one of the core functions of SRS Object Servers for it is used all the time to convert SRS entries after a query, after loading a complete databank, after linking an entry to a databank and after launching an application on an entry.

Implementation and performance

Implementation

The Loader Generator is part of the SRS system. It represents about 1000 lines of Icarus code and 2000 lines of C code (and SRS C-API). It takes as input the loader specifications written in Icarus and generates both the IDL definition and the C++ implementation of the loader objects. The IDL definition and C++ implementation of the Common Interfaces are then merged with the generated ones and an SRS Object Server can then be built for a specific ORB in a specific environment (computer and operating system). Current experiments are performed using the ORB ORBacus. The size of the common objects is approximately 250 lines of IDL and 1500 lines of C++. The size of the loader objects depends greatly of course on the number and complexity of the loaders defined. In one of our test servers providing seven loaders for fifteen databanks, there are about 150 lines of IDL and 3000 lines of C++ produced.

Performance

Figure 2 shows the results of a very simple experiment that compares the response time of the SRS command line program getz and SRS CORBA Object Servers. A lot of applications use SRS in a very basic way to fetch entries by their identifier by embedding calls to the command line program getz in the application code. To compare SRS getz and SRS Object Servers, we have written two very simple CORBA
clients (clients and server were running on different computers but on the same local network) that fetch entries by their identifiers through an SRS Object Server (through QueryEvaluator objects actually). The first client (SRS OS: 1 connection per entry) opens and closes a connection to the server for each entry fetched. This solution is not very efficient since a large amount of time is spent in CORBA communications to establish and cut connections. However, this solution is already 2.5 times faster than using SRS getz. The second client (SRS OS: 1 connection) is a much more efficient solution in which the connection with the server is established, then all entries needed are fetched and the connection is then closed. This solution is 12 times faster than the previous one and thus 30 times faster than SRS getz (all solutions use the same API but a complete initialisation of the SRS system is needed for each call to getz).

**Conclusion and related works**

The SRS system is widely used in the Bioinformatics and Molecular Biology communities to access biological information in the form of flat files. SRS now offers a CORBA interface. SRS Object Servers can be accessed and used remotely by client applications to extract and query information from databanks managed by SRS servers. SRS Object Servers build in a flexible way the objects that are sent back to the clients: loader objects glean data from one or several databanks and load them dynamically according to some conversions schemes based on loaders specifications provided by SRS administrators to fit the clients’ needs.

There is a rising interest for CORBA and distributed objects in general in the Bioinformatics community. Some systems now offer CORBA access to biological databanks (Rodríguez-Tomé et al., 1997; Hu et al., 1998). These works and the work of the Life Science Research Task Force (LSR-TF) of the Object Management Group (OMG) are based on the idea that standard interfaces should be defined for most common biological entities and data sources. We do not really believe in this approach. These standards interfaces could not encompass all already available or upcoming data sources or systems; nor specific clients’ needs. Our approach is quite completely different since loader objects which provide the data are not hard-coded but generated in order to provide customized views on databanks — and the way the data is actually stored is almost irrelevant to SRSOS users. SRS Object Servers do not only offer access to specific databanks but flexible accesses to potentially hundreds of databanks and it can link data from these databanks — provided these databanks are available in a flat file format the SRS system can deal with (which is a very weak requirement). Note however that our approach is not incompatible with the LSR-TF effort since loader specifications can be provided to match LSR-TF evolving standard interfaces.

**References**


