An API for Autonomous and Client-Side Service Substitution

Herman Mekontso Tchinda*†, Julien Ponge*, Yufang Dan*‡, and Nicolas Stouls*
*Université de Lyon, INRIA, INSA-Lyon, CITI, F-69621, France – Email: first.second@insa-lyon.fr
†UMMISCO, LIRIMA, Université de Yaoundé I, BP 812 Yaoundé Cameroun
‡Departement of Computer Science Chongqing University, Chongqing, China

***Abstract*—The service oriented approach is a paradigm allowing the introduction of dynamicity in developments. If there are many advantages with this approach, there are also some new problems associated to service disappearance. The particular case of service substitution is often studied and many propositions exist. However, proposed solutions are mainly server-side and often in the context of web-services. In this paper, we propose a client side API-based approach to allow service substitution without any restart of the client and without any assumption on external services. Our proposition is based on a transactional approach, defined to automatically and dynamically substitute services, by preserving the current run and collected data.**

***Keywords*-OSGi; Stale References; Substitution; Self-Healing Software.**

I. INTRODUCTION

The service oriented approach is a paradigm introducing loose-coupling into software architectures. A developer can simply choose an Application Programming Interface(API) describing a requested service and develop its software without knowing which implementation will eventually be installed on the final client system. Currently, most popular uses of this approach are done by web services, Android systems and the OSGi framework [1]. Main studies are about Web services, but with the *server-side* point of view [2]. It means that the service provider can make any assumptions on provided services with the objective that a service substitution can be done without any consequence on the client, even if the service is state-full, State-full services are the one that maintain internal state across successive invocations from the same requester.

Dealing with dynamism issues of services in SOA is a real challenge today. Every model implementing this architecture faces the problem of deprecated references caused by the services mobility. The OSGi component framework is one of the several models implementing SOA and in which stale references can be very harmful. In this paper, we propose to study the dynamic substitution of a service in the context of the OSGi framework. In order to introduce the problem of service substitution in OSGi, we briefly describe in the following the OSGi component framework and the problem of stale references.

A. OSGi Component Framework and Stale References

The OSGi platform allows a remote loading and dynamic deployment of applications. We propose to study the client point of view. A service is a running java implementation, whose interface is available in an open repository. Using a reference of service instead of a service object itself. But, this reference also has a drawback: the referenced service can be stopped and its dependencies deprecated at the moment of its use, leading to a stale reference. We are focusing on the case of a mobile platform with OSGi that can discover or lose connection to some service providers. In such a case, a service requested by a client can be lost while in use.

B. Shielding From Stale References

As mentioned previously, bundles can be dynamically unloaded and a service may be stopped without a prior notification, leading to *stale references*. A stale reference is a reference to a service that is no longer available, either because of the bundle offering that service has been stopped or the service associated has been unregistered [1]. When a bundle becomes unavailable, all the references to objects it provided should be released to allow garbage collector to do its work correctly.

Writing safe code for handling OSGi service references boils down to properly listening to the OSGi service registry and tracking which services are in, and which services are out. This also requires that each call to a service in a client code makes extra steps to ensure that it is effectively going to invoke a method on a service whose reference is not stalled. This is not easy as it seems, as concurrency is involved. Indeed, a thread may be invoking a service while another one is unregistering it. This easily defeats guarded accesses to a service reference if no intrinsic locks or fine-grained reentrant read/write locks are being used. While solutions such as *BluePrint Services* help in handling service events, it does not shield from concurrency nor it enforces that references are properly discarded in client code when a service is unregistered [1].

Hence, we cannot make any hard hypothesis on services lifetime, but we can propose some good practices in client development in order to be resistant to the substitution of services. The substitution is well known as a self-healing software technique [3], [4].

 ISBN: 978-1-61208-215-8

C. Dealing with Dynamic Substitution in OSGi

The problem of dynamic substitution in OSGi is linked to the problem of stale references: assume that a client Bundle is using a service supplied by any Bundle server of the environment and at one point, the service disappears whereas the client Bundle has not finished with it. If there exists an available service to replace the disappeared one, a substitution can take place. But because of stale references, client bundles programmers may not be aware of the unregistering of the service and should not look at a new service. A good policy should then be implemented to deal with the problem of stale references.

So, in order to solve the problem of stale references, the main steps are: *(i)* to detect the service unloading, *(ii)* to choose a new service and *(iii)* to load the new service by preserving the internal state of the unloaded one. We do not want to focus on the service selection problem, since this step has been largely studied elsewhere in the literature, e.g., [2]. We will focus on the two other steps.

In this paper, we propose an API-based approach for the development of the client, inspired from the transactional approach of concurrent systems. We will consider the problem of detecting unload and the one of loading the new service. In our proposition we will first consider the easy case of using a single service, before introducing the substitution of a service while using a set of state-full services.

If a software is developed by using correctly this API, we guarantee that it can, according to its preferences: *(i)* be actively notified of the unload by a specific exception, or *(ii)* continue its execution with a new service that has been automatically substituted, even if the service is a state-full one, with a very light overhead of code to write. We also claim that the development cost is low in comparison with the development cost of a similar software with the same capabilities but developed without this API. Finally, we will show that our approach does not restrict the expressiveness of developed software, which means that every program using service can be rewritten to use the proposed API.

Section II cites some other works of the domain and shows the gap we will try to fill with this proposition. Section III describes the contribution of this article, about service substitution. In order to fix the global understanding of the reader, Section IV describes the tool developed to show the feasibility of the approach. Finally, Section V concludes this work.

II. Related Work

In this paper, we propose a solution to deal with dynamicity in OSGi. Our proposition is a *client-side* solution allowing state-full services substitution in OSGi. Client-side means that we do not make any assumption on services, potentially provided by some different providers. In [5], we have an example of a case in which the substitution process fails because of a mishandling of stale references. In this section, we present some related works on service substitution in general. The section is ended by a presentation of some existing approaches dealing with dynamicity and stale references in OSGi.

In [6], authors propose an algorithm for CORBA service reconfiguration, that involves a passive link to the unavailable service and an active link to the new service, while keeping the application consistency and with a few execution disruption. In the case of stateless services, it is straightforward. But for state-full services, it is more complex. One should restore the state of the substituted service. In SIROCO [2] framework, there is a registry system, where a service can register its current internal state and thus make a checkpoint. When a service fail, the framework try to manage the new service in order to set its internal state in the late one of the previous service. A synchronization mechanism has been presented in [7]. The configuration manager provides a run-time kernel which provides a message repository for messages that has been sent by components.

OSGi specification releases some advises to use ServiceFactory Interface or Indirection mechanism in service object implementation in order to limit, *"but not completely prevent"*, the consequences of stale references [1, Section 5.4: Stale References]. In [5], by using Aspect Oriented Programming techniques, the authors propose a tracking stale references tool named Service Coroner that helps to find stale references for developed or maintained OSGi applications, and apply it in two cases study. Others approaches such as using Service Binder [8] or IPOJO [9] suggest to separate functional and non-functional aspects, by describing the services dependencies management information in meta data XML files and merge both at run time. Each of these approaches tackles a particular case of the stale references problem, but a general solution is not yet provided. An alternative solution is the use of a proxy [10], instead of a service references. The proxy manages load/unload of services and the client services do not longer keep a reference to a likely disappeared service and the problem of stale reference is then avoided.

Almost all the aforementioned approaches are *server-side* and do not tackle state-full services. For state-full services substitution, one should implement a transaction mechanism to restore the state of the substituted service. Our approach is based on a proxy that make the substitution possible, but we manage state-full services by adding a transaction mechanism.

III. Contribution

We propose to add a “safe service use” layer into a service framework such as OSGi, in order to make softwares being more fault tolerant. This layer is an API that can be used by clients to be aware of services unload.

 ISBN: 978-1-61208-215-8

To describe what have to do this API we first introduce usual approaches of fault tolerant systems. Next, we describe our solution for the simple case where the client use a single service. Finally, we extend solution to take into account the case where several services are in use at same time.

A. Fault Tolerant System

Usually, fault tolerant systems are systems whose execution can continue to deliver correct service even if a fault occurs. In such a system, the first problem consists in identifying that a fault occurs. In our proposition, we define precisely what is a fault: the unload of a used service.

There are usually three families of treatment to recover an error [4]:

- to mask the error;
- to roll-forward in the execution until a new stable state is reached;
- to roll-back to the previous stable state and restart the execution from it.

Usually, to mask an error consists in having redundant information. Since we can not have it, we will focus on the two other treatment. We propose some mechanisms associated to the last two treatment families. In order to implement the roll-forward mechanism when a service disappears, we propose to throw an exception that explicitly advice the client that the service is no more available. Finally, to implement the roll-back mechanism when a service disappears, we propose an automatic substitution of the service by another one equivalent[1]. This substitution will be state-full service resistant.

In the following, we present these solutions in the context of a single service use and a multiple services use.

B. Safe OSGi Service Reference – Single Service

When a service is unloaded, its instance is kept in memory until the garbage collector dispose it, then while there is at least one reference to it. However, both the Java language and the Java virtual machine specifications do not support a notion of *"volatile / dynamic"* references [11], [12]. References to object instances cannot be changed "under the hood" unless explicitly re-assigned as part of a program control flow. This means that encoding a thread-safe and dynamic-aware behavior of service references need to be captured as part of a proxy indirection.

1) Proxy Indirection: A very common pattern for transparently mediating interactions between client code and a component in object-oriented languages is the introduction of a *proxy object*. They are most often used to enrich existing classes with cross-cutting concerns code such as logging, security or remote object exposition. A good example are the *Enterprise Java Beans*, where developers write simple Java classes, and EJB containers enrich them with support for security, transactions and other useful features. In our context, we will try to transparently add some enrichment without that services know that they can be substituted.

2) Proxy Requirements and Functionalities: The requirements for an OSGi service proxy depends on the usages. Hence, two kinds of policy and then requirements can be defined: Roll-forward policy and Roll-back policy.

In a *Roll-forward policy*, method invocations must throw an unchecked exception if the underlying service reference is staled. The client itself just need to take account the possibility of such exception.

In a *Roll-back policy*, when a method invocation reached a stale reference problem, we will try to transparently replace the unloaded service by another service, and then to make the invocation on the new service. However, if the unloaded service is state-full, the substitution can be the source of many unexpected problems. We then need to replay a part of the last commands. For instance, if the service need to be logged in, when the service is substituted, the login method has to be invoked again before any other use of the service. The part of the code that the API need to re-execute is called a *transaction*.

Transactional systems have been widely studied for several classes of problems and applications. The type of problem that we are tackling is actually close to a transactional memory [13]. However, the service and the client are developed by knowing that if a transaction fails, then it can be executed again. Our proposition is an adaptation of these existing results in the context of OSGi, where services are developed without knowing that such a substitution can occur. The client is the only one knowing this.

Since the API need to know precisely which part of the code has to be execute again in case of substitution, then the designer of the client must declare a part of code as the transaction. However, this code can be executed many times, since many substitutions can occur. Hence, this code has to be pure. It means that no side effect has to be done in the client by the transaction.

Finally, here are the sufficient requirements in the case of using a single service:

- **Awaited Behavior:** When a method is invoked, if the underlying service reference is staled, then the awaited behaviors are the following, for each policy:
  - Roll-forward policy: unchecked exception is thrown.
  - Roll-back policy:
    - If no other service: unchecked exception is thrown.
    - Else: substitution of the service, restarting the invocation from start of the transaction method.
- **Proxy Requirements:** It depends on policy:
  - Roll-forward policy: the client would consider the

[1]In OSGi, two services are equivalent if and only if they provide the same service interface.

 ISBN: 978-1-61208-215-8

possibility of an exception for each service call.
- Roll-back policy: the client must provide a pure method making the transaction.

C. Generalizing to the Invocation of Multiple Services

While a proxy is sufficient at the granularity level of a method invocation on a single OSGi service, generalizing the approach to the coherent execution of multiple services is more involving. Indeed, consider a block of instructions where several services are being used, and having a strong requirement for that block to be executed with a stable set of non-stale OSGi references. Given that, we cannot make any assumption on concurrency and the possibility for service references to become stale in the middle of a block execution. We need to provide a more powerful transactional-like framework to execute such blocks.

1) Requirements and Assumptions: Coping with the traditional definition of a transaction, we assume that a *transacted block* is a portion of code invoking a set of services, and that the whole block shall be successfully executed as a coherent whole. However, by opposition with the case of using a single service, we can generate side effects in used services. Hence, the transaction is pure only by the client point of view. Hence, in the context of multiple services, executing a transacted block requires:

- a declaration of the service interfaces it operates on,
- methods implementations to:
  1) put the block into a coherent initial state before its execution,
  2) execute the actual block code,
  3) finalize work upon successful execution,
  4) compensate possible side-effects in other services, if a stale reference caused a failure in the block,
- a retrial policy to control how the block execution is attempted again when a stale reference caused a failure.

The context of an OSGi platform imposes very loose assumptions on the transacted block implementations. Especially, services in use are not aware of being used in a transactional context, unlike Java EE resources that implement transactional APIs. Consequently, the correctness of performing a compensation operation or the ability to retry a block execution greatly depends on such services suitability in such a context, and their public specifications.

2) Invocation Atomicity – a Correctness Hypothesis in a Multi-Processed System: The OSGi specification states that OSGi service event listeners is notified when a service is unregistered [1]. A service reference becomes staled when all event listeners have been notified from the OSGi framework notification loop. Finally, we can take advantage of making a proxy to a service event listener in order to keep atomicity. Indeed, we can make a lock on the proxy object when performing a method invocation or when receiving a service unregistration event. This ensures a safe method execution as a reference cannot become staled in the middle of a method invocation.

3) Discussion: The generalization of the transacted execution of a set of services relies on strong assumptions:

1) services offer APIs to compensate effects in case an execution is aborted,
2) transacted execution blocks properly call compensation APIs,
3) intended compensation APIs are honored in service implementations,
4) services taking part in a transacted execution do not have further side-effects, or compensate them if client make a compensation.

In more traditional approaches, a transaction API is designed for resources to be managed by a transaction monitor. In the case of OSGi services, this would be translated to service interfaces extending such an API, making it impossible to use other types of services even if they offered compensation capabilities. We instead opted for a more open approach even if the usage of bad services or incorrect transacted block implementations can easily defeat the intended purpose.

IV. Implementation

The contributions presented in the previous section apply not just to OSGi environments. Indeed, any service-oriented architecture is based on the assumption that client code has no control over the services, including their availability and upgrades. We now detail how we implemented those contributions in OSGi in 2 steps. First we propose a simple service for building safe proxies to OSGi services, then we offer a service and an API for executing and defining transacted blocks. The interested reader can download the whole API and some examples at:

https://bitbucket.org/jponge/osgi-substitution

A. Configurable Service Proxy References

1) Overview: Proxies can be created at runtime in Java by creating a class that implements the *java.lang.reflect.InvocationHandler* interface and passing it to *java.lang.reflect.Proxy* for obtaining a proxy that is a subtype of one of more interface types. What we propose here is a very simple and minimalist API for generating proxies to OSGi services. It is exposed as an OSGi service of its own with the following interface:

```
public interface ServiceProxyBuilder<T>{
  public T getService(Class<T> c,
                      ServiceReference sr,
                      ProxyMode pm);
  public T getService(Class<T> c, ProxyMode pm);
  public T getFirstServiceMatching(Class<T> c,
                                   String filter,
                                   ProxyMode pm)
          throws InvalidSyntaxException;
  public ServiceBroker<T> getServices(Class<T> clazz,
                                      String filter)
          throws InvalidSyntaxException;
}
```

The interface mimics the OSGi service reference retrieval. *ProxyMode* parameters allow to specify whether a service reference becomes disabled after its backing service has been unregistered, or if another available service can be used in place. This allows to cater for both stateless and state-full types of services:

```
public enum ProxyMode {
  DISABLED_AFTER_UNREGISTERED, RELOAD_AFTER_UNREGISTERED
}
```

A *ServiceBroker* is used when dealing with several services for the same interface.

```
public interface ServiceBroker<T> {
  public Set<T> currentServices()
          throws InvalidSyntaxException;
  public void discard();
}
```

It is really close to the OSGi service trackers, except that it has the following semantics:

- *currentServices()* returns a set of service proxies currently matching the service interface and filter specification,
- returned service proxies have the *DISABLED_AFTER_UNREGISTERED* proxy mode,
- *discard()* is equivalent to the *close()* method of an OSGi service tracker.

2) Usage: The following code, extracted from the tests suite that we defined along with our implementation, shows an idiomatic usage of the service proxy builder OSGi service, in order to be substitution resistant.

```
ServiceReference ref = bundleContext.getServiceReference(
    ServiceProxyBuilder.class.getName());
serviceProxyBuilder =
    (ServiceProxyBuilder) bundleContext.getService(ref);
EchoService service = serviceProxyBuilder.getService(
    EchoService.class, RELOAD_AFTER_UNREGISTERED);

for (int i=1 ; i<= 10000 ; i++) {
    assertThat(service.echo("plop"), is("plop"));
}
```

B. Transaction Block API and Execution Service

1) Overview: We propose an OSGi service to execute transacted blocks whose interface is as follows:

```
public interface TransactedServiceExecutor {
  public T executeInTransaction(
        TransactedExecution<T> execution,
        RetryPolicy retryPolicy)
      throws TransactedExecutionFailed;
}
```

A transacted execution is specified through the following interface:

```
public interface TransactedExecution<T> {
  public void prepare();
  public T execute();
  public void finish();
  public void rollback();
}
```

It uses a parametric type T which is the expected return value type of a transacted block successful execution. The retry policy is a simple interface which is notified of potential stale reference errors, and can in turn decide whether a further attempt can be performed. It can also be used to implement delays between retrials. An example would be an exponential back-off delay over at most 10 executions.

The interface is defined as follows:

```
public interface RetryPolicy {
  public void notifyOf(Throwable throwable);
  public boolean shouldContinue();
}
```

By the way, a possible "retry forever" policy can be implemented as follows:

```
public class RetryForeverPolicy implements RetryPolicy {
  @Override public void notifyOf(Throwable throwable) { }
  @Override public boolean shouldContinue() {return true;}
}
```

2) Usage: A definition of a transacted execution implements the *TransactedExecution* interface. Given some fictious service interfaces *SomeService* and *OtherService*, an implementation could be as follows:

```
private class SomeTransaction
    implements TransactedExecution<Void> {

  @ServiceInjection public SomeService someService;

  @ServiceInjection(type = OtherService.class,
                    proxyType = MULTIPLE)
  public Set<OtherService> otherReferences;

  @Override public void prepare() { }
  @Override public <Void> Void execute() {
    for (OtherService s : otherReferences) {
      s.doThis(someService.doThat());
    }
  }
  @Override public void finish() {
    someService.release();
  }
  @Override public void rollback() {
    someService.undoThat();
  }
}
```

A more complete example would take greater care in the *prepare()*, *rollback()* and *finish()* steps. Fields annotated with *@ServiceInjection* are injected with service proxies. The definition for this annotation is as follows:

```
@Retention(RUNTIME)
@Target(FIELD)
@Documented
public @interface ServiceInjection {
  Class<?> type() default ServiceInjection.class;
  String filter() default "";
  ProxyType proxyType() default SINGLE;
  ProxyMode proxyMode()
      default DISABLED_AFTER_UNREGISTERED;
  public static enum ProxyType {SINGLE, MULTIPLE}
}
```

It is used to configure how proxies shall be configured. Especially, they can have service reloading capabilities enabled, and they can support a single reference or a set of instances like it is the case for the *otherReferences* set in the previous example. An OSGi service filter can also be specified. The block can be passed to the transacted executor service, which is also an OSGi service:

 ISBN: 978-1-61208-215-8

```
ServiceReference reference =
    bundleContext.getServiceReference(
        TransactedServiceExecutor.class.getName());
TransactedServiceExecutor transactedServiceExecutor =
    (TransactedServiceExecutor)
        bundleContext.getService(reference);
transactedServiceExecutor.executeInTransaction(
      new SomeTransaction(), new RetryForeverPolicy());
```

We used an optimistic approach. Having service proxies being injected into transacted blocks, we could have taken advantage of them to perform a giant lock spanning for the transaction execution lifespan. Indeed, it is possible to block the thread notifying that a service is going to disappear, thus keeping the reference valid until all receivers have been notified. Such an approach would avoid the need for rollbacks at the greater cost of limiting parallelism and breaking the OSGi framework requirements that service event notification handlers shall not block [1].

V. Conclusion

In this paper, we proposed an approach and a tool[2] to make a service aware to the stale reference problem. If a software is developed by using correctly this API, we guarantee that it can, according to its preferences: (i) be actively noticed of the unload by a specific exception, or (ii) continue its execution with a new service automatically substituted, even if the service is a state-full one.

Main properties of this contribution are: (i) this solution is client side, (ii) it does not make any assumption on used services and (iii) it can be used even if used services are state-full services.

This contribution is based on the fact that the client designer knows how to use desired services. Hence, we do not try to compute which behaviors are authorized by a service. The client designer has just to make a normal use of the service and to propose a sequence to rollback in a stable state before to make another try with another service, in case of substitution.

However, if a rollback is done on the external service, a rollback would also be done on the client itself in order to hold in a consistent global state. Since such a development model is risky, we prefer to give the following guideline in the client development: *do not make any modification of the client state from its transactional part.*

We also claim that the development cost is low in comparison with the development cost of a similar software with the same capabilities but developed without our API. Indeed, as explained and illustrated in the OSGi core specification [1, Section 5.4: Stale References], to make a correct use of a service without stale reference can be a little bit tricky. Moreover, we do not introduce any restriction to the expressiveness of services. Indeed, in the worst case, we can include the whole program in one transaction. Hence, if a service is substituted, then the whole program is restarted and fully executed with the new service. Hence, this API do not add any restriction in the software development.

In future work, we will consider the use of a service call logger, such as the Logos tool [14], and of a specification of used services, in order to propose an autonomous solution. We would try to propose some heuristics to automatically compute a call sequence to roll-back services in order to restart an interrupted transaction.

[2]Freely downladable at https://bitbucket.org/jponge/osgi-substitution

References

[1] The OSGi Alliance, "OSGi Services Platform, Core Specification, Version 4.2," June 2009, http://www.osgi.org [retrieved: June, 2012].

[2] M. Fredj, N. Georgantas, V. Issarny, and A. Zarras, "Dynamic Service Substitution in Service-Oriented Architectures." IEEE Computer Society, 2008, pp. 101–104.

[3] D. Ghosh, R. Sharman, H. Raghavrao, and S. Upadhyaya, "Self-Healing Systems - Survey and Synthesis," *Decision Support Systems*, vol. 42, no. 4, pp. 2164–2185, 2007.

[4] R. de Lemos, P. A. de Castro Guerra, and C. M. F. Rubira, "A Fault-Tolerant Architectural Approach for Dependable Systems," *IEEE Software*, vol. 23, pp. 80–87, 2006.

[5] K. Gama and D. Donsez, "Service Coroner: A Diagnostic Tool for Locating OSGi Stale References," in *34th Euromicro Conference on Software Engineering and Advanced Applications, SEAA*. IEEE, 2008, pp. 108–115.

[6] C. Bidan, V. Issarny, T. Saridakis, and A. Zarras, "A Reconfiguration Service for CORBA," *International Conference of Configurable Distributed Systems*, 1998.

[7] I. Warren and I. Sommerville, "A model for Dynamic Configuration which Preserves Application Integrity," in *3rd ICCDS*. IEEE Computer Society, 1996.

[8] H. Cervantes and R. S. Hall, "Automating Service Dependency Management in a Service-Oriented Component Model," in *CBSE*, 2003.

[9] C. Escoffier, R. S. Hall, and P. Lalanda, "iPOJO: an Extensible Service-Oriented Component Framework," in *Services Computing, IEEE International Conference on*. IEEE Computer Society, 2007, pp. 474–481.

[10] H. Ahn, H. Oh, and J. Hong, "Towards Reliable OSGi Operating Framework and Applications," *Journal of Information Science and Engineering*, vol. 23, no. 5, p. 1379, 2007.

[11] J. Gosling, B. Joy, G. Steele, and G. Bracha, *Java(TM) Language Specification, The (3rd Edition) (Java (Addison-Wesley))*. Addison-Wesley Professional, 2005.

[12] "Sun Microsystems, Java(TM) Virtual Machine Specification, The (2nd Edition)," Paperback, Apr. 1999.

[13] M. Herlihy and J. E. B. Moss, "Transactional Memory: Architectural Support for Lock-Free Data Structures," *SIGARCH Comput. Archit. News*, vol. 21, pp. 289–300, 1993.

[14] S. Frénot, F. Le Mouël, J. Ponge, and G. Salagnac, "Various Extensions for the Ambient OSGi framework," in *Adamus Workshop in ICPS*, 2010.

 ISBN: 978-1-61208-215-8