A flexible framework for the design of knowledge-discovery clients

S. Ramirez\(^1\), J. Karlsson\(^1\), García M.\(^1\), Rios Perez J.\(^1\), O. Trelles\(^1\)
\(^1\) Department of Computer Architecture, University of Malaga, Spain
{cerr, johan, maxgarcia, jrios, ots}@ac.uma.es

Abstract. This paper describes a flexible and expandable framework for the
development of distributed architectures which enable the discovering and
exploitation of knowledge based resources. The framework provides modular
functionality for sharing and discovering different resources metadata related to
tools, users, and datatypes registered in central repositories, including the
ability for handling data storage and tool invocation. We demonstrate with
several clients how different components of the framework can be combined to
produce distributed architectures to better exploit project-specific requirements.

Keywords: Service integration, biomedical informatics, Bioinformatics

1 Introduction

Distributed computing has shown to be useful in domains with high-performance
computing requirements. Typical solutions to enable distributed computing use web-
or grid-services. Higher-level tasks can be solved by combining these services into
workflows. However, service interoperability has remained a difficult challenge
because each service protocol use specific data types/formats and data transportation
mechanism. A frequent approach to improve service interoperability is central
repositories with standardized metadata to allow clients to discover tools, and service
providers to specify shared data formats. Biomoby [14], FETA [16] and others
systems make use of this strategy to implement integration architectures.

Web services usually do not have any sort of user interface built in, and it is
generally up to the client to process input and display output. Therefore, Web-services
repositories provide some run-time APIs so that service consumers or intermediaries
can dynamically discover the most suitable service. In this way, clients can be written
to access the repository, obtain the needed information to discover, invoke and
process the response from the Web service.

The clients that exploit these repositories have been developed in most cases with
specific orientation to the repository used as source of metadata information:
Gbrowse-Moby [15], Ahab [2], MOWServ [10], Seahawk [6], Remora [4], Dashboard
[3], myGrid [8], Taverna [12], etc. Service interoperability between service protocols
remains difficult because each repository provides its own protocol with particular
specifications.
As a contribution towards the development of ‘high level’ semantic integration of services registered in diverse repositories, we have developed a flexible and expandable framework to allow fast development of clients that access distributed computational resources. A software framework [13] is a code library designed to facilitate software development.

In summary, the framework uses a shared and searchable catalogue to store, query and access metadata for different resources, such as tools and data types. For example, tool metadata instances are presented to the end-user as a single catalogue, hiding the details of specific service protocols. Differences in the protocols are handled by specific software components for each protocol.

In section 2, we will describe the architecture of the framework. The architecture specifies how the functionality of the framework has been organized in several modules (components) and how the modules are related.

Several components of our framework have been used in two projects; the Spanish National Institute for Bioinformatics [11] and the EU-project Advancing Clinical-Genomic Trials (ACGT) [1]. Both INB and ACGT maintain particular catalogues of available tools, but while INB uses BioMOBY services, ACGT bases its system on a Globus computational grid [5]. In section 3 will be shown some examples of clients developed in both projects.

2 Architecture

The framework has been organized in several modules, each in charge of specific task, such as tool metadata maintenance and discovery, file browsing and user administration. By splitting the functionality in modules, is possible to provide a flexible framework where modules can be combined and adapted to project-specific requirements. For example, a project might need to only handle tool and data type metadata because other types of metadata are already handled by other software components.

These modules are organized following a hierarchy of dependencies, as can be seen in the Fig. 1. The most basic module allows the definition of available data types in the system. Two main branches of dependencies correspond to Tools (software components that consume and/or produce data) and instances of a particular data-type (Data). The enactor combine these two branches by sending the data to the tools for execution using the information stored in the mirror module. The modules related to access-rights (users and groups) can be combined with every module providing authentication and access rights for resources.
Each module is composed by several layers. The layers can be combined in several ways, allowing clients to work independently of metadata location (local or remote) and the access protocol. Clients connect to modules through the following layers:

- **Application interface layer**: This is the highest level programming interface. All clients (user program) access and use this interface.
- **Local/remote access layer**: Depending on configuration, the application interface layer accesses either a local implementation (if metadata is available locally) or a remote access layer (if metadata is available on a remote server).

Servers provide access through the following layers:

- **Remote interface layer**: This interface is used by the remote access layer of the client if the client uses the remote access layer. The functionality of the interface is exposed as SOAP web-services.
- **Local data access layer**: This interface is used by the remote interface layer of the server or directly by the local access layer of the client if metadata is available locally for the client software.

These layers can be used to connect to different repository systems. Access layers have been developed for BioMOBY tool repository; e.g. MobyCentral (http://moby.ucalgary.ca/gbrowse_moby); INB (http://www.inab.org/MOWServ) and the ACGT tool repository (http://www.siveco.ro:8080). We provide a uniform view of metadata by handling multiple repositories in a common way, independently of the access protocol and location of the metadata.

### 2.1 Modules

In this section we will briefly motivate and describe each module of the framework. Each module provides functionality for creation, administration and discovery the metadata of their specific resources.

- **Datatype**: This module defines a shared taxonomy of data types. This is an important requirement when building workflows because the integration of tools implies that tool providers agree on data types. The
taxonomy allows relationships between data types to be expressed. In particular, hierarchies can be expressed, allowing one data type to extend the definition of another data type and provide additional structure.

- **Tool**: This module handles tool metadata. The concept of tool is used to represent different types of web-services, workflows, applications, etc in a uniform way. For example, ACGT services are described with the same metadata as BioMOBY services or workflows. The module can also produce the necessary information for generating a graphical user interface where the end-user may enter input data needed to invoke/enact the tool.

- **Mirror**: This module handles service endpoint metadata. An endpoint represents a computer resource where a web-service is located and can be executed. Each service may be associated with any number of endpoints. This allows clients to choose between several alternate endpoints and adds robustness to service invocation when some endpoints are not available.

- **Functional category**: This module handles functional category metadata. A functional category is a keyword that describes a functionality of a tool. The keywords are arranged in a hierarchical structure in such a way that each tool can be annotated with one or more of these keywords from very specific description to more generic functional keyword.

- **Data**: This module is mainly concerned with data handling (the results of tool invocations), not metadata handling. It provides loading and saving of data, access to the internal structure of the data, handles sets of data, etc.

- **File system**: This module provides features to access files and folders to recover the information stored without concern about where they are physically stored.

- **Execution**: This module loads data via the data module and invokes the tools using appropriate protocols. A set of software components (workers) are associated to each implemented service protocol. These components contain all the specific functionality needed for invoking a tool using a specific protocol (e.g. a specific worker is needed to interact with BioMOBY services). Thanks to of the layered design of the module; it is possible to work in remote or in local, using a serial dispatch system or a scheduler. The scheduler is able to dynamically choose the best endpoint using different configurable algorithms. These algorithms enable the scheduler to perform load balancing between the different instances of the tool and offer robustness when a particular tool instance is not available.

- **User**: Normally in a distributed system it is necessary to provide authentication and authority mechanisms. This module adds new functionality to the system to define and handles information of types of users (end-users, metadata-owner etc). Unix-like rights can be assigned to metadata instances, allowing certain instances to remain available (for using or modifications) to a certain user.

- **Group**: This module extends the functionality of the user module to allow the definition of users groups to allow of sharing resources between members of the same group.
3 Clients

In this section several clients developed with different aims will be described to shown how the modules of the framework can be combined to fit specific architectural environments.

3.1 jOrca

![jOrca GUI](image.png)

**Fig. 2.** Service invocation with jOrca. The active tab shows the service parameter interface for the service ‘getBestHitsFromBlast’. Users can choose existing data from the file system or create new data as input. The lower part of the tab also shows the advanced options that allow an experienced user to manually select a specific mirror or request status/results for asynchronous services.

jOrca ([www.bitlab-es.com/jorca](http://www.bitlab-es.com/jorca)) is a powerful and portable desktop client, highly customizable to cover a broad range of user skills. The client provides several interesting features: access to several repositories with different protocols, searching for compatible tools based on user data or keywords, embedded file system for handling local user files and user-defined lists of favorite tools. (Figure 2)

This client uses the tool and mirror modules with their dependencies, and the execution module with a local access. The file system module is not used because the system must handle external data as well (presentations, documents etc.).
3.2 ACGT portal

Several portlets (web pages) have been developed for the ACGT portal to let registry administrators manage tool metadata. These portlets provide a graphical interface to the functionality of the framework, such as browsing and registering resources (data types, services/workflows etc). The modules used in the portlets are Tool, Mirror, DataType and Functional Category. In figure 3, available data types are displayed as a tree. The structure shows inheritance relations between the data types (see section 2.1). A short description of the data type is shown automatically to guide user selection. These portlets will provide functionality to invoke individual tools in future versions.

![Data Type Tree Browser]

**Fig. 3.** Browsing for datatypes. This tree form represents the hierarchical relations between data types.

Additionally, a workflow editor (see figure 4) has been implemented in the project. This editor enables users to combine services as pre-defined workflows where the output from one service is used as input to the next. This workflow editor accesses tool metadata through the tool, mirror, functional-category and data type modules and uses this metadata to dynamically build the user interface.
3.3 Magallanes

Magallanes [9] is a versatile and platform-independent Java library of algorithms that use the framework. It can be used to create search engines which assist in tool discovery. A discovery process aims to identify the set of services or data-types that satisfy a given number of constraints from the pool of all available tools/datatypes (e.g. what tools are available to process my sequence?). Typically, clients that exploit these repositories supply some discovering mechanism, most of them with syntactic orientation such as search for web services by name and also by function or input/output parameters. However, for end-users the syntactic discovering process is not efficient since it assume a prior knowledge on tool or data type names.

Magallanes offers a flexible “google-like” search engine based on “Levenshtein distance” [7] to improve search results. The framework allows Magallanes to connect with external clients in such a way that results can be not only for discovering but also to invoke web-services, recover a data-type description, etc. The functionality is also available as a desktop application (see figure 5).

One of the main advantages of Magallanes is the ability to configure the client to deal with metadata modifications in repositories (only parsers need to be adjusted), and its ability to work with new repositories (standardized by the framework).
4. Conclusions

This paper describes a flexible framework for the implementation of distributed computing systems. It provides functionality (separated by functionality in modules) for sharing and discovering tool metadata, data storage handling and tool invocation:

- The tool, tool instance, data type and functional category modules provide the functionality needed for tool discovery. Clients, developed using our framework, are able to seamlessly connect to different metadata repositories.
- The execution module provides functionality for tool invocation. We address protocol differences with specific software components (workers) for each protocol.

Secondly, in general, it is difficult to create simple clients that are usable by users with potentially different background knowledge. Beginners need simplified user interfaces while “advanced-users” need full access to details such as mirrors. We therefore suggest adaptable and customizable software frameworks for client developments. Clients that use these frameworks, such as jOrca, expose advanced options for expert users while supplying easy-to-use, simplified interfaces for beginners or casual users. However, client development is complicated and developers could potentially spend a lot of effort in details related to service communication protocols.

In section 3, we show examples of several clients to demonstrate how different components of our architecture have been combined to fit different requirements. By clearly separating the functionality from the presentation of the functionality (the user interface), client developers can focus on user interface issues and use ready-made components (modules) for repetitive tasks such as tool discovery based on
input/output data types and/or functional descriptions. This greatly simplifies client development.

We plan to refine the semantic descriptions of tools and data types to allow more complex tool discovery in future versions of the framework. This is necessary when the number of tools grows and users are presented with many possibilities. By storing metadata about earlier successful tool invocation (even by other users), clients can rank tools based on this information and only present the tools that most likely solve the task.

The data module will be also extended with a flexible and automatic data transformation system. This will make it possible to combine services expecting different data formats (in the case where the input data represents the same data from a conceptual point of view). The list of formats can be extended according to user needs.

In the user module, we are working on designing and implementing different access layers to provide user authentication (using passwords or certificates). This will allow control of metadata because metadata instances (for example metadata for a service) could be protected from unauthorized changes. Additionally, this user authentication could be used by both clients (using the execution module) and services to restrict service invocations to specific users and/or groups.

5. Acknowledgements

This work has been partially financed by the National Institute for Bioinformatics (www.inab.org) a platform of Genoma España and the EC project "Advancing Clinico-Genomic Trials on Cancer" (EU-Contract No.026996).

References

1. Advancing Clinical-Genomic Trials (ACGT, EU-project): http://www.eu-acgt.org/
8. myGrid project: http://www.mygrid.org.uk/
16. Wroe Chris, Stevens Robert, Goble Carole, Roberts Angus, Greenwood Mark; (2003), A suite of daml+owl ontologies to describe bioinformatics web services and data; International Journal of Cooperative Information Systems special issue on Bioinformatics, ISSN: 0218-8430