How to Implement Software Connectors? A Reusable, Abstract and Adaptable Connector

Selma Matougui and Antoine Beugnard

ENST-Bretagne, Technopôle Brest-Iroise,
CS 83 818, 29 238 Brest, France
{selma.matougui, antoine.beugnard}@enst-bretagne.fr

Abstract. In recent software developments, applications are made up of a collection of reusable software entities (components) and mechanisms that permit their interaction (connectors). These latter mechanisms have many forms. On the one hand, industrial approaches use simple connectors that are mainly point-to-point connections. On the other hand, academic approaches, like Architecture Description Languages (ADL), recognize complex connectors as first class design entities. However, these concepts are restricted to the architectural level since they have almost no implementation. The current application developments use simple connectors, and high level specifications are under exploited.
In this article, we propose a means to fill the gap between connector specification and implementation. For a better reuse of design effort, and to avoid using only simple connectors when realizing applications, we propose to define connectors as complex communication and coordination abstractions and to implement them as a family of generators. We illustrate the development and use of such generators through a full example.

1 Introduction

As software systems become more complex, applications are being constructed from reusable, interconnected software components and connectors. For a long time, components have been considered as the fundamental building blocks of software systems. Although there is still no universal definition of components, there is a kind of consensus about what a component is: it offers and requires services, and performs some computation. The interconnection mechanisms, or connectors, play a deterministic role in establishing the global system properties but their level of understandability is still far from the level reached by components, as argued in [1]. The goal of this paper is to propose a better way to implement abstract connectors.

In the remainder of this article, section 2 analyses the problem and depicts what is wrong with current understanding of interconnection mechanisms. Next, section 3 details our proposal where we define what a connector is and explain its life cycle. Then, section 4 illustrates the way to implement and use such connectors with a load balancing connector. After that, we make relationships with other work in section 5, and finally we conclude in section 6.

2 Motivation

Industrial component approaches like CCM [2] and EJB [3], do not refer the notion of connector. Interaction mechanisms are low level. They refer to a limited set of simple communication abstractions among which we can cite the synchronous communication embodied as Remote Procedure Calls (RPC). These interaction mechanisms are the most used in current application developments and several implementations exist like CORBA, RMI, and SOAP. Their main functionality is to transport data and synchronize the interacting components. They are not able to make any decisions for coordinating components. The lack of abstraction implies realizing a complex communication with a sophisticated combination of simple interactions included in the functional components. Developers are constrained to construct and construct again the complex communications over these platforms and applications, anytime they need to use them, in spite of the fact they reuse them. The effort of realizing a complex communication is not well exploited. For a better quality of software and separation of responsibilities, however, we would like to be able to include in connectors more communication and coordination functionalities than only those of transporting data [4,5].

In academic approaches like Architecture Description Languages (ADL [6]), the interaction mechanisms have been recognized as first-class design entities in software architectures and are defined as connectors [7]. We identify 3 kinds of ADL that express connectors differently. The first type defines a set of connectors implemented as simple ones, like Unicon [8]. They almost meet the interaction mechanisms of industrial approaches. The second type defines connectors that model complex interactions between components like Wright [9]. Unfortunately, these connectors are reduced to specification descriptions and have no implementation. These specifications are under-utilized and current developments always refer to simple connectors. The third type defines complex connectors, like Rapide [10], but provide a fixed set of connectors types designed as components that have explicit interfaces. Hence, to interact with these connectors, the functional components explicitly call the services offered by the connector. If the connector were changed for one reason or another, the component would use the new interfaces of the new connector. Thus, we often need to use adaptors between the components and the connectors if their interfaces do not meet. These various definitions of connectors indicate how poorly they are understood. We are far from a consensus about what a connector is.

Hence, there is not a standard way to represent connectors. When comparing industrial and academic approaches we notice that industry uses simple connectors because they are implemented and available. Unfortunately, complex communications are always constructed (and constructed again) over existing platforms and are mixed with components. This makes applications hard to maintain and limits component and connector reuse and evolution. In academic approaches, some ADL define complex communication abstractions, but they do not offer a standard way to design and implement connectors. They either specify complex connectors without offering means to implement them, specify

and implement simple ones [6], or implement them as components, but this last solution lacks adaptability. Almost all languages claim to make interactions as connectors but without meaning the same thing all the time.

Regarding this discussion, we support the need to model and specify complex interactions as autonomous entities in software architecture just as many other works have done. Nevertheless, the question to ask is how to implement these connectors in order to offer and preserve the abstraction and to ensure an automated adaptation. Existing approaches offer ad hoc and predefined answers. In this article, we propose a global vision of what is a connector and propose to implement it as a family of generators in order to ensure both abstraction (of connectors) and adaptation (to components). In this way, we offer a better and easy configuration or reconfiguration of applications, component and connector reuse, and a better exploitation of design effort.

3 Connectors: Definition and Life Cycle

A connector is a reification of an interaction, a communication or a coordination system. At an abstract level, it exists to serve the communication and coordination needs of unspecified interacting components. Later in the life cycle, it describes all the interactions between special components by relying on the own interface specifications of these components. It also offers application-independent interaction mechanisms like security and reliability, for instance. Three points distinguish a connector from a component:

- It is an autonomous entity at design level but neither deployable nor compilable (it does not exist in conventional programming languages). The connector is an architectural entity. It must be connected in order to be deployed.
- It does not specify offered or required services but patterns of services. Hence, the interfaces are abstract and generic. They become more concrete later in the life cycle, on assembly with the components. This is done by adopting the interfaces type of the interacting components.
- In order to distinguish services grouped in interfaces of component ports, the word *plug* is introduced to name the description of connector ends. A plug is typed with a name and can have a multiplicity.

We distinguish four stages in the life cycle of a complex connector, summarized in table 1 and illustrated in figures 1 and 2. The connector evolves in time and, for each stage, we give it a different name and a particular graphical representation with abstract parts (dotted lines) and concrete parts (solid lines). Connectors are represented by an ellipse, in order to differentiate them from components, and plugs are represented by circles[1] around the ellipse.

[1] In opposition with components representation as boxes-and-lines [11], where components are boxes and their defined interfaces are squares. We illustrate our complex connectors by an ellipse in order to differentiate them from the simple ones that are represented by lines.

Level	Architecture (abstract)	Implementation (concrete)
Off-the-shelf	*Connector*	*Generators*
Assembled	*Connection*	*Binding components*

Table 1. Vocabulary by level of abstraction and refinement

In its most abstract state, figure 1 (a), this entity is called a *connector*. The sole concrete parts that are defined are the properties that have to be ensured and the number of interacting components able to be connected (number of different plugs). They are in fact the sole invariants that exist during the whole life cycle. The abstract parts to be specified later are the protocols of the underlying platforms on which the connector will be implemented, and the interfaces of the interacting components (both in gray in figure 1 (a)).

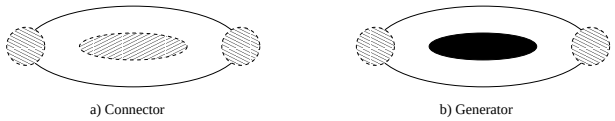

Fig. 1. The connector alone : (a) abstract, (b) implemented

An isolated connector is concretized as a family of off-the-shelf *generators*. These generators are in charge of the implementation of the connector abstraction over different communication protocols or platforms. The plugs are still abstract, the interacting components are not specified yet. When activating the generators, the plugs are destined to be transformed into proxies towards which the effective communication will take place. Figure 1 (b) shows the transformation of the internal part of the connector into a concrete entity.

At an abstract level, it is possible to use the connector in a software architecture. The connector is linked to other components, and form the *connection*. At this stage, the generic interfaces of the connector (plugs) become concrete and adopt the applicative interfaces (APIs) of the interacting components. In figure 2 (a), the plugs (circles) are concrete because their type is specified, they form the connection interfaces. Hence, we connect the components without worrying about the underlying platform.

The most concrete form of the connector is represented in figure 2 (b). Both the protocol and the actual interfaces are known. The generator, that was developed for the chosen protocol, can use these actual connection interfaces. It generates the proxies that represent the realization of a communication or coor-

Fig. 2. The connector assembled : (a) abstract, (b) implemented

dination property for the connected components' requirements (APIs) over the underlying system. We refer to this entity as the *binding component*[2].

From an application developer's point of view, the software architecture definition involves selecting components and connectors off-the-shelf, assembling (*i.e.* establishing connections), and generating the application onto the target system with the appropriate generators.

From a connector developer's point of view, defining a connector involves the specification of the abstract communication properties (load-balancing, for instance), and the development of the generators that use the interface definition of connected entities (IDL, for instance) over a target system (TCP/IP, for instance). As we can see, a family of generators could be developed for a single abstract connector.

An example of an existing generator, as defined in this section, is the CORBA generator as an instance of the RPC connector. The properties it ensures are distribution, language interoperability, and a RPC semantics. It can rely on several communication protocols like TCP/IP or other low level protocols. The proxies it generates are the stub and the skeleton.

Hence, we use this definition and life cycle of what a connector is to implement or realize complex communication abstractions. Indeed, to ensure properties like authentification, data compression, consensus, cryptography, or load balancing, current developments mix them with the application code. By dedicating one connector to realizing such non functional properties thanks to the generation process, we offer the possibility to provide separation of concerns transparently to the application and independently of the platforms.

In the following we will demonstrate the whole process of a connector specification, implementation and use through a load-balancing connector.

4 Load Balancing Connector

In this section we illustrate, throughout an example, the whole connector life cycle and the benefits of using our approach. We begin by presenting briefly the main features of the chosen complex interaction property that the connector holds: load balancing. Then, we detail the description of the connector and the different steps it goes through. Finally, we show an implementation of the load

[2] It is a refinement of the *binding object* defined in Open Distributed Processing (ODP) [12].

balancing connector. Actually, the aim of this section is to show the feasibility of our proposal and the advantages of such a software engineering approach.

4.1 Load Balancing Connector Features

Some applications, like e-commerce systems and online stock trading systems, concurrently service many clients transmitting a large, often bursty, number of requests. In order to improve the overall system responsiveness, an effective way to deal with this great demand is to increase the number of servers — replicas offering the same service — and to apply a load balancing technique. Load balancing mechanisms distribute client workload equitably among back-end servers by indicating which server will execute each client's request. These mechanisms may follow either a *non-adaptive* or an *adaptive* policy. Non-adaptive policies do not take the servers' workload into account. The round robin is such a policy. Adaptive policies, however, take the servers' workload into consideration. The policy of the least loaded server is an example. Many load metrics can be used; the CPU's idle cycles or the number of requests, for instance. In the following example, we assume a distributed application where a client needs a defined service offered by a server. As a client needs to send a huge number of requests, these requests would be balanced among n replicas of servers following a special load balancing policy.

Performing load balancing at an OS [13] or a network [14] level does not allow to use application metrics or control the load balancing policy without modifying the application itself. Hence, we propose to achieve load balancing mechanisms at a high level (eg. application level) but *transparently to the clients' and the servers' codes*. We reify connection abstractions as an independent entity. For instance, we consider load balancing as a complex interaction that involves many participants and is achieved with different protocols and policies. This abstraction is the *connector*. The property it ensures is load balancing and its number of plugs is two. Indeed, the connector distributes incoming requests of one kind of component (clients) to one kind of another component (servers). These plugs are of multiplicity n because the connector is able to make interacting several (same) clients with several (same) servers. The service offered and required is the same both side, but is unknown *a priori*. This service is not known when the load balancing connector is developed.

4.2 Load Balancing Connector Life Cycle Description

The load balancing connector is designed and implemented as set of code *generators*. A generator is in charge of producing the code that allows to ensure the load balancing mechanisms on a target platform or system. The different generators differ according to different criteria. They can differ in relation to target platform or policy differences. For example, if we are in a distributed environment, the generators could be built over different available platforms that allow remote communications like CORBA or RMI. Differentiating the generators from a policy point of view gives birth to as many generator implementations as existing

policies. By combining these criteria, we obtain several generator variants for one connector. Hence, we can have a load balancing generator with the round robin policy on the CORBA platform or on the RMI platform. These off-the-shelf generators are intended for ensuring load balancing mechanisms on different platforms for initially unspecified components. So, at this stage, the plugs are still generic, they wait to be associated with components' interface descriptions that are used by generators for the effective code generation.

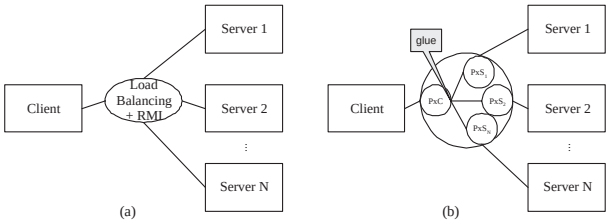

Fig. 3. Complex communication with load balancing connector. (a) connection, (b) binding component

The load balancing connector is assembled with the components of the application: the client and the servers. Figure 3 (a) represents the result of this assembly: *the connection*. Henceforth, connector plugs know who is interacting, so they take form and adopt the interacting components interfaces. The plug at client's side will embody the services offered by the servers. The client deals with the plug as if there was only *one* server (as a *shared* proxy). The plug at the server's side will embody the services needed from *one* client. The server is not concerned by managing the different clients that require the service. This process of transforming plugs into complementary interfaces of the interacting components maintains the original functionality of the application; that is the client needs a service from the server. Therefore, the load balancing abstraction is ensured without taking into consideration what is offered or not by a target platform. At this stage, the connection interfaces are specified. They can be provided for one or several generators that are responsible for resolving platform details. These connection interfaces differ from one application to another, as the component services they have adopted differ.

Once the connection established, the resulting connection interfaces are given as parameters to a chosen off-the-shelf generator[3] that activates the generation process of *the binding component*. Both the plugs and the underlying platform are now well known. The generation process can transform the connection interfaces, mapped on the generator plugs, into dedicated proxies. Figure 3 (b) highlights these generated proxies that represent the realization of the load balancing property for the client and the server interfaces over the chosen under-

[3] The generator choice can be done to meet the components' platform for example.

lying platform, for example RMI. If the selected generator is built over RMI, the generated proxies include both load balancing and distribution mechanisms. Therefore, the proxy (PxC), at client side, is in charge of converting data and deciding which server will serve the client's requests according to the policy. The proxies (PxS_i), at server side, are in charge of converting data, managing client requests, and reporting server workload in the case of an adaptive policy. For one connection we obtain different binding components by applying different generators.

The load balancing connector represents a new communication abstraction. It is realized as a family of generators. Each generator is designed and implemented, once for all, for generating complex and repetitive details over a platform. The load balancing connector is assembled each time with components that need to balance load. These components may use different services. As connector's plugs are generic, they fit to any interacting component interfaces. Once the generators selected and the plugs associated with its missing interfaces, the binding component can be generated.

This approach ensures both the separation of concerns and the separation of the abstraction and the implementation. The communication and coordination concerns are separated from the functional concerns of the applications. The client and the servers have no extra code; they are entirely dedicated to their functional properties. Hence, this approach offers the means to discharge the application from cumbersome code. Applications become more understandable, easier to maintain and to make evolve. The "know-how" of the abstraction implementation is built up into generators that can be reused any time a new connection takes place. So, it is clearer to application developers. They no longer need to worry about the implementation of connectors. This allows them to change or add a policy without affecting the application, and even use the client without any load balancing connector. They are not concerned with planning the use of load balancing when designing components.

This plug/connector approach is different from the classical role/connector approach. In the latter, the client has to play the role defined by the connector. This implies, most of the time, that the client is adapted to the interfaces of the connection. However, in the plug/connector approach, the client does not need to add extra code to be adapted to the needs of the interaction, it is the plug that adopts the clients needs. The client required service is provided to the client improved by the load balancing and distribution details, that are superfluous for the application.

4.3 Load Balancing Connector Implementation

We have specified the load balancing connector and realized the associated generators. In order to evaluate the whole connector life cycle, we have implemented the load balancing connector in the open source project Jonathan [15]. This framework basically offers a CORBA and RMI-like generators for the RPC connector. We have developed generators to realize the load balancing connector with 2 variant strategies: round-robin and least loaded sever over Jeremie, the

RMI personality of Jonathan. In the following, we briefly explain the implementations of the load balancing generators and their use. All the details can be found in[16].

The first implementation is for the round robin policy. For this non-adaptive strategy, monitoring the servers' workload is not necessary. We have realized the generator as an extension of the JRMICompiler, the standard RMI generator of Jeremie. This new generator, called JRMICompilerLB, has been implemented to ensure both load balancing and distributed properties. It generates the classical proxies that hold the code for converting data to be sent through the network. In addition, JRMICompilerLB augments these proxies with the code of the round robin policy. The standard JRMIRegistry of Jeremie, that holds the different remote server references, has also been modified to JRMIRegistryLB.

In our client/server application, the client uses the following interface which is implemented by the N servers put at the client's disposal to balance load.

```
public interface ApplicationInterface{

    public int operation(); // the server's service

}
```

At assembly time, when an architect assembles the off-the-shelf components and the load balancing connector, the abstract plugs of the load balancing connector adopt this ApplicationInterface in order to apply the semantics of the communication to it. Hence the connection interfaces become concrete. Since these connection interfaces are known, they are provided as parameters to the generator in order to generate the appropriate proxies. The following command line is used:

```
> JRMICompilerLB ApplicationInterface
```

Once the proxies generated, they are deployed with their attached components. So when the client calls the server's operations, the proxy intercepts them locally. The proxy always performs its initial functionality of converting data, but in addition, has the responsibility for coordinating the back-end servers by applying the round robin policy. It forwards every request to the next replica in the group of servers registered in the JRMIRegistryLB. This is done by replacing the current remote reference of the (PxC) proxy (see figure 3) at each request by the delivered reference from the JRMIRegistry. Hence, at run time, the client sends a request locally to the sole proxy in its possession; the proxy embodies the server's service but does not perform it; it forwards the request to the selected server according to the appropriate policy.

In the second implemented load balancing connector, the generator generates code to ensure the the least loaded server policy. In this adaptive policy, clients' requests are sent to the servers according to their load. The policy has been implemented in a specific publish/subscribe component. The generated proxies (PxS_i), at the server side, publish the load they are in charge of monitoring. The

servers are not aware of this load monitoring. The publish/subscribe component chooses the least loaded server that will serve the request. All subscribing clients' proxies are informed with this new reference and use it when a service request is made. We choose to implement this generator as an extension of JRMICompilerLB genetrator with the option *-lbAdaptive*. The publish/subscribe component, the proxies and the JRMIRegistryLB, therefore, make up the binding component.

We have evaluated performances of the new generator with the round robin policy over a network of 32 machines with 1 to 8 servers and 1 to 22 clients[4]. The results obtained from a latency test showed no overhead due to reference switchings and demonstrated a good scalability. For example, the time needed to serve the 22 clients with one server using the load balancing connector was 1.85 ms, and the time needed to serve the same 22 clients with 8 servers was 0.75 ms. All the results can be consulted in [16].

It is worth noticing that Jonathan's previous generator can still be used to implement point-to-point RMI connections. So, introducing load balancing, or changing the load balancing strategy, requires a simple proxy regeneration from the same interface description. This approach helps to easily reconfigure the application and change the policies only by changing the connector and regenerating the proxies.

5 Related Work

There are very few works related to the generation of connector implementation.

In [17], Garlan and Spitznagel define operators to create new complex connectors by composing simple ones. As we have seen in section 4.3, in our approach we can build complex connectors from simple ones (extension of RMI). We can also build them from a combination of a component with a generator (using a publish subscribe component [16]). Their approach is simple to implement, because they reuse existing connectors without any modification. However, this involves several intermediate steps because of the combination of several connectors. Thanks to the generation process, our approach enables us to change some interaction mechanisms that are not appropriate to the interaction requirement. Moreover, Garlan and Spitznagel's implementation passes directly from the specification to the binding component step through the connection—according to our life cycle. Their approach does not have off-the-shelf generators for different technologies, so they have to make the same transformations whenever a composition has to be created over a different platform. We argue that the two approaches are complementary. Owing to the fact that our approach is more complex to realize, we believe it to be more efficient, since the transformations they propose could be automated with the development of generators.

The work on ArchJava [18] also uses a kind of generator for connectors. It is more focussed on implementing a connector on a Java/ArchJava platform and

[4] One machine was reserved for a registry and another one for controlling the benchmark.

uses dedicated classes to implement the connectors. These specific classes could be seen as our generators. Our work is an attempt to offer a better conceptualization frame in order to generalize the use and implementation of connectors.

6 Conclusion

Software architecture is playing a more and more important role in software engineering. It is well recognized that architecture is of crucial importance in all nonfunctional properties of software applications. We have shown in this paper that, despite their importance, some key concepts such as connectors are still ill-defined. We have proposed a new vocabulary in order to help software architects and designers to better conceptualize architectures.

Connectors have adaptable implicit interfaces that provide communication properties transparently to the application. For instance, changing a consensus connector for a load-balancing one requires a new proxy generation that does not require any modification at the application level but does have a strong impact on the nonfunctional properties of the whole system. From a software development life cycle, we have introduced a clear vocabulary in order to distinguish all the connector stages. A *connector* is an abstract software object whose intention is to implement a communication property. This abstraction may have several implementations, called *generators*, depending on design choices and on the target platform. When abstractly used in an architecture in order to link components, we call it a *connection*. Finally, the deployed and executable stage is called a *binding component*.

We believe that this conceptualization suggests to developers to design new kinds of generators, as we have illustrated with a load balancing connector, instead of always reusing the same "old-good-one". We believe that accumulating connector development experience through developing generators is the right way to use and reuse connectors. It ensures abstraction through the generation process, and adaptability thanks to plugs.

Finally, we believe there are two ways of producing reusable software parts: usual off-the-shelf components, that can be *directly* integrated in a software architecture, and off-the-shelf generators, that are used *indirectly* to generate the adaptable binding components that link application components together.

References

1. Mehta, N.R., Medvidovic, N., Phadke, S.: Towards a taxonomy of software connectors. In: the 22nd International Conference on Software Engineering (ICSE 2000). (2000) 178–187
2. OMG: CORBA Component Model RFP (1997) `http://www.omg.org/docs/ orbos/97-05-22.pdf`.
3. Inc, S.M.: (Enterprise java beans technology) `http://java.sun.com/products/ejb/`.

4. Shaw, M.: Procedure calls are the assembly language of software interconnection: Connectors deserve first-class status. Technical Report CS-94-107, Carnegie Mellon University, School of Computer Science (1994)
5. Ducasse, S., Richner, T.: Executable connectors: Towards reusable design elements. In Verlag, S., ed.: ESEC/FSE'97 (European Software Engineering Conference). Volume 1301 of LNCS (Lectures Notes in Computer Science). (1997) 483 – 500
6. Medvidovic, N., Taylor, R.N.: A classification and comparison framework for software architecture description languges. In: IEEE Transaction on Software Engineering. (2000) 70–93
7. Show, M.: Procedure calls are the assembly language of system interconnection: Connectors deserve first-class status. In: Workshop on Studies of Software Design. (1993)
8. Shaw, M., DeLine, R., Zelesnik, G.: Abstractions and implementations for architectural connections. In: Third Int'l Conf. Configurable Distributed Systems. (1996)
9. Allen, R., Garlan, D.: A formal basis for architectural connection. ACM Trans. Software Eng. and Methodology **6** (1997) 213–249
10. Luckham, D., Kenney, J., Augustin, L., Vera, J., Bryan, D., Mann, W.: Specification and analysis of system architecture using rapide. IEEE Trans. Software Eng. **21** (1995) 336–355
11. Abowd, G.D., Allen, R., Garlan, D.: Using style to understand descriptions of software architectures. ACM Software Engineering Notes **18** (1993) 9–20
12. Putman, J.: Architecting with RM-ODP. Prentice Hall (2001)
13. Coulouris, G., Dollimore, J., Kindberg, T.: Distributed systems: Concepts and design (2001)
14. Inc., C.S.: High availability web services (2000) `http://www.cisco.com/warp/public/cc/so/neso/ibso/ibm/s390/mnibm_wp.htm`.
15. Middleware, O.O.S.: (Jonathan: an Open Distributed Objects Platform) `http://jonathan.objectweb.org/`.
16. Sanchez, F.J.I., Matougui, S., Beugnard, A.: Conception et implémentation de connecteurs logiciels : Une expérimentation pour le calcul de performance sur une application de répartition de charge. Technical report, ENST-Bretagne, Brest, France (2004)
17. Spitznagel, B., Garlan, D.: A compositional approach for constructing connectors. In: The Working IEEE/IFIP Conference on Software Architecture (WICSA'01). (2001) 148–157
18. Aldrich, J., Sazawal, V., Chambers, C., Notkin, D.: Language support for connector abstractions. In: European Conference on Object-Oriented Programming (ECOOP '03), Darmstadt, Germany (2003)