On Distributed Systems Design Mechanisms

6-Aug-99

Alexandre Sztajnberg
(alexszt@ime.uerj.br)
IME/UERJ and GTA/COPPE/UFRJ

Orlando Loques
(loques@caa.uff.br)
CAA/UFF

Endereço Postal

Abstract

Open distributed systems we are going to conceive in the future will present challenging design and operating requirements demands. Integrating dynamically autonomous distributed modules, allowing module reuse, coping with specific user requirements, such as QoS, are going to be common tasks a designer will have to face. In this context, it is convenient to make available suitable models to design such systems and provide flexibility to handle dynamic requirements.

In this paper, we investigate some techniques that are being proposed to facilitate the construction of future distributed systems. In addition, we aim to provide some guidelines that would help a designer to select those techniques best suited for a given application. Initially, a set of relevant requirements regarding distributed systems design is established. With this set in mind, some of the classical and emerging software engineering technologies are selected and investigated in their fitness to provide design solutions to large distributed systems. For each technology, the appropriate mechanisms are analyzed regarding their capability to attend the basic requirements. Finally, we provide some With the results of this investigation we present a consistent model towards efficient design of large distributed systems that tries to benefit from the best parts of all presented technologies.

1. Introduction

There is evidence that present open distributed systems need support for dynamic changes in their structure in order to adapt to continuously evolving operational and functional demands [Lad 97]. Many applications, such as video-conference, process control, and workflow management, have complex functional nature and strong non-functional and operational requirements. These requirements can be related to the management of run-time environment resources that can be modified dinamically by the application user or by external agents, such as other systems sharing or managing these resources. Clearly, there is a run-time software updating problem.

Those systems will have to support applications that will demand flexible and efficient design techniques to cope with basic requirements inherent to the application and robust schemes to cope with the underneath environment requirements.

The key question regarding this scenario is what cast of features would be desirable to build those (possibly large and complex) distributed systems ? An immediate approach to the answer is stating what are the essential means by which we would like to conceive those systems. The first step in our investigation is to establish a consistent suit of requirements that can guide distributed systems design and address the following issues:

- use of autonomous modules developed by independents entities, possibly reaching them trough a ubiquitous high-speed network (the Internet);
- reuse or recycle basic modules, code (and embedded expertise) to build our own application;

- add, transform, adapt or up-grade system behavior dynamically affecting only partially the application, without having to stop the whole system;
- adapt systems dynamically to attend user requirements on expected quality of the services provided by the system (QoS) and to react in face of an altered or evolved environment (failure situation, hardware upgrade, technology upgrade).

The second step in this investigation is to select candidate technologies that can contribute to support this requirements, at least partially, or playing important roles in system design. Object-based, computational reflection, configuration and other selected prospective candidates technologies are presented and analyzed in this work regarding their potential in addressing large open distributed systems design issues. This analysis will allow us to deeply comprehend until what extension each technology can be used in developing applications and how, if possible, can we combine the power of each of them to turn more efficient this task and how they address the relationed design requirements. Comparing and reasoning about the presented designing techniques we will propose a set of guidelines to improve and ease large distributed systems construction and associate this guidelines to available tools or frameworks.

The first part of this paper states a minimum set of desired criterias towards efficient design of large open distributed systems. In the second part, section 3, based on this criterias, prospective technologies and mechanisms such as object-orientation, computational reflection, configuration and aspect oriented programming are presented and analyzed. The next part, section 4, is intended to reason about the use of these concepts in the development of distributed applications and with this results, relate and compare each mechanism. The sequel builds a series of guidelines on how and where to apply the established parameters on the design of large distributed applications. In the last sections the concepts are consolidated and concluding remarks are presented.

2. Basic Requirements

As stated in the introduction, we selected a set of requirements that are relevant for distributed system design and construction. In this section they are named and described in order to provide a context for discussions related with current proposals in the area. In the following sections, we analyse how the concepts and mechanisms embeded in these proposals try to match the requirements. It is interesting to note that the requirements are of different nature and may not be directly related. However, we think that they together provide a sound basis for engineering distributed systems.

Reusability. A good design approach should allow parts of former conceived systems to be reused in order to minimize development efforts. In an extreme case an entire system could be created in a leapfrog style with parts of previously existing systems. In a restricted context, reuse can be achieved within a programming language, were data types and procedures already coded are available. In the context of this paper reuse is considered regarding distributed systems. We want to be able to construct systems composing them with parts previously conceived that may be interconnected trough a potentially heterogeneous network. If this is possible, great amount of time is saved and the efforts can be concentrated on the architecture of the system and not in the designing of basic parts. Two other concepts are closely associated to reuse: modularity and interoperability. Designing modular components concerns two aspects: autonomy, were behavior is concisely and completely defined and interface standardization, were interface functionality is clearly defined. Interoperability is essential to support distributed applications composed by modules, offered by independent providers, that must communicate and cooperate [Weg 96]. These modules may reside in different sites, and can use different languages, protocols and data formats, requiring special care for interoperation. Preparing a component to be modular and interoperable is half way towards achieving reuse. These concepts forms the base of the component-based programming technology [Bro 96].

Separation of concerns. Separation of concerns is a paradigm largely accepted as a form to cope with large systems conception [#referencias# Agh96A]. When designing a large system, the functional, non-functional and operational specifications can be done separately. Aspects related specifically to the conception of base-functions of a system can be treated separately from problems concerning interactions with the underneath environment aor other non-functional needs of that system. This approach allows designers to firstly concentrate efforts on the designing of the functional modules and in a next phase search solutions regarding the adaptation of the system to the specific requirements posed by the environment over which the system is going to run. Clearly, if modularity and interoperability are considered during the process, functional and non-functional domain components can be potentially reused.

Dynamic Evolution. In the scope of this work, dynamic evolution is a requirement that can support inclusion of new components or modifying existing components in distributed systems that cannot be completely stopped for those changes. Dynamic evolution can address new functional or operational demands while the system is operating. For instance, fault-tolerance, software up-grading, the introduction of new QoS parameters, communication styles or time restrictions are activities that may be require changes at run-time. etc. On the application level, the designer may have to be obliged to plan for this, although in some cases some degree of transparency can be achieved. The run-time environment must also support mechanisms to allow the inclusion of modules and their linkage with the previously existing modules.

Dynamic changes also depend on the granularity and context where the change is performed. In some cases it is interesting to have access to the internal structure of parts of the system, for example, to extend its functionality or take decisions based on some internal state. This will have strong implication on the conception and implementation of the support mechanisms required (compiler and run-time levels).

Dynamic changes can introduce non trivial problems regarding security issues as some kind of access to internal structure of the systems and components may be provided. However, in some situations this is the price to pay for it.

Coverage. For coverage requirement we mean that a given technology has to be widely available and accepted as a means to design and program systems. A facet of the availability is ease of use and dissemination. A good design technology should be easy to use and become rapidly disseminated in order to create critical mass of experts and to evolve. Another aspect that can be put in this same requirement is the ability of a given technology to solve efficiently a large number of problems, and can be called expressiveness. Nevertheless, as we observed through the elaboration of this paper there is no universal solution regarding this requirement. No technology is yet comprising enough to address all system development requirements. Actually, as we will discuss, this requirement will only be addressed with an adequate combination of technologies.

With this set of requirements in mind, in the next section a selection of the most prospective technologies will be laid-out and for each of them verified if these basic requirements are addressed.

3. Selected Technologies

To fulfill the desired requirements in the process of building distributed applications, designers should have available appropriate tools, techniques and models. In the following is presented a set of selected technologies that provide mechanisms with which distributed systems can be design and that can addresses, at least partially, the desired requirements. Some of these technologies are basic concepts but are associated to available tools that implement them, others are emerging paradigms or recently studied techniques.

3.1 Objects

Object technology is based in two main concepts: encapsulation and polymorphism [Rum 91 and Pou 94]. Encapsulation is used by objects to assure that it's internal structure, composed by data types variables and procedures (*methods*) to manipulate them, is only accessed through a well defined interface. Polymorphism allows a method to switch the actual object that is going to handle it according to the signature of the parameters that are passed in invocation time. Encapsulation and polymorphism concepts provide an useful mechanism to design modular components and are of interest in our work.

There are two general approaches to implement object technology programming languages. Class-based programming languages precludes that an object's implementation is defined by its class. A class is a lifeless template of behavior and interface description that is used to create objects of that kind. An object is a live instance of a class, occupying memory space and having an associated context. The other approach is object-based or prototype-based programming languages. In prototype-based languages the concept of class is not strong. Every element is said to be an object and objects are not instances of a class. There are interesting conceptual differences between the two approaches [Tai 96] but we are concentrating in the effects of this difference on extension mechanisms associated to object-technology. For our purposes we are using class-based languages in most of the examples.

Object-oriented languages are usually associated to mechanisms that address extension. Three important extension mechanisms can be found in object-oriented languages implementations and are going to be inspected in the next subsections: inheritance, composition and delegation.

3.1.1 Inheritance

With inheritance mechanism a new class can be defined using and sharing as base structure another class definition and adding new functionality or modifying (overriding) some original parts of the former class. Inheritance principle is mostly related to object-class interface and behavior extensions. Depending on the point of view, the inherited class is called the *parent* or *superclass*. One way to see inheritance is the specialization of a more generic parent class. In that sense components based on classes of a hierarchy of classes can be used to compose an application. There are some variations on the semantic associated to inheritance. We can summarize this in the following:

- reuse previous definition of the structure of a class with no strict relation to what this class does (interface inheritance);
- extend the behavior of a class creating a specialized class based on, and altering some public parts (overriding) of the former (nonstrict inheritance);
- extend the behavior of a class creating a specialized class strictly based on the former class interface and adding other features, but not altering the original ones (strict inheritance or specialization);
- create a specialized class based on two or more superclasses, inheriting interface and behavior of all former classes (multiple inheritance);
- creating a generic class (superclass) encapsulating two or more classes and their relationship (aggregation);
- creating a generic class based on the union of a set of partitioned subclasses, generally the superclass is an abstract class, were there is no implementation provided, just the interface, and therefor cannot be instantiated (generalization).

3.1.2 Composition and Concatenation

The distinction among the terms composition and concatenation is in some way blurred. All the two terms are used to name similar extension mechanism styles. Composition enables the creation of new classes with more complete features by sharing code of simpler former classes in a *black-box* style. In a composition the structure of two or more classes are copied and combined to generate another class. Unlike (multiple-)inheritance, composition can be implemented without any necessity of breaking the encapsulation of objects: the component classes interface and associated implementation are concatenated and new features can be added but former interfaces are not modified.

3.1.3 Forwarding and Delegation

Delegation is a special extension mechanism featuring in some object-oriented languages that allows an object handling a request to involve another object, a delegate object, in the operation of that request. However, due to a self-reference pointing the operation is done in the context of the original receiver object.

In [Pou 94] delegation and inheritance are considered distinct concepts where one can emulate the other. In [Tai 96] delegation is considered a mechanism by which inheritance can be implemented. The common consideration here is that inheritance and delegation have the same expressive power.

The forwarding mechanism is used when an object does not know how or does not want to handle an arriving request and thus relays this request to another object (a parent or another object in a chain). The object that is able to handle the forwarded request, does it in its own context.

Delegation can be achieved in languages which do not implement delegation natively by using forward operations an passing the forwarder object-reference as part of the request to be manipulated by the receiving object.

Addressed Requirements

One of the major promised advantages of object-oriented technology is its ability to provide (possibly) consistent mechanisms to allow incremental modification and extensions to well tested and debugged code. This mechanisms should allow reuse, combination and reorganization of elements (objects, classes or even entire applications) in behalf of the design of new systems. Among the mechanisms presented that provide reuse features, the most accepted is inheritance with all is possible semantics, but even though there is not still universal consensus around the support of basic features embedded in the compilers to implement it, nor regarding the complete semantic associated. Besides, the use of inheritance or the other mechanisms are not synonym of reuse. The planning of objects to compose an application must be careful to allow reuse and easy interaction with other objects. There is not yet unanimity concerning concepts around these mechanisms and different object-oriented languages provides normally only one of them.

When distributed systems are considered other factors as reference location and reference of remote objects, solved by the compiler in centralized applications, becomes important. Another problem that has to be faced is the lack of standardization among different object-oriented languages regarding implementation of inheritance mechanisms. This last problem is even worst if implementation of classes and super-classes reside in different sites. As we shall se in the last section, solutions for those kind of problems are being addressed by standard environment supporting distributed objects like OMG's CORBA. Delegation, by its turn, is a more adequate mechanism for distributed environments given the message-based nature of that mechanism [Pou 94].

Regarding dynamic evolution, distributed systems based on object technology environment should support separate compilation (in space and time) of their components [Gue 96] and New classes should be freely added to a system at run time supported by late and dynamic binding. If this kind of flexibility is not provided, the possible alternative is to stop the running system, recompile new code and start the system again. This alternative is quite static and frequently, source code of the modules to be extended must be available.

As mentioned in [Tai 96] two distincts models for inheritance are generally implemented in object-oriented languages: class-based and prototype-based. The short review of inheritance mechanism presented until this point is class-based. Prototype-based inheritance uses existing objects in a running application from which to inherit interface, behavior and actual context. Dynamic extension based on inheritance and composition mechanisms are not easily implemented in distributed environments. If a class-based object-oriented language is used, dynamic inheritance cannot be achieved, mostly because static type checking, systematically used in most object-oriented compilers, cannot be easily done in a distributed environment and relations between objects are defined in compile time. In case of use of a prototype-based object-oriented language dynamic inheritance can be achieved given that the new object will be conceived by cloning a running object and the problems arised form the lack of type checking are already solved by the implementations of languages of this nature. Inclusion of dynamic inheritance is being strongly considered in object-oriented languages to provide dynamic extensibility.

The semantic and mechanisms of inheritance are closely related to the object-oriented language being used and for that reason code reuse can be limited as there is no standardization in how to implement them (in [Tai 96] a comprehensive survey on inheritance mechanism presents more details). A possible solution is to limit inheritance possibilities to interface-inheritance as suggested in CORBA. In the next sections we are going to present some possibility to flexibilize inheritance mechanism at the programming level and better modularize overall conception of complex systems, although basic modules construction can still be done using an object-oriented language or using an object-based systematic for programming.

3.2 Computational Reflection

3.2.1 Basic Concepts

The idea behind computational reflection is to allow a system to execute processing on it’s own behalf, to modify or adjust its own behavior A module that uses reflection is composed by a base component and a reflex component. The reflex component has an interface to the base component and works as a filter, exposing internal informations of the base component or intercepting calls that should go directly to the basic component, altering or adjusting their characteristics. An original invocation already pre-treated by the reflex component can (or not) be forwarded to the original destination. (figure 3.1). The reflex component can be conceived to have access to the internal environment that hosts the modules, allowing adjustments to the base-module reactions triggered by low level context events.

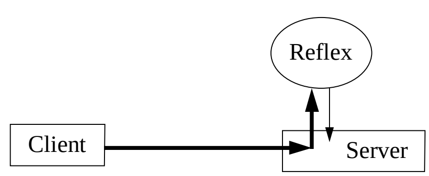

Figure 3.1 - Reflection as code interposition

Computational reflection concepts are being strongly considered and implemented in programming language level, mainly in object-oriented languages. Some experimental implementations have been concentrated on special extensions of compilers for object-oriented languages, such as C++[Chi 95 and Gow 96], Smalltalk[], CLOS[Kic 97] and Java [Kle 96]. The combination of object technology features mentioned in section 3.1 and controlled flexibility provided by computational reflection is going to be exploited in this subsection.

3.2.2 Reflective Object-Oriented Programming Languages

Reflection technology widens object technology expressiveness in two basic aspects: allowing more flexible extension mechanisms and enforcing separation of concerns design. This discussion if furthered in section 4. In object-oriented languages that provide reflection features, a class, now called a base-class, can be associated to a meta-class. The language syntax has to provide a way to establish connection between base-class and meta-class. This connection is normally expressed by a reification construct, to indicate that a method invocation messages directed to this base-class is going to be treated by it’s respective meta-class and to turn relevant informations about the base-class available to the meta-class, and a reflection construction that effectively intercepts the invocation and handles it in the context of the meta-class. The rules and syntax do describe the connection between base-classes and meta-classes are called a Meta-Object Protocol (MOP). In that context, reflection is said to be computational or behavioral [Lis 95] as it actuates in the behavior or computations of a base-object.

Instantiation of a base-object is accompanied of the instantiation of the corresponding meta-object. All base-objects are treated in a base level that is concerned with the external computation of the application [Lis 95] and meta-objects are treated in a meta-level that operates the internal computation of the application. Similarly as described on the preceding section, meta-objects are responsible for modify, adjust or filter base-object’s reaction to arriven messages or that are sent from these in response to a external request.

Comments

In large distributed systems, a base-object that offers generic services, can have those services specialized or differentiated through the use of adequate meta-objects. Important features to distributed systems as safeness, real-time or fault tolerance [Fab 95, Lis 95, Fra 97], for example (figure 3.4) can be added by reflection. In that way, we can say that reflection flexibilizes reuse and extensions possibilities of native inheritance mechanisms on object-oriented languages.

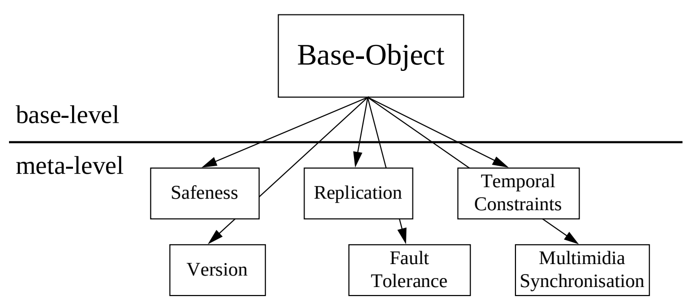

Figure 3.2 - Using reflection to add non-functional characteristics

Some operating systems research include reflection as a mechanism to allow extensions and adjustment of their functions [Mit 97, Chi 97 and Gow 95]. Microkernel based systems [Str 93], orient basic system services to execute in user level and for that reason have to provide efficient reflection mechanisms. More basic mechanisms as named-pipes and streams are frequently used to allow efficient implementation of reflection modules and the construction of filters in kernel level, mainly in the case of communication protocols [Jon 93]. Distributed systems benefit from these features as specific behavior regarding the operation environment can be customized to an application's need by reflection.

Potential advantages of the use of reflection through meta-objects are limited by compiler technology. Implementation of meta-object concepts in object-oriented languages like C++ does not seam to be trivial. An example of implementation is Open C++ [Chi 95], a C++ pre-processor that offers reflexive characteristics. It is advocated that reflection concepts are employed for the generation of code more easily, before compilation. In this way, the base-object and the meta-object would be linked in the same executable module, having no need of a run-time system supporting meta-level.

A different possibility to reflective languages is to use MOP to provide extensibility to semantic behavior of the own programming language and syntax constructions, opening its implementation to addition of new possibilities [Kic 97]. This feature can turn the use of computation reflection more easy if a simpler syntax is built for programming base and meta object association. Other implementations of reflective object-oriented languages proposes an approach that use an executive run-time support that manages base-object and meta-object instantiation and enforces coherence of systems that are created based on those elements [Cam 96, Zim 96]. Other approaches as [Gow 96] provides a slightly dynamic use of reflection triggering reification (whether the base-object will use reflection or not) by an expression result.

Nevertheless, if we consider that in distributed systems there is a strong need for flexibility in application operation, the solution of a system compiled as a unique module cannot attend the majority of the situations. Even so one could conceive a meta-object library with implementation of the most popular or performance critical extensions [Loq 96] as meta-objects and use them when needed.

Addressed Requirements

Among the basic requirements regarding distributed systems design, reflection better provides separation of concerns. The distinction between base-level and meta-level perfectly addresses separation of concerns requirement. Functional and non-functional issues can be planned and treated in different domains, respectively in base-level and meta-level domains.

As object technology, reflection is also not a direct instrument to achieve modularity or reuse. This will be a concern for the application programmer. Nevertheless, reuse requirement can be fulfilled if meta-objects are carefully conceived to be used in diverse situations and stored into a library of meta-objects.

Dynamic aspects of reflection are the most challenging issue. By one side, with computational reflection, important aspects for distributed systems, as already related, can be added by reflection of base-objects, given that the programming language in use has the adequate features to do so and adequate meta-objects are available. By another side, dynamic selection, dynamic intention to use or dynamic evolution of systems based on base-objects and meta-objects are difficult to address. Most MOPs have to be programmed in a static style, were classes or objects to be reflected and their respective meta-objects are selected in the code and then compiled. After that nothing more can be done. Environments contemplating dynamic selection of meta-objects are not yet largely available [Gow 96].

Benefits offered by computational reflection are still not evident for two main reasons: programming meta-objects is not easy and reflective programming languages are not widely available nor disseminated. Reflection can be used more easily by an application programmer if a meta-object library is available, but a skilled meta-level programmer has to build the library. Besides, computational reflection depends on the use of a specialized programming languages.

3.3. Configuration

3.3.1 Basics

The basic principle of configuration-based design supposes the existence of available software components which will be used for construction of systems or applications with particular characteristics. This particularity is achieved by customizing and interconnecting those components. In that way, definition of the software components and application are aspects of distinct domains. When establishing a configuration, it is possible to select specific attributes, among a set of available options for each component and connection [Sha 95 e Mag 96].

Configuration can be handled in tow ways: statically, when configuration descriptions are used to generate an application that, once configured, will run with operating system's support, or dynamically, when it will run over an executive support environment (as described in [Szt 95]) that allows interactive configuration commands to be launched form an external controlling agent or initiated by any active module. In the scope of this paper, configuration environments are assumed to have this agent, external to the application, that can positively interfere in the present structure of the application, adding new modules, removing existent modules, altering the connection of modules or modifying module or connection behavior while the application is operational.

The configuration of an application has to be supported by some tool that allows the designer to express the structure of selected modules and their interconnection regarding a specific application. Traditionally expressing a configuration is undertaken by a language. Depending on the intended abstraction level the terms: configuration programming languages (CPL), architectural description language (ADL) [Bis 96], module interconnection languages (MIL) or Composition Language (CL) [Iss 96] can be found in the literature. Generically this languages have to translate configuration scripts, expressing the desired application structure, with their related level of detail and complexity, in primitive constructions or commands of the targeted execution environment that will materialize the application. The utilization of graphic tools [Wer 95] as interfaces to configuration languages permits that the systems can be described in a visual and intuitive form. Configuration languages have interesting characteristics like:

- allow hierarchic construction of components (composition);
- facilitate the reuse of modules and systems that have been composed and debugged at some time;
- allow generation of more low level and efficient code automatically (for example RPC and CORBA IDL).

A widely known mechanism conceived to allow module connection is generically called **connector** [Bis 96]. A connector is an abstraction used on configuration level to provide the programmer an uniform interface to the native communication mechanisms of the operating systems, with a simple and standardized approach, with out the necessity of care with specific details. The modules are designed in such a way that the interfaces through which they can exchange informations with other modules is completely defined by connectors. With a configuration command, connectors of one or more modules can be interconnected. A connector can be associated to the interface signature of a component. By using the correct combination of components, any style of application design can be mapped into a configured or composed scheme: peer-to-peer, bus based, hierarchical, event-based, etc. [Gar 94 and All 94]. Figure 5.2 illustrates that mechanism.

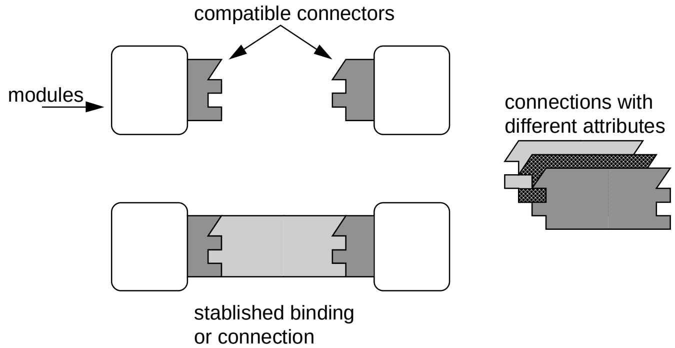

Figure 5.2 - Connection of two modules

One of the advantages of explicitating the connection between two modules is the clarity experimented during the programming of functional and non-functional requirements, that can be done separately [Bis 96 e Sha 95]. In figure 5.2 this is schematized by the possibility of selecting one of the different available attributes for the connections. Besides, many other characteristics can be associated to the action of binding a connector pair:

- data types and structures that are supported by the connectors (also called a signature) can be statically verified when configuring a connection, avoiding that connectors of different types, that would transport different data (or messages) can be interconnected. Modules with matching connector signature can be automatically interconnected. Nevertheless, signature incompatibility can be overcome with module bridging [Weg 96] were a specialized module handles the task of translation.

- semantic and protocol used for communication through that connection can be dynamically administered in a more flexible way if that communication is easily identified. For example, in the case of a connection of modules that reside in the same node, the communication can, in a more detailed level, be done with sockets, pipes, streams or shared memory. In the case that the communication of those modules are distributed over the Internet, a protocol like TCP, UDP or RTP can be selected to support that connection. The execution environment should allow the selection of the mechanism to be used and should suggest the best option.

In opposition to the advantages of the connector concept, one of the known challenges to realize an environment with connector schema is the difficulty to implement it with well defined structures and boundaries inside a module's code [Bis 96]. Traditional implementation of connectors inside the modules can be presented in a diffuse form in the code. This problem has been surrounded with code skeleton generation by the own configuration languages or by integrating communication mechnisms into programming languages.

Comments

Distributed environments supporting externally commanded configuration can facilitate the design of complex distributed applications. The designer can address functional requirements of the application more easily (re)using basic modules previously developed and debugged and respect operational requirements by selecting specific interconnections modes for each of them, including communication protocols. The result is depicted in figure 5.1. Among the operational requirements eventually necessary we can cast out fault tolerance, that can be selectively configured for each module [Loq 97], applications with modules that have time synchronization needs and with strong demands of quality of service (QoS) contracted with the underneath support and communication subsystems, typically multimedia applications [Bla 96, Geo 96 e Flo 96].

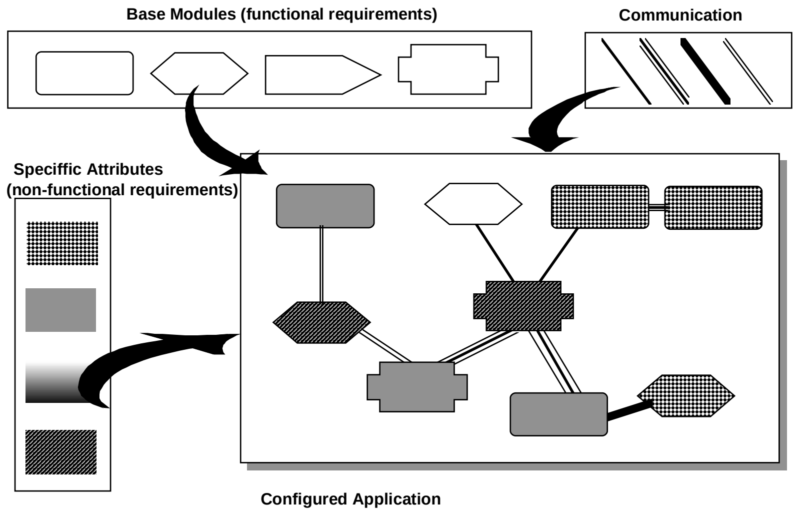

Figure 5.1 - support for distributed applications configuration

There are two basic approaches to combine object and configuration technology. The main problem to be solved is how to map connector abstractions in a class or object. An immediate solution is to map each method present in the class interface into a connector. The principal task of the external configuration agent is to bind every caller invocation instruction to a method regarding a callee object, resolving the reference address that was unknown to that caller before the binding. A more elegant approach is to consider connectors as first-class objects. In this solution the base-object must have a standard interface to the connector-object. When two objects have to be linked, the configurator instantiates and attaches a connector-object to each involved object and then binds the two connector-objects. The flexibility of this approach is very attractive. The object-connectors instantiated by the configurator can be selected by the user and customized regarding the desired communication style. Another advantage of this approach is that connector-objects can be prepared to translate different patterns of communication allowing objects from different environments to connect through a bridge.

Addressed Requirements

Configuration technology satisfies almost all basic requirements in some way. Reuse of existing modules allows is achieved when building application by composition and combination of this existing modules. Modularity is essential to configuration technology and is achieved by conventioning a set of standard connectors by a combination of which, modules can communicate; when generating code skeleton or by enforcement of the environment that supports configuration. Extension mechanism is provided by allowing complex or specialized modules to be built and customized by composition of more primitive modules.

Separation of concerns is also contemplated in configuration. The correct selection of basic modules that will be used to implement functional characteristics, and customizable behavior and interconnection styles that will define the non-functional characteristics of an application, is the manner configuration technology provides separation of concerns. In another level, besides describing interconnection styles and specific behaviors of modules, a designer can use specialized modules that implements non-functional aspects of an application such as translation bridges to compatibilze interfaces or to add features that cannot be done otherwise.

In a configuration environment, dynamic evolution is provided with the possibility to aggregate more modules or reconfigure existing ones with no stop of the whole system. Reconfiguration primitives and adequate support from that environment enforces consistence of the new configuration built dynamically. It can be observed that it is not uncommon that configuration can, besides being triggered by an external agent (a configuration manager module), be initiated by a module that is participating of the application [Kra XX].

One draw back of configuration, despite of all advantages, is the availability factor. Until now, there is no universal accepted standardization or a most used environment or configuration language. This problem is being overcomed with the growing adoption of OMG's CORBA as a modular environment to develop and support distributed objects. Deploy configuration mechanism over CORBA is feasible and benefits form all embedded advantages of object technology and OMG's standardization effort.

4. Relating the Concepts

Now that the essential concepts of the selected technologies have been presented and we known how each of them addresses the set of basic requirements, we go a step further in this analysis and relate those technologies. First some detailed reasoning and combination of technologies are going to be discussed. Following, a comparison on how they address the set of basic requirements closes this subsection. With these informations a good measure of suitableness of each technologies are achieved and the set of guidelines on open distributed systems design, goal of this paper, can be derived.

4.1 Combining Technologies

The use of reflection as an extension mechanism is associated to a specialized programming language, normally object-oriented, that embeds meta-objects features through a MOP. Moreover, there will be a time waste for the programmers to learn how to use and work with meta-level concepts. One way to convey with this *inconvenience* is to provide a set of meta-objects that can be selected by the programmer as commented early.

A full fledged configurable system, requires a configuration environment with external configuration agents and a configuration description tool. Modules have to be programmed following the interface model standardized by the configuration scheme. The later does not represent a problem as we can have automatic skeleton code generation from CLs or graphic tools and connector libraries.

When choosing between reflection and configuration, a trade off between efficiency and flexibility should be searched. If a reflective object-oriented language is used we can lose modularity because the base-object and associated meta-object will be compiled in a sole object. It would be desirable to have at our disposal an object repository with their respective meta-objects and select them when necessary. Proper run-time support and dynamic binding technique can be used to provide more convenient facilities. A standard interface associated to the base-object would allow that meta-objects were selected and instantiated only at the execution moment, as figure 4.1 illustrates. Some efforts are canalized to provide this characteristic in reflexive programming languages compilers, but they are not yet a complete reality, for this would demand that reflection must be associated to the configuration approach. This last point has special interest in our investigation.

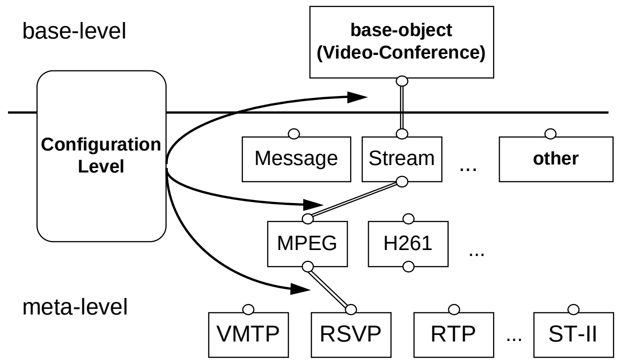

Figure 4.1 - Configurable multiprotocol environment based on reflection

Is must be observed that on programming languages with reflective features, simple mechanisms that allow dynamic selection of the meta-object to be used are not widely available. So, in the example of figure 4.1, the compression technique to be used has to be selected in programming time by indicating an association between the base-object and the selected meta-object. As an alternative, reflection concepts can be used by adopting, on the system description, the configuration technique. The relation of a base-object and meta-object can be implemented connecting two modules that implements respectively the behavior of the base-object and meta-object with the additional advantage that this can be done dynamically.

In the RIO Environment [Szt 95] a joint approach of configuration and computational reflection was used to provide flexibility in communication subsystem to support dynamic handling of specific protocol features. In this environment a meta-level MM (Multiprotocol Manager) module has the mission to carry out particular interactions between application modules and communication subsystem, enabling the correct operation of the selected protocol. MM modules with different semantics can be configured and exchanged to address specific comunication requirement of modules composing an application. Similar approach was used in [Pan 93] with Actor-based system and in [Ber 96] with Composition- Filters abstraction.

An utilization example of reflection and meta-objects configuration would be a multimedia application, like video conference, were a series of parameters indicating QoS requirements determine the adequate selection of meta-objects to be interposed in a meta-object hierarchy. In figure 4.1 the base-object, video conference application, uses a meta-object that implements a coded continuous stream mechanism to realize the communication, that by its turn uses RSVP protocol. The knowledge of the base-object, nevertheless, is limited to a simple generic communication.

The advantages of computational reflection to the application programmer reside in the transparency of the new functionalities aggregated by meta-objects and in the syntactical comfort. In the following code expert [Chi 95] it can be observed that in the program without reflection, the *Node* object must inherit characteristics form *PersistentObjetct* class to be a persistent object, while in the program using reflection, persistency is implemented as an extension of C++ language.

```
class Node: public PersistentObject
{
   public:
      Node *next;
      doublevalue;
};
```

```
persistent class Node
{
   public:
      Node *next;
      doublevalue;
};
```

4.2 Comparing Technologies

Reuse. Regarding reuse requirement, the selected technologies have similar approaches. New components can be built with pre-conceived components. Object technology can relay on libraries that offer useful classes or objects and specialized objects can be created by inheritance, delegation or aggregation over these former classes or objects. Reuse can be maximized with the adoption of design patterns technique [Gam 94] were entire object-based solutions can be used. Reflection technologies goes a step further and allows base-objects and meta-objects reuse. Various applications can benefit of available meta-object library to implement elements regarding non-functional requirements faster. Another possibility is to reuse the entire application base-level modules and switch the meta-level components, if possible, providing an entirely new behavior to the original system. Configurable environments allows reuse when primitive modules are used to compose complex modules. Regardless of the possibility of reuse, in all technologies, object-based and reflective systems have to be conceived with the intention to be reused. Configuration approach is more reuse-driven as construction, as standardization of module interface and interaction styles is a meter of survival in this technology. Thus, modules conceived with configuration technology are inherently reusable**.**

Separation of concerns. Separation of concerns is addressed by each technology in different ways. Object technology does not provide any mechanism to address separation of concerns explicitly. Even though object technology can be used to implement applications were separation of concerns is observed. Computational reflection by its turn is an enabling technology that precludes this requirement as a basic mechanism. Reflective programming languages allows a clear distinction between base-level and meta-level programming. Each of these levels are concerned with different aspects of the application. Configuration addresses separation of concerns allowing basic-modules design, specific module behavior and interconnections styles to be planned separately. Non-functional and operational requirements are deployed externally in configuration domain. To achieve equivalent characteristic, reflection technology depends on the run-time environmet.

Dynamic Evolution. As discussed in the first two sections, dynamic evolution requirement englobe two aspects: the need of extensibility to let systems evolve from former existing systems and the need for these extensions happen dynamically. The last issue is not trivial to be implemented and it is rather found in specific environments

Extension is achieved in object technology by using inheritance and similar mechanisms. Base-objects can be extended to specialize or generalize its behavior. With reflective languages meta-objects can provide extension of base-objects, modifying or adding features to the original base-object behavior. Inheritance and reflection have similar objectives regarding extension but different approaches: inheritance allows creation of new classes and reflection allows transformation of classes (in [Jag 96] inheritance is modeled by reflection and some formal reasoning about their expressiveness is presented). Extension approach of configuration is more clean: modules or systems can be extended by adding new modules and reconfiguring a target architecture. Different behavior can be imposed to selected modules changing non-functional characteristics or connections of these modules with the additional benefit that this can be done dynamically.

Dynamic evolution considering pure object-oriented environment is somewhat difficult, however feasible if a running time executive allows new classes and inheritance to be defined on-the-fly. The first step to accomplish that is to have an environment that supports late and dynamic binding of objects, and maintains consistence of references. The same problems can be identified in reflective technologies. In most of the available MOPs meta-objects are defined statically. Which classes or objects are going to be reflected and which parts are going to be reified are defined in the code. Extending or adding meta-objects normally is achieved by modifying the original code. If this points could be addressed dynamically, systems built with object and computational reflection would be allowed to evolve dynamically. Once again, configurable systems have a more clean solution compared with the other technologies. Configurable systems can evolve dynamically by reconfiguration actions, when diverse characteristics of an application can be modified. Environments supporting external and dynamic configuration have mechanisms to enforce consistent reconfiguration of modules, assuring that interactions with other modules are gracefully frizzed during reconfiguration leading to a *smooth* evolution.

Universal comprising. Universal comprising is the requirement least fulfilled, mostly because the lack of uniformity and standards. While reuse is normally achieved in all analyzed technologies, different implementations of the same technology cannot interoperate easily. This is also true regarding different technologies. For instance, object-oriented languages are largely available but some issues regarding internal implementation of embedded mechanisms as inheritance are still opened. Thus, regarding distributed systems the creation of a new class derived from a parent class defined from different object-oriented programming languages, were parent class and derived class can reside in distinct sites is difficult to achieve. Programming languages or execution environments with reflection features are also still not largely available and there is no standard features universally accepted. Configuration technology have positive characteristics: basic modules can be written with any programming language, the technology is already mature, comprouvedly useful and largely available. Even so, configuration environments suffer from the same problem as computational reflection: there is still no universally accepted implementation of configuration environments nor bridging standards to let them interoperate. As stated in the section were configuration technology was mentioned, CORBA can bring a step further the solution for standardization regarding most technologies analyzed, aggregating them around it's the core distributed object architecture.

Table 4.1 summarizes the differences and overlapping characteristics of the mentioned technologies. In most cases there is a clear distinction on why and when apply each of them.

Model	Objects	Reflection	Configuration
separation of concerns	not addressed directly by object technology	base-object and meta-object separation	basic-modules, configurable behavior and interconnections styles are planned separately.
reuse	object library can be used specialized objects can be created by inheritance, delegation or composition. Reuse can be maximized with the use of design patterns.	base-objects can be reused meta-objects can be reused in various applications if available from a meta-object library..	primitive modules are used to compose complex modules. Architectures are constructed based on existing modules. requires standardization of interface and interaction styles
dynamic evolution	extension can be achieved using inheritance, delegation or composition, but dynamic extension is hard to achieve dynamic evolution is hard to achieve but supporting late and dynamic binding can help	extension is provided through meta-objects in most of the available MOPs meta-objects are define statically. dynamic extension of base-objects must be supported by an specialized environment	achieved with on-the-fly reconfiguration capability of simple or composed modules
universal comprising	object-oriented languages are largely available no standardization for extension mechanisms	programming languages or execution environments with reflection features are still no largely available there is no standard features universally accepted	ADLs and configuration environments are available. There is no uniformity or standard or specific implementation universally accepted.

Table 4.1: comparison of the technologies

Comparing the available mechanisms for the construction of an application exclusively with objects, reflection or configuration technologies we can verify that they have equivalent expressive power and the programming effort is equivalent. This fact was comprouved in [Loq 97] in fault tolerance area comparing reflection and configuration. Modules, built with object-based technology have to be programmed in some language and then combined to produce an application. This combination can be done using reflection or configuration, depending on the factors pointed out in later sections. What differs the two techniques is that, in the case of reflection, application structure is described explicitly through a specialized programming language, while that same structure is described externally, through an ADL, independent of the programming language, in the case of using configuration approach.

5. Building the Set

It is not our intention to state that a designer has to use all the selected technologies in this work to be well succeeded, but it would be very comfortable to have them available and be allowed to choose the best mechanism to solve the problem in hand. Considering the strong parts of each technology we can view a chain of decisions regarding what mechanisms will be adopted to implement an application as following:

1. In a basic level of programming the use of object-based technology or object-oriented language should be selected. This language should provide the adequate mechanisms to build basic modules. Regarding more complex modules, an adequate set of design patterns should be selected.
2. Computational reflection cam be used in the programming of basic components where a fine adjust of non-functional characteristics is necessary, in a very particular level of detail. Functional extensions and specific environmental-related behavior should also be added by reflection. A flexible reflective language should be preferred. It would be even better if adoption of reflection or selection of meta-objects could be done during execution dynamically or through configuration commands. Nevertheless, if a MOP is not available, configuration approach can be used;
3. Configuration technology can be used in many levels of granularity. In a higher level configuration can be used to build large distributed systems based on previously composed modules. In an other level this same technology can be used to compose complex modules in a first moment and then structure the application.

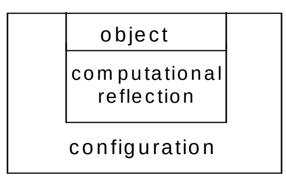

Figure 5.1: Proposed Model

Figure 5.1 depicts the guidelines in to a schematic model and table 5.1 consolidates this statements, and complements table 4.1.

Model	Objects	Reflection	Configuration
where to apply	basic module conception	module extension	complex module conception system architecture construction
Examples	C++, Smaltalk, CLOS and Java	Open C++, CLOS, Metajava	RIO[Mal 93], Write[], Regis[], Uni-Com[], OMG-STRUDL [OMG 97]

Table 5.1: where to apply each technology

The presented guidelines considered the present level of maturity, evolution and standardization of the mechanisms analyzed. There is still a large work to be done before it can be delineated a consistent and synergetic set of tools regarding every point mentioned in the last sections.

5.1 CORBA

Many works conducted in distinct areas regarding distributed systems design consider OMG's CORBA [OMG 96A] as a valid infrastructure (some of them are referred in the next section). Comparing the results of these works and the guidelines stated in our work, we argue that OMG's CORBA has potentially almost all the features well suited to design large distributed systems:

- it allows distributed programming by it's own nature and has adequate support for distributed systems (called common facilities);
- it is object based, what is a promise of encapsulation and reuse and OMG-IDL can map interface descriptions in programming languages as C, C++, Smaltalk and ADA;
- provides adequate semantics for distribution of objects and imposes adequate semantic restrictions when necessary;
- meta-level features were proposed recently in papers as [Sin 97] and officially by OMG in [OMG 97B]. An extension of the ORB interface allows the application meta-objects to control some parts of the core behavior. If reflection can be used with the base components of CORBA, the ORB can be flexible extended.;
- compositional features were also proposed recently in [OMG 97]. IDL was extended to incorporate configuration commands resulting on a Structural Definition Language (OMG-STRUDL). One of the issues less addressed in OMG's compositional proposal is the availability of external run-time configuration. However academic research teams are working on that [Loq 97 and Szt 97] and this feature could be included as a vertical facility;
- it has a growing set of general frameworks (called Common Vertical Facilities) by which well accepted domain of applications can be rapidly built;
- software bridges are available to integrate with other architectures as OSF's DCE and Microsoft's COM and OMGs effort on standardization is concerned to map CORBA to other standards as OSF-ODP, ANSA and TINA;
- it provides interoperation protocols as with IIOP *Internet Inter-ORB Protocol*, an ORB gateway, allowing the use of CORBA technology over large networks as the Internet;
- there is a proposed framework on object migration addressing availability of resources, fault tolerance or deploying autonomous agents paradigm;
- extensions in communication styles [Her 96 and Maf 95] accepting group communication and connection oriented streams are being studied. There is a RFP (Request for Proposal) opened by the OMG regarding this issue. With this extensions applications with video and audio media can be designed over CORBA and other QoS requirements can be addressed following those experiences.

6. Related Works

All presented mechanisms an problems raised in this work are being recently studied. However, there are no many researches attempting to extract the good parts of each of them and combine them in a coherent form. As far as we are concerned the aspects mentioned in this works are being studied isolatedly. Nevertheless [Weg 96] present a very comprehensive panel of what kind of evolution are being expected towards a consensus in large distributed systems design. [Loq 97, Cra 97, Mag 97, Pry 97 and Iss 96] present alternatives in using configuration with distributed objects over an ORB. [Ber 96] approach combines objects, composition and limited reflection but the main feature regards messages exchanged between objects. [Gow 95], [Mit 97], [Sin 97], [Fab 95] and [Fra 97] present solutions to extend application and system functionality using computational reflection and exploit flexibility aspects of using meta-objects by providing the needed flexibility when coding meta-objects.

Our work complements other researches in that we are identifying really useful concepts and mechanisms of every possible technology in providing suitable means to design large distributed systems. In this phase we are concentrated in achieve a design model derived form de adequate combination of these tools. Specificaly, we are driving our efforts towards the use of configuration technology as a main enabler of large distributed systems design.

7. Concluding Remarks

All presented technologies have their merits and address some of the requirements stated in section 2, but until this point we are not convinced that any one solves all problems as required. In [Agh 96A, Gue 96 and Weg 96] it is mentioned that today's available object technology does not provide solution for every type of software problemas and other techniques must be used. The adequate combination of configuration and other technologies, leading to the model depicted in figure 5.1, showed to be the correct answer to the questions in the first section regarding the design of large open distributed systems.

With object technology, inheritance, delegation and fine-grained composition are mechanisms to be used in the programming level of basic objects. Due to its nature, object technology and embedded mechanisms, by their selves, cannot be used to conceive large structured systems. Object technology can be viewed as a building block technology.

Beyond objects, the two other technologies selected for distributed systems design were evaluated: configuration and reflection (this one based on a specialized programming languages). Configuration, on the architectural level, and reflection, on the language level, can be used for the construction of complex systems. The use of reflection in combination with object-oriented language showed to be promising in the construction of complete modules with different operational requirements, while the advantages of the use of this technique in the design of large distributed applications, if compared with the advantages obtained with configuration techniques, are not evident. Reflection offers flexibility on basic component construction, but does not allow external interference to adjust the application's behavior. This adjust can be predicted, however, during the application programming. Reflection solution is not much dynamic regarding operational requirements. Configuration, on the other hand, obliges that modules be programmed using connectors for message interchange, but permits the modules to be interconnected under demand and communication be adjusted before and during application's execution. Operational requirements fulfillment with configuration is simple and dynamic.

AOP technology was not involved in the evaluation and comparison sections for it is a very new paradigm. There is not yet a formal structure associated with this technology and no critical mass of work done. Nevertheless [Kic 96] and [Lop 97] presents and evaluates distributed system design following AOP paradigm and this technology should be seriously considered. In AOP separation of concerns and reuse requirements are strongly supported but regarding the other requirements nothing concrete can be said.

AOP's precursor team [Kic 96] claim that extremely complex systems cannot be built using traditional style of modularity. AOP enables designers to reason about natural aspects of a system that are captured separately, possibly with different representation, and then interconnect them automatically, generating a final resulting system.

Aspectual decomposition of a problem is very different of the traditional module decomposition. When the programmer is using AOP he should express implementation strategy at an appropriate abstract level through an appropriate aspectual view and should not be concerned with implementation details.

Considering distributed systems, one could separate the whole problem in basic-functions, location, communication and synchronization aspects. Then, each of this aspects would be captured but, very important, a relation among the aspects has to be expressed in a precise way. A special tool (called Aspect Weaver by the authors of the AOP) is responsible to combine and interrelate all the aspects into a executable form. The generated executable can be entangled and efficient, depending upon the smartness of the tool.

An important point in AOP is expressing separation of aspects with coordination allowing the weaver to work properly (and to be conceived at all). Available implementations of AOP environments are exploiting the concepts of base-level and meta-level separation extending the ideas of reflection. Different interfaces can be used to provide basic-functionality ("what to do") related to the base-level and to monitor or adjust the basic-functionality ("how to do it") related to the meta-level.

Other important requirements not considered directly in this work, but also relevant, are **formalization** and **efficiency**. Technologies with formal background are more likely to be accepted. Configuration technology associated to connector abstraction, for example, is known to be well formalized. Connector abstraction facilitate formal properties verification as deadlock absence, corretude and termination [Mor 95] and operational properties (as for example, the coherence and viability of QoS parameters required by each module in a multimedia system, that can drive the network admission mechanisms). Efficiency by its turn is a default requirement for every issue related to distributed systems. Therefore, it is expected that the mentioned technologies provide adequate performance.

As a last remark, we are furthering these investigations analyzing the possibility of implementation of the proposed design model over a platform that reunites CORBA and Java. Even though Java does not presents strong communication characteristics, it is portable and can provide reflective features [Way 96 e Wu 97], that can be useful in the programming of basic objects. By another sight, WWW browsers are incorporating Java interpreters and the IIOP, that permits the use o CORBA and Java on an integrated fashion. Besides there are already available some CORBA implementations with IDL mapping to Java, turning this environment attractive for open distributed systems implementation.

References

[Agh 93] Agha, G., Frolund, S. et *alli*, "Abstraction and Modularity Mechanisms for Concurrent Computing", IEEE Parallel and Distributed Technology, Systems and Applications, IEEE Computer Society, May, 1993.

[Agh 96A] Agha, G., "Linguistic Paradigms for Programming Complex Distributed Systems", ACM Computing Surveys, Vol. 28, No 2, pp. 295-296, June, 1996.

[All 94] Allen, R., Garlan, D., "Beyond Definition/Use: Architectural Interconnection", ACM SIGPLAN Notices, Volume 29, No 8, IDL Workshop, pp. 35-44, August, 1994.

[Ber 96] Bergmans, M. J. L., "The Compositional-Filters Object",

[Bis 96] Bishop, J. and Faria, R, "Connectors in Configuration Programming Languages: are They Necessary?", IEEE Third International Conference on Configurable

Distributed Systems, Annapolis, Maryland, May, 1996.

[Bla 96] Blakowski, G. and Steinmetz, R., “A Media Synchronization Survey: Reference Model, Specification, and Case Studies, IEEE JSAC, Vol 14, No 1, pp. 5-35, January, 1996.

[Bro 96] Brown, A. W., and Wallnau, K. C., “Engineering of Component-Based Systems”, Component-Based Software Engineering: Selected Papers from the Software Engineering Institute, IEEE Press, 1996.

[Cam 96] Campo, M., and Price, N.H., “Meta-Object Manager: A framework for Customizable Meta-Objetc Support for Smaltalk 80”, I Simpósio Brasileiro de Linguagens de Programação, Belo Horizonte, MG, September, 1996.

[Chi 95] Chiba, S., “A Metaobject Protocol for C++”, OOPSLA 95, 10th Conference on Objetc-Oriented Programming Systems, Languages and Applications, Austin, Texas, EUA, October, 1995.

[Chi 97] Chiba, S. et alli, “Weak Protection for Reflective Operating Systems”, ECOOP’97 Workshop on Reflective Real-time Object-Oriented Programming and Systems, Jyvaskyla, Finland, June, 1997.

[Cra 97] Crane, S. and Dulay, N., “A Configurable Protocol Architecture for CORBA Environments”, International Symposium for Autonomous Decentralized Systems”, Berlin, Germany, April, 1997.

[Fab 95] Fabre, J.C.; Perennou, T. e Blain, L., “Meta-Object Protocols for Implementing Reliable and Secure Distributed Applications”, LAAS REPORT 95037, CNRS, February, 1995.

[Flo 96] Florissi, P. GS., e Yemini, Y. “Management of Application Quality of Service”, Distributed Comuting and Communications Lab, Computer Science Dep. Columbia University, NY, 1996.

[Fra 96] Fraga, J., Farines, J., Furtado, O., Siqueira, F., “Programação de Aplicações Distribuídas Tempo-Real em Sistemas Abertos, XXIII Seminário Integrado de Software e Hardware, Recife, PE, August, 1996.

[Fra 97] Fraga, J., Maziero, C., Lung, L.C., e Loques, O. G., “Implementing replicated services in open systems using a reflective approach”, Third International Symposium on Autonomous Descentralized Systems, Berlin, Germany, April, 1997.

[Gam 94] Gamma, E. et. *alli*, “Design Patterns - Elements of Reusable Object-Oriented Software”, Addison-Wesley Publishing Co., USA, 1995.

[Gar 94] Garlan, D., Allen, R., Ockerbloom, J, “Exploiting Style in Architectural Design Environments”, SIGSOFT 94, pp. 175-188, New Orleans LA, USA, December, 1994.

[Geo 96] Georganas, N. D., Steinmetz, R., Nakagawa, T., “Synchronization Issues in Multimedia Communcations”, Guest Editorial, IEEE JSAC, Vol 14, Nº 1, pp. 1-4, January, 1996.

[Gol 97] Golm, M. and Kleinöder, J., “Implementing Real-Time Actors with MetaJava”, ECOOP’97 Workshop on Reflective Real-time Object-Oriented Programming and Systems, Jyvaskyla, Finland, June, 1997.

[Gow 95] Gowing, B. and Cahill, V., “Making Meta-Object Protocols Practical for Operating Systems”, Forth International workshop on Object Orientation in Operating Systems, pp. 52-55, 1995.

[Gow 96] Gowing, B. and Cahill, V. “Meta-Object Protocols for C++: The Iguana Approach”

[Gue 96] Guerraqui, R. et *alli*, “Strategic Directions in Object-Oriented Programming”, ACM Computing Surveys, Vol. 28, No 4, pp. 691-700, December, 1996.

[Her 96] Herbert, A., “CORBA Extensions for Real-Time and Multimedia”, APM.1311.02, ANSA Phase III, 12 pages, 1996.

[Iss 96] Issarny, V. and Bidan, C., “Aster: A Corba-based Software Interconnection System Supporting Distributed System Customization”, International Conference on Configurable Distributed Systems, Annapolis, USA, May, 1996.

[Jag] Jagannathan, S. and Agha, G., “A Reflective Model of Inheritance”,

[Jon 93] Jones, M. B., “Interposition Agents: Transparently Interposing User Code at the System Interface”, ACM 14th ACM Simposium on Operating Systems Principles, Asheville, EUA, 1993.

[Kic 96] Kiczales, G. and Irwing, J. et *alli*, “Aspect-Oriented Programming”, Position Paper for the ACM Workshop on Strategic Directions in Computing Research, MIT, June, 1996. (http://www.parc.xerox.com/spl/projects/aop)

[Kic 97] Kiczales, G., Rivières, J. and Bobrow, D.G., “The Art of the Metaobject Protocol”, MIT Press, 1991.

[Kim 95] Kim, W.Y. and Agha, G., “Efficient Support of Location Transparency in Concurrent Object-Oriented Programming Languages”, Supercomputing’95, San Diego, USA, December, 1995.

[Kle 96] Kleinöder, J. and Golm, M., “MetaJava: An Efficient Run-Time Meta Architecture for Java”, International Workshop on Object Orientation in Operating Systems, IEEE, Seattle, Washington, October, 1996.

[Lad 97] Laddaga, R. and Veitch, J., “Dynamic Object Technology”, Communications of the ACM, Vol. 40, No. 5, May, 1997.

[Lee 95] Lee, A. H. and Zachary, J. L., “Reflections on Metaprogramming”, IEEE Transactions on Software Engineering, Vol. 21, No. 11, November, 1995.

[Lis 95] Lisbôa, M. L., “Reflexão Computacional no Modelo de Orientação a Objetos”, 25º Jornadas Argentinas de Informatica y Investigacion Operativa, Buenos Aires, Argentina, September, 1995.

[Lop 97] Lopes, C. V. and Kiczales, G., “D: A Language Framework for Distributed Programming”, Xerox Palo Alto Research Center, 1997, (http://www.parc.xerox.com/spl/projects/aop)

[Loq 96] Loques, O., Leite, J., Botafogo, R. and Lobosco, M., “Configuração, Reflexão e CORBA: Algumas Considerações”, III Workshop do Projeto ASAP, UFCE, Fortaleza, Ceará, 1996.

[Loq 97] Loques, O., Botafogo, R., Leite, J. “A Configuration Approach for Distributed Object-Oriented System Customization”, IEEE Third International Workshop on Object-oriented Real-time Dependable Systems, Newport Beach, California, February, 1997.

[Luc 95] Luckham, D.C., et *alli*, “Specification and Analysis of System Architecture Using Rapide”, IEEE Transactions on Software Engineering, Vol. 21, No. 4, , pp. 336-355, April, 1995.

[Maf 95] Maffeis, S. “Adding group communication and fault-tolerance to CORBA”, Proceedings of the 1995 USENIX Conference on Object Oriented Technologies, Monterey, CA, EUA, June, 1995.

[Maf 96] Maffeis, S., “Electra 2.0b Programer’s Manual”, Dept. of Comuter Science, Cornell University, February, 1996.

[Mag 96] Magee, J., Kramer, J., “Dynamic Structure in Software Architectures”, Fourth Symposium on the Foundations of Software Engineering, San-Francisco, California, EUA, October, 1996.

[Mag 97] Magee, J., Kramer, J., “Composing Distributed Objects in CORBA”, Third International Symposium on Autonomous Descentralized Systems, Berlin, Germany, April, 1997.

[Mal 93] Malucelli, V. V, "RIO - Linguagem de Configuração - Tolerância à Falhas", Nota Interna, DEE / PUC-RJ, February, 1993.

[Mit 97] Mitchell, S. E., et *alli*, “Adaptive Scheduling using Reflection”, ECOOP’97 Workshop on Reflective Real-time Object-Oriented Programming and Systems, Jyvaskyla, Finland, June, 1997.

[Mor 95] Moriconi, M., Qian, X. and Riemenschneider, R.A, “Correct Architecture Refinement”, IEEE Transactions on Software Engineering, Vol 21, No 4, pp. 356-372, April, 1995.

[OMG 96A] OMG, “The Common Object Request Brocker: Architecture and Specification”, Revision 2.0, July, 1996.

[OMG 96B] OMG RPF Revised Joint Submission, “Joint Meta Object Facility, Revision 1.0”, OMG CFTF, RPF-5, January, 1997.

[OMG 97] OMG RPF Revised Joint Submission, “Multiple Interfaces and Composition, OMG TC Document orbos/97-05-17, Version 4.2.7, June, 1997.

[OOP’90] ___, “Designing Reusable Designs: Experiences Designing Object-Oriented Frameworks”, OOPSLA/ECOOP’90, Addendum to the Proceedings, pp. 19-24, Ottawa, Canada, October, 1990.

[Pan 93]

[Pou 94] Pountain, D. and Szyperski, C., “Extensible Software Systems”, Byte Magazine, May, 1994.

[Pry 96] Pryce, N. and Crane, S.,”A Uniform Approach to Configuration and Communication i Distributed Systems”, International Conference on Configurable Distributed Systems, Annapolis, USA, May, 1996.

[Rod 92] Rodden, T., Blair, G. S., “Distributed Systems Support for Computer Supported Cooperative Work”, Computer Communications, Vol. 15, No. 8, October, 1992.

[Rub 97] Rubira, C. M. F., Corrêa, S. L. and Buzato, L. E., “Um Framework Orientado a Objetos Reflexivo para a Construção de Software Tolerante a Falhas”, VII Simpósio de Computadores Tolerantes a Falhas, Campina Grande, Paraíba, July, 1997.

[Rub] Rubira, C.M.F and Randell, B., “Object-Oriented Environmental Fault Tolerance”,

[Rum 91] Rumbaugh, J., et *alli*., “Object-Oriented Modeling and Design”, Prentice-Hall Inc., 1991.

[Sha 95] Shaw, M., DeLine, R., Klein, D. and Zelesnik, G., “Abstractions for software architecture and tools to support them”, IEEE Transactions on Software Engineering, Vol 21, No. 4, April, 1995.

[Sin 97] Singhai, A. et alli, “Reflective ORBs: Supporting Robust, Time-critical Distribution”, ECOOP’97 Workshop on Reflective Real-time Object-Oriented Programming and Systems, Jyvaskyla, Finland, June, 1997.

[Str 93] Strouds, R., “Transparency and Reflection in Distributed Systems”, ACM

Operating Systems Review, Vol 22, Num 2, April, 1993.

[Szt 95] Sztajnberg, A and Loques, O. G., "Ambiente RIO: Estado do Projeto”, XXV Congresso da SBC, Canela, RS, August, 1995.

[Szt 97] Sztajnberg, A. and Loques, O. G., “Reflexão Computacional e Multimidia Distribuida”, relatório de curso, COPPE/UFRJ, São Carlos, SP, Brazil, May, 1997. (in portuguese)

[Tai 96] Taivalsaari, A., “On the Notion of Inheritance”, ACM Computing Surveys, Vol. 28, No. 3, pp. 438-479, September, 1996.

[Tur 96] Turletti, T. e Huitema, C., “Videoconferencing on the Internet”, IEEE/ACM Transactions on Networking, Vol. 4, Num 3, pp. 640-350, June, 1996.

[Wal 96] Wallnau, K., Long, F. and Earl, A., “Toward a Distributed, Mediated Architecture for Workflow Management”, NFS Worksop on Workflow and Process Automation in Information Systems: State-of-the-Art and Future Directions, 1996.

[Way 96] Wayner, P. “Better Java Programming”, Byte Magazine, pp 63-64, September, 1996.

[Wec XX] Weck, W., “Inheritance Using Contracts & Object Composition”,

[Weg 96] Wegner, P., “Interoperability”, ACM Computing Surveys, Vol. 28, No. 1, March, 1996.

[Wer 95] Werner, J.A.V., "Metodologia e Suporte para Sistemas Distribuídos Configuráveis", dissertação de mestrado, DEE / PUC-RJ, March, 1995.

[Wu 97] Wu, Z., Schwiderski, S., “Design of Reflective Java”, APM.1911.01, ANSA Phase III (approved) Technical Report, 30 páginas, 1997.

[Yan 96] Yang, Z and Duddy, K., “CORBA: A Platform for Distributed Object Computing”, ACM Operation Systems Review, Vol 30, No. 2. pp. 3-41, April, 1996.

[Zim 96] Zimmermann, C., Cahill, V. “It’s Your Choice - On the Design and Implementation of a Flexible Metalevel Architecture”, IEEE, Proceedings of The Third International Conference on Configurable Distributed Systems, Maryland, EUA, May, 1996.