INTELLIGENT AND DYNAMIC PLUGGING OF COMPONENTS – AN EXAMPLE FOR NETWORKED ENTERPRISES APPLICATIONS

Moisés L. Dutra; Ricardo J. Rabelo
Federal University of Santa Catarina, BRAZIL
moises@floripa.com.br; rabelo@das.ufsc.br

This paper presents an approach to minimize the problem of reduced functional flexibility in the complex industrial systems, where they are bought as a whole package or module, quite expensive, even though they are not used at all or do not fit the enterprise's needs completely. The approach is based on the idea of a dynamic and intelligent plugging of software components. This plugging will occur only when the components functionalities are effectively needed, adapted to the current computing environment in use. The plugging is made on demand, applying a new perspective to the Application Service Providers, under the form of a Federation of Application Providers.

Keywords: Components, Application Service Providers, Plugging on Demand, Functional Flexibility.

1. INTRODUCTION

Substantial investment on financial, technological and computing resources has been required from the companies to deal with the problem of increasing complexity of enterprise systems. This is even more problematic if it is taken into account that more than 90% of the companies are small ones, implying that most of the systems solutions that might leverage their competitiveness cannot be acquired.

This paper presents an approach to that problem, providing a model where the system to be used by a company is dynamically and intelligently built up and adapted at execution time according to the current user needs, having the system's *kernel* as the basis. The envisaged scenario is based on the systems paradigm where the user should work only with the necessary software functionalities, only when (s)he needs, at the necessary environment (Dutra et al., 2003).

In spite of some results achieved by a number of international initiatives / research projects towards larger systems functional flexibility, the situation can still be considered primary. The best that software vendors have been doing nowadays is to offer smaller, less cost and easier installable versions of their software (e.g. ERP systems) in the form of "*My system*", more adapted to the needs of a given company.

Actually, this business model does not solve the essential problem. Software modules remain with a high degree of granularity and they are made available just as instances of a wider system, i.e. companies have to buy them with their full subset of functionalities no matter what their effective needs are. This is a very important aspect as practice shows that most of the software functionalities are not used by the end-users. Therefore, if they do not use them, why should companies pay for them and waste computing resources to host them?

The proposed approach increases very much the functional flexibility of a system, where companies can use only the functionalities they need, when they need and at the required computing environment (PC, palm, etc.). The system is no longer developed as a monolithic system but rather as a set of small software *components*, independent and logically integrated, providing the full set of the system functionalities when put together. In this approach a new vision of Application Service Providers is given, transforming them into distributed *Application Providers* of components.

The model has been validated in a scenario of virtual enterprises, where the supporting tools used by its members to manage businesses can be adapted to members' current needs.

This paper is organized as follows: Section 2 stresses the main technologies used. Section 3 addresses the dynamic plugging technique. Section 4 presents the proposed model. Section 5 depicts the implemented prototype and preliminary results. Section 6 provides the main conclusions.

2. INVOLVED TECHNOLOGIES

This section depicts the three main technologies used as the basis for the proposed approach. These technologies have been chosen as they allow the construction of open, interoperable, adaptive systems, providing a larger system life cycle.

2.1 Components

A software component is a unit of composition with contractually specified interfaces and explicit context dependencies, which can be deployed independently and is subject to composition by third parties (Szyperski, 1997). A component-based development provides a more flexible approach than the traditional software development method, in which the system is designed globally by deploying and integrating small modules inside the same application.

Components can be designed to execute simple or complex tasks, with variable granularity, i.e., they can be implemented as simple functions or even as larger and complex modules. (Beneken et. al, 2003) see several advantages of using components: they can run in adaptive environments; can be exchanged or partly deployed partly without failure of the whole; can be written in different languages, using different technologies, in different operating systems; explicit interfaces allow to connect and decouple cooperating parts of the overall system.

2.2 Application Service Providers

According to (Dewire, 2002), an *Application Service Provider* (ASP) provides a contractual software-based service for hosting, managing, and providing access to an application from a centrally managed facility. For a certain periodically fee, the ASP provides content and other services for users connected through the Internet or any other network platform, and the users do not need to be concerned with software versions and upgrades. ASP provides access to applications that are located outside the client work environment. Several specialists believe that, with the appearance of the Internet, it would make more sense to provide software as a service than to sell it as a product "closed in a box" (Stardock Corporation, 2000).

Despite being a good model, it presents some relevant limitations when observed under the envisaged functional flexibility scenario: its processing is logically and physically centralized (the component is executed in the ASP); the granularity of its modules is very large; and they usually are not adapted at all to the client needs.

2.3 Peer-to-Peer

Peer-to-Peer (P2P) is an architecture where the resources and service sharing are made directly among the involved system peers, without the intervention of a central server (Parameswaran et al., 2001). The term "peer-to-peer" refers to a class of systems and applications that employ distributed resources to perform a function in a decentralized manner (Milojicic et al., 2002). Therefore, a P2P-based system is suitable to support large scale and geographically distributed / decentralized systems.

3. DYNAMIC PLUGGING OF COMPONENTS

In traditional approaches of component-based systems, the final system is "fully integrated" during design time. Each component is a "mini subsystem", which can be developed, deployed and tested separately. Its replacement by another component does not affect the global system operation, thus supporting some level of functional flexibility.

However, this flexibility is not as large as it could be. Firstly, because the traditional plugging of components is static and manually done. Secondly, because once plugged, the component remains in the same system even if it is no longer needed.

Some authors have made contributions in that direction, such as (Lauder, 1999) (Seiter et al., 1999), applying patterns for dynamic plugging. These patterns were based in generic frameworks to support the plugging (usually of an inherited class) in runtime. However, in the approach proposed in this paper, the dynamic plugging of components occurs transparently to the user, without any framework and on demand. The plugging is intelligent as it should adapt itself to the current computing environment (hardware and software), to the sources of download and to the type of components.

3.1 Designing Dynamic Components

Dynamic components need to be firstly adaptable to several types of hardware and operating systems, including PCs and mobile devices. As said before, the components granularity can vary substantially. The focus in this work is on components of small granularity in order to better fit the needs of the client system.

The two main component models are the *Enterprise Java Beans* (EJB) (Sun Microsystems, 2002) and the *CORBA Component Model* (CCM) (OMG, 2002). In both models a structure called *Container* is required to support the plugging and the components execution. A container provides supporting services for the component life cycle, transactions management, communication security, and events notification. In the dynamic model proposed in this paper, the container is no longer required. This provides a more agile transfer and component plugging, and it creates the basis for a solution independent of technologic, thus enlarging the system life cycle. The same direction has been followed by some international efforts (Agedis, 2003) (Adapt, 2004).

The communication between the application and the components (dynamically plugged in) can be carried out in several ways, e.g. by changing registered messages in the operating system, by *Application Programming Interfaces* (APIs) (Coach 2004), by *Dynamically Linked Libraries* (DLLs), by local components management (like COM) and distributed models (like CORBA) (Calim 2001) (Combine 2002), and simply by direct access (Liang et al., 1998), where the component functionalities are used directly, without the need of an integration middleware.

An application that enables dynamic plugging is composed of its core functionalities and "pluggable" areas where the components can be plugged in by means of their interfaces, which enables the communication between the component and the external world. The components' interfaces must be extremely well defined (parameters, generality and communication) so that plugging can be accomplished successfully.

4. PROPOSED APPROACH

In general, the proposed approach works as follows (Figure 1). The company has the software kernel, comprising its essential functionalities. When the client (user or groups) calls for a system option / functionality whose code is not presented in the kernel, a *requisition* for the associated code component is dispatched to the representative (*Coordinator*) of a central of components (*Federation of Application Providers - FAP*) that will search, over the Internet, for the most suitable repository (*Application Provider - AP*) that can supply that particular need / component. Once it has been found, a *peer-to-peer* communication is established between the application and the repository, and the component is sent out to the client application to be plugged in, in a transparent way, according to the requisition's specifications.

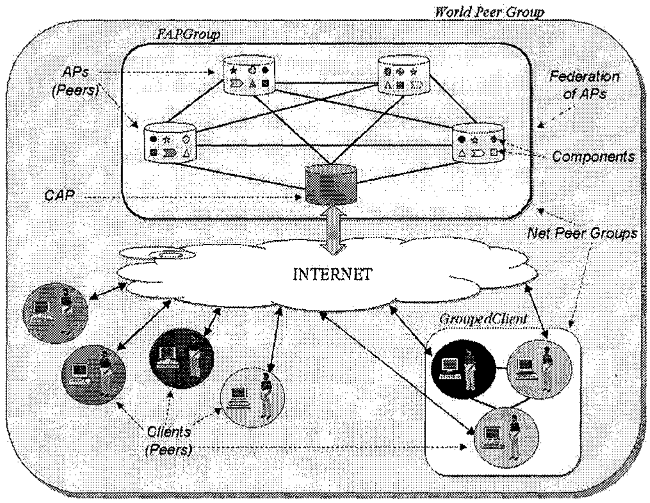


Figure 1 – Global Architecture

4.1 Application Provider (AP)

An AP is a repository of components. Unlike the traditional Application Service Providers (ASP), here the components dynamically plugged in run at the client's host, meaning that APs do not provide services but rather applications / components. Moreover, the proposed model is totally distributed, with the components coming from several APs located at different places, selected based on some decision criteria (e.g. geographic distance, bandwidth). Different versions / implementation models of the same component can be available in the APs, but all of them must follow the pre-designed component's interface.

4.2 Federation of Application Providers (FAP)

The FAP was introduced in the model to give scalability to the global architecture. It represents a cluster of APs from which the components should come. The FAP has a coordinator (*Coordinator of Application Provider - CAP*), which is the visible external entity to the FAP clients. The CAP is in charge of: i) seeking the AP which better matches the component advertisement; ii) creating a log file of all the received requests and the respective APs that were selected to supply the components; and iii) managing the components *contract*.

A contract is directly related to the business model involving the APs and the clients. For instance, a company can pay for the components a fixed monthly fee, or based on the number of components plugged in.

AP Structure

An AP has five cooperative modules (Figure 2). The first module is the *CAP Listener*, which receives component requests. Each request is checked by the *Specification Valuator*, which analyzes the request, validates it and searches for it in the *Component Repository*. The *Component Sender* makes the component transfer to the client, and the *Unsuccessful Message Sender* informs the client about the lack of the component.

CAP Structure

The CAP structure comprises six modules (Figure 3). The *Client Listener* receives the component requests from the clients. These requests are validated by the *Request Valuator*, which analyzes the received specification and verifies the client's contract terms. The *Component Advertisement Researcher* seeks the component's advertisement inside the FAP. The *FAP Listener* waits for the answer of the advertisement search, and the *Request Forwarder* redirects the request to the AP which has posted the advertisement. The *Unsuccessful Message Sender* will notify the client either if the FAP does not have the requested component advertisement or if the client's request was not approved by the request valuator.

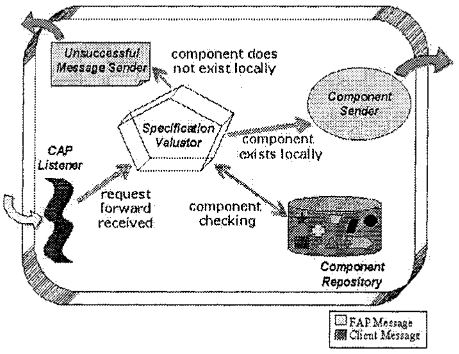


Figure 2 – AP Structure

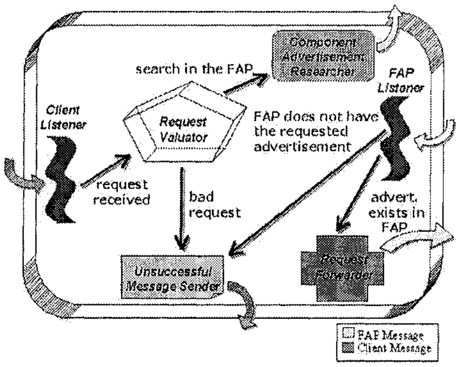


Figure 3 – CAP Structure

4.3 FAP Client

A FAP client is represented by a computer or a group of computers in a local network which hosts applications that request components from the FAP "server" (Figure 4). These applications in turn can be i) stand-alone; ii) distributed applications running either in a single computer or in several computers; and iii) the same application running its copies in several computers. For the cases *ii* and *iii* it is called *grouped client* (Figure 5) This allows computers to run more than one FAP-Client-application, developed in different languages and in different platforms, i.e. heterogeneous applications can request heterogeneous components no matter their languages and operating systems are. Grouped clients have just one contract with the FAP.

A FAP client has a module called *Component Management Module*, which manages the entire plugging process and that is composed of four sub-modules:

- *Component Fault Treater*: It acts whenever the system recognizes that the needed component is currently not present in the client. The treater then looks for the component in the local repository / cache. If it is found, a notification is sent to the *Component Plugger*. If not, the *Request Dispatcher* is called.
- *Request Dispatcher*: It builds the component request specification based on the client needs and environment characteristics, and sends the request to the FAP.

- *FAP Listener*: It waits for an answer from the FAP concerning the component that was requested. In the case of a positive answer, the *Listener* receives the message that encapsulates the component itself, stores it in the local cache, and calls the *Component Plugger*; otherwise a failure notification is sent to the application.
- *Component Plugger*: It performs the dynamic plugging itself. It loads the component from the cache so that the application can use it thereafter.

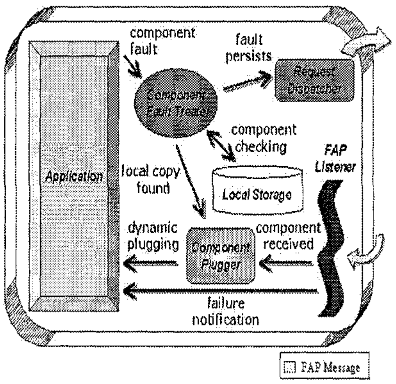


Figure 4 – FAP Client Structure

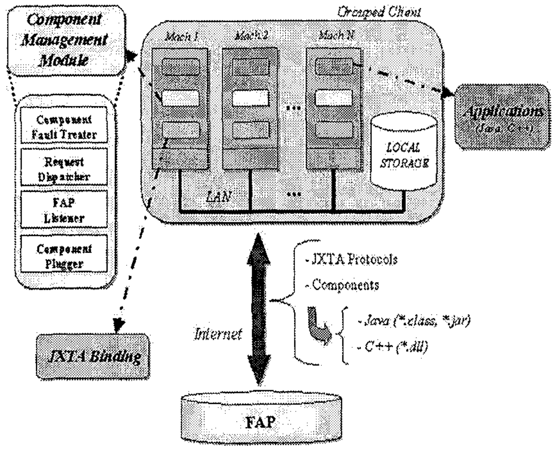


Figure 5 – Grouped Client

Each FAP client has a central local repository to cache the components already transferred. Once the components are used they can be later discarded from the application kernel (depending on the business model in use), but they are kept stored in the cache for future use, i.e. the client does not need to request it again to the FAP. This results in lower network latency and provides more agility to the global plugging process. The components can be transparently added to or removed from the application without interfering in the other components already plugged in. A component can be integrated with other (heterogeneous or not) components as well as with legacy ("re-engineered") systems. For the automatic component updating, the FAP client has a process that frequently asks the CAP about new versions. If a new version is found close to a certain AP, the same global plugging process is triggered again.

The approach proposed in this paper has some general similarities with the model proposed in (Camarinha-Matos et al., 2001) for Virtual Organizations, which is called *service federation*. It represents an approach to support the interoperation among heterogeneous, autonomous and geographically distributed entities. The services are available at service providers that publish their services in a catalogue that can be consulted by the users whenever they need and wherever they are. The service providers form a cluster, composing a federation. Each of those entities interested to provide services should announces them in a *catalogue* that will serve as the central source of information for clients. Once a given client selects a given service, the catalogue sends the service's interface to the client application so that a direct / remote service invocation can be carried out between the application and the service provider.

Both approaches, *service federation* and the one being proposed here (*federation of application providers - FAP*), involve distributed and autonomous clusters of providers and allow transparent access to what the user / client application requires, no matter where it is. Yet, both make use of a kind of central broker. However, the FAP approach presents some differences, namely:

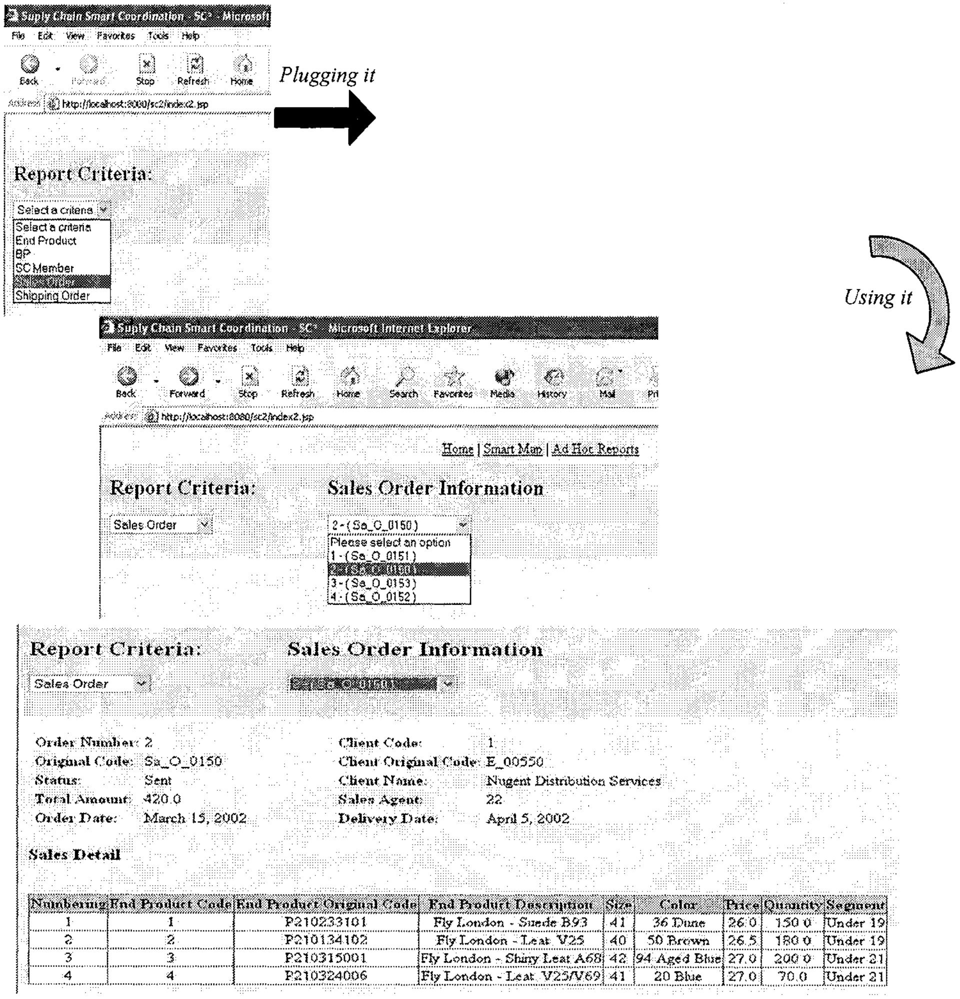
i. FAP does not provide services, but system components.
ii. FAP does not require that providers are previously registered in the "broker" as the supporting platform that is used (JXTA – see section 5.1) is able to look for the components in the APs automatically.
iii. FAP does not provide the services' interfaces to the client application. It finds the most suitable AP for the required component, a transparent P2P connection is established between the AP and the application, the plugging process is carried out, and the "service" is executed locally.
iv. The FAP client application is the one which sees what is missing, therefore there is no human intervention.

5. PROTOTYPE

In order to test and to preliminary validate the proposed model, a prototype has been developed taking a virtual enterprise (VE) application into account. This application consists of a VE management system that provides several functionalities to the end-user (Rabelo et al., 2002). Applying the proposed FAP approach on that system meant to rethink it with the objective of defining what would be the system's kernel (i.e. the FAP client) and hence its "optional" functionalities (i.e. the pluggable components to be put available in APs).

The prototype was based on only one of the macro-functionalities of the system, called *Ad-hoc Reports,* which provides a number of managerial reports about a given VE (Figure 6). The user has several report options, such as the list of the VE members, the parts being produced, and the involved sales and shipping orders. The display of these options is executed by the ad-hoc's kernel. When the user selects, for instance, the report option *sales orders*, the system detects that this function is not there and requests the respective component to the CAP. After the whole plugging process is accomplished (see Section 4), the component is executed and then other graphical interfaces are shown, listing all the sales orders related to that given VE. In this simple case, the purpose of the component is to have access to the local database and to get those orders using SQL queries. It also has the purpose to provide the user with detailed data about each of these sales orders (from the database too), shown in the interface in the bottom of the figure 6.

It has to be noted that the way the component's graphical interfaces were shown (i.e. in HTML in that case) was specified (besides some other basic parameters) in the requisition for the component sent out by the FAP client regarding its computing environment and needs. For instance, another component with the *same* functionality but built up to run over another operating system (e.g. Linux) and non-web environment can exist in FAP. Therefore, the system does not need to have all possibilities to show the ad-hoc reports embedded in its kernel. Only the required possibility is (dynamically) linked to the kernel and exactly when it is needed, providing an effective functional flexibility.


Suply Chain Smart Coordination - SC² - Microsoft Internet Explorer
File Edit View Favorites Tools Help
Back Forward Stop Refresh Home Search Favorites Media History Mail
http://localhost:8080/sc2/index2.jsp
Plugging it
Using it
Home | Smart Map | Ad Hoc Reports
Select a criteria
End Product
BP
SC Member
Sales Order
Shipping Order
Please select an option
1 - (Sa_O_0151)
2 - (Sa_O_0150)
3 - (Sa_O_0153)
4 - (Sa_O_0152)

Report Criteria: Sales Order

Sales Order Information

Order Number:	2	Client Code:	1
Original Code:	Sa_O_0150	Client Original Code:	E_00550
Status:	Sent	Client Name:	Nugent Distribution Services
Total Amount:	420.0	Sales Agent:	22
Order Date:	March 15, 2002	Delivery Date:	April 5, 2002

Sales Detail

Numbering	End Product Code	End Product Original Code	End Product Description	Size	Color	Price	Quantity	Segment
1	1	P210233101	Fly London - Suede B93	41	36 Dune	26.0	150.0	Under 19
2	2	P210134102	Fly London - Leat. V25	40	50 Brown	26.5	180.0	Under 19
3	3	P210315001	Fly London - Shiny Leat A68	42	94 Aged Blue	27.0	200.0	Under 21
4	4	P210324006	Fly London - Leat. V25/V69	41	20 Blue	27.0	70.0	Under 21

Figure 6 – Prototype Interface

The JXTA Model
In order to implement the envisaged decentralized / distributed model the JXTA peer-to-peer platform (JXTA Project) was chosen. JXTA is a very recent technology and it is constituted by a networked programming platform to cope with distributed computing and interoperable platforms, independent of operating systems and programming languages (Gong, 2002). JXTA has a set of communication protocols, each one containing one or more messages. This platform supports the several issues required in the proposed FAP model, namely location protocols, content search and transfer of data among peers, also including confidentiality, integrity, authentication, access control, auditing, cryptography and data security.

Every FAP client is seen as a JXTA peer, possessing a local JXTA Binding to enable the communication with the other peers. All peers belong to a group called *World Peer Group*, which is the logical reference of the JXTA structure. The peers can also belong to subgroups (*Net Peer Groups*). In the case of FAP a peer group was created, called *FAPGroup*. It means that every AP is part of this group. Yet, grouped clients (see section 4.3) also are represented as groups, belonging to the *World Peer Group*.

The APs' structures are implemented using the JXTA protocols. The communication process (messages and data transfer) has used the *Pipe Binding* and the *Peer Discovery* protocols. *Pipes* work as communication channels, and they can be of type *Pipe In* and *Pipe Out*. A *pipe in* is created to establish a communication and to wait for external connections, whereas a *pipe out* is used to locate some pipe and to connect with it. This localization is done using advertisements, which is the basic structure to announce the components specifications to the federation of APs. In the JXTA platform the advertisements are expressed in the XML standard.

This prototype has been developed using the following tools: Java SDK 1.4.2.02, JXTA platform version 2.1.1, Borland database Interbase 6.0, webserver TomCat 5.0.16 and JDBC driver FireBirdSQL 1.0.0., in the Windows XP environment.

6. CONCLUSIONS AND NEXT STEPS

This paper presented an approach called Federation of Application Providers (FAP) as a more flexible alternative to the Application Service Providers (ASP) existing in the market. Using FAP an application can be seen as a building block system, where its functionalities – implemented as software components – are dynamically plugged into the system in the exact moment they are required, adapted to the current computing environment. This approach creates rooms for new business models as the components come from a group of distributed, interoperable and autonomous application providers, which supply components, not services. Moreover, this approach makes it possible for small companies to acquire modern software (e.g. ERPs), usually too costly and with many unnecessary functionalities.

The development of dynamic and multi-platform components, with rigorous specifications, has been seen as a prominent approach to maximize code reusability. More comprehensive scenarios can arise from when large repositories of generic components were built up, especially if they were based on reference models for processes, information and ontologies. The preliminary results reached with the developed prototype showed that the FAP model seems very promising in supporting systems functional flexibility.

The model is strongly based on the JXTA platform, which supports most of the system requirements in a generic peer-to-peer scenario.

Further research should involve a deeper reflection on business models and the contracts between the FAP and their clients, and on how these aspects should be connected with the dynamic plugging and unplugging processes (specially in Java-based components), without human intervention. Yet, a more complex application should be developed to comprise multi-language components and hence to evaluate their interoperability.

Acknowledgements

This work was partially supported by CNPq – The Brazilian Council for Research and Scientific Development and in the scope of the IFM project (www.ifm.org.br).

7. REFERENCES

1. Adapt, *Middleware Technologies for Adaptive and Composable Distributed Components* – http://adapt.ls.fi.upm.es/adapt.htm, in March 2004.
2. Agedis, *Automated Generation and Execution of Tests Suites for Distributed Component-based Software* – http://www.agedis.de/, in March 2004.
3. Beneken, G., Hammerschall, U., Broy, M., Cengarle, M. V., Jürjens, J., Rumpe, B., Schoenmakers, M, *Componentware – State of the Art 2003.* In Understanding Components Workshop of the CUE Initiative at the Univerità Ca' Foscari di Venezia, Venice – Italy, October 7th-9th 2003.
4. Calim (2001) – *Corba Architecture for Legacy Integration and Migration* – http://dbs.cordis.lu/fep-cgi/srchidadb?ACTION=D&SESSION=118262004-3-5&DOC=2&TBL=EN_PROJ&RCN=EP_RCN:53593&CALLER=IST_UNIFIEDSRCH, in March 2004.
5. Camarinha-Matos, L. M., Afsarmanesh, H., Kaletas, E., Cardoso, T. F. (2001), *Service Federation in Virtual Organizations, in Proceedings of IFIP TC5 / WG5.2 & WG5.3 Eleventh Int. PROLAMAT Conf. On Digital Enterprise – New Challenges*, Kluwer Academic Publishers, pp 305-324, Hungary, 2001.
6. Coach (2004) – *Component Based Open Source Architecture for distributed Telecom Applications* □ http://dbs.cordis.lu/fep-cgi/srchidadb?ACTION=D&SESSION=121472004-3-5&DOC=11&TBL=EN_PROJ&RCN=EP_RCN:61829&CALLER=IST_UNIFIEDSRCH, March 2004.
7. Combine Project (2002) – http://www.opengroup.org/combine/overview.htm , March 2004.
8. Dewire, D. T., *Application Service Providers - Enterprise Systems Integration,* 2nd Edition, pag.449-457. Auerbach Publications, 2002.
9. Dutra, M. L., Rabelo, R. J., *Dynamic Functional Instantiation of Industry Systems* [in Portuguese], in proceedings of VI Brazilian Symposium of Intelligent Automation, Brazil, September 2003.
10. Gong, L., *Project JXTA: A Technology Overview* – Sun Microsystems, Inc., October 29, 2002.
11. JXTA Project - http://www.jxta.org/ – in February 2004.
12. Lauder, A., C++ Report Magazine, *Pluggable factory in Pratice*, pp. 27-32, v. 11, n. 9, Oct 1999.
13. Liang, S., Bracha, G. (1998), *Dynamic Class Loading in the Java Virtual Machine. Proceedings OOPSLA '98*, Vancouver, Canada, October, 1998.
14. Milojicic, D., Kalogeraki, V., Lukose, R., Nagaraja, K., Pruyne, J., Richard, B., Rollins, S., Xu, Z., *Peer-to-Peer Computing* , Technical Report HPL-2002-57, HP Labs. 2002
15. Object Management Group, CORBA Componentes – formal/02-06-65.2002.
16. Parameswaran, M., Susarla, A., Whinston, A. B. (2001), *P2P Networking: An Information-Sharing Alternative* – IEEE Computer Society's – Computing Practices, pag. 31, July 2001.
17. Rabelo, R. J.; Klen, A. P.; Klen, E. R., A Multi-agent System for Smart Coordination of Dynamic Supply Chains, Proceedings PRO-VE'2002, pp. xx-yy, 2002.

18. Seiter, L., Mezini, M., Lieberherr, K., *Dynamic component gluing. In Ulrich Eisenegger, editor, First International Symposium on Generative and ComponentBased Software Engineering*, Springer , 1999.
19. Stardock Corporation (2000), *ASPs – A Primer – April/2000 –* http://www.stardock.net/media/asp_primer.html – in February 2004.
20. Sun Microsystems, *Enterprise JavaBeans Spec. version 2.1*, 2002.
21. Szyperski, C., *Component Software: Beyond Object-oriented Programming*, Addison-Wesley, 1997.