A Disconnected Service Architecture for Unanticipated Run-time Evolution of Code

Manuel Oriol, Giovanna Di Marzo Serugendo

Centre Universitaire d'Informatique, University of Geneva
24, rue Général Dufour,
1211 Geneva 4, Switzerland
{oriol,dimarzo}@cui.unige.ch

Abstract. Run-time evolution of applications is an important issue for safety critical systems, such as nuclear power plants management systems; or for night-and-day used software like mail/Web servers and banking systems. It may also be useful, during an application development, to let portions of code evolve at run-time, not restarting the debugged application, and still using it. In addition, today's software is surrounded by a highly dynamic environment, in terms of softwares, network topologies, or communication means. Thus, evolution of code cannot always be anticipated, i.e., foreseeable at design time. At our sense, unanticipated run-time evolution of object code is favored by disconnection of communicating components, i.e., by avoiding as much as possible dependencies among components. Anonymous and asynchronous communications are two means for realizing disconnection. This paper presents first a model for unanticipated run-time evolution of code, based on these principles. The model enables communication among components through asynchronous services, which are described rather than designated, thus respecting the anonymity of communicating parties. Second, we describe two implementations of the model: a local one allowing to add and remove services at run-time on a local host; and a distributed one, which enables the distribution of an application, while it is still running. We report as well our experience in realizing a restricted Web server and a tic-tac-toe game, which demonstrate that our evolution model allows to hot-swap, duplicate, and remove parts of an application at run-time.

Keywords. Evolution, Run-time, Disconnection, Components, Associative Naming, Asynchrony, Anonymity, Distribution.

1 Introduction

Programs managing safety critical systems, such as nuclear power plants, satellites orbits, or rocket flights may rarely be stopped. It becomes then difficult to operate updates of these programs, even when driven by debugging or security purposes. Long-running, heavily used software, such as operating systems, mail accounts managers, e-commerce applications, telebanking, or mobile phone

management appear to have the same needs. Indeed, stopping and restarting a software offering services to its users 24 hours a day, necessarily implies "delogging" users, unacceptable delays for customers, or careful halt of on-going electronic transactions. We identify these needs as the need for software *dynamic*, i.e., run-time, evolution.

In the dynamic software evolution paradigm we distinguish two types of evolution: *anticipated evolution* and *unanticipated evolution. Anticipated evolution* is an evolution that has been foreseen, at design-time, by the programmer. This means that plug-ins technologies (like Java servlets [16], Adobe Photoshop plug-ins [25], or object oriented language inheritance) are mechanisms that allow programmers and maintainers to extend functionalities, but not to modify the heart of the application itself. *Unanticipated evolution* consists in evolution that has not been foreseen by the programmer. Changes have thus to be supported either by the language or the execution platform. In the following we have only interest to the case of unanticipated evolution.

Dynamic evolution of code allows the transparent replacement of a whole or a part of a system while running. Current approaches to run-time software evolution usually address evolution problems either in the functional programming paradigm [15, 11], limiting thus the problems related to objects transfer; or on an object-oriented basis but restraining a lot possible changes [34, 21, 9]. In this case, run-time changes do not necessarily ensure consistency, or preservation of desired properties. Object-oriented approaches usually impose rigid constraints on APIs, ensuring dynamic replacement of compatible sub-types at run-time, but preventing more flexible evolution schemas.

The evolution model, presented in this paper, supports *unanticipated dynamic* evolution of code. It is based on a run-time platform that provides built-in facilities for code evolution, and ensures desired properties to be preserved despite run-time changes. Our model focuses on disconnection among software entities (components, agents, objects), favoring as much as possible time, space, reference, and APIs decoupling among communicating entities. It is thus founded on a disconnected service architecture favoring evolution, through *asynchronous* and *anonymous* communications between entities. Entities do not reference services, but provide a *description* of them. This ensures anonymity, preservation of services properties despite run-time changes, and frees entities from communicating through fixed APIs.

This paper is organized as follows. First, Section 2 presents the proposed evolution model based on a disconnected service architecture. Second, Section 3 describes two implementations of the run-time platform (LuckyJ), a local and a distributed one. Third, Section 4 describes two case studies built on top of the implemented platforms. Finally, Section 5 discusses some related works.

2 Evolution Model

The evolution model, presented in this paper, results from the observation that code evolution is hindered by connections among software entities that insert

dependencies, and thus complicate or limit severely updating of running applications. Connections take several forms: inheritance/instantiation links between classes/objects; direct references on data structures between classes/objects; references between caller and callee of methods or services; synchronization constraints between caller and callee of methods or services; and fixed APIs between caller and callee of methods or services. The nature of these connection links among entities suggests that evolution is facilitated if there is time, space, references, and APIs decoupling among entities. We chose the following means to realize decoupling.

Asynchronous communication enables the removal or replacement of entities without other entities being obliged to wait for all answers to be computed, delivered, or received.

Anonymous communications allows entities to communicate with each other with no reference either on the partner's identities or on the partner's code or objects. In our model, anonymous communications heavily rely on *service descriptions*. Indeed, anonymity implies that an entity, asking for an invocation, cannot reference either an entity or a service. The approach taken here is that an entity instead describes the required service. This means that programmers then specify required actions to be invoked on entities. Similarly, when creating services, programmers describe actions provided by entities. Service description, instead of service naming, offers several advantages in the framework of code evolution. First, entities providing equivalent services can be replaced transparently. Second, qualifying a service, rather than simply relying on a naming reference, offers a greater assurance on the service functionality, I/O parameters, and QoS. In addition, service description prevents entities to rely on rigid APIs. It ensures consistency, since properties, specified in the service description, are guaranteed despite code evolution.

We chose to restrict interactions among entities, by authorizing parameters of *primitive types* only. Indeed, interactions, where parameters carry complex data structures, prevent easy update of those data structures, since both communicating entities heavily rely on them. Code evolution is encouraged if parameters are only of primitive types, such as Strings, Integers, Chars, Boolean, etc. These types are not subject to change, since they are not specific to any program. This restriction does not limit the programmer from using complex data structures inside an entity. It does not restrict as well the richness of communications. Indeed, any object may be serialized. The deserialization thus creates a clone of the original object, which is free from any reference.

Finally, we chose a *service based architecture* that offers a convenient granularity level. Code evolution then occurs at the services level, which offer each a specialized task, whose replacement or removal does not affect the whole system.

The combined use of the above mechanisms results in a disconnected service based architecture, where: time decoupling is realized through asynchronous communications; space decoupling is obtained by a distributed service architecture; reference and APIs decoupling is the result of anonymous communications using service descriptions.

Unanticipated evolution of code is based on the unique constraint that the new service has to provide the same service description as the old one. Since it is an abstract description of the service, any implementation respecting this description can correctly serve the request. For instance, this allows programmers to replace an outdated service by a newer one without caring for translation methods, ensuring that calls to the old service can still be satisfied by the new one; or for inheritance or sub-typing consistencies between the old and the new version.

2.1 Disconnected Service Architecture

The model for code evolution, presented here, is based on a *disconnected service architecture*, following the above principles. Entities communicate exclusively by requesting services fulfilled by other entities. In order to avoid as much as possible connection links among entities, communication occurs anonymously and asynchronously. *Service description*, instead of service references, and a *coordinating element* among the entities achieve disconnection, since they enable anonymity, asynchrony, and avoid fixed APIs.

Figure 1 shows the basic elements of the architecture. The model consists of several *entities* offering some *services*. Due to anonymity, entities do not communicate directly, they address their requests, using a service description, to a *service manager* that is in charge of finding an adequate service to invoke.

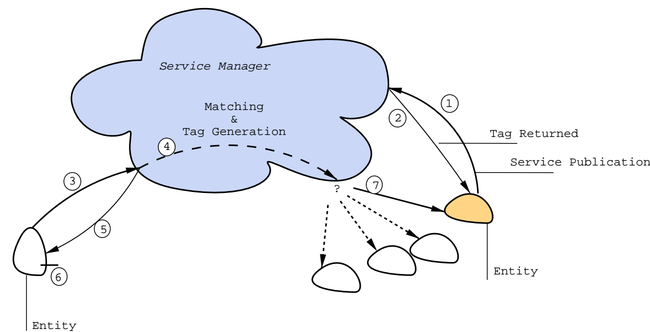

Fig. 1. Service Publication, Request, and Invocation

The model is intended for a distributed scheme, where the service manager and the entities do not run all at the same location; and where the service manager is itself made of several distributed pieces.

In a sense, our approach may be considered as a special case of the publish/subscribe mechanism [10], where subscribers express a long-term interest in events furnished by publishers. Similarly to the publish/subscribe mechanisms communications occur asynchronously and anonymously. The main differences

reside in the fact that, in the publish/subscribe model, subscribers are continuously informed of events in which they are interested, and all subscribers of a given event receive the notification of that event through a bus of events. In our model instead, service requests do not express long-term interest in an event, they mean that a given computation is requested (only once). In addition, even if several requests are performed for the same service, that service is computed for each subscriber, in a customized way with the subscribers own parameters. Finally, regarding the infrastructure, our model is not based on a bus of events, but more on a middleware performing an asynchronous and anonymous connection between one subscriber and one publisher.

Service Manager. At the heart of the model, the service manager acts as a coordinating element among the anonymous entities. Entities, offering services, submit the descriptions of their services to the service manager. Entities, requesting a service, ask for it by submitting a description of the requested service to the service manager. The service manager stores service publications from entities; and satisfies requests for services from entities. It matches services publications and services requests, chooses the best adapted service, and actually invokes it. The coordinating element, here the service manager, is the only element that knows the identities (references, pointers) of the entities.

From now on, the terms *calling/answering entities* will be used for the entities performing service requests, and those satisfying them respectively. We will also use the terms *service publication* and *service request* for a service description submitted to the service manager announcing an available service, and for a service description requesting a service, respectively.

Entities. An entity is a running software coupling data and services, such as a program, a thread, a servlet, a component, or an object. In the framework of our disconnected architecture, entities communicate with each other anonymously and asynchronously.

Anonymity means that a calling entity *cannot know* which entity, and which service, will answer its service request. In fact, it is not necessary to know the identities, but it is necessary to have the assurance that the requested task will be correctly done: the service, actually satisfying the request, should conform to the service request. Anonymity implies also that there is no reference on other entities code.

Asynchrony prevents the entity, requesting a service, to wait for the service to return. The answer, if there is any, will be delivered to the calling entity, by requesting one of its services. This also means that entities are programmed in an asynchronous-aware style.

Services. Our evolution model presents a service-based architecture. Interactions among communicating entities occur through service requests exclusively, i.e., the return of a service occurs also through a service request.

A service is a piece of code, performing some customized computation whenever it is invoked with appropriate parameters. A service is coupled with a formal description of itself including: the service functionality, the behavior (parameters and returned values), and some quality of service features.

Evolution of a service can then occur transparently. Indeed, replacing an old service can occur independently of the entities that are requesting it. It suffices that the new service presents the same, or a stronger description, as the one of the old service[1].

Tags. A tag is an anonymous reference, uniquely denoting an entity, a service, or an interaction (a communication). The coordinating element (the service manager) is the only element of the model able to associate to each tag the corresponding reference designating explicitly the entity or the service. Indeed, the service manager stores service publications along with their service tag, entity tag, and direct reference.

Tags are generated by the service manager, and serve different purposes. Whenever an entity is loaded in the system, it receives from the service manager, an *entity tag*, which will serve to further publish/remove its services, or to remove itself from the system. Every time a service publication is submitted to the service manager, a unique *service tag* is returned to the entity publishing the service. This tag is further used by the entity to replace/remove the service. Whenever a service request is submitted to the service manager, a unique *communication tag* is returned to the calling entity, which will then use it for declaring a new service able to receive the answer of the requested service. This communication tag is also passed to the actually invoked service, which will use it for addressing its answer. The communication tag acts as a temporary anonymous reference between the calling and the answering entities.

It is worth noting that the returns of the tags are the only interactions occurring synchronously in the model. They occur between the service manager and the entities. This fact does not prevent the global interactions among the entities to occur asynchronously.

2.2 Communication

We now describe a whole interaction among anonymous entities. Service publication, request and invocation are depicted by Figure 1, while service return is shown in Figure 2.

Service Publication. Publishing a service actually consists in announcing it to the service manager, by submitting a description of the service ①. The service manager generates the service tag, stores the service description along with the tag, and returns the tag to the entity publishing the service ②.

---

[1] Subsection 2.3 discusses service descriptions, and Subsection 2.4 the matching process.

Service Request. Requesting a service simply results in submitting a service description to the service manager ③. The service description contains as well the parameters necessary for the communication.

It is worth noting that service descriptions are not publicly available for entities or for their programmers. An entity actually tries to have its request fulfilled, by asking for some service. It can happen that there is no service available that can satisfy it.

Matching. Every time it receives a service request, the service manager performs a matching, i.e, it looks in the service publications storage for the service description which best matches the request ④.

Two cases occur: (a) no corresponding service is found, then the service manager sends `null` to the requesting entity. In this case, no communication tag is generated for the potential answer. The entity, waiting for the tag, then knows that there is no corresponding service available; (b) one or more services have been found, the service manager chooses the one that best matches the request[2]. In case, several services match in an equivalent manner the request, one of them is randomly selected. The service manager generates a new communication tag, that will be used for a potential answer ⑤. The calling entity, receiving the communication tag, will immediately use it for registering a new service ⑥, which simply waits for the answer from the actually invoked service, if there is any.

Service Invocation. We differentiate the service request, from the service invocation, where the service is actually called by the service manager to satisfy a service request.

Once a service satisfying the service request has been selected, the service manager actually invokes the selected service, passing to it: its own service tag, the service request (containing potential parameters), and the communication tag generated after the matching ⑦.

Service Return. A service, invoked by the service manager, performs the corresponding computation, using the parameters included in the service request received at invocation time. The satisfaction of a service may either consist in a computation without particular notification to the entity that requested it, or returns some resulting value or computation notification. In this latter case, the answering entity performs a service request for delivering its result.

The return of a service is an operation completely symmetric to the service request. Indeed, the answering entity furnishes the answer, by *requesting a service*, more precisely the service previously registered by the calling entity (at step ⑥). In order to unambiguously address the answer to the calling entity, it builds

[2] Subsection 2.4 defines matching, and best matching.

a service request using the communication tag it received during the invocation ⑧.

The service manager then performs a matching ⑨. It invokes the corresponding service ⑩, previously registered using the communication tag ⑥, enabling then the calling entity to receive, asynchronously, the return value. The answering entity, since it performs a service request for delivering the result, receives a *new* communication tag, as a result of the matching process realized by the service manager. However, in this case, this communication tag is not used for a further response.

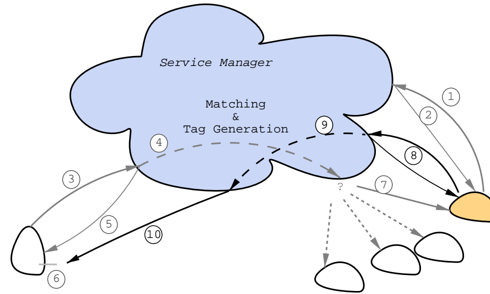

Fig. 2. Service Return

2.3 Service Description

Since there is no reference, shared among entities, there is no possibility to namely reference services. The philosophy adopted by our model consists in *describing* the requested service. Focus is put on the service and not on the particular entity that will satisfy it. Service description has two main goals:

- Allow the infrastructure to decide which service publication is the one that fits the best the service request;
- Let programmers specify, as precisely as they need, services they request or provide.

In this context, we need to know *what* kind of service we want, *how* the answering entity satisfies the service, and *how well* the entity satisfies the service. Informally, services may then be described in three parts answering these three questions respectively:

Functionality. This is the message part that describes the functionality of a service by characterizing its kind. An analogy with traditional object programming could be the names of a method, its class name, and its package name. For example, a service allowing to read on a file system has the following functionality, expressed in a pseudo-language:
`Functionality: "FileSystem": "Read"`.

Behavior. This is the information that will guarantee the compliance of the functional check. It contains basically: arguments proposed to the service to realize its task; and the way the service has to work (e.g. does it return an answer, if so, is it answering through a particular service or a service built with the communication tag, etc.). An analogy with traditional object programming could be the types of a method and its arguments. In the reading files example, the behavior specifies that it takes in parameter one String and returns the result which is also a String, using the communication tag (implicit by the use of `return`). The Behavior is expressed as follows: `Behavior: String: "return": String`.

Quality of Service. This is the part of the message that allows to select a particular service from a group of equivalent services by matching most exactly the quality of service desired with the quality of service proposed. There is no analogy with traditional programming since the choice between methods is not an issue for programmers. The reading file example would then indicate that it should be performed locally. In the direct line of the previous parts we express the quality of service as follows: `QoS: "local"`.

A service description is made of four fields: one for each part - Functionality, Behavior, and QoS; and a triple of numeric values $[d_1, d_2, d_3]$, indicating the minimum matching depth of each part.

Each part is described using: first a *preamble*, specifying the related part: `Functionality`, `Behavior`, or `QoS`; second, a *series of labels*.

In the case of the Functionality and QoS parts, the sequence of labels, serve to specify the functionality and the quality of service respectively, using an increased level of details. For instance, a service offers the functionality `Read`, in a `FileSystem` application. In the case of the Behavior part, two cases occur: either the service description is a service publication, or it is a service request. In the first case, the labels specify the types of the input parameters, the possible return value, and its type. In the second case, the labels specify typed values, a possible expected returned answer, and its type. Parameters types are only primitive types, such as String, Integers, etc.

The fourth field of a service description is a triple of numeric values indicating the required minimum matching depth each part must reach, when a service publication and a service request are compared. The matching depth of each part is the maximum number of consecutive labels that match.

For instance, the following service publication:

```
(Functionality: "FileSystem": "Read",
 Behavior: String: "return": String,
 QoS: "local",
 [3,2,1])
```

means that: the minimal functionality is 3, i.e., the service furnishes a reading file service; the minimal behavior is 2, i.e., the service requires an input parameter, but does not necessarily return an answer; and the quality of service is 1, i.e., the service prefers a local interaction, but it is not necessary.

Even though labels are well-known, agreed names, such as `Read` or `local`, it is important to note that it is not necessarily the actual name of the service that will answer the request. It is simply a way of qualifying the kind of service, used in the service description.

2.4 Matching

Matching is then executed, comparing a service request with the available service publications. The first three fields are compared separately:

- The Functionality part of the service request is syntactically compared with the Functionality part of a service publication. The matching mechanism matches labels according to their position in the Functionality part, going from left to right. The Functionality part matches at a depth n with another Functionality part, if n is the greatest number at which they match;
- The matching of the Behavior part is similar to the matching of the Functionality part, except that two different labels, related to parameters, can match, provided that one is a type name, and the other one is a parameter value of the same type;
- The matching of the QoS part is similar to the matching of the Functionality part.

A service publication is *equivalent* to another one, if it matches at a same depth for all three parts. A service publication is *stronger* than another one, if it matches at a higher depth for at least one of the three parts.

A service request *matches* a service publication if the 3 matching depths are greater than the minimum required matching depths $[d_1, d_2, d_3]$ specified in both the service request and the service publication.

The *best adapted service* for an invocation is the one that matches at the highest depth for functionality, behavior and quality of service (first we consider functionality, then behavior and finally quality of service). The minimum required matching depth ensures a minimal adequacy between what is required for a service and what is offered by the selected one, which will be actually invoked. In file system example, the service publication :

```
(Functionality: "FileSystem": "Read",
 Behavior: String: "return": String,
 QoS: "local",
 [3,2,1])
```

matches with the following service request:

```
(Functionality: "FileSystem": "Read",
 Behavior: "myFile.html" : "return": String,
 QoS: "local",
 [3,4,1])
```

Indeed, the matching depth is `[3,4,2]` and it is bigger than the one specified in the service publication (`[3,2,1]`), and that of the service request (`[3,4,1]`). In this case, the service request needs the service to return a value, therefore the required depth for Behavior is 4.

Although, in the above example, we use a flat structure to represent each part, the service description supports a tree-like structure enabling alternatives [27]. Fields corresponding to Functionality, Behavior, and QoS are defined as a tree. It is possible to express conjunctions (between an upper level and its associated subtree) and disjunctions (among different branches) in a service description.

2.5 Unanticipated Run-time Evolution

Entity Evolution. An entity is first loaded in the system, and further replaced or removed. In the case of replacement, a state capture functionality (i.e., a method of the entity called by the coordinating element) enables the transfer of the state of the old entity to the new entity. Both for replacement and removal, services publications related to the entity are removed from the repository.

Service Evolution. Service evolution consists in replacing a service by another one which has the same or a stronger service description. Once the new service has been declared to the service manager, it can then be selected during a matching process. The replacement can occur after the old service has been removed; or before its removal, ensuring the continuous availability of the service. The service manager is notified of a service removal. It then removes the service description from its service publications repository. If both the old and the new services are available simultaneously, the service manager will indifferently choose one of them during a matching and selection process. It is worth noting that a service evolution does not imply an entity replacement or removal.

Figure 3 shows a service request ①, followed by a matching process ②. The old version of the service is no longer available, but the new one is already registered. Since it satisfies the service request, it is selected by the service manager and actually invoked ③. The interaction then proceeds as usual. The new version returns the answer (if there is any), by requesting the appropriate service ④. After a new matching process, the service manager invokes the service dedicated to receive the answer ⑤.

Removing a service is independent of service requests, and generally does not raise problems, except in some situations, where it may lead to an absence of service satisfaction. Let us consider the different cases: (1) if the answering service is removed before a matching service request, or after it has returned, the service is not available, and no problems arise; (2) if the service is removed after a matching request, but before a matching selection, the service cannot take part in a matching process, it is simply not available. If no equivalent service is found, then the calling entity receives `null`; (3) the service is removed after a matching selection, where it was selected, but before its actual invocation from the service manager. The calling entity is notified that the computation will

occur, however if the called entity is actually removed, nothing happens; (4) the service is removed after its actual invocation, but before it returns. This is similar to (3), the calling entity never receives an answer. The last two cases (3) and (4) show that in an asynchronous schema, an entity must be programmed to take into account unexpected cases: missing computations and results.

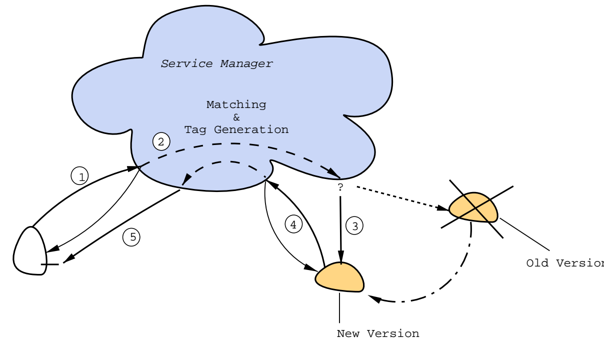

Fig. 3. Service Evolution

Unanticipated Evolution. The proposed evolution model favors unanticipated evolution at run-time in the following sense. There are no pre-defined entry-points were code can be modified at run-time. There are no typing or hierarchical constraints on new versions of services, and entities with respect to old versions. Calling and answering entities are not known at design time. In addition, their service descriptions are not needed by other entities, either at design time, or at run-time. In the distributed implementation of the model, entities are located on different hosts, but none of them knows the location of the others. The new version of a service can likely be located on a different host than the old version.

However, of course, entities and services must be programmed according to the model, and consequently, their design integrates, in an anticipated manner, mechanisms for run-time evolution: service descriptions, anonymity and asynchrony of computations, as well as possible lack of service return.

2.6 Model Properties

The proposed model, based on a disconnected service architecture, actually enables disconnection: service descriptions enable anonymity and asynchrony, and remove the need for relying on fixed APIs at run-time; the service manager acting as a coordinating element frees the entities from being aware of references.

In the case where several services, available simultaneously, are able to satisfy a given service request, consecutive service requests may likely be satisfied by

different services. At each request, the service manager performs a matching and a random selection among equivalent services, i.e., among those having the same level of matching.

Replacing an old service by a new one does not impose any inheritance or sub-type constraints on the new service. The only condition is related to the service description of the new service, which must be equivalent or stronger than the service description of the old one.

When the removal of an old service is preceded by the registration of the new version, the *permanent availability* of the service is guaranteed.

The matching performed on service descriptions is a syntactical matching, the labels have to match perfectly at the syntax level (except for the parameters part, whose types only have to match). Labels implicitly refer to an ontology, for instance, it is clear for all parties that a `Read` actually stands for reading. The presented matching scheme supports the insertion of additional label categories. It can be made more flexible, by allowing semantical labels, instead of syntactical ones.

All entities and services can evolve at run-time. The coordinating element, maintaining the connections, is the only element that cannot evolve without requiring to stop and restart the whole system. However, depending on the way it is implemented, some parts of it can be implemented as entities, and thus be subject to run-time evolution. This will be shown in Section 3.

3 Implementation

There are currently two implementations of our model. The first one authorizes the evolution of *local* Java applications, while the second one enables the *distribution* of the entities computations among several hosts.

3.1 Local Implementation

This subsection presents the local implementation, called LuckyJ, of our model. The name we chose comes from the "I'm feeling lucky!" functionality of the Google[3] search engine, which redirects the user to the Web page that best matches the request. Similarly, our infrastructure invokes the service that best matches the service request.

In the framework of this implementation, entities and the coordinating element are all loaded in the same JVM, while in the distributed case entities can all be loaded in different JVM.

Figure 4 shows the different elements of the platform. As described in section 2 we have entities, services, and tags. In the implementation, the coordinating element consists of two different parts: a *description passer*, which maintains entities references, and a service manager, which performs matching, and service selection. This division is due to the observation that in previous works on

[3] http://www.google.com

evolution [15], the main reason to stop the whole platform was due to a bug discovered in the platform itself. Thus, we chose to modularize the platform, in order to enable evolution of some parts of it. The description passer, maintaining connections, cannot evolve at run-time. However, the service passer becomes a regular entity, and can likely evolve dynamically.

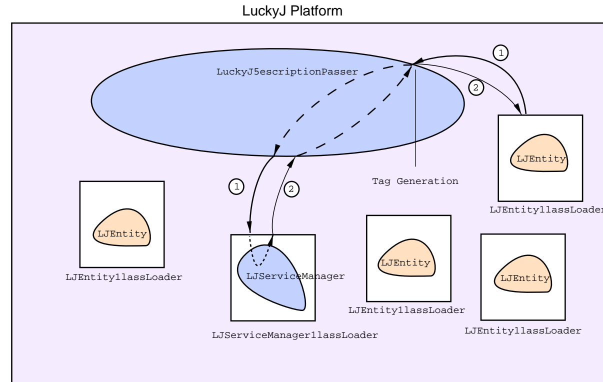

Fig. 4. LuckyJ Platform: Service Publication

The *description passer* is the coordinating component of the platform, redirecting all the messages from the entities to the service manager, and vice-versa. It maintains references, and generates tags. The description passer receives requests for loading entities, as well as service requests, and service publications. Due to its central nature, we put in it administrator interfaces, keeping a trace of: all published services; all entities; and all active threads present in the service manager. These interfaces allow the administrator of the platform to kill an entity or a service manually, and to ensure consistency in the platform and in its threads.

The *service manager* is the element of the platform that matches service requests with service publications, and performs a random selection among equivalent matching services. The default implementation stores the different service publications in a labeled tree that simplifies the matching process and maintains a consistency between announced services and invoked services. It is, for example, impossible to remove a service during a matching process (while the algorithm goes all over the tree). Two threads control a request and an invocation respectively. The first thread, originating from the calling entity, serves the request and reception of the communication tag. The second thread, created by the service manager, whenever a matching service has been found, serves the invocation of the selected service. The use of two threads provides asynchrony in service calls, and computation.

An *entity* is an aggregate of user defined classes and user-defined instances. This means also that an entity is loaded in its own `ClassLoader`. In that manner,

different versions of the same classe(s) can be loaded in the same JVM. It means also that once an entity is deactivated (does not have any more active thread) and that there is no more reference on it and its code, the garbage collector may remove the code itself as specified in the JVM specifications [20].

Tags are built using two values: a unique number for each tag in the platform (an incremented integer), and a sparse value (a big size random number). The size of the random number guarantees the uniqueness, in the sense that the probability to generate two identical sparse numbers is very low.

Figure 4 shows a service publication. An entity contacts the description passer for registering one of its services ①. The entity sends its entity tag, and the service publication. The description passer simply forwards the service publication to the service manager along with a newly generated service tag, returned to the entity ②. The description passer keeps a trace of the service tag only for administrative purposes. The different parts of the service publication are stored into three labeled trees maintained by the service manager. The nodes of these trees contain a description object, together with a list of the services providing it.

Figure 5 shows an entity that requests a service, from another entity, by contacting the description passer. It sends its own entity tag, together with the service request ③. The description passer simply forwards the request to the service manager ④.

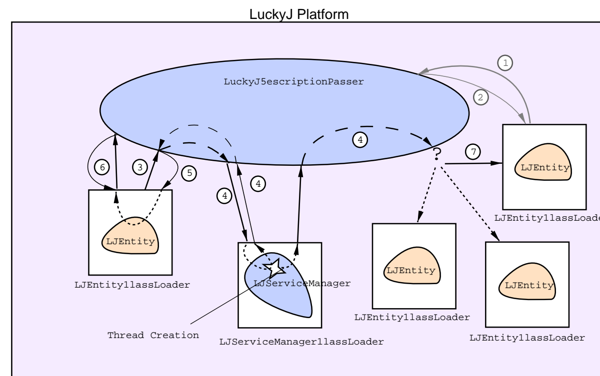

Fig. 5. LuckyJ Platform: Service Request and Invocation

Once it receives a service request, the service manager performs a matching, i.e, it looks in the service publication tree for the service description which best matches the request.

Once a service satisfying the service request has been selected, the service manager creates a new thread dedicated to the selected service invocation. The service manager passes the service request along with the service tag, and the communication tag to the description passer ④. Finally, the description passer actually invokes the corresponding method of the answering entity ⑦.

Evolution. This subsection describes various evolution scenarios allowed by the platforms. This allows us to show additional features, provided by the platforms such as state capture (and retrieval) functionality, and finalization of an entity.

The replacement of an entity by another one is decomposed as follows: (a) the description passer loads and instantiates the new entity, (b) once loaded, the description passer captures the sate of the old entity and transfers it to the new one, (c) it then removes the old entity (see next paragraph for more details). The default implementation of the state capture functionality consists in capturing by reflexivity all the public fields value into a single String. In order to capture as well the state of non public fields, programmers of components must override the default implementation of the state capture method.

Removing an entity means that the description passer stops all threads created by this entity (an entity has its own thread group). It then unregisters all its published services, and finalizes the entity.

Adding an entity is simply the act of loading a new entity. Nevertheless, from an evolution point of view, it is possible to have several versions of the same entity simultaneously. Thus, it is also important to allow micro-evolutions, where the administrator manipulates (adds, removes) directly the published services.

Due to the intangible nature of services, replacing a service is assimilated to publishing a new description and removing the old one. Adding a service simply consists in registering an additional service publication. Removing a service means that the service manager removes the service publication from the storage tree. This can occur only when no matching process is running. The service tag maintained by the description passer is also removed. Invocations cannot be made, even if the service has just been selected during a matching process.

3.2 Distributed Implementation

The LuckyJ distributed implementation is intended first, to offer the *same evolution model*, but with services distributed on several LuckyJ platforms, allowing thus space decoupling. Second, the distributed implementation allows maintainers of an application, written on top of LuckyJ, to distribute it *on-the-fly* on several physical networked hosts, and *without rewriting any piece of code.*

These constraints imply, among others, that the whole system guarantees to a requesting entity, running in the system, the same behavior as in the local case, i.e., the best matching service, present in the (global) system, will satisfy its request.

The chosen architecture is distributed (among several platforms), but still centralized, as depicted in Figure 6. Platforms are organized hierarchically in a tree-like structure. A root platform stores all service descriptions, performs the matching for service requests, and generates tags. Service publications, and requests are forwarded from local platforms up to the root platform, while matching results, tags, and service invocations are forwarded down to the local platforms.

In order to realize this architecture, and to satisfy the constraints explained above, we have introduced a new kind of entity, called *server entity*, whose

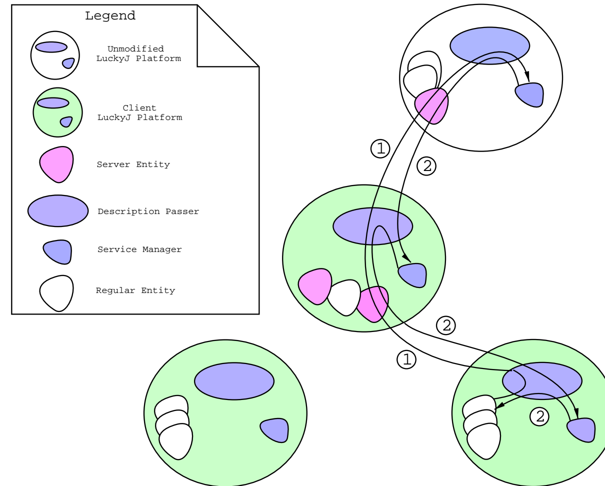

Fig. 6. Distributed LuckyJ: Service Publication

role is to enable communication among two platforms, and forwarding of the information up or down the hierarchy of platforms. Server entities are present in all platforms, excepts leafs in the hierarchy.

All platforms, except the root platform, have the particularity that they have been equipped with a *client description passer* that relays service registration/removal and invocations. Such platforms are called Client LuckyJ platforms. The client description passer is a key part of our distributed platform. First, it forwards up in the hierarchy any service publication, and service request. Second, it passes any communication tag, and matching results to both: the server entity (for forwarding the information down the hierarchy); and to its local service manager (for registering any newly published service).

The top platform is an unmodified LuckyJ platform, i.e., its description passer is the same as in the local implementation. Nodes in the hierarchy are clients of the platforms that are above them in the hierarchy.

Service and communication tags are generated by the service manager in the root platform, hence they are uniquely built. Entity tags are generated by the local platform, where the entity is loaded. The uniqueness in this case is guaranteed by combining a newly generated sparse name, with a sequence number.

Communication in a cluster of LuckyJ platforms fits to the model, since introduction of the distribution is made transparently for the entities. Indeed, entities still register their services, or address their service requests locally to their description passer. The further forwarding of the publication or requests up in the hierarchy is transparent for the entities. The only difference may reside in the latency introduced by the networked communication. Indeed, registration or request of a service, followed by the reception of the service tag or the com-

munication tag respectively, are the only synchronous operations in the model, thus subject to latencies.

Figure 6 shows a service publication in the distributed implementation. A service publication is always effected as many times as there are levels in the clusters from the local LuckyJ platform to the top platform. The entity, publishing a service, first contacts its local description passer (as in the local implementation), which in turn immediately forwards the publication to its parent description passer, communicating through the server entity. The publication then reaches the root platform, and the registration occurs in its service manager ①. This service manager generates the service tag, and returns it to its local description passer, which forwards it back to the client description passer, where one thread is waiting for the service tag. The service tag propagation follows then the same path in reverse order as the publication ②. At each platform, the service manager registers as well the service.

Evolution. We chose to prohibit *remote* loading and removal of entities, since these operations may arise in different administrative domains. The default case is to allow such possibilities on local platforms only.

Transfer state functionality from an entity loaded in a platform to another loaded in another one (see figure 7) is prohibited. In the case where a whole entity is delocalized, the new entity starts with its own initial state.

Figure 7 shows a service replacement. Unregistering the old version of the service follows the same steps as the registration (up and down the hierarchy). The new version of the service registers regularly to its local description passer (and hence, is available globally). Once a request for the service arrives, the matching process (in the root platform) simply chooses the new version of the service ④, leading to its actual invocation ⑦.

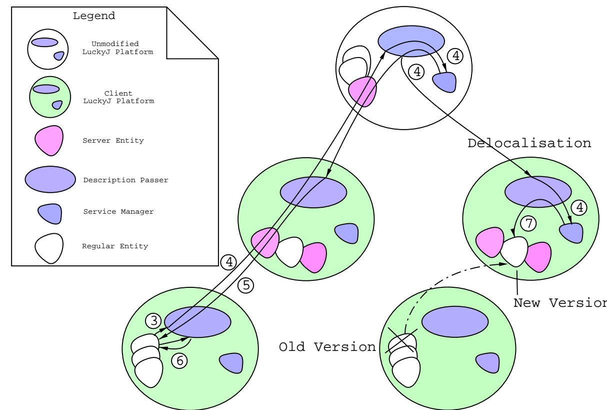

Fig. 7. Distributed LuckyJ: Service Evolution

3.3 Implementations Properties

The LuckyJ local platform is relatively lightweight as the core platform is coded in approximatively 5000 lines (25 Java classes) and the graphical administrator interfaces are coded in approximatively 1000 lines (7 Java classes). Although our primary goal was not to completely optimize the code but to provide a first prototype for testing purposes, we chose to have structures (especially for matching) that may use more memory, but reduce the algorithms complexity. Measurements made with 1000 identical services (worst case) show that 100 requests are effected in approximatively one minute (on a Pentium II at 450 MHz under Linux). In fact the more diverse the proposed services are, the more the matching is fast, as shown by the benchmarks below.

The distributed implementation sticks to the original model, described in Section 2. It benefits from its intrinsic properties of disconnection, asynchrony, associative naming and anonymity. The distributed architecture guarantees (globally) the best matching.

Additional properties of the distributed architecture are: (a) distribution of the processing load is performed *transparently* and at *run-time* for the entities; (b) the same code is used for distributed entities and for non-distributed ones. It is not necessary to change the code of the entities. The anonymous communication used for requesting services works in the same way for both the local and distributed case.

Finally, we can mention the fact, that if one the branches of the tree structure of platforms is disconnected from the whole tree (due to network problems), this branch can still continue working, even though in a restricted form, i.e., with the services available under this branch.

Benchmarks. Several benchmarks have been realized on the local platform [29]. The *scalability benchmark* measures the time needed by the infrastructure to invoke a service. This benchmark shows that time is constant and low, independently of the number of services, provided their descriptions are different (e.g., 90ms for 20000 different services descriptions). Time increases exponentially already for a few number of services, when most services have all the same description (e.g., invocation time is 17s for 20000 services and 3 different services descriptions). The *service description* benchmark measures the time employed to invoke a service with respect to the service descriptions' lengths and to the number of services. It shows that lengths over 800 labels cause time picks, while under 800, time is constant. In both cases, the result is independent of the number of services. The *evolution benchmark* measures the time needed to replace large numbers (45000) of services with respect to the number of entities. State transfer has been realized using the default implementation of the state capture functionality. Results show that the time grows in a polynomial manner with respect to the number of entities. In addition, the maximum average time length for replacing a single simple entity is less than 2ms (52000ms/45000).

In target applications, we consider that the common case is to have, in a local application, less than 500 components that use few services each of a length less

than 50. In such an environment, invocations and component replacements are effected in less than 50ms in any configuration.

The above benchmarks show few things:

- The more services have distinct descriptions the more the platform is fast when an invocation is made. This indicates that other solutions should be found for large peer-to-peer systems in which service descriptions may probably be very similar.
- Structures for representing services use a lot of memory space. Thus, in a potentially global application built on LuckyJ, other ways should be found to stock the service descriptions and effectively perform matching. This implies using massive storage applications like databases and probably delocalizing the matching to external structures.
- Massive run-time evolution of components is a task that appears to behave with at most a polynomial complexity. In fact, our algorithms should show a linear behavior, but the internal JVM memory management implies that deviations may arise.

4 Case Studies

This section describes two case studies realized on top of the local implementation of our evolution model.

4.1 WEb Evolving SErver on LuckyJ (WeeselJ)

WeeselJ is a Web server, with a restricted functionality. It is on-line since its early stage of development (October 2002). It displays a lot of information (like complete API of LuckyJ) and is accessible on-line[4].

In WeeselJ, there are two types of entities: (1) resident entities that provide services to other entities, and may keep values in the long term; and (2) servlet entities that are loaded on demand by the Web server, according to the clients requests. The use of two types of entities is motivated by the fact that, on one hand, there are entities that should remain in WeeselJ indefinitely, like the entity that listens to HTTP requests. But on the other hand, it is also convenient to provide a light-weight mechanism, for server-side code execution, through servlet-like entities.

In its actual version, the entire WeeselJ counts 7 classes for a total of approximatively 1000 lines of code. The entity, listening to HTTP requests, has had more than 160 different versions and each of them had been tested without stopping the server. The other entities have had at least 20 different versions each, that were tested independently. Web pages have been showed continuously since the third day of development[5] except in 5 occasions where the application

[4] http://www.weeselj.org

[5] Early October 2002. (Current time: October 2003)

had to be restarted: once for correcting a bug in the description passer related servlet-like entities, once because of an NFS problem, once for migrating on another platform (distributed implementation was not yet available), and once because of an infinite loop in an entity that crashed the JVM, and once because of a power cut. At this moment, the Web server has shown an availability of 99.99 percents.

4.2 Tic-Tac-Toe

After two hours presentation, this case study constituted a practical work for third year students at University of Geneva, and has subsequently been used for open days of our department. During this event, visitors were given the possibility to freely build their own version of the Tic-Tac-Toe using the diverse implementations students made. Unadverted public could then compose on-the-fly its own tic-tac-toe game, and change parts of the application *during the play*. This example demonstrates that the evolution model is also powerful for component development and integration, allowing to duplicate and remove parts of an application at run-time.

There are two kinds of components available, each in several versions:

- Game Display : this component represents the basic user interface to communicate with the rest of the application.
  - Colored Graphical User Interface
  - Textual User Interface
- Partner's choice
  - Physical adversary on the network
  - Computer with different levels

The user combines two components, one of each kind, in order to build his game. It then can modify a component at run-time to either change the display, passing from a colored GUI to a textual interface, or to change its adversary, e.g., changing the level of the computer adversary.

Students who had to compute the different versions of the components, where given the LuckyJ platform, and the service description of the Game Display, and of the Partner's choice component services.

Each student group then implemented the entities, and two versions of the corresponding services, leading to as many different implementations of a service as the number of students. These implementations where then all registered in the system, and all available simultaneously.

The key feature in the evolution consisted in allowing the visitor to change the graphical interface of the game or the adversary level, while still being able to continue the game. Changes were unanticipated since participants could freely chose the components forming their own application.

Even though, the evolution in this example is limited, it nevertheless shows the power of the proposed architecture, since unexperienced people could easily build an application, and subsequently let it evolve at run-time.

4.3 Development Feedback

The feedback we have from developing with LuckyJ results from either the previously shown Web server, and Tic-Tac-Toe examples and from diverse other development we made like a small text editor [28], distributed stickers, and a semi-centralized version of the distributed implementation useful for P2P applications [35].

Testing on-the-fly. When developing using the LuckyJ environment, the main interest is not to care for implementing a complete behavior. Indeed, we made tests and, when entities were evolution-oriented (see next point), we began an iterative test-and-program cycle. Usually, with LuckyJ we may test every step of programming, since input/output is generated by the transfer state protocol, and having incomplete functionalities may only lead to invocations that have no effect. While testing, we used the transfer state functionality to initialize data correctly without re-launching the platform. When functionalities were missing, we did not have to program any wrapper. It seems that programming with LuckyJ is similar to programming with languages such as Smalltalk, since the platform is permanently running and, any changes are allowed. But although Smalltalk is fundamentally untyped, the Java language guarantees a typing consistency inside an entity. Thus, we are able to make applications evolve during the development cycle, while developing them.

Building evolution-oriented entities. It appears that developing entities and testing them on-the-fly is an easy task with LuckyJ under certain conditions. The first condition is that entities should always have a transfer state functionality implemented. This frees the programmer from relying on the basic transfer state protocol, which may loose information like `private` variables state. The second condition is to always finalize entities, and make sure that there are no other Java objects that reference objects created by the entity. This allows the garbage collector to collect the code of the entity. Consequently, this results in a long-living application that does not grow continuously and that also has better performances and features.

Strengths and Limitations. Evolution of code can be performed at any moment, e.g., after a service request, but before a service return. The communication tag can be transferred from an old service to a new one, thus ensuring the continuity of a request.

There are no constraints on a new version of a service, such as inheritance or sub-typing adequacy. Even its service description may change, in this case calling services may simply not have their request satisfied. Absence of answer, or no satisfaction of a service request is considered a normal case. This implies that services must be programmed in order to take into account missing computations, and missing results. This turns out to be useful to make the applications more robust to network or peer failures.

The model is scalable in space, since entities may easily and transparently be distributed on a network, without changing any line of code.

The implementation of the matching algorithm follows an efficient algorithm based on trees, avoiding duplicate storage of identical labels, and consequently duplicate searches. However, due to the potential complexity and number of service descriptions, the whole matching process reveals to be rather low.

We did not perform a formal change impact analysis [3] of our platform. Nevertheless, we can mention that changes of services entities implementations, including changes to the service passer, should not affect other services or entities, provided the new version of the service correctly implements the service description. However, if changes are made on the core LuckyJ Java package itself, the platform, and consequently all entities, need to be stopped. Indeed, this package defines Java parent classes for entities, services, primitive types, description passer, and service manager. Clearly, such changes have an impact on the entities that may need to be reprogrammed or tested. At a higher-level, if new versions of the description passer or the service manager modify interfaces for publishing services, and requests, or for loading, unloading entities, etc., it is clear that services will need as well to be reprogrammed.

5 Related Work

On Dynamic Loading. In literature, run-time evolution of applications is a preoccupation that migrated from the operating systems to the programming paradigm. Indeed, early works have been proposed to build dynamically loadable libraries for C and C++ programs [8, 14, 30, 32]. The main problems addressed by these approaches were mainly technical ones based on dynamic loading of code in a compiled environment. Several restricted solutions have also been presented [14, 8] for managing the evolution of versions in these environments. In the Java language, the `ClassLoader` abstraction for loading code [19, 20] opened a wide range of perspectives for code evolution. All these techniques constitute basic building blocks for programming evolution oriented platforms, but usually the models still consider evolution as an exceptional event. In addition, for security of execution purposes, these models restrict possibilities of evolution. This means that, usually, a textual identifier (like a function name) is considered as sufficient to identify a functionality. This implies that there are difficulties to have different implementations of the same functionality coexist. In the case where some versioning possibilities are open, it is assumed that there is a single code provider for a given functionality. This seems to us too much restrictive to be applied to a, potentially, fast evolving environment, where it is common to have several implementations of a single functionality.

On Evolution Infrastructures. In the past few years, several works have been realized in order to provide infrastructures (i.e., run-time platforms) to programmers and system administrators, allowing run-time evolution of applications. In compiled environments the works of Gupta [12, 13] study the time points at which

on-line software version change may occur depending on the threads execution. This work, however, cannot be applied to object programming [11] because of the many object inter-dependencies in such languages. Indeed, this model relies on state mapping, which is difficult to implement for dynamic object creation.

Hicks [15] proposes an infrastructure for allowing type-safe evolution of applications programmed in a typed C by adding an indirection level for each function call. In a sense, this work focuses on similar goals as the work presented in this paper. However, it does not address object-orientation approach, and its solution is not easily applicable to objects.

In the object-oriented world, most works are concerned: by safely transferring state of objects from one class to another and vice-versa [34, 9]; by the typed safe replacement of methods at run-time [7]; or possibly both [21]. These approaches are more low-level than ours and lack the possibility to make arbitrary changes. Indeed, changes imply that new classes are either sister classes, or sub-classes of old classes, restraining a lot the evolution possibilities.

On Components Evolution. In component-oriented software development, approaches often consider that the key feature, to make an application evolve, is to be able to change the inter-component connectors and to replace components at run-time [26], possibly using reflexive mechanisms [31]. This works resemble to ours for their granularity and also for connecting components which constitutes also a main ingredient of our model. However, our approach focuses on the fact that these connectors are resolved each time a communication is made in the architecture, allowing to consider the evolution mechanism as a common one rather than as an exceptional one.

In industrial approaches like Enterprise JavaBeans [36], CORBA [23], services are referenced through a naming service (namely JNDI [18] for JavaBeans or the naming and trading services [22] for CORBA). A component willing to use another, unknown one, must make a research through these services and, once found, it has to decide if it is the right one, given some specific informations, and then use it in a potentially long-term communication process. This has several implications: (1) this means that a programmer should code the part that decides to use one service or not, (2) if the communication process is long enough, it should be stopped when updated. In our architecture, clients use associative naming and communications are basically anonymous. This has several advantages as the fact that we are able to split a component into several independent parts, we may easily use delegation mechanisms, we may transparently make the whole system evolve leaving an old version of the code while a newer one is present.

The work that is probably the closest to ours is the work of Sadou *et al.* [33] which describes a way to compose services. In this work, linking service calls with services invoked is made at invocation. There is an agreement made to find out if a common environment provides the desired services. In fact, the agreement is made on groups of method signatures that are called a *role* in the environment. Although we also rely on a late binding mechanism, we do not constrain

ourselves to having a common referential for defining a role, but we rather build a completely open matching mechanism. Thus, we consider that services may come and leave from our system anytime without being bound to specific entry-points, our only entry-point being the service manager. This constitutes a more adapted mechanism for peer-to-peer networks and open systems in general.

On Ad-Hoc and Peer-to-Peer Applications. A common problem for the ad-hoc networks paradigm and peer-to-peer application paradigm is service discovery [4]. Numerous applications are possible for a system that would allow to find and use services directly when needed. Our approach fits completely to these requirements and we proposed diverse implementations in this direction [27]; the distributed architecture described in this article is only one of them. In JXTA [24], for example, a common limitation is that peer-group services are statically defined. This implies that specific search services must be defined when willing to use a non-peergroup service [38]. Although it is always possible for a peer to propagate a service announcement through the peer network, our approach would allow to have a peer-group service that adds dynamically services at the peer-group level.

On Dynamic Services and Service Description Languages. Few service description languages have been described. As an example WSDL [5] is a norm for describing web services. The description is constituted by a document that binds some parts to other documents (like protocols description). The goal of WSDL is mainly to allow programmers to describe services. Once found, active components have to decide if the described service corresponds to a valid possibility or not. Comparing to our infrastructure, there is no service manager infrastructure that manages the matching between services although one could easily imagine how to integrate our infrastructure with WSDL descriptions in order to automate the matching. UDDI [37] allows to fill the gap between describing and finding a service. It constitutes a phone-book used to find services. Services described using WSDL can be published and retrieved through UDDI [6]. Adding WSCL to the whole system [2] allows to decouple conversations, i.e., the order in which messages are exchanged, from the exchange of message itself. The main difference with our solution resides in the fact that, in the WSDL case, it is the programmer's role to choose between services matching the description. This also means that there is no way to quantify the quality of matching. Communication is not anonymous and components have references on each other when the server and the client have been defined. In our model, we consider simple service specifications, so that programmers can easily integrate the service descriptions to their code. Nevertheless, we are also considering the possibility of using more powerful description service formalisms and thus, possibly let our infrastructure be integrated with components that propose those services.

The Adaptation Description Language (ADL) is part of WSXL [1], and is used to adapt the output of a WSDL component without invoking it. Adaptation points (data, presentation and control adaptation) convey updates (location, op-

erations, and constraints) made to the component's output, such as changing a page's color, or button's effects. At design-time, an intermediary (between the application provider and the end user) defines adaptation points that adapt output to particular end-users. At run-time, the supporting middleware, intercepts user inputs and performs the adaptation, defined by the intermediary at the corresponding adaptation point. In this case, changes are statically foreseen at design-time by both the application provider and the intermediary. The components, whose output is adapted, is itself not modified, it did not undergo code evolution. In this case, the more adaptations are needed, the more adaptation points and corresponding code are added to the whole application.

6 Conclusion

The evolution model proposed in this paper is based on a disconnected service architecture. Entities communicate exclusively through services, favoring fine-grained modular code and data structures replacement. In addition, interactions among entities and services occur anonymously and asynchronously, relying on service descriptions. This frees entities from synchronization, references and APIs constraints. In this paper, we described our model, two prototype implementations, and the implementation of a tic-tac-toe game. Preliminary results and experiments show that this lead seems promising for run-time evolution, but also for software engineering purposes, and for a wide range of applications that possess unanticipatedly evolving internal organisation, and finally for those consisting of autonomous components.

Future work on this subject will be concentrated on enlarging the scope of applicability the model has, and in implementing distributed platforms adapted to these application domains. These application domains are typical domains for which there is a need for disconnection, such as: peer-to-peer networks and *ad-hoc* networks, where peers may appear/disappear asynchronously [27]. A possible application would be to allow dynamic services in JXTA [24] peer-groups.

Another lead we follow is the possibility to describe services using a more detailed specification than the current service description. These specifications would then possibly be automatically generated at compile-time (and verified at load-time). This will be particularly useful for building self-healing and self-organizing applications [17]. In this case, our platform would act as a middleware allowing entities to publish their specifications and requests. The self-organising mechanism will then be obtained, by the fact that entities, by their requests, anonymously, and in an unpredictable manner, indirectly activate other entities, without even knowing them. This constitutes a long-term project for which the work we described in this article constitutes only a first step.

7 Acknowledgements

This research is partially supported by Swiss NSF grant 21-68026.02.

References

1. A. Arsanjani, D. Chamberlain, D. Gisolfi, R. Konuru, J. Macnaught, S. Maes, R. Merrick, D. Mundel, T. Raman, S. Ramaswamy, T. Schaeck, R. Thompson, A. Diaz, J. Lucassen, and C. Wiecha. (WSXL) Web Service Experience Language Version 2. Technical Report IBM Note 10 Avril 2002, IBM, 2002.
2. D. Beringer, H. Kuno, and M. Lemon. Using WSCL in a UDDI registry 1.02. http://www.uddi.org/pubs/wsclBPforUDDI_5_16_011.doc, May 2001.
3. S. Bohner and R. Arnold, editors. *Software Change Impact Analysis*. Tutorial Series. IEEE Computer Society Press, Los Alamitos, CA, USA, 1996.
4. H. Chen, A. Joshi, and T. W. Finin. Dynamic Service Discovery for Mobile Computing: Intelligent Agents Meet Jini in the Aether. *Cluster Computing*, 4(4):343–354, 2001.
5. E. Christensen, F. Curbera, G. Meredith, and S. Weerawarana. Web services description language (WSDL) 1.1. http://www.w3.org/TR/2001/NOTE-wsdl-20010315, march 2001.
6. F. Curbera, D. Ehnebuske, and D. Rogers. Using WSDL in a UDDI registry 1.05. UDDI Working Draft Best Practices Document, http://www.uddi.org/pubs/wsdlbestpractices-V1.05-Open-20010625.pdf, June 2001.
7. M. Dmitriev. Towards Flexible and Safe Technology for Runtime Evolution of Java Language Applications. In *OOPSLA Workshop on Engineering Complex Object-Oriented Systems for Evolution*, 2001.
8. S. M. Dorward, R. Sethi, and J. E. Shopiro. Adding new code to a running C++ program. In USENIX Association, editor, *USENIX C++ conference proceedings: C++ Conference, San Francisco, California, April 9–11, 1990*, pages 279–292, Berkeley, CA, USA, Spring 1990.
9. S. Drossopoulou, F. Damiani, M. Dezani-Ciancaglini, and P. Giannini. Fickle : Dynamic object re-classification. In J. L. Knudsen, editor, *Proceedings ECOOP'01 – Object-Oriented Programming*, volume 2072 of *LNCS*, pages 130–149. Springer-Verlag, 2001.
10. P. T. Eugster, P. Felber, R. Guerraoui, and A.-M. Kermarrec. The many faces of publish/subscribe. *ACM Computing Surveys*, 35(2):114–131, 2003.
11. D. Gupta. *On-line Software Version Change*. PhD thesis, Department of Computer Science and Engineering, Indian Institute of Technology, Kanpur, 1994.
12. D. Gupta and P. Jalote. On-line software version change using state transfer between processes. *Software/Practice and Experience*, 23(9):949–964, 1993.
13. D. Gupta, P. Jalote, and G. Barua. A formal framework for on-line software version change. *IEEE Transactions on Software Engineering*, 22(2):120–131, February 1996.
14. G. Hamilton and S. Radia. Using interface inheritance to address problems in system software evolution. *ACM SIGPLAN Notices*, 29(8):119–128, 1994.
15. M. Hicks, J. Moore, and S. Nettles. Dynamic software updating. *ACM SIGPLAN Notices*, 36(5):13–23, May 2001.
16. J. Hunter. *Java Servlet Programming*. O'Reilly & Associates, Inc, 1998.
17. J. O. Kephart and D. M. Chess. The Vision of Autonomic Computing. *Computer*, 36(1):41–50, January 2003.
18. R. Lee and S. Seligman. *JNDI API tutorial and reference: building directory-enabled Java applications*. Java series. Addison-Wesley, Reading, MA, USA, 2000.

19. S. Liang and G. Bracha. Dynamic Class Loading in the Java^{TM} Virtual Machine. In *Proceedings of the 13th Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA-98)*, volume 33, 10 of *ACM SIGPLAN Notices*, pages 36–44, New York, October 18–22 1998. ACM Press.
20. T. Lindholm and F. Yellin. *The Java Virtual Machine Specification*. The Java Series. Addison Wesley Longman, Inc., second edition, April 1999.
21. S. Malabarba, R. Pandey, J. Gragg, E. Barr, and J. F. Barnes. Runtime Support for Type-Safe Dynamic Java Classes. In E. Bertino, editor, *Proceedings ECOOP'00 – Object-Oriented Programming*, volume 1850 of *LNCS*, pages 337–361. Springer-Verlag, 2000.
22. Object Management Group. Naming service specification–revised edition. http://www.omg.org/cgi-bin/doc?formal/01-02-65, February 2001.
23. OMG. The comon object request broker: Architecture and specification. Technical Report PTC/96-03-04, Object Management Group, 1996. Version 2.0.
24. Ra Ti On. Project JXTA: An open, innovative collaboration. http://www.jxta.org/project/www/docs/OpenInnovative.pdf, 2001.
25. D. O'Quinn. *Photoshop in a Nutshell: A Desktop Quick Reference*. O'Reilly & Associates, Inc., Newton, MA 02164, USA, second edition, 1999.
26. P. Oreizy, N. Medvidovic, and R. N. Taylor. Architecture-based runtime software evolution. In *Proceedings of the 1998 International Conference on Software Engineering*, pages 177–186. IEEE Computer Society Press / ACM Press, 1998.
27. M. Oriol. Peer Services: from Description to Invocation. In G. Moro and M. Koubarakis, editors, *First international workshop on agents and peer-to-peer computing, AP2PC 2002*, volume 2530 of *LNCS*, pages 21–32. Springer-Verlag, 2002.
28. M. Oriol. LuckyJ: an Asynchronous Evolution Platform for Component-Based Applications. In P. Constanza and G. Kniesel, editors, *USE'03 - 2nd International Workshop on Unanticipated Software Evolution*, pages 40–49, 2003.
29. M. Oriol and G. Di Marzo Serugendo. Application evolution: A disconnected and service-based approach. Technical report, Centre Universitaire d'Informatique, University of Geneva, Switzerland, 2003.
30. A. J. Palay. C++ in a changing environment. In USENIX Association, editor, *Proceedings: USENIX C++ Technical Conference*, pages 195–206, Berkeley, CA, USA, 1992.
31. B. Redmond and V. Cahill. Supporting unanticipated dynamic adaptation of application behaviour. In B. Magnusson, editor, *Proceedings ECOOP'02 - Object-Oriented Programming*, volume 2374 of *LNCS*, pages 205–230, Malaga, Spain, 2002. Springer-Verlag.
32. A. Richter. DLOPEN(3). Linux Programmer's Manual, December 1995.
33. S. Sadou, G. Koscielny, and H. Mili. Abstracting services in a heterogeneous environment. In R. Guerraoui, editor, *Middleware 2001 : IFIP/ACM International Conference on Distributed Systems Platforms, Proceedings*, volume 2218 of *LNCS*, pages 141–159. Springer-Verlag, 2001.
34. M. Serrano. Wide classes. In R. Guerraoui, editor, *Proceedings ECOOP'99 – Object-Oriented Programming*, volume 1628 of *LNCS*, pages 391–415. Springer-Verlag, New York, NY, 1999.
35. D. Stadler. Invocations de Services dans un système d'Information de type Peer-to-Peer pour la Plateforme LuckyJ. Master's thesis, University of Geneva, 2003.
36. Sun Microsystems. *Java Beans(TM)*, July 1997. Graham Hamilton (ed.). Version 1.0.1.

37. uddi.org. UDDI technical white paper, September 2000.
38. S. Waterhouse. JXTA search: Distributed search for distributed networks. http://spec.jxta.org/v1.0/docbook/JXTAProtocols.html, 2001.