American Journal of Engineering Research (AJER)
e-ISSN : 2320-0847 p-ISSN : 2320-0936
Volume-02, Issue-10, pp-52-62
www.ajer.org

Research Paper Open Access

Medical image processing using a service oriented Architecture and Distributed Environment

Himadri Nath Moulick[1], Moumita Ghosh[2]
[1]CSE, Aryabhatta Institute of Engg & Management, Durgapur, PIN-713148, India
[2]CSE,University Institute Of Technology,(The University Of Burdwan) Pin -712104,India

Abstract: - The aim of this paper is to present a services based architecture for medical image processing in assisted diagnosis. Service oriented architecture (SOA) improves the reusability and maintainability of distributed systems. In service oriented architectures, the most important element is the service, a resource provided to remote clients via a service contract. We propose a generic model for a service, based on a loosely coupled, message-based communication model. Our service model takes into account the possibility to integrate legacy applications. Specialized image processing services can be dynamically discovered and integrated into client applications or other services. Complex systems can be created with the help of some SOA concepts like Enterprise Service Bus (ESB). DIPE is a distributed environment that provides image processing services over integrated teleradiology services networks. DIPE integrates existing and new image processing software and employs sophisticated execution scheduling mechanisms for the efficient management of computational resources within a distributed environment. It can also be extended to provide various added-value services, such as management and retrieval of image processing software modules, as well as advanced charging procedures based on quality of service. DIPE can be viewed as the natural evolution of the legacy field of medical image processing towards a service over the emergent health care telematics networks.

Keywords: - *service oriented architecture, image processing, web service.*

I. INTRODUCTION

In service oriented systems, operational entities are distributed across the network in order to improve availability, performance and scalability. These entities are called *services*. The service provides access to its functionality. The whole system is viewed as a set of interactions among these services. SOA promotes the reuse of services. The system evolves through the addition of new services. SOA is not tied to a specific technology. It can be implemented using a large variety of technologies, programming languages and communication protocols. Interactions between services and clients in SOA are based on a very dynamic model [1]. A service can be discovered at runtime, can be replaced if has become unavailable or can be used to create a new service (and a new functionality). With these characteristics, SOA offers a powerful support for adaptability. The adaptability can take many forms, depending on the terminal capabilities, the network connection, etc. Microsoft has proposed a SOA based platform for healthcare [2]. Healthcare is an extremely fluid industry. Each change requires an adaptation of systems. Point-topoint integration becomes costly and complex to maintain for healthcare providers and consumers. The benefit of SOA to the healthcare industry is that it enables systems to communicate using a common framework, integration of new elements becomes less complex and the system can be adapted more rapidly. In recent years, advances in information technology and telecommunications have acted as catalysts for significant developments in the sector of health care. These technological advances have had a particularly strong impact in the field of medical imaging, where film radiographic techniques are gradually being replaced by digital imaging techniques, and this has provided an impetus to the development of integrated hospital information systems and integrated teleradiology services networks which support the digital transmission, storage, retrieval, analysis, and interpretation of distributed multimedia patient records [1]. One of the many added-value services that can be provided over an integrated teleradiology services network is access to high-performance computing facilities in order to execute computationally intensive image analysis and visualisation tasks [2]. In general, currently available products in the field of image processing (IP) meet only

specific needs of different end user groups. They either aim to provide a comprehensive pool of ready to use software within a user-friendly and application specific interface for those users that use IP software, or aim for the specialised IP researcher and developer, offering programmer‟ s libraries and visual language tools. However, we currently lack the common framework that will integrate all prior efforts and developments in the field and at the same time provide added-value features that support and in essence realise what we call a „service‟ . In the case of image processing, these features include: computational resource management and intelligent execution scheduling; intelligent and customisable mechanisms for the description, management, and retrieval of image processing software modules; mechanisms for the "plug-and-play" integration of already existing heterogeneous software modules; easy access and user transparency in terms of software, hardware, and network technologies; sophisticated charging mechanisms based on quality of service; and, methods for the integration with other services available within an integrated health telematics network. In this paper we present the architecture of DIPE, a novel distributed environment for image processing services. DIPE is based on a distributed, autonomous, co-operating agent architecture [3]. It is designed so that it is modular, scaleable and extensible, and it can be readily implemented on different hardware and software platforms, and over heterogeneous networks. DIPE consists of a functional core which supports the persistent distributed execution of IP algorithms, and can be extended to support other added-value services such as macros, resource management, algorithm retrieval, charging, etc. Here we describe the functional core of the system and discuss the mechanisms and notions employed to allow integration of third party IP algorithms and the development of new IP software. Finally, we describe the functional extensions of the core that support macro execution and resource management. DIPE has been developed to support distributed medical imaging processing, an added-value teleradiology service within the integrated regional health telematics network, currently under development by the Institute of Computer Science (ICS), Foundation for Research and Technology - Hellas (FORTH), on the island of Crete [4].

II. SOA BACKGROUND

The term Service Oriented Architecture, SOA for short, contains some important notions. We have the following definitions for these notions [3]: An **Architecture** is a formal description of a system, defining its purpose, functions, externally visible properties, and interfaces. It also includes the description of the system's internal components and their relationships, along with the principles governing its design, operation, and evolution. A **service** is a software component that can be accessed via a network to provide functionality to a service requester.The term **service-oriented architecture** refers to a style of building reliable distributed systems that deliver functionality as services, with the additional emphasis on loose coupling between interacting services.

1.Service

The **service** is the core element in SOA. A **service** is defined as "a mechanism to enable access to one or more capabilities, where the access is provided using a prescribed interface and is exercised consistent with constraints and policies as specified by the service description" [4]. A service can have the following characteristics: A service provides a contract defined by one or more interfaces (just like a software component). This allows the change of the service implementation without reconstructing the client as long as the contract is not changed. Implementation details (programming languages, operating systems, etc) of the service are not the concern of the service requestor. A service can be used as stand-alone piece of functionality or it may be integrated in a higher-level service (composition). This promotes reusability. Legacy applications can be transformed in services by using some wrapper techniques. Services communicate with their clients by exchanging messages. Typically, the request/ response message pattern is used. From the client point of view, a synchronous or asynchronous communication mechanism can be implemented. In SOA model is not fixed a specific communication protocol. Many protocols can be used: HTTP, RMI, DCOM, CORBA, etc. Services can participate in a workflow (the term is *service choreography* in SOA terminology). A workflow is "the movement of information and/or tasks through a work process" [5] and it's based on a workflow engine. Services need to be discovered at design time and run time by clients. This mechanism is provided by a service directory (service registry). A service provider can publish (register) his service. Services communicate with other services and clients using standard, dependency-reducing, decoupled message-based methods such as XML document exchanges. This characteristic is called **loose coupling** This term implies that the interacting software components minimize their knowledge of each other: more information is achieved at the time is needed. For instance, after discovering a service, a client can retrieve its capabilities, its policies, its location, etc. The characteristics of **loose coupling** are [6] Flexibility: A service can be located on any server and relocated as necessary (with the condition to update its registry information) and clients will be able to find it. Scalability: Services can be added and removed depending on the needs. Replaceability: With the condition that the original interfaces are preserved, a new implementation of a service can be introduced, and outdated

implementations can be retired, without affecting the service clients. Fault tolerance: If a server, a software component, or a network segment fails, or the service becomes unavailable for any other reason, clients can query the registry for alternate services that offer the required functionality, and continue to work in the same way.

2. SOA Interaction cycle

In figure 1 is depicted the basic case of using a service with three components: a service provider, a service requester and a service directory (service registry). Some simple, bi-directional interactions (synchronous request/response pattern) are represented as an *interaction cycle* [7]. A real-world implementation can be more complex. A SOA architecture has three important elements:

2.1 Service directory

It acts as an intermediary between providers and requesters. Usually, services are grouped by categories.

2.2 Service provider

The Service Provider defines a service description and publishes it to the service directory.

2.3 Service requester

The service requester can use the search capabilities offered by the service directory to find service descriptions and their respective providers.

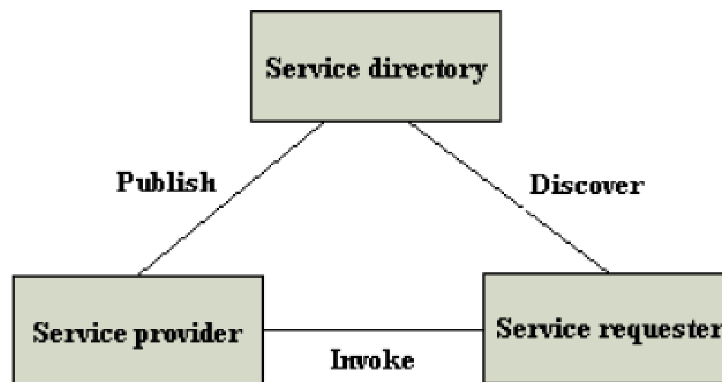

Fig 1. SOA interaction cycle

The service provider has to publish the service description in order to allow the requester to find it. Where it is published depends on the architecture. In the discovery the service requester retrieves a service description directly or queries the service registry for the type of service required. In this step the service requester invokes or initiates an interaction with the service at runtime using the binding details in the service description to locate, contact and invoke the service.

3. Enterprise Service Bus

The Enterprise Service Bus (ESB) is sometimes described as a *distributed infrastructure* [8] and it's a logical architectural component that provides an integration infrastructure consistent with the principles of SOA. Two different issues are being addressed: the *centralization of control*, and the *distribution of infrastructure* [9]. ESB and centralize control of configuration, such as the routing of service interactions, the naming of services, and so forth. ESB might deploy in a simple centralized infrastructure, or in a more sophisticated, distributed manner. ESB does not implement a service-oriented architecture (SOA) but provides the features with which one may be implemented. ESB is not mandatory in SOA but is usually used in large (enterprise) systems with many services. The ESB might be implemented as a distributed, heterogeneous infrastructure. Minimum ESB capabilities considered in IMB view [8, 9]:

3.1 Communications

Routing and addressing capabilities providing location transparency, administrations capabilities to control service addressing and at least one form of messaging (request/response, publish/subscribe, etc), support for at least one communication protocol (preferable a widely available protocols such HTTP).

3.2 Integration

Support for multiple means of integration to service providers, such as Java 2 Connectors, Web services, asynchronous messaging, adaptors, and so forth.

3.3 Service interactions
An open and implementation independent service messaging that should isolate application code from the specifics of routing services and transport protocols, and allow service implementations to be substituted.

III. MODEL FOR SOA-BASED IMAGE PROCESSING SYSTEMS

In this section we propose a model for implementing SOA-based system oriented to medical image processing. The model is generic enough to be used in other areas. The model contains a programming model, a service model and a messaging model.

1. Programming Model

The programming model, depicted in figure 2, is composed by four layers: the service layer, the component layer, the object layer and the technology layer.

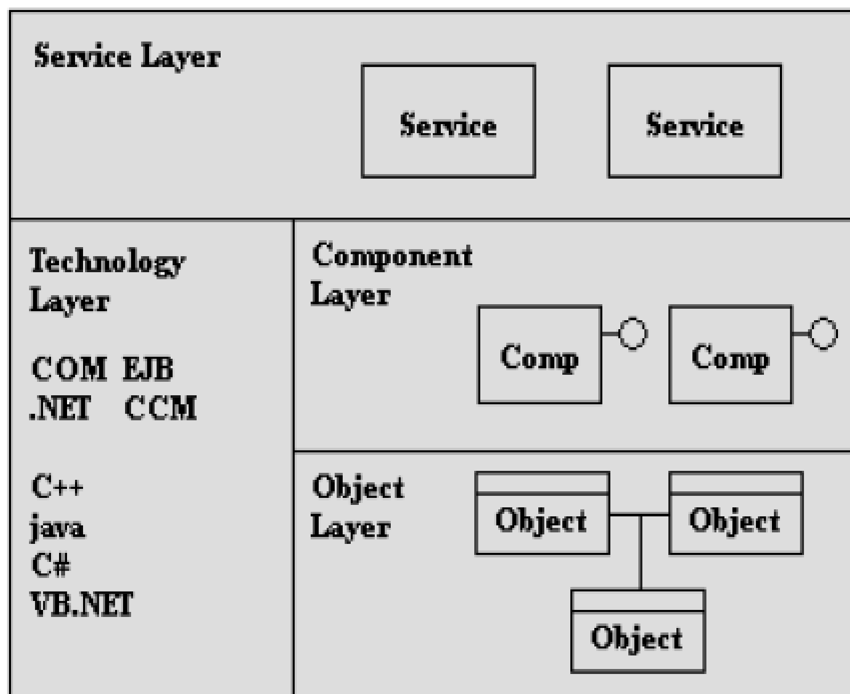

Fig 2. Programming model

Typically, a service is created using one or more components and a component is created using one or more objects. The service layer contains business services. A service is created with the help of the component oriented programming (COP). The component layer relies on software component technologies like: COM (component object model), EJB (Enterprise Java Beans), CCM (CORBA Component Model), OSGi (Open Services Gateway Initiative) or .NET Component Model. The software components can be of two types: functional components (business components) and non-functional components (like data access components, communication components or any other components). A component is implemented using object oriented techniques (the object layer). This layer is based on object oriented technologies (programming languages) like: C++, java, C#. Our model addresses the problem of integrating legacy applications (existent applications that are not servicebased). In order to integrate these applications, a wrapper pattern (adapter pattern) can be used. The wrapper can be applied in every layer. For instance, if the legacy application is object oriented but is not based on components, the wrapper should be applied in the component layer. If the legacy application is implemented in C, the wrapper should be applied in the object layer. This respect the proposed model: functionality is encapsulated in components, and components are created using objects.

2. Service Model

The service model is depicted in figure 3. It's composed from 3 layers: the interface layer, the business layer and the resource layer.

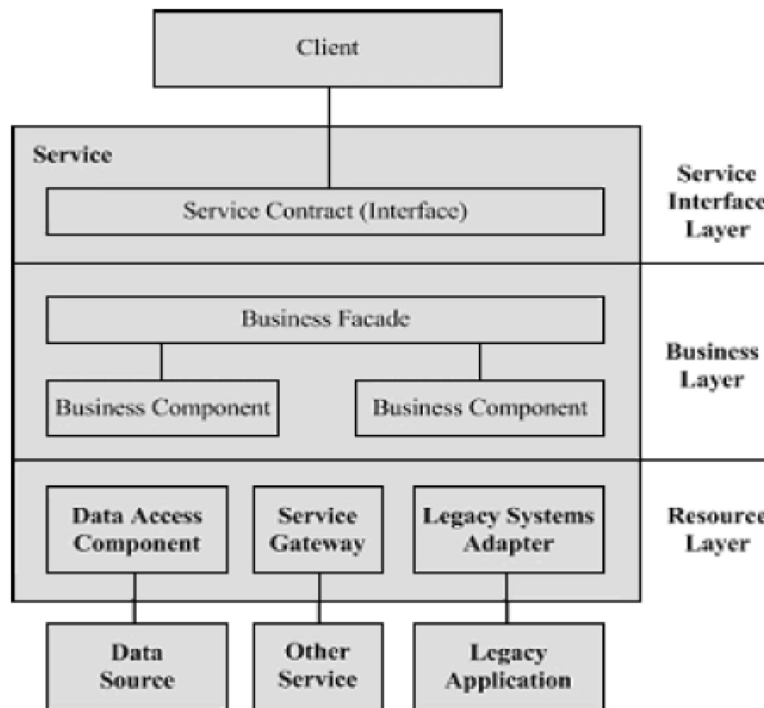

Fig 3. Service model

The Service Interface Layer contains the service contract (service interface) and it's detailed in the next section. The Business Layer contains a business façade and business components (sometimes called functional components). A business component performs (implements) operations described in the service contract. The business façade (façade pattern) is optional and it may be used in a complex architecture, with many business components. The resource layer contains different components (nonfunctional components) with the roll of interacting with external resources. In the figure are represented three of the most common types of resource access: a data access component for accessing database systems, a service gateway for accessing other services (in SOA a service can be a consumer for another service) and a wrapper (adapter) component for accessing legacy systems. The resource layer is not mandatory if the service does not use external resources. Accessing a legacy application was treated in section 3.1 from a programming point of view. The service model is extensible, new facilities like security, transactions or QoS capabilities can be introduced.

3. Messaging Model

Usually, a service communicates with its clients by sending and receiving well-defined messages. A proposed messaging model is presented in figure 4. A service interface is similar with an interface in object oriented programming. The service interface has the role to describe the service operations and the types of messages needed by those operations.

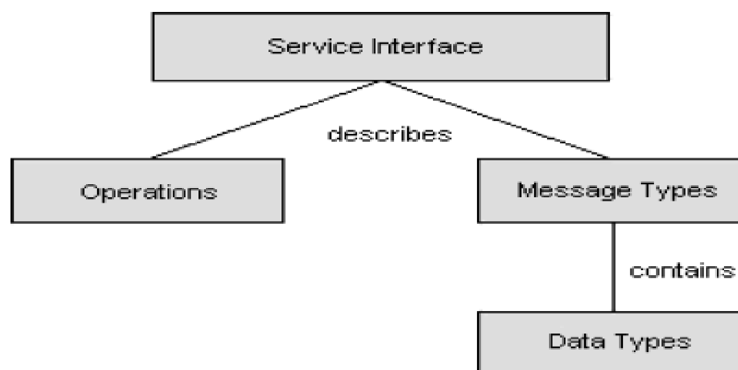

Fig 4. Messaging model

A message type contains one or many data types that can be translated in build-in or custom data types from a programming language. In many cases, marshalling techniques may be used to provide compatibility between server data types and client data types. Typically, this is the case when the client and the server are implemented using different technologies. For instance, an image processing service interface can describe a user defined data type (a class in object oriented programming) containing the image name, the image type, the image data (as a specific format), etc. If the service is implemented as a web service, the data types are encapsulated (serialized) in XML documents and send over network using SOAP. Messaging exchange patterns (MEP) can be used for accessing a service. The most common access pattern used is the request/response (also known as request/reply) pattern. In this case, the service consumer sends a request to the service and receives a response. This access pattern is used in the web services applications. The client can use a synchronous or asynchronous communication mechanism. The asynchronous mechanism is preferred when communication costs are high or the network is unpredictable. Another pattern that can be used is publish/subscribe. This pattern is based on the message queue paradigm. For instance, an image capture service allows to other services or clients to subscribe to it. When a new image is captured all subscribers receives the new image. The publish/subscribe pattern is typically used with an asynchronous communication mechanism.

IV. ARCHITECTURE AND IMPLEMENTATION

The core of the system consists of several communicating components: user applications, execution agents, pools of IP algorithms, and management agents. [20] The management agent is the central element. Its main purpose is to realise the network of individual modules (applications and execution agents) and initialise the communication among them. However, the main body of messages is communicated directly among the individual modules. The local cluster can be further expanded through a network of management agents, within the same or even different organisations. Thus, the management agent ensures the scaleability of the environment, a basic requirement of an integrated teleradiology services network [21]. Additionally, the management agent authenticates users and provides unique image ids by using standard digital signature technology.

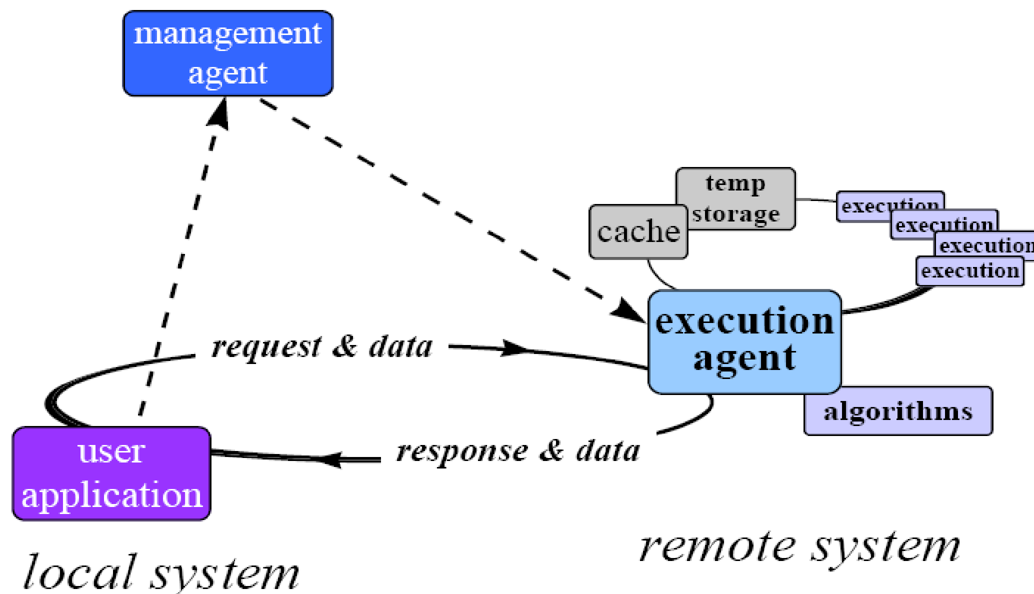

Fig 5 . Communication within a DIPE cluster

The execution agent is responsible for the execution of a specific algorithm. It receives requests for execution through the management agent and creates a communication link with the requesting application in order to receive further information and input data required for the execution (Figure 5). After this point, this agent can proceed autonomously to the execution of the algorithm. It stores input data into a local cache area and executes the requested algorithm. Output generated through the execution of the algorithm is sent back to the agent. The execution agent is responsible to forward this output to the requesting application. In case there is a network failure or the requesting application is not running any longer, the agent keeps the results of the execution in temporary storage for delivery upon request. This ensures persistent algorithm execution and enhances the robustness of the system. The user application is the front end of the system and consists primarily

of a customisable graphical user interface. A virtual temporary storage management module ensures that the application can handle synchronously a considerable number of large data sets. An important feature of the user application is that it incorporates certain image processing algorithms that require real-time response, and thus it is not sensible to redirect their execution to an agent or over the network. These include routines necessary for image visualisation (e.g., zoom, focus, resize, contrast adjustment, etc.), as well as certain algorithms for local, real-time image processing. Finally, the graphical user interface provides toolkits that support the various functionalities of the environment (algorithm insertion, monitoring of the system‟ s status, resource management, macro composition and execution, etc.). A typical screen of the application is shown in Figure 6 . The basic requirement that DIPE is readily implemented on various operating systems and over heterogeneous networks poses certain implementation constraints. Thus, inter-process communication is based on the TCP/IP network protocol, while operating system transparency is ensured by using ACE, an object-oriented network programming toolkit for developing communication software [5]. DIPE is now implemented on UNIX and Windows NT/95 workstations.

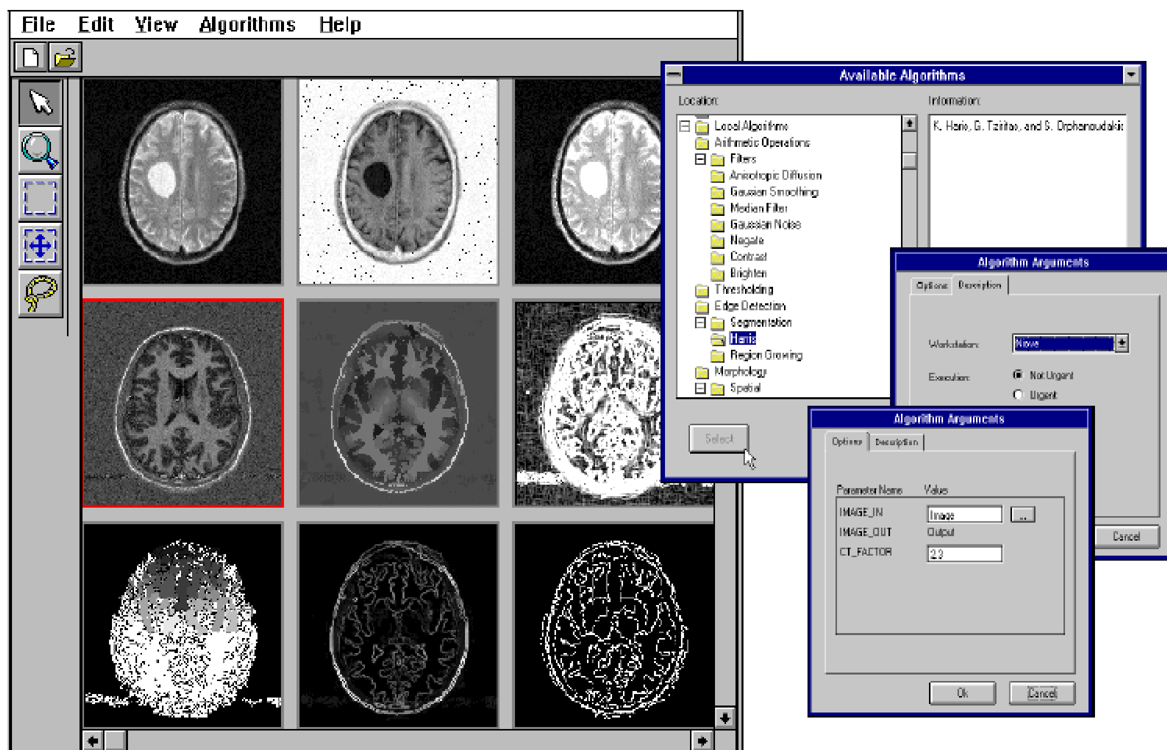

Fig 6. A typical screen of DIPE

1. The Algorithm Repository

The functional core of DIPE is the set of available image processing algorithms, private or public, local or network wide. An important feature of DIPE is that it allows easy integration of third party algorithms, i.e. software modules where only an executable is available and the only information known is the command line syntax, as well as the input and output data formats. The integration is achieved through the algorithm wrapper, a single generic process. The wrapper converts input data from the application format to the format that a specific IP algorithm requires, executes the algorithm and finally converts the output data of the algorithm to the format of the user application. While the algorithm is being executed, the wrapper is responsible to handle requests from the user application. Such requests include the termination or pause of the execution, or the resumption of a previously paused execution. Additionally, DIPE provides a library of ready-to-use routines for the development of new IP algorithms, which consists of basic routines related to the starting and ending phases of the algorithm, as well as of routines that support a more sophisticated mode of user-algorithm communication during execution. In routine medical image processing, a common situation involves processing images using the same set of algorithms often with a standard set of parameter values. DIPE provides the mechanisms to simplify the complicated process of executing individual algorithms sequentially, by grouping them together and thus creating a macro-algorithm (macro). In general, the DIPE macro is a set of individual algorithms that may be performed independently on the same or different data sets, or may be performed sequentially. There is no constraint on the complexity of algorithm combinations and the inter-relationships of their input and output

data. The execution of a macro is the responsibility of a special macro agent. The macro agent acts as a mediator for macro executions. It consists of three main functional parts: the interface with the application, the interface with the rest of the system (management and execution agents), and the module which is responsible for the management of the macro execution. The macro agent models macros as a directed acyclic graph, thus enabling macro decomposition and individual scheduling of its components.

2. Resource Management

Quality of service in DIPE is guaranteed by a sophisticated resource management and execution scheduling mechanism. The scheduling of a requested algorithm execution to the most appropriate processing element (PE) is a distributed decision making process based on the market metaphor, and is realised through the co-operation of the execution agents [16, 19]. Upon request for an algorithm execution, the management agent initialises an „auction" . The request is forwarded to the appropriate „bidders" , that is those execution agents that are able to perform the request. Each execution agent evaluates the request by taking into consideration the load of the local PE, the possible existence of the required input data in its local cache vs. the cost for transferring the data through the network, and the execution characteristics of the particular algorithm. Then, each execution agent makes a bid to the management agent by returning the estimated „cost" of the execution. The management agent evaluates all the bids it receives and assigns the execution to a particular execution agent. It is important to note that the execution characteristics of each algorithm are drawn from its execution profile, which includes information on size of input/output data, PE memory needed at runtime (relative to input data) and time needed for execution (normalised to input data and PE). A good approximation about the memory requirements and the execution time of an algorithm is derived from a statistical analysis based on previous execution profiles of the algorithm.

V. EXPERIMENTAL RESULTS

In this section we present two service implementation using web services standard and OSGi (Open services Gateway Initiative). OSGi [13] is a java-based service platform that implements a dynamic component model (from our point of view, OSGi is a component model).

1. Web service example

The first example is a service implementation according to our model. The service receives an image and returns a grayscale copy of that image. The service interface is named *GrayscaleFilter* (figure 7) and has a single operation, *transformImage*(). The business layer (functional layer) contains 2 components: *GrayscaleComponent* implementing the filter and *BitmapUtilsComponent* used to convert an image to byte array and vice versa. Non-functional aspects of the service like handling incoming connections are treated by the web service used to run our service. Also, on the client side, some tools (like Visual Studio .NET) greatly simplify the work with web services by generating the necessary code to access the service.

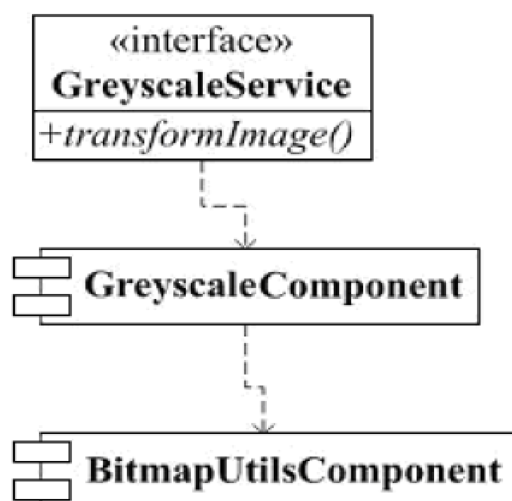

Fig 7. Web service component diagram

The class diagram is represented in figure 8. Every component is implemented by a single class.

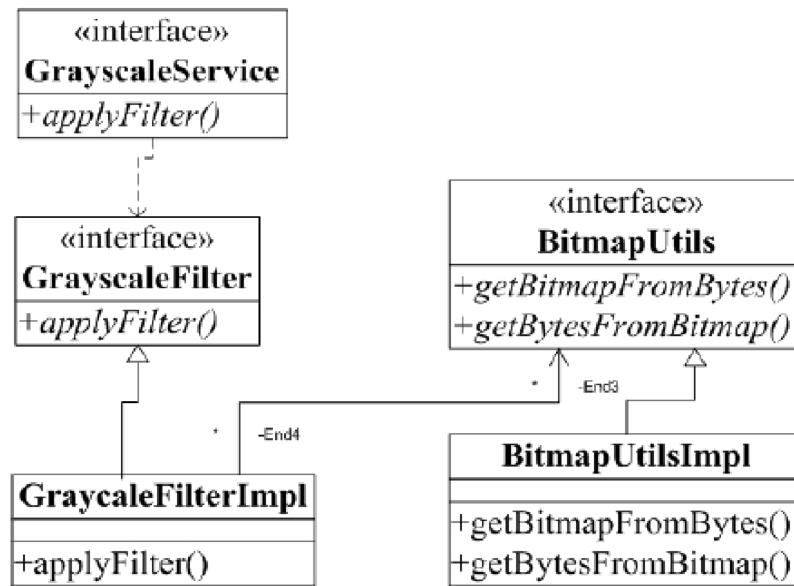

Fig 8. Web service class diagram

Note that in this simple example the business façade from our service model is not used and the resource layer is missing since no external resources are needed.

2. OSGi service example

In order to show that SOA is not based only on web services, the second example is an implementation of an image processing service using OSGi. We are using the *Knopflerfish* framework [14] as support for developing our service. In OSGi a deployment unit is called *bundle*. The framework manages the bundle lifecycle. A bundle functionality is contained typically in a jar file (java archive file). After the bundle is created it needs to be registered in the framework and other bundle can use the published service. Our OSGi service is more complex than the web service because it needs communication facilities (offered by a communication component) because OSGi does not specify a communication protocol like a web service. The component diagram for our service is depicted in figure 9

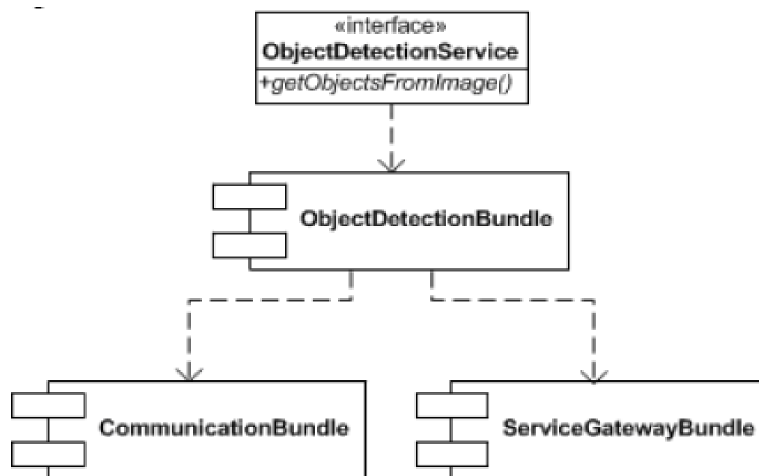

Fig 9.OSGi service component diagram

The service interface is called *ObjectDetectionService* and exposes a single operation, *getObjectsFromImage*. The input parameters (an image) and the return values (a collection of image objects) are not represented on this diagram. The *ObjectDetectionBundle* represent the functional part of the service (business layer). This component uses a communication component and a gateway component. In figure 11 is depicted the class diagram for the communication bundle.

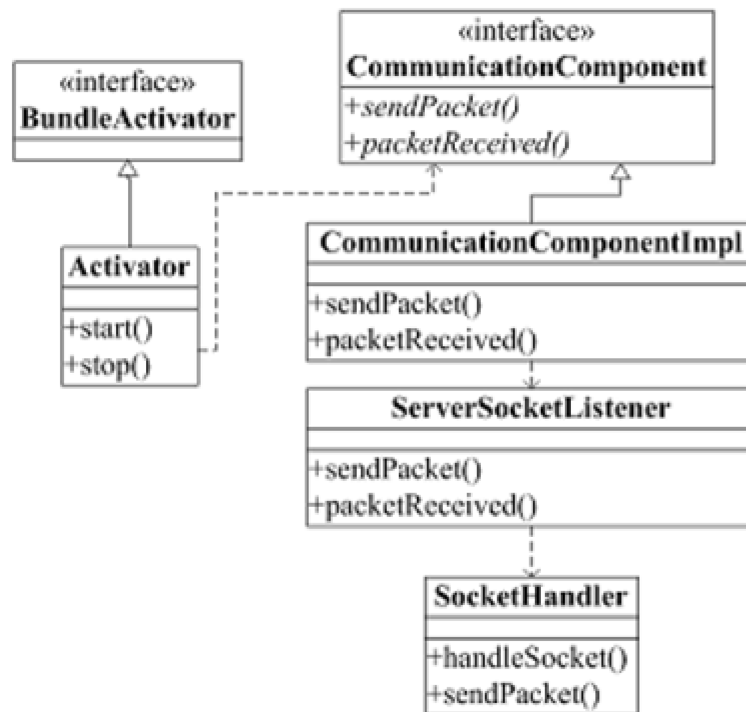

Fig 10. Communication component, class diagram

The component interface is called *CommunicationComponent* and provides two operations, one for sending a packet and the second for receiving a packet. A packet is a unit of information exchanged by the service. In our case the packet contains the image as a byte array. The Activator class implements *BundleActivator* interface and is necessary in order to allow the *Knopflerfish* framework to manage the bundle (start and stop the bundle). To be used, a bundle must be started. The bundle interface has an implementation provided by CommunicationComponentImpl. The communication is based on standard sockets (with the help of ServerSocketListener and SocketHandler). For this service, the resource layer contains a component (*ServiceGatewayBundle*) for accessing other services. The object detection algorithm implemented needs to use a grayscale image in order to provide good results. This component contains the logic to access our grayscale web service presented in section previous.

VI. CONCLUSIONS

In this paper, we have proposed a model for implementing SOA-based image processing systems. The model contains a programming model, a service model and a messaging model. We have focused on the concept of service. The service is represented as a layered architecture with a service interface layer, a business layer and an optional resource layer. The service interface layer contains the service contract (service interface). The business layer contains the service functionality, contained in business components. The resource layer contains non-functional components, used to access external resources like database systems, other services or legacy applications. Service Oriented Systems are very flexible. A service can be discovered at runtime, can be replaced if is unavailable or can be incorporated in a new service (a powerful support for adaptability). Our future goals are to create a SOA based platform for adaptation with applicability in medical domains. This platform may be based on ESB in order to provide full SOA facilities. DIPE has been designed and developed to offer image processing services over integrated health care services networks, and to act as an integration platform for diverse image processing software. It exhibits a modular, extensible and scaleable architecture that ensures system robustness and execution persistence. A sophisticated resource management and execution scheduling mechanism allows the medical expert to take full advantage of geographically distributed computational resources. Future research will address the development of intelligent and customisable mechanisms for the description, management, and retrieval of image processing software modules, as well as charging mechanisms based on quality of service. DIPE is currently being extended through its functional integration with other medical information systems that have been developed in our laboratory. Important examples include CoMed [17], a desktop conferencing application which allows interactive real-time co-operation among several medical experts, as well as TelePACS [16], an information system for medical image

management and communication. DIPE is one of the diverse telematics applications incorporated in the regional health telematics network, which is currently being developed by ICS-FORTH on the island of Crete.

REFERENCES

[1] Michael Herrmann, Muhammad Ahtisham Aslam, Oliver Dalferth, Applying Semantics (WSDL, WSDL-S, OWL) in Service Oriented Architectures (SOA), Universität Leipzig, Germany, Technical report, 2005.
[2] Microsoft Healthcare [online], http://www.microsoft.com/ industry/healthcare, (Jun, 2007).
[3] J. Treadwell, Open Grid Services Architecture Glossary of Terms, Hewlett-Packard, January 25, 2005.
[4] Organization for the Advancement of Structured Information Standards (OASIS), “Service Oriented Architecture (SOA) Reference Model,” Public Review Draft 1.0, February 10, 2006.
[5] Workflow definition [online], http://en.wikipedia.org/wiki/ Workflow (June 2007).
[6] Latha Srinivasan and Jem Treadwell, An Overview of Service-oriented Architecture, Web Services and Grid Computing, , HP Software Global Business Unit, November 3, 2005.
[7] Armin Haller, Juan Miguel Gomez, Christoph Bussler, Exposing Semantic Web Service Principles in SOA to Solve EAI Scenarios, May, 2005.
[8] Rick Robinson, Understand Enterprise Service Bus scenarios and solutions in service-oriented architecture, IBM, Jun 15, 2004.
[9] Patterns: Implementing an SOA Using an Enterprise Service Bus (IBM Redbooks), IBM.Com/Redbooks, 2004.
[10] WS Specifications [online], http://www.w3schools.com/ webservices/default.asp, (Jun, 2007).
[11] Jeffrey Hasan, Expert Service-Oriented Architecture in C#: Using the Web Services Enhancements 2.0, Apress, 2004.
[12] SOAP Specifications [online], http://www.w3schools.com/ soap, (Jun, 2007).
[13] OSGi Service Platform, The Open Service Gateway Initiative, IOS Press, 2003.
[14] OSGi Tutorial A Step by Step Introduction to OSGi Programming Based on the Open Source Knopflerfish OSGi Framework , Sven Haiges, http://www.knopflerfish.org /tutorials, October 2004.
[15] S.C. Oprhanoudakis, E. Kaldoudi, and M. Tsiknakis, “Technological Advances in Teleradiology”, Eur. J. Radiology, vol. 22, 205-217, 1996.
[16] S.C. Orphanoudakis, “Supercomputing in Medical Imaging” IEEE Eng Med Biol, vol. 7, 16-20, 1988.
[17] P. Maes, “Modelling Adaptive Autonomous Agents”, Artificial Life Journal, ed. C. Langton, vol. 1, nos. 1&2, MIT Press, 1994.
[18] S.C. Orphanoudakis, M. Tsiknakis, C. Chronaki, S. Kostomanolakis, M. Zikos, and Y. Tsamardinos, “Development of an Integrated Image Management and Communication System on Crete”. In: Lemke HU, Inamura K, Jaffe CC, Vanier MW, eds. Proc. of CAR‟ 95, Berlin, p. 481-487, 1995.
[19] D.C. Schmidt, “The ADAPTIVE Communication Environment: An Object-Oriented Network Programming Toolkit for Developing Communication Software”, 12th Sun User Group Conference, San Francisco, California, June 14-17, 1993.
[20] D.F. Ferguson, Y. Yemini, C. Nikolaou, “Microeconomic Algorithms for Load Balancing in Distributed Computer Systems.”, . In Proceedings of International Conference on Distributed Systems (ICDCS 88). San Jose, California: IEEE Press, 1988.
[21] M. Zikos, C. Stephanidis, and S.C. Orphanoudakis, “CoMed: Cooperation in Medicine”, Proceedings of EuroPACS‟ 96, pp. 88-92, Heraklion, Greece, October 3-5, 1996.