SOS: A Framework for Software Reuse over Open Networks

Glêdson Elias da Silveira
Federal University of Rio Grande do Norte
ges@dimap.ufrn.br

Silvio Lemos Meira
Federal University of Pernambuco
srlm@di.ufpe.br

Abstract

Software engineering research has demonstrated that software reuse can lead to higher productivity, better quality and smaller time to market. In addition, it is known that the benefits increase when reuse is carried out across multiple systems, projects, and even organizational boundaries. In such scenario, where components must be delivered to geographically distributed producers in short time and at low cost, traditional in-house libraries of reusable components are absolutely unsatisfactory.

This paper introduces SOS (Software Operating System), a framework for supporting on demand, dynamic distribution and integration of distributed, versioned, reusable components on both producers and users environments. The framework combines hypertext-based Web concepts and mobile code abstractions to define a transparent, distributed component library, which defines the basis for large scale, systematic reuse of software components during development, deployment, execution and evolution of software systems.

Key-works: software reuse, software distribution

1. Introduction

Software engineering research has demonstrated that software reuse can lead to higher productivity, better quality and smaller time to market. It has been argued that an important characteristic of infrastructures supporting reuse is the existence of a marketplace that provides access to reuse for both producers and users [1]. However, software industry has been unable to mobilize such infrastructure of production for reusable components [2].

In addition, it is known that the benefits increase when reuse is carried out across multiple systems, projects, and even organizational boundaries, enabling higher levels of commonality and spreading reuse costs [3]. In such scenario, where components must be delivered to geographically distributed producers in short time and at low cost, traditional in-house libraries of reusable components are absolutely unsatisfactory.

On the other hand, the Internet has been recognized not only as a tool for communication in the new millenium but also as an environment for enabling changes in computing and distribution paradigms. For instance, Web-based technologies make possible new classes of software systems supporting efficient, timely delivery of content to interested parties.

Therefore, in a context where producers are spread over the Internet, reuse-driven approaches based upon local libraries can make use of Web technologies to support a more powerful and useful approach based upon distributed component libraries. In such scenario, distribution issues play a fundamental role and possibly will radically change the conception, deployment and maintainability of software systems. In such approach, distribution means the efficient, automatic delivery of components to interested parties, including producers and users.

Handling software systems as monolithic entities, current reuse infrastructures install the whole set of components comprising applications, even when users require just a small portion of their functionality. In distributing settings, such approach must be radically changed since transferring enormous applications strains storage and bandwidth utilization, even more severely so when dynamic management of updates is required.

As commonality across software systems is predicted on the order of 60-to-70 percent [3], reuse infrastructures based upon distributed libraries must intrinsically support distribution capabilities for retrieving individual, high granularity reusable components according user needs and on demand. Such capabilities reduce storage and bandwidth utilization and help supporting notions like customizability, adaptability, extensibility and evolvability - all of them essential to extend and adapt software systems in order to meet changing user needs.

This paper introduces SOS (Software Operating System), a framework for supporting on demand, dynamic distribution and integration of distributed, versioned, reusable components on both producers and users environments. The framework combines hypertext-based Web concepts and mobile code abstractions to define a transparent, distributed component library, which defines the basis for large scale, systematic reuse of software components during development, deployment, execution and evolution of software systems. Like the Web, which moves distributed reusable resources to clients, SOS allows software systems to locate, retrieve, install and execute remotely available reusable components on user desktops.

In such scenario, cataloging, publishing, retrieving and executing distributed reusable components are easily and efficiently achieved by adherent platforms, which allow producers to register software components in a distributed component library and users to acquire software licenses and have them automatically installed and running. Therefore, SOS is a true way for manufacturers to provide genuine plug-and-play software systems.

The remainder of this paper is organized as follows: next section presents a comparison with related work. Sections 3 and 4 present the architectural framework and the operational model, respectively. In section 5, implementation issues of a prototype environment are presented. Section 6 in conclusion presents some final remarks, pointing out practical results.

2. Related Work

There is a number of technologies that promotes software reuse, including COM+ [4], ActiveX [5], CORBA [6], RMI [7], Java Beans [8] and EJB [9]. Those technologies enable software reuse exploring two different approaches: distributed objets and component libraries. Distributed objects promote reuse of executing components, which are running or can be activated on remote servers. Although powerful, distributed objects can not be applied to all kinds of software systems, being suitable for ones based upon distributed architectures.

On the other hand, component libraries promote reuse of executable components, which are made locally available during installation of their respective software packages on user desktops. In such case, it is usual software systems to make concurrent use of different versions of the same components. However, current technologies have adopted limited approaches to address versioning. As effect of the lack of versioning support, it is very common to install a new software package and something previously installed breaks [10]. Besides, component libraries are fulfilled with the entire set of components referred to by

software systems, not regarding which ones are really activated by users during runtime. In SOS, differently, components are just installed based on user needs.

Without laborious and complex programming efforts, none of those technologies [4-8] has capabilities to support on demand, transparent download, installation and activation of components on user desktops. SOS supports such capabilities enriching both approaches, distributed objects and components libraries. It defines a distributed component library that dynamically and transparently maintains local component libraries on both user and producer desktops. Besides, it adopts a versioning mechanism enabling applications to make use of different versions of components. Although SOS does not directly address issues related to distributed objets technologies, it let components activated in user desktops make use of such technologies, playing the role of either a client or a server object.

In spite of being designed without software reuse purposes, several environments support aspects explored by SOS, like programming languages supporting code on demand and content delivery and software distribution systems allowing dynamic distribution and updating of digital resources. Java [11] loads locally available classes as they are required. Besides, Java applets [12] and extensions [13] download remotely available classes during runtime. Although powerful, they suffer from problems resolved by SOS. Java does not allow applications to share different versions of classes. Applets and extensions can only download classes from the remote host that they came from. Based upon URLs, like the Web, they have referential integrity and location transparency problems which are well known. Since extensions are not installed locally, each time applications are activated, extensions have to be downloaded once again and, occasionally, before finding classes, all referenced extensions must be downloaded. In SOS, differently, artifacts are downloaded just once when referred to during runtime, being available thereafter for all applications sharing them. Besides, SOS locates and retrieves exactly the artifact specified by the application.

A number of recent content delivery and software distribution systems based upon Web technologies are available nowadays, including Castanet [14], WebCasting [15], Netcaster [16] and NetDeploy [17]. Employing URL-based models and representing applications as a set of files to be reproduced on user desktops, those systems fetch files from URLs and automatically check and update changed files on a regular basis. Instead of files that are just a storage abstraction, SOS manages the post-development application lifecycle exploiting a compositional model based on identifiers, which solve problems related to URLs.

None of [14-17] can cope with inner composition of applications in a fine-grained way. The closest technique is used by Software Dock [18] but, as all other proposals, it deals with monolithic software systems, distributing complete applications even when users require just a small portion of their functionality. Therefore, in a context where producers are spread over the Internet, transferring enormous applications strains storage and communication resources, even more severely so when dynamic management of updates is required.

3. Software Operating System

SOS (Software Operating System) is a framework that provides means to deal with software reuse over global open networks. The framework employs the power of distribution and execution models of already-existing Web technologies for supporting a distributed component library which allows registration and on-demand, dynamic distribution and integration of reusable components over the Internet.

From the producers' perspective, SOS simplifies the development, registration, distribution and management of software systems, designed using distributed reusable components available on the network and possibly developed by multiple and geographically dispersed producers. From the users' perspective, SOS provides easy access to software systems available on the network: what users need is to get software licenses or to connect enabled devices and, thereafter, using them immediately. Upon registering software licenses or install enabled devices, whenever fine-grained components that compose software systems or device drivers are referred to at the first time, based upon a code on demand approach, they will be individually, transparently and dynamically located and downloaded from the distributed component library. Upon that, they will be installed, loaded and linked on user environments or devices. Therefore, the framework acts like a dynamic networked marketplace for allowing distribution and reuse of software components.

Loading executable code over open networks is a well-known security risk. To deal with such problem, SOS provides integrity and authenticity based upon a public-key infrastructure. It supports an open and wide distribution of components, in which they can be made available freely and without restriction but are protected from modifications.

3.1. Compositional Model

SOS is based upon a compositional model in which software systems are composed of a set of individual *metacomponents* distributed over the Internet. Metacomponents are reusable software entities that wrap and describe other software entities, *embedded components*, which in turn can be components of any specific component model [Figure 1]. For instance, a metacomponent can wrap a COM+ [4] or JavaBeans [8] component.

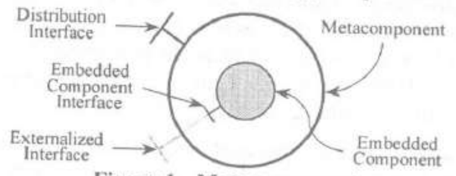

Figure 1 - Metacomponent

The metacomponent model, detailed in [19], defines the architecture of metacomponents, specifying the structure and operations of their interfaces and mechanisms by which they interact with others. Besides, metadata describing metacomponents are presented in [19]. As another contribution, [19] presents guidelines to design metacomponents that can work together to form larger applications.

The purpose of metacomponents is to specify distribution issues related to their embedded components. The *distribution interface* addresses such distribution purposes only, not dealing with features related to technologies employed to develop embedded components. However, the distribution interface defines operations for extracting embedded components, granting to visual builder tools and runtime services the ability to manipulate and activate embedded components. As access to embedded component makes visible their interfaces, the model regards such accessibility as a virtual interface, called *externalized interface*.

To facilitate software reuse, a metacomponent can be deployed independently and is subject to composition by third parties. Composition enables metacomponents to be developed integrating and using services provided by other previously available metacomponents. Hence, multiple metacomponents can be combined and interrelated to rapidly build an application or to create a new metacomponent, more comprehensive or specialized.

SOS handles metacomponents using mechanisms based upon *metacomponent identifiers*. Every metacomponent has its own identifier, which is universal and unique on the Internet.

Identifiers allow SOS to distinguish, discover, locate, retrieve, install and activate corresponding metacomponents. An identifier has the following syntax: *N/V*, where *N* and *V* are the name and the version of the metacomponent. Together, these terms make metacomponents unique in the world and over time.

For expressing namespace authority, SOS arranges the namespace as a hierarchical tree, where leaves are metacomponents and, internal nodes are *domains* representing software producers. A domain is a named collection of metacomponents and, possibly, other domains immediately underneath it. Each domain is controlled by a producer, which has authority for registering metacomponents and creating sub-domains.

Like the Web, SOS describes software systems with a hypertext structure of identifiers similar to URLs. Such hypertext structure is defined by composition of metacomponents. Identifiers differ from URLs in that identifiers allow metacomponents to be specified by name instead of location. The name of a metacomponent is unrelated to its location, making possible to move or replicate metacomponents without affecting or breaking existing references (i.e., identifiers) to them, which is not possible using conventional URLs. Identifiers, together with the resolution mechanism supported by SOS, solve the problem of referential integrity and location transparency that occurs with URLs [20].

3.2. The Architectural Framework

SOS defines a high-level architectural framework [Figure 2] that exploits the compositional model allowing the reuse of metacomponents. The framework defines a compliant *middleware* that allows producers to develop and register metacomponents and users to install and execute software systems. To effect such activities, producers and users employ a set of reuse-driven automated tools, which interacts with the middleware through well-defined interfaces. It must be stressed that reuse issues are spread throughout the entire lifecycle of software systems - from development and registration, to installation and execution.

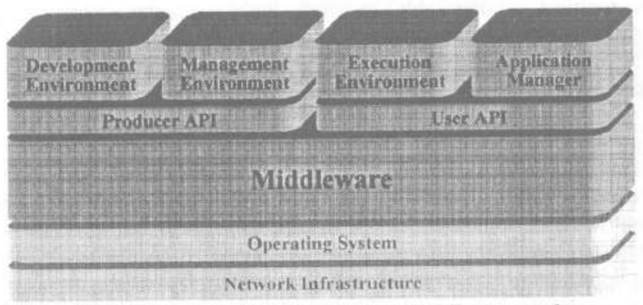

Figure 2 – High-Level Architectural Framework

On the producer side, a set of automated tools are employed by producers to design, develop and register metacomponents. *Development environments* are visual builder tools that support reuse-driven design and development of metacomponents, allowing embedded components to be codified and wrapped in metacomponents. In such process, a development environment automatically generates metadata describing the metacomponent, for instance, its composition. When generating metadata, development environments check and validate dynamically and transparently the composition of a metacomponent, assuring that it refers to already available metacomponents only.

Development environments support reuse of metacomponents based upon a code on demand approach in which required metacomponents are transparently found in a local cache or retrieved from the middleware. Handling embedded components, a development environment becomes dependent on the technology employed to develop them.

In order to be available, metacomponents must be registered in the middleware. To do that, producers employ *management environments* that allow administering namespace authority. Such environments have capabilities that allow producers to manage domains and to register metacomponents. Working with metacomponents, a management environment is not dependent on the technology employed to develop embedded components.

On the user side, the *application manager* is a visual tool employed by users to manage the post-development application lifecycle including installation, execution, update, upgrade and removal. *Execution environments* are runtime services that execute on user desktops providing an application context for metacomponents. In practical terms, based upon a code on demand approach, it provides an operating system process or thread in which metacomponents are dynamically loaded and executed. For instance, a slightly modified Java Virtual Machine (JVM) [21] can be the execution environment for metacomponents wrapping Java classes. During execution of applications, execution environments load required metacomponents only when they are referred to at the first time.

It must be stressed that the metacomponent model [19] provides capabilities that allow execution environments to assure that a new software version does not interfere on other one previously installed. Such capabilities are provided by including in each metacomponent the identifiers of other ones that it depends on. Since an identifier specifies the version of the metacomponent, execution environments can activate the correct one.

In fact, execution environments extract and instantiate embedded components, using metacomponents to manage the dynamic loading of other metacomponents only. Since embedded components can be codified using different technologies, at first sight, an execution environment is only able to work with metacomponents whose embedded components share an equivalent technology. To assure compatibility and correctness during runtime, such constraint imposes that a given application must be comprised of metacomponents whose embedded components are based upon the same technology. However, at least conceptually, using an approach similar to COM+ [4], an execution environment can be language-neutral, being able to handle embedded components developed using any technology. In such case, the execution environment must deal with adaptation of data formats used by different technologies. The tradeoff of technology interoperability is added complexity. As described in [19], a metacomponent has an attribute and associated operation that allows identifying the type of execution environment to be activated.

The *user API* and the *producer API* define a collection of operations supported by the middleware enabling development environments, management environments, application managers and execution environments to interact with the middleware to accomplish their end goals. Such operations will be concisely described in section 4.

The middleware composes a distributed component library and name service responsible for dealing with distribution issues, enabling on demand, dynamic delivery of registered metacomponents to user and producer environments. Indeed, it is a set of distributed, inter-communicating services composed of instances running in participating devices and relying upon a network infrastructure that allows communication. Such services are responsible for registering, storing, locating and retrieving metacomponents. Figure 3 shows the three-tier architecture of the middleware, where tiers define services with specific functionality through

well-defined interfaces conferring independence between them. Such independence allows the design and implementation of each tier to be performed in an autonomous way provided that an agreement exists on the technology employed to implement the interfaces.

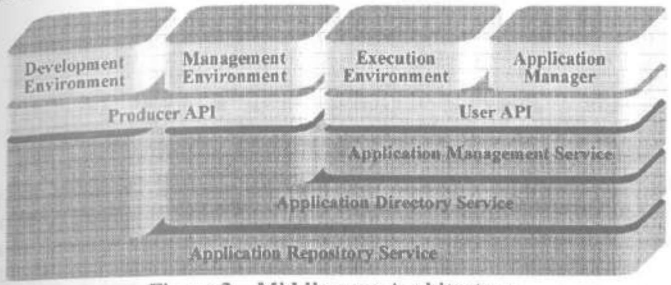

Figure 3 – Middleware Architecture

- **Application Repository Service (Repository):** composes a universal distributed component library for storing and retrieving metacomponents.
- **Application Directory Service (Directory):** builds a distributed name service for registering, locating and retrieving metacomponents.
- **Application Management Service (Manager):** locally manages the lifecycle of metacomponents on user environments.

To be available, metacomponents must be registered by producers in the directory, which in turn transparently stores them in the repository. The directory implements a resolution mechanism that translates metacomponent identifiers to their corresponding storage locations in the repository, which holds packaged data describing each and every metacomponent. Producers employ management environments to manage the directory and the repository. In several activities, the directory transparently manages the repository.

The manager enables metacomponents to be dynamically retrieved and integrated on the user's desktop at runtime. It offers services that guarantee support for every phase of the post-development application lifecycle, which includes installation, activation, update, upgrade and removal. The manager allows users to acquire a software license and get the application installed and running, enabling the availability of genuine plug-and-play software systems.

As distributed services, entities playing the role of the directory and the repository are only activated in the subset of devices employed by producers to register and store their metacomponents. On the other hand, as the manager is a locally granted service, entities playing its role is activated in all participating devices (user desktops), enabling on demand, dynamic retrieval of registered metacomponents at runtime and automatic management of every phase of the post-development lifecycle of metacomponents.

3.2.1. Application Repository Service

The repository is a universal distributed database composed of a set of distributed entities, called *containers*, responsible for managing the storage and retrieval of metacomponents [Figure 4]. Containers are accessed by *container names*, which designate locations of hosts supporting the service. The framework does not specify any syntax to container names. Although a metacomponent can be replicated in several containers, the repository does not address any replication control, which must be regarded by upper tiers.

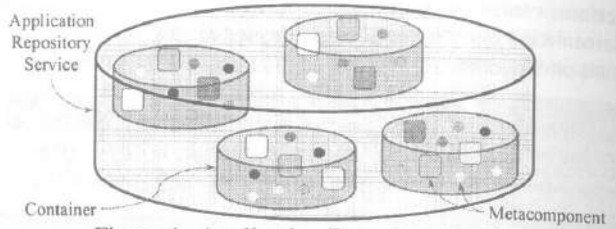

Figure 4 - Application Repository Service

Containers define relationships with a set of elements [Figure 5]. On behalf of a producer, a domain can be authorized to store and remove metacomponents in several containers. On the other hand, several domains can be authorized to manipulate metacomponents in a container. In order to control access, each container keeps a *list of authorized domains*, identified by their names and public keys. Furthermore, each container has a *container manager*, which is an institution or person responsible for managing the list of authorized domains.

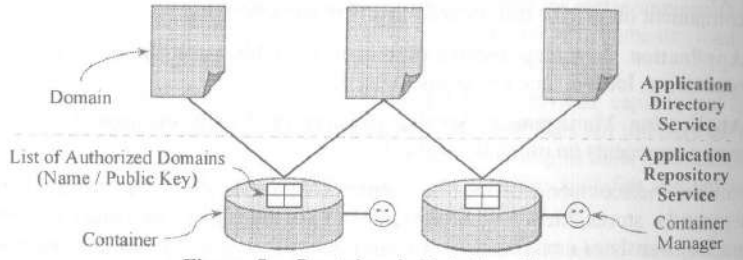
Domain
Application Directory Service
List of Authorized Domains (Name / Public Key)
Application Repository Service
Container
Container Manager

Figure 5 – Container's Relationships

3.2.2. Application Directory Service

The directory is a universal distributed name service composed of a set of distributed, inter-communicating entities, called *registries*, responsible for managing the registration of domains and metacomponents, the resolution of metacomponent identifiers and the retrieval of metacomponents [Figure 6]. In a way similar to containers, registries are accessed by *registry names*, designating locations of hosts supporting the service, and the framework does not specify any syntax to registry names.

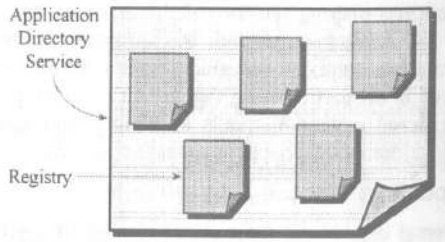

Figure 6 - Application Directory Service

Registries also constitute relationships with a set of elements [Figure 7]. On behalf of a producer, a domain can be authorized to register metacomponents in several registries. On the other hand, several domains can be authorized to inscribe metacomponents in a registry and each registry keeps a *list of authorized domains*, identified by their names and public keys. Registries must implement consistency control, assuring that the information about domains is

synchronized between them. In addition, each registry has a *registry manager*, which is an institution or person responsible for managing the list of authorized domains.

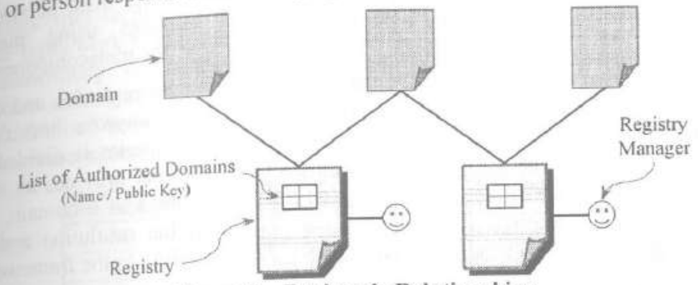

Figure 7 – Registry's Relationships

As outlined earlier, a domain is an entity authorized to work with a set of registries and containers. In addition, a registry and a container can assist several domains. Figure 8 illustrates the relationships among a domain *D* and its registries and containers.

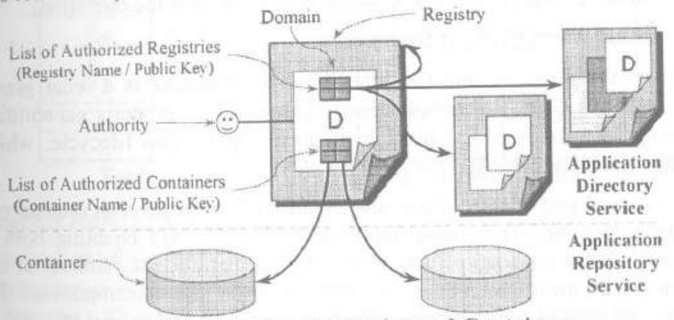
Domain
Registry
List of Authorized Registries (Registry Name / Public Key)
Authority
D
List of Authorized Containers (Container Name / Public Key)
Application Directory Service
Application Repository Service
Container

Figure 8 - Domains, Registries and Containers

Being served by several registries and containers, a domain keeps a *list of authorized registries* and a *list of authorized containers*, identifying registries where information describing the domain is kept and containers where metacomponents of the domain can be stored. Registries and containers are identified in those lists by registry and container names, together with their respective public keys. Each domain has an *authority*, which is an institution or person responsible for managing it. An authority has permission to register sub-domains and metacomponents beneath its domain and, it is also responsible for controlling the list of authorized registries and containers that serve its domain. Moreover, an authority can administer several domains, representing that a producer can maintain several domains.

The universality and uniqueness of metacomponent identifiers are granted by uniform registration and resolution processes enforced by interfaces of registries. The registration process defines the binding between identifiers and container names. Registries implement the concept of multiple binding in which a single identifier can be resolved to several container names, defining multiple locations (containers) where replicas of the corresponding metacomponent are stored. On the other hand, the resolution process translates identifiers to corresponding container names.

Registries hold only bindings between identifiers and container names, activating operations on specified containers to effectively store replicas of corresponding metacomponents. Metacomponents are not stored in all containers serving their domains, allowing frequently referenced metacomponents to have more replicas than rarely referenced ones. During registration, producers select the subset of containers where replicas must be stored. After registering a metacomponent, producers can change bindings using management environments that indirectly request containers to store or remove the metacomponent.

Based upon replication and consistency controls, the collection of registries and containers assisting a domain provides backup to each other. Further, they improve both the overall performance of the resolution process since processing load of queries is divided between registries, and accessibility as access via each registry and container provides an alternative path. Therefore, the greater the number of registries and containers of a domain, the more available is the domain knowledge and the more efficient is the resolution and retrieval processes. Besides, distributed aspects confer high-level of scalability to the framework.

The framework is not dependent on specific technologies to implement registries and containers, as long as those technologies offer facilities to support the operations defined by the interface of the directory and repository. The World Wide Web, for one, is a satisfactory technology. In such scenario, URLs can be used as registry and container names, and CGI scripts [22] or Java Servlets [23] can be developed to implement the operations.

3.2.3. Application Management Service

Differently from the directory and the repository, the manager is a local granted service activated in each and every compliant device. The manager supports capabilities allowing users to manage all phases of the post-development application lifecycle, which includes installation, activation, update, upgrade and removal.

The manager is composed of a set of entities [Figure 9]. The *cache* is an internal repository of installed metacomponents. The *engine* implements the user API enabling both application managers to request the installation, activation, update, upgrade and removal of applications, and execution environments to request the load of required metacomponents. Besides, the engine has internal capabilities that support both the dynamic update and removal of software systems. The engine is responsible for managing the cache assuring that: metacomponents are just installed after being referenced by previously installed metacomponents; only one copy of a metacomponent is installed when it is shared by several metacomponents; and several versions of a metacomponent can be installed. The *explorer* is the only entity of the manager that interacts with registries, being responsible for retrieving metacomponents. The separation between explorer and engine is just formal. In order to achieve better performance, the explorer can be integrated into the engine.

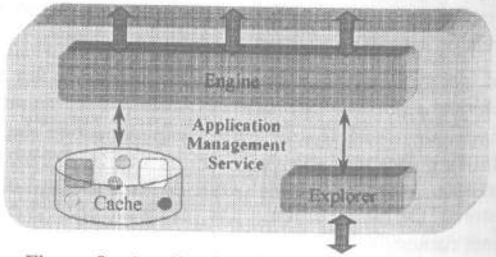

Figure 9 - Application Management Service

4. Operational Guidelines: a Sketch

SOS services offer facilities through well-defined interfaces specified by operations, which define the framework's operational model. In order to protect producers and users, SOS assures authenticity and integrity using digital signature supported by a public-key infrastructure embedded in the framework. Due to space limit, the operational model is concisely described below, abstracting away issues related to authenticity and integrity, and focusing on reuse issues. It is presented using the symbols defined in Table 1.

Symbol	Description
M	Metacomponent
I_M	Identifier of the metacomponent M
D	Domain
PU_D	Public key of the domain D
P_D	Parent domain of the domain D
R	Registry
PU_R	Public key of the registry R
R_D	Set of registries serving the domain D
C	Container
PU_C	Public key of the container C
C_M	Set of containers storing the metacomponent M
C_D	Set of containers serving the domain D

Table 1 – Symbology of the Operational Model

4.1. Application Repository Service

Containers support two kinds of interfaces. The first, *authority management operations*, allow container managers (producers) using management environments to control the list of authorized domains. The second, *functional operations*, allow registries, on behalf of authorized domains, to store, remove and retrieve metacomponents. Authority management operations compose the portion of the producer API that directly interacts with the repository [Figure 3].

In order to authorize a container C to handle metacomponents of a domain D, supplying D and PU_D, the manager of C activates the corresponding authority management operation to include D in the list of authorized domains. Once D is authorized in C, registries can activate functional operations to store, remove and retrieve metacomponents of D. To cancel the authorization, the manager of C removes D from the list of authorized domains. To do that, supplying D, the manager activates the corresponding authority management operation.

The retrieval of a metacomponent M from a container C in C_M can be requested by any registry that, providing I_M, activates the corresponding functional operation. On the other hand, the storage and removal of a metacomponent M of a domain D are restricted to registries R_D. In order to store M in a container C in C_M, supplying M, any registry R in R_D activates the corresponding functional operation. Similarly, to remove M from a container C in C_M, providing I_M, any registry R in R_D activates the corresponding functional operation.

4.2. Application Directory Service

Registries offer four interfaces. The first, *authority management operations*, allow registry managers using management environments to control the list of authorized domains. The second, *domain management operations*, allow authorities (producers) using management environments to define registries and containers assisting their domains. The third, *functional operations*, support registration and retrieval of metacomponents, identification of new versions of metacomponents, and resolution of metacomponent identifiers. The fourth, *replication control operations*, internally ensure consistency of domains replicated in several registries. All operations, except replication control operations, compose the portion of the producer API that directly interacts with the directory [Figure 3].

In order to authorize a registry R for handling metacomponents of a domain D, supplying D and PU_D, the manager of R activates the corresponding authority management operation to include D in the list of authorized domains. To cancel the authorization, the manager of R removes D from the list of authorized domains. To do that, the manager activates the corresponding authority management operation supplying D.

Supposing that registry and container managers have already authorized the domain D in their respective registries R_D and containers C_D, the list of authorized registries and the list of authorized containers owned by D must be initialized with R_D and C_D, respectively. In order to initialize the list of authorized registries, for each R in R_D, the authority of D activates the corresponding domain management operation providing R and PU_R. Similarly, to initialize the list of authorized containers, for each C in C_D, the authority of D activates the corresponding domain management operation informing C and PU_C. Domain management operations are activated in only one registry R in R_D, which automatically activates replication management operations for keeping synchronized all registries R_D.

The resolution of metacomponent identifiers requires that domains keep information about all registries serving their sub-domains. The authority of the domain declares such information activating domain management operations. Hence, in order to register the domain D in its parent domain P_D, the authority of P_D must activate the corresponding domain management operation providing D and, for each registry R in R_D, R and PU_R. Again, the domain management operation is activated in only one registry serving P_D, which automatically activates replication management operations for keeping synchronized registries serving P_D.

The registration process enables producers to make and cancel metacomponent inscriptions, managing the binding of metacomponent identifiers and container names. In order to register a metacomponent M of a domain D, the producer activates the corresponding functional operation in a registry R in R_D, supplying M and the subset of containers C_M ($C_M \supseteq C_D$) serving D. To keep synchronized all registries R_D, R automatically activates replication management operations. Besides, R automatically activates functional operations on all containers C_M to store replicas of M. On the other hand, to cancel the inscription of M, providing I_M, the producer activates the corresponding functional operation in a registry R in R_D. Again, synchronization is accomplished in all registries R_D internally activating replication control operations and R automatically activates functional operations on all containers C_M to remove replicas of M.

The metacomponent retrieval can be requested to any registry. In order to retrieve a metacomponent M of a domain D, supplying I_M, a requesting entity (explorer or development environment) activates the corresponding functional operation in any registry R'. Based on a cooperative resolution process, R' discovers the registries R_D and sends a request to any registry R in R_D for fetching M. Since all registries R_D know the set of containers C_M, R selects any container C in C_D and requests the retrieval of M. Upon retrieving M, R returns M to R', which in turn returns M to the requesting entity.

4.3. Application Management Service

Manager's facilities are provided by interactions between the engine and the explorer. The explorer supports an interface that allows retrieving a metacomponent and verifying if a new version of a metacomponent has been released. To retrieve or verify a metacomponent M, the engine calls the explorer supplying I_M. The explorer directly activates registry's functional operations for processing its requests.

The engine offers two kinds of interfaces: *functional* and *management operations*. Functional operations allow users employing application managers to control the post-development application lifecycle. Besides, functional operations enable execution environments to request the load of metacomponents. Functional operations define the user API [Figure 3].

The installation of an application is basically to register in cache the identifier of the metacomponent representing the application. Hence, in order to install an application M, a user just informs I_M. At this point, the application manager requests the engine, which in turn forwards the request to the explorer for retrieving M from the directory. Upon retrieving M, the engine stores it in the cache. It must be pointed out that only the metacomponent M is installed. All other metacomponents composing the application M will be just retrieved when referred to during runtime.

Upon installing the application M, using the application manager, the user can activate M. The engine finds M in the cache and activates the execution environment, which is responsible for loading in memory referenced metacomponents. At any moment, the user can request the removal of the application M, deleting M and its internal metacomponents, not reused by other ones.

The update operation requests to the explorer for verifying whether the application M, whose identifier I_M was specified by the user, has a new version M'. If so, the engine requests the retrieval and installation of M'. Again, only M' is retrieved and installed. M and its referenced metacomponents, not reused by other ones, can be optionally deleted from the cache. Hence, metacomponents are kept in the cache if other ones reuse them.

In contrast with update, upgrade does not represent just the installation of a new version that fixes bugs. Instead, the upgrade consists in replacing one application by another very similar but including new functionality. When the user request the upgrade of the application M to M', specified by their identifiers I_M and $I_{M'}$, respectively, the upgrade operation requests to the explorer for retrieving and installing the upgraded version M'. At this point, what happens is very similar to the update process. It must be noted that the update automatically finds the new version of an application and the upgrade finds the version specified by the user.

The engine's management operations internally control the automatic update and removal of applications. The internal update process keeps applications up to dated by automatically polling the explorer, in any desired time interval, to verify whether new versions have been released. Polling intervals are predefined by producers during developing of applications but can be locally configured by users. The internal removal process supports a lease approach based on time intervals specified by producers. Upon interval expirations, the engine automatically removes applications and their internal metacomponents, not reused by other ones. Polling and lease intervals are attributes of applications [19].

As pointed out before, execution environments are responsible for loading metacomponents in memory, allowing component-based software systems to get correctly assembled and integrated on user desktops at runtime. Exploring composition, execution environments perform dynamic loading and linking of metacomponents. Whenever an executing metacomponent *M* refers to another metacomponent *M'*, not loaded in memory yet, the executing environment sends a request to the engine. First, the engine verifies whether *M'* is locally installed. If so, *M'* is returned and immediately loaded in memory. Otherwise, the engine forwards the request to the explorer for retrieving *M'*. Upon retrieving *M'*, the engine stores it in the cache and returns it to the execution environment.

The retrieval mechanism installs metacomponents only when they are referred to at the first time. However, since network infrastructures can introduce significant delays, SOS allows users to adjust the retrieval mechanism enabling a prefetch approach. Such adjustment is performed by users changing a metacomponent attribute [19], which identifies the number of composition levels whose metacomponents must be retrieved in advance. For instance, if that attribute is equal to 1, every time a metacomponent *M* is referred to at the first time, the engine immediately retrieves and installs the other ones directly referenced by *M*. So, when *M* refer to any of those metacomponents, they are already available.

5. A Prototype Environment

In order to validate and evaluate SOS, a prototype middleware, called CoDelivery, was developed for Java applications. A detailed description of CoDelivery can be found in [24], where a study case based on a simple Java application is also presented demonstrating actions carried out by producers and users, for releasing two versions of an application and installing, executing and updating them.

CoDelivery employs the power of already-existing Web technologies for distributing, integrating and executing Java applications over open networks. CoDelivery manipulates metacomponents wrapping Java classes. Compositions of metacomponents are defined in terms of runtime dependencies between Java classes. Dependencies are characterized by references to other classes: *extends* and *implements* clauses, class attributes or method parameters and results. CoDelivery does not handle classes in default Java packages as metacomponents so far, only those developed by producers.

CoDelivery's services are implemented as Java RMI distributed objects. Directory and repository objects are activated in a subset of devices, which offer their services through a network infrastructure. On the other hand, as a locally granted service, Manager objects are activated in each participating user desktop. To protect producers and users, CoDelivery provides authenticity and integrity based upon digital signature supported by remote methods.

Containers and registries are RMI objects accessed by URLs. For instance, a container can have a container name like *rmi://zeus.enginz.com/sos.container*, representing the protocol, the host and the name of the RMI object. Containers and registries store information about authorized domains and their metacomponents in an abstract storage device defined by a Java interface. Such approach enables different implementations of the storage device interface to make use of different technologies to store information. For instance, directly accessing the file system or interacting with a database management system. During configuration time, producers can define a Java class that implements the storage device interface. In the present, there is only one implementation of the interface, which stores information as a collection of directories and files generated using Java object serialization.

The engine is a local RMI object accessed by a URL and provides facilities to application managers and execution environments. The explorer is just a Java object instantiated by the engine. The cache is also an abstract storage device defined by a Java interface, which can be implemented using different technologies. Such approach allows users of a local network to share metacomponents. For instance, an implementation of the cache interface can request a

remote RMI object, running in a server available in the local network, for storing and retrieving metacomponents. However, the available implementation of the interface stores metacomponents as a set of directories and files generated using Java object serialization.

CoDelivery provides an execution environment that supports execution of metacomponents wrapping Java classes. The execution environment extends the default Java class loader for requesting metacomponents to the engine. In the present, capabilities provided by application managers, development environments and management environments are implemented as a set of Java packages, one package for each capability. For instance, on the user side, there are packages for installing, activating, updating, upgrading and removing Java applications. On the producer side, there are packages for managing registries and containers serving domains and registering metacomponents.

6. Concluding Remarks

SOS provides means for dealing with software reuse over the Internet. It is not, at first sight, a great leap forward. But it is clear, by comparison with other, already existing approaches that we are putting forward a new way to manage software reuse, that may cause a serious impact on current models, solving some hard problems and overcoming critical limitations. What makes this proposal unique is its support for software reuse triggered by producer and user needs and based upon Web-based development and computing models, which take into account inherent features of software systems such as composition and commonality.

Exploring software composition in a fine-grained way, SOS defines a reuse-driven approach to develop, distribute, execute and evolve software systems comprised of reusable metacomponents. Indeed, from the producers' perspective, SOS enables a component-based development approach that uses metacomponents available on the network and developed by multiple and geographically dispersed producers. From the users' perspective, SOS manages the entire post-development application lifecycle maintaining a local component library whose components are individually and dynamically installed triggered by user needs. Therefore, the framework acts like a dynamic networked marketplace for allowing reuse of software components based upon a code on demand approach.

SOS is an advancement in its own, given that no prior proposal could show the following properties: dynamic distribution of versioned components; on demand, progressive installation of components; automatic update of software systems; and dynamic removal of software systems based upon a lease approach. In addition, performance analysis and preliminary practical results are encouraging, as we implemented the support for: location transparency and referential integrity of metacomponent identifiers; reusability of metacomponents; scalability and resiliency based on distribution and replication features; incrementability supporting on demand usage of communication and storage resources; and extensibility and evolvability triggered by user needs. Security concerns are also taken care of, protecting producers and users based on a public-key infrastructure.

Performance analyses and practical evaluations were performed using a set of small applications, each one designed for assessing different features of the environment. In order to make performance analyses, scripts were developed for simulating a high number of queries for registries and containers.

More extensive practical evaluation of the prototype environment is under way to assess SOS in several application domains, including a real-life test with tens of thousands of users distributed nationwide in Brazil. Interesting work remains to be done, including the implementation of a GUI-based development environment, management environment and application manager. Besides, SOS can be enriched with extensions to cope with a business model and mechanisms allowing search engines to find, evaluate and select metacomponents.

7. References

[1] Kontio, J. *OTSO: A Systematic Process for Reusable Software Component Selection*. University of Maryland. Technical Report CS-TR-3478. 1995.
[2] Cox, B. *No Silver Bullet Revisted*. American Programmer Journal, November 1995.
[3] McClure, C. *Reuse: Re-Engineering the Software Process*. Extended Intelligence, Inc. 1994. < http://www.reusability.com/papers7.html>
[4] Kirtland, M. Object-Oriented Software Development Made Simple with COM+ Runtime Services. Microsoft Systems Journal. November, 1997.
[5] Microsoft Corporation. *How to Write and Use ActiveX Controls for Microsoft Windows CE*. June, 1999. <http://msdn.microsoft.com/library/techart/activexce.htm>
[6] Object Management Group. *CORBA/IIOP 2.3.1 Specification*. October, 1999.
[7] Sun Microsystems, Inc. *Java Remote Method Invocation (RMI)*. <http://java.sun.com/products/jdk/rmi/index.html>
[8] Sun Microsystems, Inc. *JavaBeans API Specification – Version 1.01*. July, 1997.
[9] Thomas, A. *Enterprise JavaBeans Technology: Server Component Model for Java Platform*. Sun Microsystems. December, 1998.
[10] Szyperski, C. *Greetings from DLL Hell*. Software Development. October, 1999.
[11] Gosling, J. and McGilton, H. *The Java Language Environment: A White Paper*. Javasoft. May, 1996. <http://www.javasoft.com/doc/language_environment>
[12] Campione, M. and Walrath, K. *The Java Tutorial*. Addison-Wesley. 1998.
[13] Campione, M. et al. *The Java Tutorial Continued: The Rest of the JDK*. Addison-Wesley. 1998.
[14] Marimba, Inc., *Introducing the Castanet Product Family*. <http://www.marimba.com/products/castanet-intro.htm>
[15] Microsoft Corporation. *Webcasting in Microsoft Internet* Explorer *4.0 White Paper*. September, 1997. <http://www.microsoft.com/ie/press/whitepaper/pushwp.htm>
[16] Netscape Communications Corporation, *Netcaster Developer's Guide*. September, 1997. <http://developer.netscape.com/docs/manuals/netcast/devguide/index.html >
[17] Open Software Associates, *NetDeploy 4 Technical Specifications*. December, 1999. <http://www.osa.com/docs/tss/tssowb.phtml>
[18] R. S. Hall et al., *The Software Dock: A Distributed, Agent-based Software Deployment System*. International Conference on Distributed Computing Systems. May 1997.
[19] Elias, G. and Meira, S. L. *A Metacomponent Model to Support the Extensibility and Evolvability of Networked Applications*. TOOLS USA 2000. August, 2000. United States.
[20] D. Ingham, S. Caughey, and M. Little, *Fixing the Broken-Link Problem: The W3Objects Approach,* Computer Networks and ISDN Systems, Volume 28, Nº 7-11, May 1996.
[21] Lindholm, T. and Yellin, F. The *Java Virtual Machine Specification*. Second Edition. Addison-Wesley. 1999.
[22] Gundavaram, S. and Oram, A. *CGI Programming on the World Wide Web*. O'Reilly & Associates. April, 1996.
[23] Sun Microsystems, Inc. *Java Servlet Specification, v2.2*. December, 1999.
[24] Elias, G. and Meira, S. L. *CoDelivery: An Environment for Distribution of Reusable Components*. TOOLS Europe 2000. June, 2000. France.