

Clustering OSGi Applications using Distributed Shared Memory

Anthony Gelibert*, Walter Rudametkin[†], Didier Donsez[†] and Sebastien Jean*

*Grenoble Institute of Technology,
LCIS Laboratory, CTSYS Team
Valence, France

e-mail: firstname.lastname@lcis.grenoble-inp.fr

[†]Grenoble University,
LIG Laboratory, ADELE Team
Grenoble, France

e-mail: firstname.lastname@imag.fr

Abstract—Modern applications are no longer monolithic and centralized, they have become distributed, modular and dynamic. The OSGi Service Platform is thus commonly used for building modularized and dynamic Java applications, but its distribution model however relies on communication protocols using remote invocation semantics. Distributed Shared Memories (DSM) provide an unusual approach to ease distribution by transparently sharing the state of targeted objects in an application without changing the initial data access paradigm.

This paper describes how DSM principles can be efficiently used in order to simplify the clustering of dynamic services. The proposed approach consists in transparently integrating DSM into the OSGi service model using containers and annotations. It has been implemented using the Apache Felix framework, the iPOJO component model and the Terracotta DSM, and has been validated in three use cases.

I. INTRODUCTION

Since their conception, software applications have continued to evolve. From a design point of view, monolithic applications have turned to modular approaches. From an execution model point of view, centralized applications have become distributed. These evolutions have introduced new requirements, particularly in terms of performance and availability. Dynamic services, building on advances from components and objects, have become a common way to ease application design. Even if dynamic services are already distributable, the common strategy remains to use existing and dedicated protocols focused on remote invocation mechanisms. These protocols are not transparent and consequently induce a lot of refactoring in order to distribute a centralized application. Clustering concerns, such as, load balancing and fault-tolerance, have been tackled from a different angle by Distributed Shared Memory (DSM), by considering data as a programming paradigm and allowing for transparent distribution through state replication. However, if these mechanisms have been successfully applied to objects, dynamic service replication remains unaddressed.

OSGi, a component-based and service-based application framework [1], has become the de facto standard for modularized Java applications in research and industry ([2] and [3]).

In this paper, we propose a container-based approach for the transparent integration of DSM mechanisms within the OSGi service platform, easing the distribution and clustering of OSGi services. The next section presents the context and motivations of our work, describing OSGi and DSM principles, and explaining how both technologies can be used to efficiently build clustered dynamic services. The third section details the proposed approach, based on the use of containers, focusing on the annotation-based integration model. Then, fourth section discusses implementation and validation. Finally, the last section concludes and presents future work.

II. CONTEXT AND MOTIVATIONS

A. The OSGi Service Platform

The OSGi specification is becoming the de facto standard for modular and dynamic Java applications. It is both a programming model to develop Java modules — named bundles in the specification — and a runtime infrastructure and framework to control their life-cycle. Thus, OSGi permits developers to manipulate bundles by: adding, starting, stopping, upgrading, and removing them. These operations are dynamic, performed at runtime, and there is no need to restart the entire platform to suppress a bundle or add another. The framework ensures consistency across bundles by keeping track of their dependencies. Finally, Service-Oriented Architectures [4] can be reified thanks to a service registry which is available for publishing, subscribing to, and tracking services. Services are the main paradigm used for inter-bundle communication.

Initially, the OSGi specification only permitted invoking bundles on the same platform. In other words, a service running on a platform cannot use a service running somewhere else by direct call (i.e., direct method invocation). If such a feature is needed, explicit communication mechanisms, such as messages or sockets, have to be used. These mechanisms however require using a specific programming paradigm that, even if not necessarily prohibitive in terms of complexity and development time, impacts source code because of communication code that has to be implemented and refactoring.

In order to provide the OSGi framework with a distribution mechanism, initial approaches of the OSGi Alliance proposed bridges with two existing technologies, namely UPnP [5] and Jini [6]. While these technologies are of interest, namely in the domain of home automation thanks to their ability to communicate with devices, they still require a complex development model and are far from achieving the desired feature of transparent or semi-transparent distribution (especially with UPnP where the simplistic data typing system mismatches with Java types). Although nothing had come directly from the OSGi Alliance concerning the distribution of OSGi services, two external research projects investigated the subject: R-OSGi [7] and Rose [8], both using an RPC [9] based approach. Finally, in 2010, the OSGi alliance provided a specification called “Remote Services” [10]. This work is based primarily on the work done in R-OSGi, and as such, is also an RPC-based approach.

B. Distributed Shared Memory

To develop distributed softwares, there are basically two different approaches: *explicit* communication protocols or *implicit* implementations. DSM implement this latter, by allowing a variable to be seen as *local* even if in reality, it is not the case. So, a software (and previously an hardware mechanism) does all the work in a transparent manner for the application developer.

When classifying DSM approaches in Java, a key factor to consider is the level where the distribution mechanisms take place in regards to the JVM:

- 1) In the Operating System or in an underlying low-level library used by the JVM. The main implementation of this approach was Java/DSM [11], which is no longer developed. The advantage of this approach is its transparency with respect to the virtual machine.
- 2) In the JVM. The functionality is implemented directly into the JVM, and is provided as an alternative JVM to those provided by Sun. The main implementations are cJVM [12] and JESSICA [13]. While cJVM seems to no longer be supported, JESSICA released its fourth version in 2009 now runs on top of the JVM.
- 3) On the JVM. The DSM functionality runs on top of the JVM, and as such is a generic solution for Java. There are several techniques for this kind of implementation, such as, the use of special constructions or new keywords (e.g., JavaParty [14]), the use of special libraries (e.g., ProActive [15], JavaSpaces [16] and Aleph Toolkit [17]), by configuration files (e.g., Addistant [18], J-Orchestra [19] and Terracotta [20]), or by dynamic instrumentation (e.g., JavaSplit [21]).
- 4) In the compiler. This approach effectively bypasses the JVM by compiling code directly to native instead of bytecode. One of the main attempts is Hyperion [22] which seems to no longer be developed.

To summarize, only one approach seems to still be used: “On the JVM”. The reasons are numerous: easiness of use and

development, difficulties to synchronize modified virtual machines with the original ones, impossibility for some solutions to use JIT features (and consequently performance decreases) or even specific OS or hardware that is required. Consequently, we chose an “On the JVM” solution for our implementation: Terracotta.

The Terracotta project provides transparent JVM clustering to applications. They provide a Java agent, which performs dynamic bytecode injection, altering and enhancing applications without requiring static modifications or a special underlying platform (the JVM needs only to support agents). The modifications to the application allows Terracotta to plug into the Java Memory Model in order to maintain the semantics of Java (JLS) while offering distributed features. This allows Terracotta to convert single-node multi-threaded applications into distributed multi-node applications, often with no code changes at all. Clustering is enabled through a declarative XML configuration and provides fine-grain, field-level replication. Objects do not necessarily have to be serializable because Terracotta works on atomic fields in order to only synchronize modifications.

A really interesting point in the approach chosen by Terracotta is the result of the bytecode instrumentation. Indeed, this operation constructs an extended JVM, corresponding in fact to a JVM-level DSM, but without requiring the deployment of one, just that of its Java agent. Thanks to this clever choice, the distributed aspects of the application are totally hidden from the developers (i.e., no code is altered).

The architecture of Terracotta relies on two main components (illustrated in Figure 1).

The server is the instance in charge of coordinating the clients. If there are several servers, clients will try to connect to each of them, until they reach an active one. To ensure robustness, if a server breaks down or becomes unavailable, clients will try to contact another one.

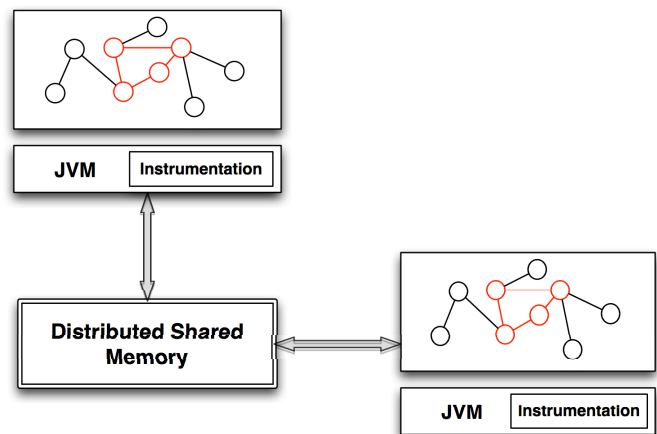


Figure 1. Distributed Shared Memory - principle

III. CONTRIBUTION

Distributed Shared Memory (DSM) is used to replicate the internal state of applications across several hosts. However, because these applications are homogenous, the provisioned code is identical for all the distributed JVMs and applications. This is, for instance, the case for a cluster of JavaEE application servers, which tend to run the same server implementation, the same version and the same applications. In this configuration, DSM is often chosen as the simplest way to enhance an application by replicating it several times and using the replicates for load balancing and failsafe features.

Our motivation is to allow different applications, composed of various and different modules, to share parts of their state. This sharing permits the creation of a “clustered service”: a service that is at least partially replicated on a cluster. Replicated services are composed mainly of two parts, a replicated part (a subset of the total class graph) and a local part containing the local references.

Figure 2 presents our objective: heterogeneous applications running on an OSGi platform, sharing a partially replicated module. The replicated instances, which are components, have their state shared and synchronized across the entire coordinated network in a way that is transparent to the applications themselves.

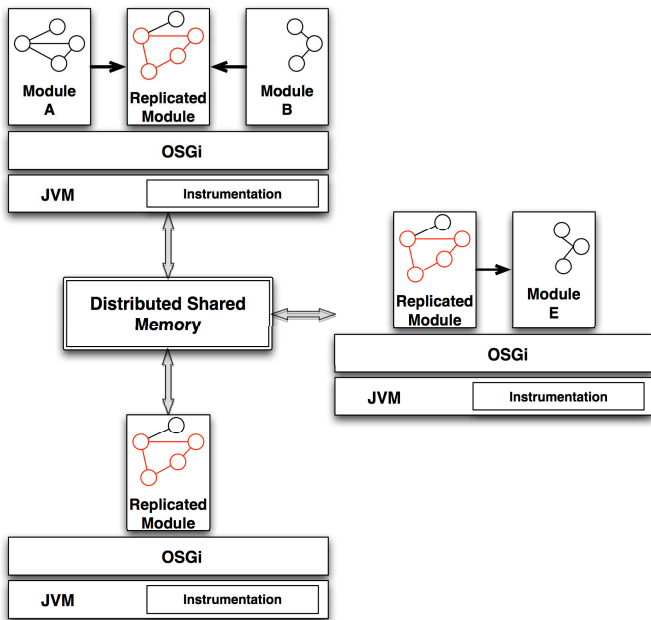


Figure 2. Terracotta Server - example of use

To facilitate the description of an application’s distributed behavior and to maintain coherency between code evolution and such descriptions, we propose a set of Java 5.0 annotations. Although our contribution targets DSM in general, some annotations are implementation-dependent (i.e., dependent on Terracotta).

In the context of OSGi, components that require services generally use mechanisms to have them injected into fields

(i.e., dependency injection). In this manner, method calls on injected fields are actually service invocations on local services (i.e., services collocated on the same platform). When replicating the component, special attention is required because, in most cases, injected services should not be replicated across frameworks, they should be considered transient and managed locally.

Figure 3 presents the complete toolchain, from the initial Java code to the final clustered service. The process is pretty simple: developers provide their annotated code, and, while the java compiler (i.e., `javac`) compiles the classes to produce the bytecode files, the annotation processing tool [23] parses the source code files using a dedicated annotation processor and produces the configuration file to use with the DSM system. In our specific implementation, it is a Terracotta configuration file. At this point, the build process is complete and the next step is to run the classes on the JVM with the configuration file and the Terracotta java agent enabled.

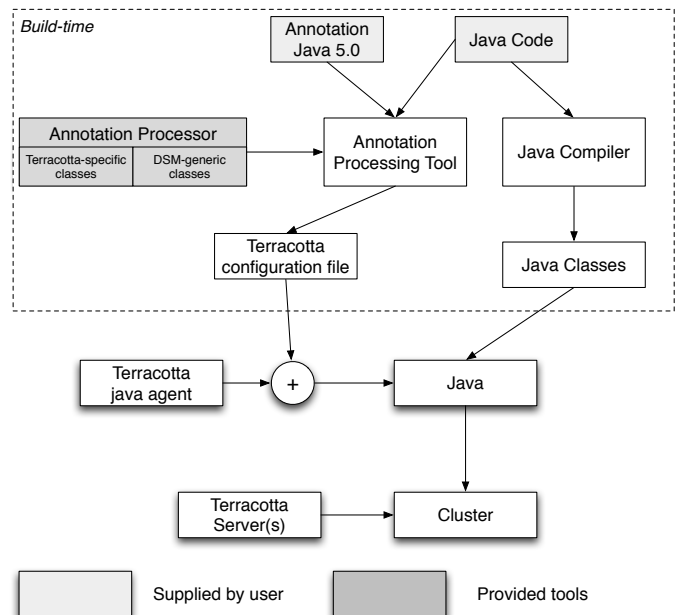


Figure 3. Contribution - complete toolchain

IV. IMPLEMENTATION

The OSGi platform has been the source of prolific research in the area of component oriented programming. Projects such as ServiceBinder [24], Declarative Services [25], iPOJO [26] and Blueprint Service [10]) have proposed applying service-oriented component principles in order to simplify the development of applications on top of OSGi.

Rather than defining a new component model that allows instance state replication, we propose a generic extension mechanism for existing component models.

Our prototype targets the iPOJO framework. Actually, iPOJO provides several convenient features that have eased our implementation: component model extensibility, dependency

handling and injection, and finally, support for annotations. However, the main reason behind our choice of iPOJO was the fact that iPOJO extends the application by performing bytecode manipulation at compile-time. Such manipulation is similar to the one performed by Terracotta, that is, our open-source DSM of choice, and thus compatible with Terracotta's bytecode manipulator.

We chose to extend the existing iPOJO model by adding replication metadata to the component. These metadata are given in the form of Java 5 annotations. In our initial implementation, we use the annotations to produce a Terracotta configuration file for replication. We have defined the following annotations:

- **TCSYSTEM**, specifies the global behavior of Terracotta (i.e., the configuration model used by Terracotta).
- **ServerConfiguration**, configures the server: symbolic name, host and ip.
- **ClientsConfiguration**, configures the client: log path.
- **InstrumentedClass**, indicates the classes that will be instrumented by Terracotta. The class and all of its dependencies have to be instrumented in order to be properly replicated (this is similar to the serialization process).
- **SharedField**, shares the field between all nodes. Each instance of this class will have the same value on all nodes (unless the field is not a primitive and contains transient fields).
- **InjectedField**, marks a field to be injected by Terracotta. For example, it is possible to inject an object representing the cluster in order to produce "replication-aware" objects (e.g, obtain its node id in the cluster).
- **DistributedMethod**, declares the method as "distributed". Each invocation of this method will be synchronously replicated on all nodes.
- **AutoLock**, specifies the level of lock used on a class or a method.

The following snippet shows the annotations of the component provisioned by the bundle A as illustrated in Figure 4. This component is partially replicated: the fields `m_clusteredCounter` and `m_myServProperty` are explicitly replicated across the distributed OSGi platforms. This component provides a service `IAService`. Since the instance of the component `AComp` is replicated in each OSGi platform, this service is registered in each local OSGi service registry: in Host 1 the service is bound to and consumed by bundle X, whereas, it is bound to and consumed by the bundle Y in the Host 2. Moreover, the service property set thanks to the shared field `m_myServProperty` has the same value in all the registries of the OSGi platforms. The iPOJO component `AComp` requires a service `IBService` which is injected by the container into the field `m_BService`. This field should be declared transient, because Terracotta does not know iPOJO will inject it dynamically, to avoid replicating its state when replicating the component instance. The `@AutoLock` annotation is set at the class level in order to propagate the transactional semantic on all methods provided

by the component. However, concurrency controls can be specific for each method for a fine grain approach.

```

/* Annotations for iPOJO */
@Component
@Instantiate
@Provides
/* Annotations for the Terracotta container. */
@InstrumentedClass
@AutoLock
public class AComp implements IAService {

    /* This block is only required if the used OSGi
       framework is Felix */
    static {
        final NamedClassLoader ncl = (NamedClassLoader)
            AComp.class.getClassLoader();
        ncl.__tc_setClassLoaderName(AComp.class.
            getCanonicalName());
        ClassProcessorHelper.registerGlobalLoader(ncl);
    }

    @SharedField
    @ServiceProperty(name="my.prop",
        value="foo",
        mandatory=true)
    private String m_servProp;

    @Requires(filter="(qos.level>3)",
        optional=true,
        nullable=false)
    private transient IBService m_BService;

    @SharedField
    private Integer m_clusteredCounter = 0;

    public void run() {
        synchronized(m_clusteredCounter) {
            m_clusteredCounter++;
        }
        m_BService.doSomething();
    }

    public int getInvocationCounter() {
        return m_clusteredCounter;
    }
}

```

Our toolchain is based on APT, a *standard* tool which allows processing Java source code annotations in order to produce new files. In our toolchain, the processor extracts Terracotta and iPOJO annotations and automatically produces the Terracotta manifest (`tc-config.xml`). During the parsing of annotations, a configuration structure is generated and finally transformed into a valid manifest.

Since APT is a standard tool, processing is seamlessly integrated into the development cycle via the use of Maven and Ant plugins. This two-phase approach permits easily adding new annotations and extending the support of Terracotta. However, our current implementation has some limitations. For instance, it does not check incompatibilities between Terracotta and iPOJO annotations such as `@SharedField` and `@Requires`. This could be easily solved by using an additional processor such as Spoon AVAl [27] which aims to check such coding rules.

The toolchain is part of the OW2 Chameleon [28] open-source project.

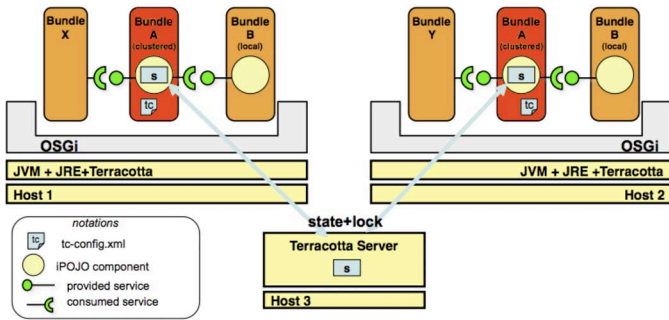


Figure 4. Clustered components across distributed OSGi platforms

V. VALIDATION

Our work does not aim to prove that DSMs provide better performance than RPC, each one having their own strengths and weaknesses, but rather to show the feasibility and interest of using a DSM approach, which provides a simpler development model and transparent distribution, in dynamic service-oriented environments.

This section describes two types of validations we have done against our proposition: functional and qualitative.

A. Functional validation

An OSGi event is the message format used by the *Event Admin Service*. This service is specified in the OSGi Service Compendium and provides a standard way to exchange messages in the OSGi environment using the well-known publish/subscribe model. The service does not persist nor distribute events between platforms.

To functionally validate the container, we decided to implement the proposition in [29] with our container. In this work, the authors proposed an approach permitting to spread OSGi events across multiple virtual machines. To achieve this, the authors proposed to use a dedicated OSGi bundle that listens to all events in the platform and sends them to others virtual machines using a dedicated link. They provide several implementations using different technologies: Ivy [30], Flash [31] and JMS [32]. The first is a diffusion protocol for Local Area Network, the second uses sockets and the last is a message bus specification from Sun.

So, to exchange messages transparently between several frameworks, the only thing to do is deploying on each of them a special service, which will make the propagation using a particular technology. Like these technology aren't necessarily OSGi-dependent, events are equally available outside of the framework to others applications.

Our implementation uses the distribution facilities provided by Terracotta. Each platform starts a service which is configured to be partially replicated among other platforms. This service listens for all the events transmitted on the local framework. When it receives a new transmittable message, it uses a special method that can be synchronously triggered on all the nodes of the cluster. Its parameter will be a structure composed of the propagated events and the emitter's identifier.

This allows each node of the cluster to distinguish a remote event from a local one and publish the duplicated remote events locally.

With this distributed service, it is possible to obtain transparent communication between two virtual machines without modifying the code of events producers/consumers.

Figure 5 shows an example using a producer and a consumer on two different virtual machines linked by our bridge.

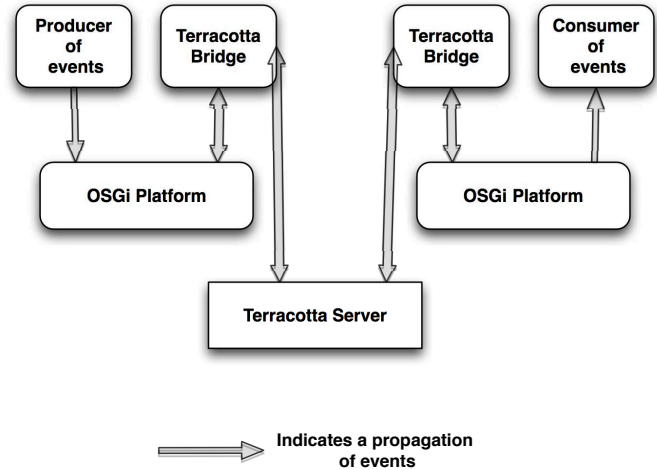


Figure 5. Terracotta bridge for Event Admin

B. Qualitative validation

In this section we propose several scenarios targeting different projects to show the interest of our approach.

1) *JOnAS*: JOnAS [33] is an open-source implementation of the Java Enterprise Edition specification. The application server, which is used in production servers, is composed of OSGi bundles and consists of over 3 million lines-of-code.

In this software, DSM may be useful in two ways.

The first is simple, like our container is compatible with OSGi and iPOJO, we can envisage using it for applications running in JOnAS. We can imagine several ways to provide this service, JOnAS could be provided with a Terracotta installation, or just indicating where found the software. Additionally, they could provide our container as an external library, with a specific documentation. It is a simple solution that can easily be validated with our previous works. This one has been realized and checked with the spread of OSGi events and the examples provided by Terracotta. In term of re-engineering, nothing was required. It is only the deployment of our examples, and like JOnAS is just an "enhanced" OSGi framework (with additional bundles), it is immediate.

The second way is far more technical. It is the use of Terracotta inside the bundles of JOnAS. The current point of interest for replication is the clustering of EJB, an interesting solution for the industry.

This platform is pretty interesting for the validation of our proposition on a big project. But, we have not forgotten our main objective. Thus, it is not the final performances that

will be investigated but the easiness of integration, and here this example is clearly interesting. If we choose the first approach, there is not any modifications required in JOnAS. The second approach is more complex and will require an initial investment to be used, which seems unlikely.

2) *uGASP*: GASP [34] is a pure Java middleware for mobile multiplayer online games. It implements the OMA GS [35] specifications, enabling mobile multiplayer gaming. uGasp [36] is an OSGi/iPOJO middleware dedicated to ubiquitous multiplayer gaming keeping all GASP functionalities.

The targeted application is to provide a Delay-Tolerant Networking [37] service for Bluetooth games. The DTN is a network architecture that seeks to address the technical issues in heterogeneous networks that may lack continuous network connectivity.

Currently, there is a problem of state migration when using the uGASP framework for the games using Bluetooth and requiring that the player move (in the real world). Indeed, like Bluetooth is not IP compliant and has not a large range of transmission and reception, there is a problem to maintain the continuity of identification between all the nodes in the path of the user.

The usual solution is to design a protocol doing the migration of states between the servers and permitting for a player connecting to a server to move to a second one without losing its previous state. These protocols are often *ad hoc*, usually induce performance problems and require that the rest of the architecture takes in account this specificity. So, the impact in term of abstraction and integration is really important.

Our idea is to synchronize the states of the different servers with Terracotta. In other words, the current state of the game is replicated among all the nodes. In this manner, the state replication and the migration of the user is totally transparent for the rest of the program. In term of performances, it exclusively depends of Terracotta but from our perspective, it is really interesting for the easiness of integration and use. Figure 6 illustrates this scenario.

3) *H-Omega*: H-Omega [38] is a flexible application server for home gateways. It provides an execution platform based on the OSGi platform and the iPOJO component model, and technical services required to create home-context applications. The application server is composed of OSGi services and permits developers to integrate new appliances by adding new bundles to communicate with and control them. To exchange information between bundles, such as, temperature, luminosity, electric consumption, among others, the bundles use the OSGi Event Admin publish/subscribe service.

There are two different ways we have identified to integrate replication into this project that would be useful. The first is to remove the single point of failure of this network by allowing using several OSGi platforms in the house. By using the special bundle presented in the functional validation, we can envisage the spreading of events between the platforms. Like our service is totally transparent for the producers and the clients, the creation of a totally replicated system induces no direct cost. We can envisage more advanced scenarios, using

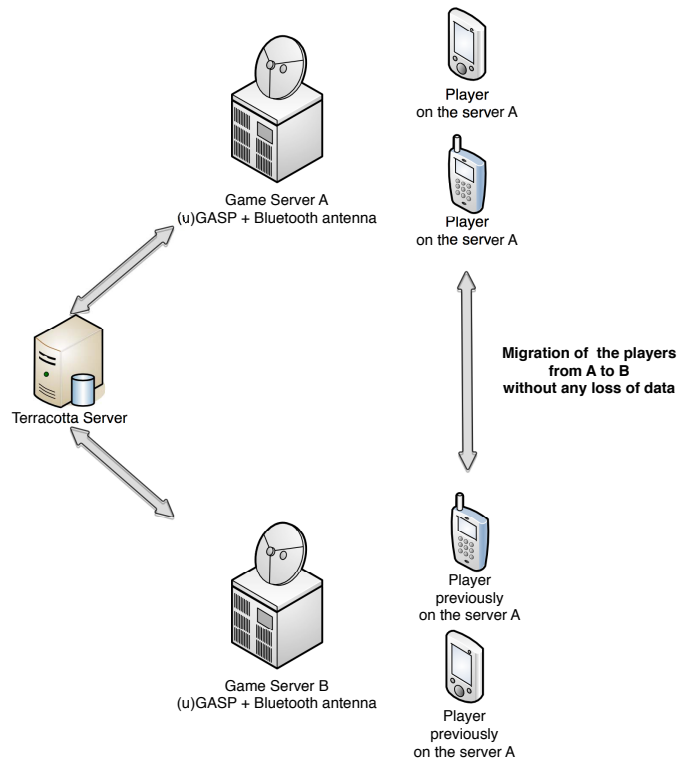


Figure 6. uGASP scenario

the spreading of events and an identification mechanism to allow not only replication but also replicates coordination. For example, we can envisage deploying on a highly available JVM a platform with the critical services. These ones will receive all the messages, but will respond only to their own triggers. Like they are on a dedicated machine, their answer will not suffer of the other services execution.

The second is to globally remove the events and replace them by shared memory. Thus, there will have no more messages passing and only a state for the temperature, for the luminosity, etc. This solution equally permits to deploy multiple platforms all synchronized by Terracotta. This solution will simplify the further development and extension of the platform but will require large initial modifications because of the different development paradigm.

4) *Cilia/ALE*: Cilia [39] is a lightweight data mediation and integration service component model, built upon OSGi. It provides a framework and toolset that simplify the development and the maintenance of large-scale, complex integration solutions that interacts with heterogeneous and complex integrated systems. ALE is the integration of Cilia in the AspireRFID project [40]. The AspireRFID project aims at developing and promoting middleware along with tools to develop, deploy and manage RFID-based applications and sensor-based applications. So, ALE is a version of Cilia dedicated to RFID tags and sensors.

Before anything, even if our scenario aims AspireRFID, it is also completely usable on Cilia alone.

A chain of mediation is a set of linked mediators with at their source one or more producers and at their output a generated report on these sources. The produced elements depend of the nature of the application, in the case of AspireRFID it is naturally RFID tags but for Cilia, it could be anything. The generated report depends of the chosen sequence of mediators.

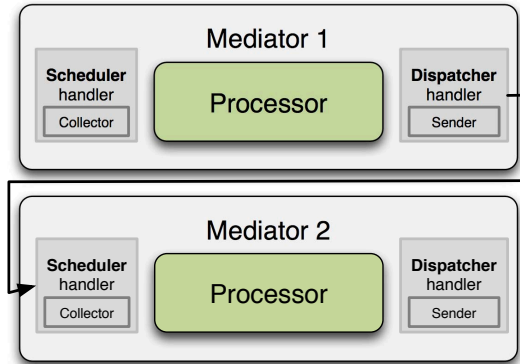


Figure 7. Cilia Component Model

Each mediator is made of three parts (illustrated in Figure 7). The scheduler receives the list of elements and gives it to the processor. All the behaviors are allowed: introducing a delay, changing the order of the lists, discarding entire lists, etc. The processor calls a specific function on the received list of elements and sends the result to the dispatcher. The dispatcher sends the result to the following(s) mediator(s).

Our scenario uses the state-full mediators like the difference, the aggregate, etc., and proposes to replicate these mediators. The internal state which is replicated permits to offer interesting features for the previous dispatcher (load balancing, fault tolerance, etc.) without loosing any information. Depending on the mediator, the internal state can be the previous list, a fixed number of previous lists, etc.

After this first step, we have obtained the possibility to create replicated processors and we can use them in existing scenarios to enhance them. For example, the possibility to change the “path” of an event, based on the states of the mediators is already possible. But currently, like there is neither memory distribution neither synchronization, the previous state is lost during the change of path. Here, our processors permit to keep this state and consequently a failure becomes transparent for the final mediator. Another typical scenario is the load balancing.

In a last step, we can envisage to integrate other features in the chain of mediators. For example, the persistence is a feature that could be really useful in this framework.

Lessons Learned

The projects we have targeted to validate our approach are very different. Their sizes vary from a few hundred lines-of-code (e.g., EventAdmin Bridge) to a couple of million lines-of-code (e.g., JOnAS in its extended version). Large variations concerning the number of bundles and the number of

components can also be found in these applications. Regarding replication, we observed that generally a few bundles need to be replicated in order to distribute the application. For example, the propagation of events requires only one bundle be shared by the platforms, even if a large number of bundles use the service. The same observation can be made on Cilia with the replicated mediators, each of them is a single bundle. One lesson we have learned is that interesting levels of distribution and replication can be obtained using a small number of replicated components in regard to the total number of components in the application.

The annotation mechanism permits developers to concentrate on the source-code and to keep metadata concerning distribution and replication properly synchronized. The integration of APT in the build process is beneficial because it increases the impression of fluidity and transparency.

But *a contrario*, we discovered that even with a tool capable of transparent distribution, like Terracotta, distribution is not trivial. Attention needs to be paid to the global architecture of the cluster, especially when distributing methods or synchronizing variables, and to the consequences this may have on replicates. Moreover, because instrumentation is done runtime, the compiler cannot make statical verifications on application’s code (or at least, not efficiently), conducing to runtime errors. Furthermore, IDEs used at development time, cannot see all the implications runtime instrumentation and dynamic injection of bytecode may have on distribution. This creates complicated debugging scenarios when problems, especially transient ones occur.

Using the current approach, the time required to distribute a component will depend not on the size of the component, but on its structure. For example, distributing the EventAdmin service took us a few hours, but to distribute a mediator took considerably less time because it only requires replicating the internal state of a small piece of the architecture. Regarding H-Omega, which uses a message passing architecture, the migration from this paradigm to a distributed shared memory would be much longer than even the EventAdmin service.

VI. CONCLUSION AND FUTURE WORK

Modern applications are increasingly distributed, modular and dynamic. Modularization and dynamicity have been, for instance, achieved by dynamic service platforms, among which OSGi is the de facto standard. Concerning distribution, several approaches have been proposed to enable the construction of distributed applications. These approaches generally consist in using remote invocation protocols or by integrating distribution within the middleware. However, each of them requires changing the programming paradigm, from a centralized approach to a distributed one, which introduces large modifications to source code and increases complexity.

Distributed shared memory (DSM) has been well researched in the past. The approach spares developers from focusing on distribution by allowing them to use shared objects as if they were local. They can also be used for creating dynamic clustered services in service oriented environments, where a

set of services may run on different platforms but share state and be executed in synchronization. Our work has focused on simplifying the clusterization of dynamic services by using distributed shared memory. This provides application developers a simplified development model for both distribution and replication. The approach that we have proposed is based on component containers and a set of annotations that allow to transparently integrate DSM mechanisms within the OSGi framework. These mechanisms have been implemented using the Apache Felix OSGi platform and Terracotta DSM, and have been validated under the scope of several scenarios.

At the moment, our current use cases have centralized servers, but in the case of pervasive applications, there might not be a central point to deploy the DSM server. We envision several solutions for this and which we will investigate in the future. An initial approach could be to use another feature of the Terracotta software which allows the declaration of several servers and dynamic switching and failover between them. For a more extensive solution, we may provide introduce special nodes or local memories. This work is currently in progress.

Another topic that we are interested in is the dynamic update of running services in order to improve availability, fix functionality (quickly apply patches and bug fixes), adapt resource consumption and add new features. The emerging topic of runtime updates of dynamic services is of recent interest to researchers. Novel techniques are being applied to individual services in order update them. Some of these techniques are capable of avoiding state-loss under certain conditions, but dynamic updates of clustered services seems to be all but unaddressed. Our future work will focus on dynamic updates of clustered services. Our approach will require extending the current container to support such features. Special attention will need to be brought to evolution safety, defining which services are allowed to evolve and how to evolve shared services without breaking compatibility among distributed frameworks.

REFERENCES

- [1] Website of the OSGi Alliance.
<http://www.osgi.org>
- [2] Gruber et al., *The Eclipse 3.0 platform: adopting OSGi technology*. IBM Systems Journal (2005) vol. 44 (2) pp. 289–300
- [3] Desertot et al., *A Dynamic Service-Oriented Implementation for Java EE Servers*. IEEE International Conference on Services Computing (2006) pp. 159–166
- [4] Papazoglou, Traverso, Dustdar and Leymann, *Service-Oriented Computing: State of the Art and Research Challenges*. IEEE Computer 40(11): 38–45 (2007)
- [5] UPnP Forum.
<http://www.upnp.org>
- [6] Waldo, *The Jini architecture for network-centric computing*. Communications of the ACM (1999) vol. 42 (7) pp. 76–82
- [7] Rellermeyer et al., *R-OSGi: Distributed applications through software modularization*. Middleware (2007) pp. 1–20
- [8] Bardin, Lalanda and Escoffier. *Towards an automatic integration of heterogeneous services and devices*. Asia-Pacific Conference on Services Computing. 2006 IEEE, 0:171–178, 2010. doi: <http://doi.ieeecomputersociety.org/10.1109/APSCC.2010.89>
- [9] Birrell and Nelson, *Implementing remote procedure calls*. ACM Symposium on Operating Systems Principles (SOSP) (1983).
- [10] OSGi Alliance, *OSGi Specification Release 4.2*.
<http://www.osgi.org/Download/Release4V42>
- [11] Yu and Cox, *Java/DSM: A platform for heterogeneous computing*. Concurrency: Practice and Experience (1997) vol. 9 (11) pp. 1213–1224
- [12] Aridor et al., *cJVM: a Single System Image of a JVM on a Cluster*. IEEE International Conference on Parallel Processing (1999) pp. 4–11
- [13] The University of Hong Kong Department of Computer Science, *JES-SICA 4 Project*.
<http://i.cs.hku.hk/~clwang/projects/JESSICA4.htm>
- [14] Philippsen et Zenger, *JavaParty – Transparent Remote Objects in Java*. Concurrency Practice and Experience (1997) vol. 9 (11) pp. 1225–1242
- [15] Baduel et al., *Programming, Deploying, Composing, for the Grid*. Grid Computing: Software Environments and Tools (2006)
- [16] Freeman et al., *JavaSpaces principles, patterns, and practice*. (1999)
- [17] Herlihy, *The Aleph Toolkit: Support for Scalable Distributed Shared Objects*. International Workshop on Network-Based Parallel Computing: Communication, Architecture, and Applications (1999) pp. 137–149
- [18] Michiaki et al., *Addistant: An Aspect-oriented Distributed-programming Helper*. Transactions of Information Processing Society of Japan (2002) vol. 43 pp. 17–25
- [19] Tilevich et Smaragdakis. *J-Orchestra: Enhancing Java programs with distribution capabilities*. ACM Transactions on Software Engineering and Methodology (TOSEM) (2009) vol. 19 (1)
- [20] Terracotta.
<http://www.terracotta.org>
- [21] Factor et al., *JavaSplit: A runtime for execution of monolithic Java programs on heterogeneous collections of commodity workstations*. IEEE International Conference on Cluster Computing (CLUSTER) (2003) pp. 110
- [22] Antoniu et al., *The Hyperion system: Compiling multithreaded Java bytecode for distributed execution*. Parallel Computing (2001) vol. 27 (10) pp. 1279–1297
- [23] Oracle, *Annotation Processing Tool*
<http://java.sun.com/j2se/1.5.0/docs/guide/apt/index.html>
- [24] Cervantes and Hall, *Automating Service Dependency Management in a Service-Oriented Component Model*. International Workshop on Component-Based Software Engineering, Portland, USA, 2003
- [25] OSGi Alliance, *OSGi Specification Release 4.0*.
<http://www.osgi.org/Download/Release4V40>
- [26] Escoffier, Hall and Lalanda, *iPOJO: an Extensible Service-Oriented Component Framework*. IEEE SCC 2007:474–481
- [27] INRIA, *Spoon AVaL*.
<http://spoon.gforge.inria.fr/AVaL/Main>
- [28] OW2 Consortium, *Chameleon*.
<http://wiki.chameleon.ow2.org>
- [29] Donsez et Thomas, *Propagation d'événements entre passerelles OSGi*. Atelier de travail OSGi (2006)
- [30] CENA Toulouse, *Ivy*.
<http://www2.tls.cena.fr/products/ivy>
- [31] Adobe, *Adobe Flash Player*.
<http://www.adobe.com/fr/products/flashplayer>
- [32] Java Community Process, *JSR 914: Java™ Message Service (JMS) API*.
<http://jcp.org/en/jsr/detail?id=914>
- [33] JOnAS, *Java Open Application Server*.
<http://wiki.jonas.ow2.org/xwiki/bin/view/Main/WebHome>
- [34] OW2 Consortium, *Gaming Services Platform*.
<http://gasp.ow2.org>
- [35] Open Mobile Alliance.
<http://www.openmobilealliance.org/default.aspx>
- [36] OW2 Consortium, *uGASP*.
<http://gasp.ow2.org/ubiquitous-osgi-middleware.html>
- [37] Delay Tolerant Networking Research Group.
<http://www.dtnrg.org/wiki/Home>
- [38] LIGforge, *H-Omega*.
<http://homega.ligforge.imag.fr>
- [39] OW2 Consortium, *Cilia*.
<http://wiki.chameleon.ow2.org/xwiki/bin/view/Main/Dev>
- [40] OW2 Consortium, *AspireRFID*.
<http://wiki.aspire.ow2.org/xwiki/bin/view/Main/WebHome>