A General Brokering Architecture Layer and its Application to Video on-Demand over the Internet

Franco Cicirelli and Libero Nigro
Laboratorio di Ingegneria del Software
Dipartimento di Elettronica Informatica e Sistemistica
Universitá della Calabria, I-87036 Rende (CS) - Italy
E-mail: f.cicirelli@deis.unical.it, l.nigro@unical.it

Keywords: service oriented computing, application framework, middleware, peer-to-peer, Internet, video on-demand, Java, Jini, Java Media Framework, RTP/RTCP protocols

Received: February 7, 2006

GOAL -General brOkering Architecture Layer- is a service architecture which allows the development and the management of highly flexible, scalable and self-configurable distributed and service-oriented applications over the Internet. GOAL centres on a design pattern which decouples the design of service functionalities from the distribution concerns. A service wrapper is specifically responsible of the distribution aspects. The wrapper is weaved at runtime to its corresponding service by a dynamic proxy object. GOAL makes it possible to augment, in a transparent way, the behaviour of a software object in order to permit it to be remotely accessible. Current implementation of GOAL depends on Sun Microsystems' Jini as the brokering/middleware layer. The paper describes GOAL and demonstrates its practical use through the design and implementation of a Video on-Demand system. Java Media Framework is used for pumping multimedia data at the transmitter side and for rendering purposes at the receiver side, RTP/RTCP protocols are used for multimedia streaming.

Povzetek: Predstavljen je GOAL – arhitektura za napredne spletne aplikacije, npr. video na zahtevo.

1 Introduction

Service Oriented Computing [1] emerged in the last decade as a computing paradigm centred on the concept of *service* [2, 3] as basic building block. Services are suitable for developing and organising applications for large-scale open-environments. They are effective in improving software productivity and quality, as well as fostering system evolution and maintenance. A service is a coarse-grained software component virtualizing a hardware or software resource which is made exploitable for on-demand use. Binding to the service/resource typically occurs just at the time the component is needed. After usage the binding may be discarded. Applications, tailored according to user requirements, are constructed through a combination and composition [4] of independent, outsourced service components exposing a well-defined interface. Main features of the service paradigm include dynamism and transparency. The former refers to services which can appear or disappear in a community without centralized control, possibly notifying about their presence/absence. This behaviour depends on the use of the so called discovery protocols [5, 6, 7]. The latter feature means that services can be used without knowledge about service provider platforms and service provider locations. Service dynamism and interaction model strongly relate service architectures to peer-to-peer architectures [8]. Service computing relies on the high-level abstraction entities defined by Service Oriented Architecture (SOA) [9, 10] in order to (i) characterize and organize service-based applications and (ii) capture the relationships existing among these entities. Basic entities in a SOA are the *service provider*, the *service client*, and the *service registry* which acts as a broker among clients and providers. Each service, offered by a provider, preliminarily requires to be advertised in a registry in order for it to become subsequently discoverable and utilizable by a client (e.g. a human user or another service).

GOAL, the General brOkering Architecture Layer proposed in this paper, is a novel service architecture allowing the development of highly flexible, dynamic and self-configurable service-based applications. GOAL aims at simplifying the management of service lifecycle by reducing the burden of designing, developing and deploying software objects suitable to work in a distributed context. Different distribution aspects like data consistency, fault tolerance, security and remote communications are treated as cross-cutting aspects. In particular, the latter two concerns are directly addressed by the system and are the responsibility of *proxy* objects. GOAL offers a minimal *framework* [11], easy to understand and use, and a few *meta-services*. A *service design pattern*, enforcing common guidelines for service development, is provided. Designing a new service does not introduce dependencies from a particular API or system components. Weaving customer-

objects and system-objects occurs only during service operation. Meta-services are system entities which allow one to publish, search and use customized services. A service, once advertised, becomes available within a GOAL community. Matching criteria can be specified during a searching phase. In addition, when a new service appears/leaves the community, interested clients can be automatically notified. The notification mechanism ensures system flexibility and scalability and permits the development of highly dynamic and self-adapting software. A service can leave the community due to an explicit removing operation or due to a crash. In the latter case, self-healing properties are carried out using a fail silent model [12] based on a leasing mechanism.

Current implementation of GOAL depends on Jini [7, 13, 14] as the underlying service infrastructure, borrowing advantages of dynamic registration, service lookup, notification of remote events, distributed object access and platform-independence enabled by Java. However, the brokering layer, i.e. Jini, is fully abstracted and can possibly be replaced. Communication among services is based on the exchange of Java objects and fully exploits benefits of *runtime code mobility* [15].

GOAL can be used as the starting point for building further abstraction layers targeted to specific application domains. As a significant example, a Management Architecture for Distributed meAsureMent Services -MADAMS- [16] was developed directly on top of GOAL mechanisms. MADAMS is tuned to the requirements of distributed measurement systems [17, 18, 19]. MADAMS rests on the concept of *measurement service* as the basic abstraction entity modelling a (physical or virtual) measurement instrument, and the concept of *connector* which provides inter-instrument communications. MADAMS also supports recursive service composition. MADAMS was successfully employed for demand monitoring and control [16] and for remote calibration of distributed sensors [20].

This paper describes GOAL features and the general design guidelines for achieving distributed services. As an application, GOAL capabilities are demonstrated through the development of a distributed Video on-Demand (VoD) system [21, 22, 23]. The VoD system depends on Java Media Framework (JMF) [24] which is responsible both for pumping multimedia data into a network connection at a sender side and for presenting multimedia data at a receiver side. Data streaming depends on RTP/RTCP protocols [25].

The structure of the paper is the following. Section 2 summarizes some related work. Section 3 presents the proposed service architecture along with its programming model. In particular, a description about the service design pattern, system security and the catalogue of meta-services is provided. Section 4 illustrates design and implementation and service catalogue concerning the prototyped VoD system. Finally, an indication of directions which deserve further work is furnished in the conclusions.

2 Related Work

As with other technologies suited to the development of distributed systems, ensuring transparency and hiding management of distribution concerns allow developers to focus only on domain-specific problems. For these purposes, different infrastructures and middleware layers have been proposed centred on the service metaphor. Sun Microsystems' Jini [13, 7, 26] permits the construction of service-based applications in terms of fundamental mechanisms of service publication/discovery, leasing management, remote event notification and transaction support.

In [27] an approach based on *tuple-space* [28] for building service frameworks is proposed. Concepts like *actor*, which execute client requests, *virtual resource* and *virtual service* are introduced. Virtual entities enable abstraction layers to be achieved on top of either physical resources or services thus ensuring a common and uniform way for accessing them. Spaces are used to manage (e.g. create, destroy, search) agents, services and resources.

Other solutions are targeted to abstracting and hiding details of the adopted middleware/brokering layer in order to favour its interchangeability. By providing a well-defined set of components (i.e. interfaces and objects) and through code generation mechanisms, Iceni [29, 30] allows an automatic service management in different computing contexts such as Open Grid Service Infrastructure or Jini service community. In [31] a framework is proposed which hides behaviour of underlying transport layer and separates *coordination patterns*, i.e. request/response interactions, from *computational logic*, i.e. service functionalities.

Colombo platform [32] introduces the concept of *servicelet* as the unit of development and deployment. A servicelet is a stateless object corresponding to a single service or to a collection of them. Context information are managed by specific *Servicelet Context* entities which are handled by the runtime system. Management of explicit metadata in the form of machine-readable service descriptions, including functional and non-functional QoS characteristics, is an important characteristic of Colombo. The goal is to avoid generating a gap between the internal representation of service capabilities and the external, interoperable service view which is defined by the service contract.

Sirena framework [33] defines an architecture to seamlessly connect heterogeneous (resource constrained) devices and services furnished by such devices. Sirena comprises an incoherent set of tools having the responsibility of generating service stubs and skeletons, managing service lifecycle, supporting visual composition for service orchestration and so forth.

A different goal is pursued in Arcademis [34] which is a Java-based framework enabling the implementation of modular and highly customizable middleware architectures for specific application domains. A distributed system built on top of Arcademis is structured according to *three abstraction levels*. The first level is constituted by basic components like invokers, which are responsible for emitting

remote calls, or schedulers which are used to possibly order remote calls. Although these are abstract classes and interfaces, Arcademis also provides concrete components that can be used without further extensions. The second level is represented by the concrete middleware platform obtained from Arcademis basic components. The framework defers to this level decisions about serialization strategy, communication and lookup protocols that will be adopted. Finally, the third abstraction level is made up by components which make services available to end users.

In the context of the above mentioned proposals, the original contribution of GOAL is twofold: (i) to allow development of new services without introducing, at design time, bindings to specific framework components (e.g. abstract classes or interfaces), (ii) to transparently handle distribution concerns as cross-cutting aspects. All of this fosters low coupling among entities, system evolution and maintenance in a natural way.

3 GOAL Service Architecture

GOAL addresses all the activities involved in the lifecycle of services by exploiting a specific service design pattern and by using a set of well-defined system components and interfaces having specific roles. A main concern rests in encapsulating and hiding implementation details of core components by using stable interfaces so that if changes occur, e.g. in the middleware layer which is replaced or in the communication protocol stack, no consequence is induced in the implemented and working applications. GOAL components and features are discussed in the following.

3.1 Service Design Pattern

The development of a generic service follows the service design pattern depicted in Fig. 1. Each remote service, i.e.

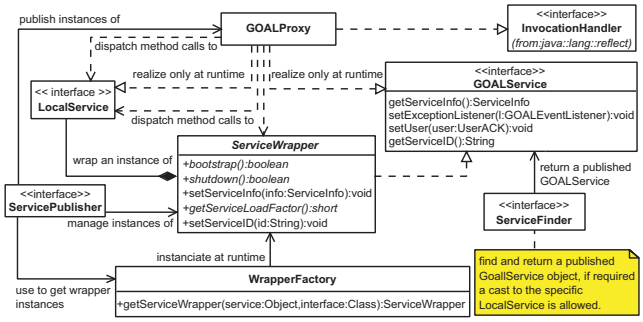

Figure 1: Components of service design pattern.

a new software object made available within a GOAL community, is first developed as a local object. This permits design efforts to concentrate only on the effective service functionalities. In this design phase, the only constraint to fulfil is in defining functional aspects of the new service by means of an interface. One such interface is shown in Fig. 1 as the *LocalService* interface. Interfaces allow a what-how decomposition [35] which ensure service client code immutability with respect to service implementation changes. Any object can be a candidate for a remote service because no restrictions are introduced in the definition of its functional behaviour except for the serializability of the parameters appearing in method signatures. Remote concerns are managed by means of a service wrapper. This kind of object extends local service behaviour with distribution aspects like transaction support and so forth. As a common guide line, the wrapper may enfold the local service and execute distributed tasks by interleaving them with method calls on the wrapped service. A service wrapper can require to be bootstrapped, for instance by initializing its internal state with information retrieved by contacting other services. A shutdown operation allows a wrapper to tear down, i.e. becoming out of work or unpublished. All of this is reflected in Fig. 1 by the *ServiceWrapper* abstract class. Other common functionalities allow: (a) setting the service identifier, (b) setting service info (e.g. a description of service behaviour and functionalities) and (c) managing an estimated load factor value of the service provider. By default, the above concerns are system addressed. When no special requirements have to be met, a *DefaultWrapper* can be transparently used. Would new functionalities be added to the local service, e.g. in order to cope with data consistency and integrity among multiple system nodes, an extension of the default wrapper may be supplied. At compile time, the local service interface and the relevant wrapper may be completely unrelated. A wrapper has to override only the local service operations whose capabilities require to be extended. Only at runtime, the wrapper and the local service behaviour will be weaved according to an aspect-oriented programming style [36]. Two problems arise when making a local service remotely accessible: (i) the service has to be advertised into a community, (ii) a representative object for the service, i.e. a *GOAL proxy*, is required to be downloaded on service client in order to support remote communications. Service advertisement is the responsibility of the *ServicePublisher* meta-service (see Fig. 1). Service finding and proxy download activities are in charge of the *ServiceFinder* meta-service (see Fig. 1). The proposed publisher/finder mechanisms help in hiding details about the actual brokering layer. Would the brokering layer be replaced, e.g. CORBA used instead of Jini, only the publisher/finder objects have to be correspondingly modified. While publishing a local service, behind the scene the service publisher (i) asks to a *WrapperFactory* for a wrapper instance, (ii) correlates service and wrapper with the proxy and (iii) makes the latter one object available to a GOAL community using functionalities of the actual brokering layer. The *GOALProxy* (see Fig. 1) is a remotely accessible object which, moving to a service client host, transparently acts as a dispatcher of request/response messages between remote user and local service provider. In the case the communication protocol changes, e.g. XMLRPC is preferred to Java RMI, only

the proxy requires to be changed. By preferring the execution of overridden methods, the proxy realizes the interweaving among local service and the corresponding wrapper. At runtime, by exploiting Java dynamic proxy mechanisms [37], a GOALProxy is able to implement a list of specified interfaces without requiring code generation. Implementing a specific interface of a local service ensures that a generic client would not be able to perceive any difference between direct service usage with respect to proxy mediate usage. The sequence diagram in Fig. 2 summarizes the effects of the service design pattern on service publication and utilization. Figure 3, instead, depicts communication details characterizing interactions among service client and the associated service provider. A GOAL-

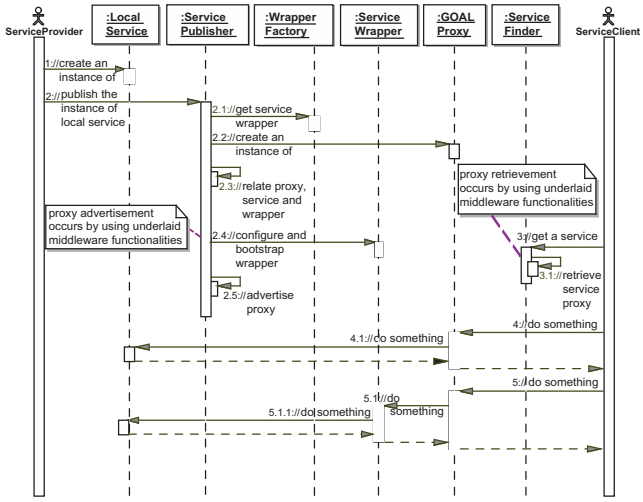

Figure 2: Sequence diagram capturing service utilization.

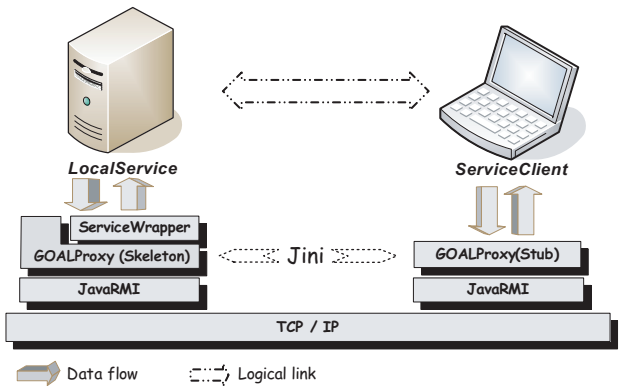

Figure 3: GOAL service usage scenario: communication details.

Proxy may enforce the download of the entire service code on a user node. This is useful when the service must be executed on the client side, e.g. for accessing to hardware or software resources hosted on a particular computing node [20]. Proxy behaviour is defined during publication simply by setting some of the so called *GOALServiceProperty(s)*. If no constraints appear in the object serializability or persistence, a service may be used according to *remote* or *downloadable* mode. A GOALProxy may be specialized in order to augment the capabilities of the overall system. For instance, to deal with fault-tolerance concerns, a proxy can be designed to abstract communications between a single client and a group of equivalent service providers, so as if one of the provider becomes unavailable or crashes, the client remains able, in a transparent way, to continue its work [38]. The class diagram of the service design pattern (see Fig. 1) makes also clear that, once published, a local service becomes a *GOALService*. GOALService interface defines a set of functionalities which are common to all services in a GOAL system. In particular, the setExceptionListener method is used to set a listener whose aim is to handle exceptions not thrown by local service methods but raised during remote operation. The setUser method is used to set a *UserACK* object especially devoted to cope with security concerns (see section 3.2). Remaining operations should be self explanatory. Other common issues of the service design pattern are user *friendliness* and *load balancing* support. Each service may possess a graphical user interface (GUI) obtained through a *GUIfactory* object previously published by using the service publisher (see also Fig. 6). Service finder is then used by clients in order to search and retrieve the factory. Advantages of this approach are: (i) a service is developed independently from its graphical interface, (ii) the GUI is instantiated only on the client side thus avoiding serialization of graphical objects, (iii) the GUI allows use of the service without any previous knowledge about it, (iv) multiple graphical user interfaces, possibly tied to different user node capabilities, can be supported. Load balancing is carried out by using the so-called *remote service attributes*. Every service has one of such an attribute that expresses its load factor, i.e. NORMAL for a low or normal load factor, and WARNING or BUSY for a high/very high load factor. Although services are usually supposed to remain context-free, remote attributes can provide a kind of context information [39, 40] exploitable during the finding phase. For instance, the service finder always tries to return services in the NORMAL state, if there are any, otherwise the first one matching searching criteria is returned. The service publisher keeps up to date remote attributes by periodically querying service wrappers state or (possibly) by monitoring the CPU usage on provider nodes.

3.2 Security Concerns

Handling security is an essential issue in a distributed and multi-user scenario. The service code downloaded from a remote site requires to be trusted along with the remote site itself. User credentials must be verified, usage of system resources granted and resource accesses controlled. Authentication and authorization mechanisms of GOAL are im-

plemented through the UserACK object (see Fig. 1) which permits user identification and acknowledgment of its roles (e.g. administrator or normal user), privileges and grants. The concept of *user groups* is introduced and users may become members of one or multiple groups. Each group, e.g. admin group, owns some *GOAL permissions* and the UserACK holds the union of all permissions relevant to the user joined groups. Information stored in a permission follow the same hierarchical schema adopted in Java package specifications. Creating a permission with a service package info and service name enables access to the corresponding service. By providing only package information, a grant is given to all services belonging to the package. A finer authorization control is achieved by specifying service method/function name(s) in the permission. The use of UserACK makes it possible, in a decentralized context, to accept or discard a user request. User grants are checked directly by GOAL proxies. Therefore, authentication and authorization concerns are transparently managed with respect to service functionalities and service implementation. During publication, a specific GOALServiceProperty can be used to state if the proxy has to enable or disable the management of security concerns, i.e. to state if the service has to be considered *secure* or *public*. In the case security aspects are to be explicitly managed by a service, the UserACK object must be transmitted as a parameter when invoking its methods.

Users can freely propose new groups and create their own UserACK. However, only *signed* UserACKs and *accepted* groups can be effectively used. A system administrator signs a new UserACK and establishes its expiration time. Signed UserACKs cannot be modified by users: the system is able to recognize when a UserACK is corrupted, modified or just invalid (e.g. it expired). UserACK and group management is responsibility of the core downloadable *Grant Management service* whose GUI is shown in Fig. 4. A UserACK submission form is offered and group membership is achieved by choosing items from a list of already accepted groups. Inspection of group properties is allowed. Likewise to UserACK, a group submission form is also available. Submitting a new group requires the group name and the list of group permissions to be provided. A reserved area, offering an overall vision of existing UserACKs and groups, is under the control of the system administrators (see Fig. 5). New UserACKs/groups can be signed/accepted and single permissions can be added/removed in a group as well as in a submitted UserACK.

The accesses to a service can be also allowed or denied depending on other criteria. Load balancing aspects, service availability or service exclusive use may often be considered during the UserACK acquisition phase. Confidentiality and privacy can be ensured by using the Secure Socket Layer for service communications, whereas trustness can be achieved by exploiting Java standard security mechanisms relevant to remote code management [41].

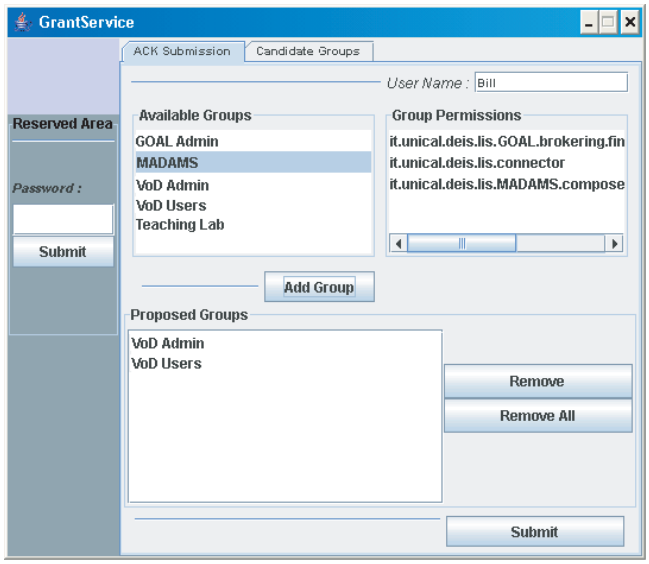

Figure 4: Grant Management service GUI.

3.3 Meta-Service Catalogue

GOAL meta-services are responsible for publishing, searching, retrieving or removing services from a community. Figure 6 portrays the UML class diagram of the publisher/finder services which depend only on interfaces. Actual objects are created by a singleton factory which ensures a coherent delivering according to the underlying middleware layer. To cope with security concerns, a valid UserACK is required when using meta-services. Only the method find(String):GOALService can be used without a UserACK. This provides a bootstrap mechanism exploitable by new users for contacting the service devoted to grant management in order to obtain the personal UserACK. The advertisement process is carried out by requiring the service to publish and a list of GOALServiceProperty. These properties allow to specify descriptive and behavioural attributes. Descriptive attributes may be used, for instance, to provide service description. Behavioural attributes must be used to specify the name of the interface through which the service will be retrieved by clients, the wrapper to use and so forth. Following a successful invocation, the publish method returns the unique service identifier. A service may be published using different interfaces thus allowing multiple views of the same local object. Among service properties it is also possible to specify a service working directory which will be used, by the system, for storing persistent information like the service identifier. Service properties may also be labelled as searchable. In this case, properties may be specified as matching attributes during the finding phase. The publishServiceUIFactory method is used to publish the UIFactory of a specified service, unpublish is used instead to remove a service from the community. Finding a service requires the service name, i.e. its interface name, and (possibly)

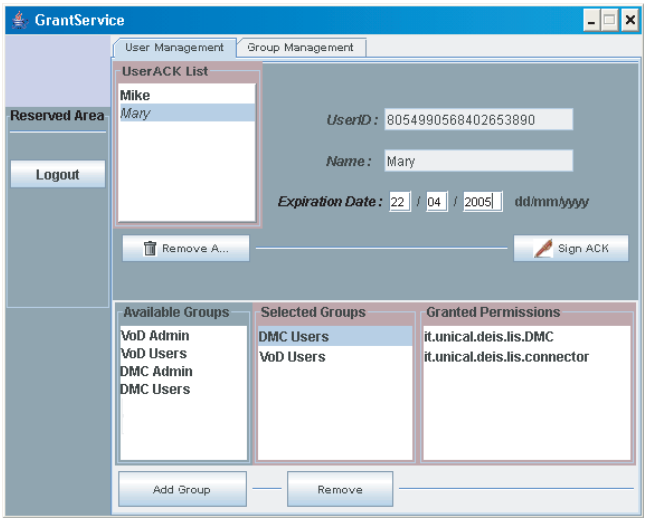

Figure 5: Administration panel of the Grant Management service.

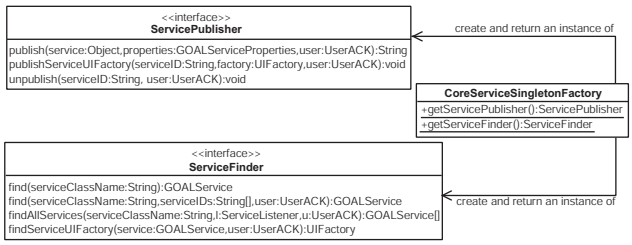

Figure 6: Publisher/finder service design.

service identifier information. Although the finding process is based on naming criteria, the matching can also occur when a published service implements any interface in a hierarchy. As a side-benefit, the use of textual name and the availability of service GUI enable usage of any published service without requiring specific Java code to be installed on the client node. The findAllService method allows service retrieval by bypassing the load balancing policy. If specified, a ServiceListener (see Fig. 6) notifies when a new searched service joins or leaves the community. Meta-services require their code to be pre installed on every GOAL node.

4 A GOAL-based VoD System

The following describes a VoD system developed on top of GOAL. The VoD system consists of a service federation which permits publishing, searching and retrieving as well as streaming and rendering of multimedia contents. Java Media Framework [24] is used for pumping (at provider side) and rendering (at client site) multimedia data. Streaming of multimedia contents relies on the RTP/RTCP protocols [42]. First the service architecture is described, then the list of developed services for the VoD system is provided.

4.1 System Architecture

The architecture of the achieved VoD is depicted in Fig. 7. It consists of five types of computing nodes having different roles in supporting VoD services. Nodes, and relevant services, can dynamically join or leave the system and when this occurs the other nodes are notified. Some

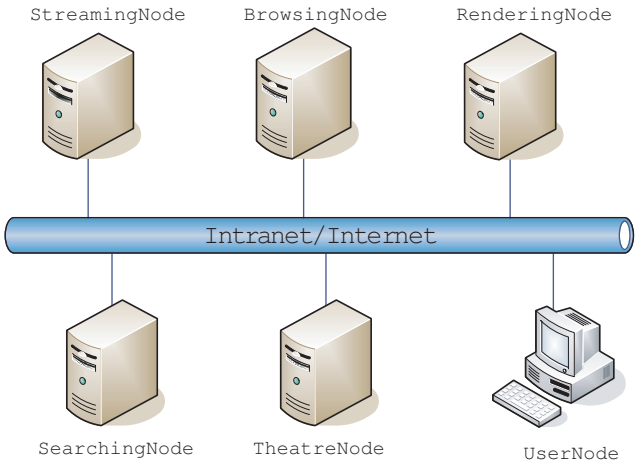

Figure 7: Architecture of the GOAL-based VoD system.

nodes may be duplicated: multimedia files and related descriptions are normally distributed across multiple *streaming nodes*. Other kind of nodes, instead, may be duplicated for fault-tolerance and load balancing issues. VoD services are ultimately requested and made it available to final users through *user nodes* (see Fig. 7). The architecture was designed so as to minimize code requirements on the user nodes. Here only some standard code like JMF and obviously GOAL meta-services code, is supposed to be statically available. All of this contributes to system evolution because, by exploiting the download of the service code, a user, on-demand, will always use the latest version of the various software components. A description of each node is provided in the following.

Streaming nodes are media servers. They contain multimedia data files and associated descriptions (e.g. title, authors etc.). Streaming nodes enable to: (i) access the descriptions of media data; (ii) add/remove multimedia files; (iii) create and manage multimedia sessions (for data streaming and control) on behalf of end users.

Browsing nodes respond to management functionalities. Relevant services offer a unified list of the multimedia content available on the various streaming nodes and allow users to organize multimedia data across them. The organization consists in adding/removing/modifying multimedia contents on different streaming nodes.

Searching nodes portray a whole vision of all the existing multimedia contents by providing: (i) the unified list of available media files distributed across streaming nodes; (ii) searching facilities, e.g. for selecting specific movies; (iii) user profiles in order to tailor media information on a per user basis or to send notifications when relevant new media data come into existence; (iv) trace facilities about media utilizations like reviews, user preferences and so forth.

Rendering nodes act as remote libraries from which user nodes can dynamically download the code required for rendering a video content, possibly by ensuring also receiver based QoS control, e.g. lip-sync [43].

Theatre nodes provide a service which is used as entry point for user interactions. In order to view a movie a user has to (i) searching it by using searching node functionalities, (ii) starting and managing multimedia sessions by using streaming node services, (iii) managing the rendering process on the user node by retrieving and using rendering libraries downloaded from a rendering node. All of this requires the utilization and the coordination of multiple VoD services which in turn are provided by different computing nodes. By using the service exported by a theatre node, a user obtains an holistic vision of the entire VoD system. In this way, issues concerning single service invocation and coordination are fully abstracted.

4.2 Service Catalogue

SessionController and VideoFileManager

Are specific of streaming nodes. SessionController negotiates and creates a *multimedia session* between a client node and a streaming server node, with distinct *control* and *streaming* bindings. The streaming binding is used for media data streaming, e.g. unicast, and relies on the RTP/RTCP protocols [25]. A negotiation phase is required for establishing port identifiers at both receiver and transmitter side. The control binding is TCP-based and is used for exchanging session control commands (e.g. play, rewind, pause and stop). SessionController does not require its functionalities to be extended for remote access. Therefore, the DefaultWrapper can be transparently used during the publication phase. SessionController service has a VCR-like GUI which is automatically made available at the end of the negotiation phase.

The VideoFileManager service is mainly devoted to adding/removing media files to/from a specific streaming node and managing media file information, e.g. title, director and language. Information about duration, file encoding and so forth are automatically detected and made available by the service. Media information are stored in XML format. VideoFileManager also notifies a set of listeners when, for instances, a new movie is added. A complete list of available movies is also provided. In order to enforce data consistency, listeners require to be notified under transactional support. Transaction management is responsibility of the VideoFileManagerWrapper (see Fig. 8) and relies on the Jini transaction mechanism. The class

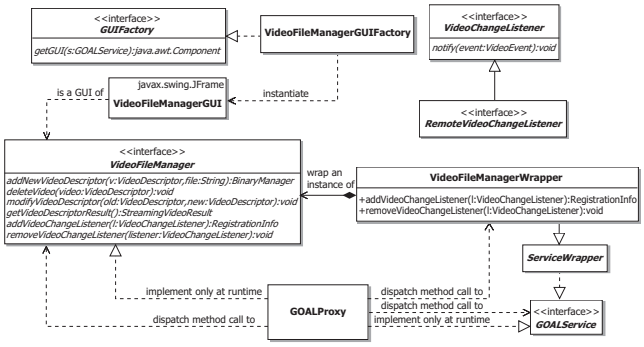

Figure 8: Class diagram of VideoFileManager and related entities.

diagram in Fig. 8 makes clear that the wrapper and the corresponding service are unrelated at compile time, i.e. they do not implement or extend any common entity. As discussed in section 3.1, the weaving between the two objects is only established at runtime by means of a GOALProxy. During the bootstrap phase (see section 3.1), the wrapper registers itself as a listener of the VideoFileManager. Subsequently, it will act as a dispatcher of notifications coming from the wrapped service and going toward remote listeners. By overriding the methods addVideoChangeListener and removeVideoChangeListener, the wrapper obtains to be the exclusive manager of the RemoteVideoChangeListener(s) which are handled under transaction. A *RemoteVideoChangeListener* is a listener whose functionalities are extended to support notification and management of remote events. In addition, the remote listener behaves as a transaction participant when notified by a transaction client [26], i.e. a VideoFileManagerWrapper. To enforce self-healing properties, the registration of a remote listener is regulated by a lease. As one can see, all the methods reported in Fig. 8 makes no use of UserACK objects, this is because security concerns are transparently handled by the GOALProxy. Figure 8 also shows the relationship existing between VideoFileManager and its GUI. Figure 9 depicts the GUI of a VideoFileManager service while an upload of a new movie occurs.

StreamSearcher

It is specific of searching nodes and provides a searching facility allowing a uniform access to all the media data available on existing streaming nodes. A StreamSearcher enriches media data with information about user activities and collects user reviews and profiles. It acts as a listener of events coming from VideoFileManager, e.g. informing that a new movie has been added to the video library, or coming from other StreamSearcher, e.g. informing that another review was added. This is reflected in the class diagram reported in Fig. 10 where a StreamSearcher interface extends the VideoChangeListener interface. As one can see,

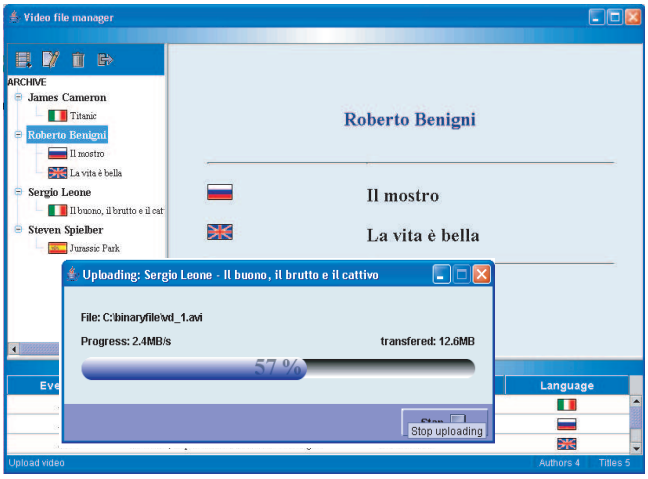

Figure 9: VideoFileManager GUI.

some methods of StreamSearcher require a UserACK object as parameter. Although security concerns are always managed by the proxy, one of such an object is required for tracing user's activities. Likewise to the VideoFileManager

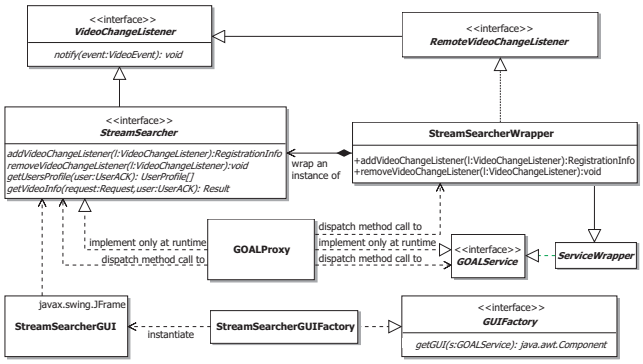

Figure 10: Class diagram of StreamSearcher and related entities.

service, a StreamSearcherWrapper (see Fig. 10) was introduced for guaranteeing consistency and integrity of the data exchanged with other services, i.e. VideoFileManager(s) and StreamSearcher(s). In this case, the wrapper extends the RemoteVideoChangeListener interface. In particular the wrapper registers itself as a listener of the enfolded service and as listener of the VideoFileManager and StreamSearcher working in the service community. At the bootstrap phase (see Figg. 1 and 2) the wrapper is in charge of initializing its own media data repository. If other searching nodes are available, a data mirroring is performed, otherwise it has to contact all the VideoFileManager(s) in order to retrieve info about movies. Media data and the so called enriched media data (i.e. title, authors, user reviews, and so forth) are represented in XML and stored in an XML DBMS such as eXist [44]. Figure 11 shows an interaction with the StreamSearcher service with a specification of searching criteria for finding a movie. The wrapper acts either as a transaction participant or a transaction client. Only at transaction commit, data received from other nodes are transmitted to the enfolded service.

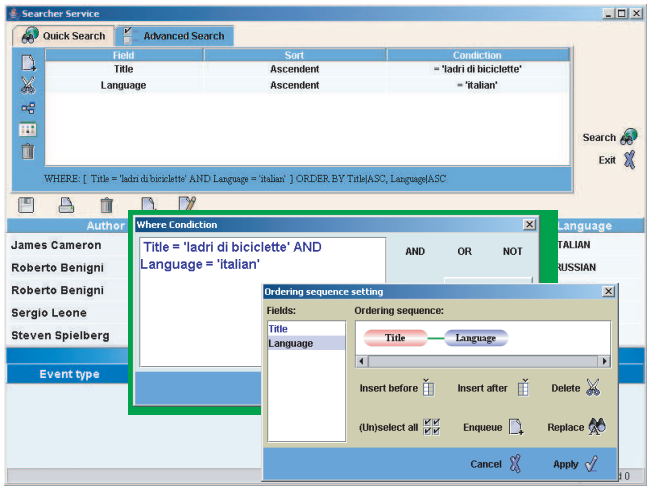

Figure 11: StreamSearcher GUI.

Renderer

It is specific of rendering nodes and, in the context of a multimedia session, it assists the audio/video rendering process on a client node. The rendering process can possibly be accompanied by a receiver-based QoS control filter [45]. Within an allowed end-to-end delay, one of such a filter separately buffers incoming audio/video packets, assembles media frames and synchronizes media frame presentation on the basis of their presentation time (this is achieved by elaborating either timestamps of RTP packets and report packets of the RTCP sender in order to periodically adjust the real-time clock of the receiver subsystem to the real-time clock of the sender). Too late arriving or corrupted packets are discarded. The filter is capable of controlling intra-medium jitter and inter-media skew directly affecting the lip-synch problem [43]. Rendering service allows management of volume and video zoom factor as well as shows reproduction time of the rendering movie. This service requires to be fully downloaded on client node and no remote functionalities are added.

Browser

Specific of Browsing nodes, this service allows the listing of the VideoFileManager available into the VoD system. This service is only for management purposes i.e. selecting a streaming node to administrate. Browser service requires to be fully downloaded on a client node and no remote functionality is added.

Theatre

Specific of Theatre nodes, this composed service provides added value to final user activities. Behind the scene a theatre asks for a searching service and for a rendering service as well as, once a movie is chosen, for the right session controller service in order to start and manage the incoming multimedia session. A negotiation phase is required between the SessionController and the Renderer for according IP addresses and port numbers. Theatre is a downloadable service which does not require remote functionalities to be added and the DefaultWrapper is used during its advertisement. The theatre service does not have an own graphical interface: it supports user interaction through the GUI(s) of the component services (see Fig. 12).

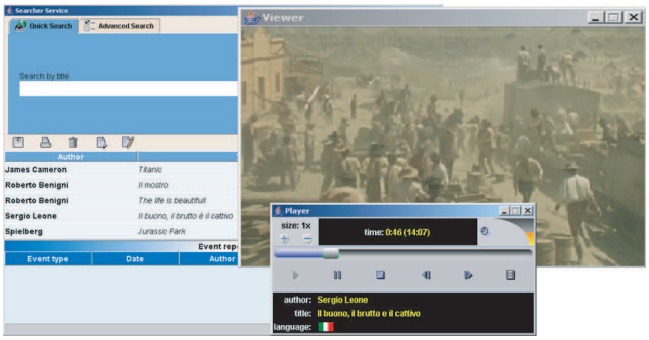

Figure 12: Theatre service vision.

5 Conclusions

The General brOkering Architecture Layer facilitates the development of general-purpose distributed service-based applications. GOAL is based on a service design pattern and makes an application development independent with respect to a specific middleware technology. By keeping a clear separation between local vs. remote concerns and by exploiting the service metaphor GOAL fosters software evolution and maintenance. Development time and design efforts as well as the initial background required for making an application remotely usable are very small. Management, though, of more complex distribution concerns like transaction support, requires a deeper knowledge about GOAL components and the underlying middleware layer. GOAL mechanisms have been successfully experimented in the realization of significant applications like distributed measurement systems [16, 20]. This paper reports about the achievement of a Video on-Demand system over the Internet. Current implementation of GOAL depends on Java/Jini technology. Directions of further work include the following:

- specializing service proxies with the purpose of allowing interoperability between GOAL services and Web Services [46]
- introducing design by contract [47] by foreseeing pre-conditions and post-condition to be transparently managed via service proxy when calling a service method
- making it available functionalities for supporting long term transaction [48] by offering a coordinator core-service accepting a list of methods to be managed under transaction
- adding management of non functional aspects [49], such as service availability, service response time and throughput, either in the service advertisement or within the finding process
- extending the VoD system in order to support multicast and cooperative multimedia sessions [23].

References

[1] M.P. Papazoglou and D. Georgakopoulos. Service oriented computing. *Communications of the ACM*, 46(10):24–28, 2003.

[2] K. Bennett, P. Layzell, D. Budgen, P. Brereton, L. Macaulay, and M. Munro. Service-based software: the future for flexible software. In *Proceedings of the Seventh Asia-Pacific Software Engineering Conference (APSEC'00)*, pages 214–221, Washington, DC, USA, 2000. IEEE Computer Society.

[3] R. Perrey and M. Lycett. Service-oriented architecture. In *Proceedings of the Symposium on Applications and the Internet Workshops (SAINT'03 Workshops)*, pages 116–119. IEEE Computer Society, 2003.

[4] J. Yang and M. P. Papazoglou. Service components for managing the life-cycle of service compositions. *Information Systems*, 29(2):97–125, 2004.

[5] E. Guttman, C. Perkins, J. Veizades, and M. Day. Service location protocol, version 2, rfc 2608. `http://www.ietf.org/rfc/rfc2608.txt`. Accessed on October 2005.

[6] UPnP. Universal plug and play device architecture. `http://www.upnp.org/download/UPnPDA10\_20000613.htm`. Accessed on October 2005.

[7] W.K. Edwards and W. Edwards. *Core Jini*. NJ: Prentice Hall, second edition, 2001.

[8] R. Schollmeier. A definition of peer-to-peer networking for the classification of peer-to-peer architectures and applications. In *Proceedings of the First International Conference on Peer-to-Peer Computing (P2P '01)*, pages 101–102, Lingkoping, Sweden, August 2001. IEEE.

[9] M.P. Papazoglou. Service-oriented computing: Concepts, characteristics and directions. In *Proceedings of the Fourth International Conference on Web Information Systems Engineering*, pages 3–12. IEEE Computer Society, December 2003.

[10] M. Shaw and D. Garlan. *Software architecture: perspective on an emerging discipline*. Prentice-Hall, 1996.

[11] M.E. Fayad and D.C. Schmidt. Object-oriented application framework. *Communications of the ACM*, 40(10):32–38, 1997.

[12] D. Cotroneo, C. Di Flora, and S. Russo. Improving dependability of service oriented architectures for pervasive computing. In *Proceeding of the Eighth International Workshop on Object-Oriented Real-Time Dependable Systems (WORDS'03)*, pages 74–81. IEEE Computer Society, 2003.

[13] Sun Microsystems. Jini network technology - specifications (v2.1). `http://www.sun.com/software/jini/specs/index.xml`. Accessed on May 2006.

[14] Jini network technology. `http://www.sun.com/software/jini/`. Accessed on October 2005.

[15] A. Carzaniga, G.P. Picco, and G. Vigna. Designing distributed applications with mobile code paradigms. In *Proceedings of the 19th International Conference on Software Engineering*, pages 22–32. ACM Press, 1997.

[16] F. Cicirelli, D. Grimaldi, A. Furfaro, L. Nigro, and F. Pupo. MADAMS: a software architecture for the management of networked measurement services. *Computer Standards & Interfaces*, 28(4):396–411, 2006.

[17] D. Grimaldi, L. Nigro, and F. Pupo. Java based distributed measurement systems. *IEEE Transactions on Instrumentation and Measurement*, 47(1):100–103, 1998.

[18] A. Furfaro, D. Grimaldi, L. Nigro, and F. Pupo. A measurement laboratory over the internet based on Jini. In *Proceedings of the Twelfth IMEKO TC4*, pages 479–501, 2002.

[19] W. Winiecki and M. Karkowski. A new Java-based software environment for distributed measuring systems design. *IEEE Transactions on Instrumentation and Measurement*, 51(6):1340–1346, 2002.

[20] F. Cicirelli, A. Furfaro, D. Grimaldi, and L. Nigro. Remote sensor calibration through MADAMS services. In *Proceedings of the IEEE International Workshop on Intelligent Data Acquisition and Advanced Computing Systems: Technology and Applications (IDAACS'05)*, Sofia, Bulgaria, 2005.

[21] L.A. Rowe, D.A. Berger, and J.E. Baldeschwieler. The Berkeley Distributed Video on-Demand System. In T. Ishiguro, editor, *Proceedings of the Sixth NEC Research Symposium*, pages 55–74. SIAM, 1995.

[22] K.C. Almeroth and M.H. Ammar. The Interactive Multimedia Jukebox (IMJ): a new paradigm for the on-demand delivery of audio/video. In *Proceedings of the Seventh International World Wide Web Conference (WWW7)*, pages 431–441, 1998.

[23] G. Fortino and L. Nigro. ViCRO: an interactive and cooperative videorecording on-demand system over Internet MBone. *Informatica*, 24(1):97–105, 2000.

[24] Java Media Framework. `http://java.sun.com/products/java-media/jmf/index.jsp`. Accessed on November 2005.

[25] C. Crowcroft, M. Handley, and I. Wakeman. *Internetworking Multimedia*. UCL Press, London, 1999.

[26] R. Flenner. *Jini and JavaSpaces Application Development*. SAMS, first edition, 2001.

[27] J. Jang. An approach to designing reusable service frameworks via virtual service machine. In *Proceedings of the Symposium on Software reusability (SSR'01)*, pages 58–66. ACM Press, 2001.

[28] N. Carriero and D. Gelernter. *How to write parallel programs*. MIT Press, 1990.

[29] N. Furmento, W. Lee, A. Mayer, S. Newhouse, and J. Darlington. ICENI: an open grid service architecture implemented with Jini. In *Proceedings of the ACM/IEEE Conference on Supercomputing*, pages 1–10. IEEE Computer Society Press, 2002.

[30] N. Furmento, J. Hau, W. Lee, S. Newhouse, and J. Darlington. Implementations of a Service-Oriented Architecture on top of Jini, JXTA and OGSI. In *Proceedings of the Second Across Grids Conference*, pages 90–99. Springer-Verlag, 2004.

[31] L. Fuentes and J.M. Troya. Towards an open multimedia service framework. *ACM Computing Surveys (CSUR)*, 32(1):24–29, 2000.

[32] F. Curbera, M. J. Duftler, R. Khalaf, W. A. Nagy, N. Mukhi, and S. Weerawarana. Colombo: Lightweight middleware for service-oriented computing. *IBM Systems Journal*, 44(4):799–820, 2005.

[33] H. Bohn, A. Bobek, and F. Golatowski. SIRENA - service infrastructure for real-time embedded networked devices: A service oriented framework for