https://doi.org/10.20378/irbo-51782

Evaluation of Actor Based Code Offloading from Android Smartphones in Real-Time

Marcel Großmann and Andreas Keiper

Computer Networks Group, University of Bamberg
An der Weberei 5, 96047 Bamberg, Germany
marcel.grossmann@uni-bamberg.de

Abstract. The widespread idea of cloud computing affects developing approaches of mobile applications. With ubiquitous computing capabilities and mobile devices, one question arises for mobile application developers: How could battery drainage be reduced? To combine the cloud computing services and mobile applications, an offloading of processes may produce a benefit to save battery life. For example, background processes are mainly uninteresting for mobile users and can be offloaded to the cloud, as well as image processing or searching. Only a feedback mechanism is necessary to obtain results of a process in the cloud. With already existing technologies, an approach based on actors seems to be able to handle code offloading. By avoiding the pitfalls of this concept, system designers, who identify battery and processing power consuming applications, can offload critical code to an actor in real-time. The signalling concept is analysed with existing web-service technologies and described by a messaging paradigm, which fulfills the actor's requirements. A trial architectural design demonstrates the functionality of real-time code offloading on mobile devices.

Key words: Android; Architectural Design; Actor Framework; Code Offloading; Real-Time

1 Introduction

Since a lot of computing power nowadays is provided by cloud services, code offloading may reduce battery drainage of mobile devices. Modern technologies offer a lot of platforms that are capable to ease the process of programming code

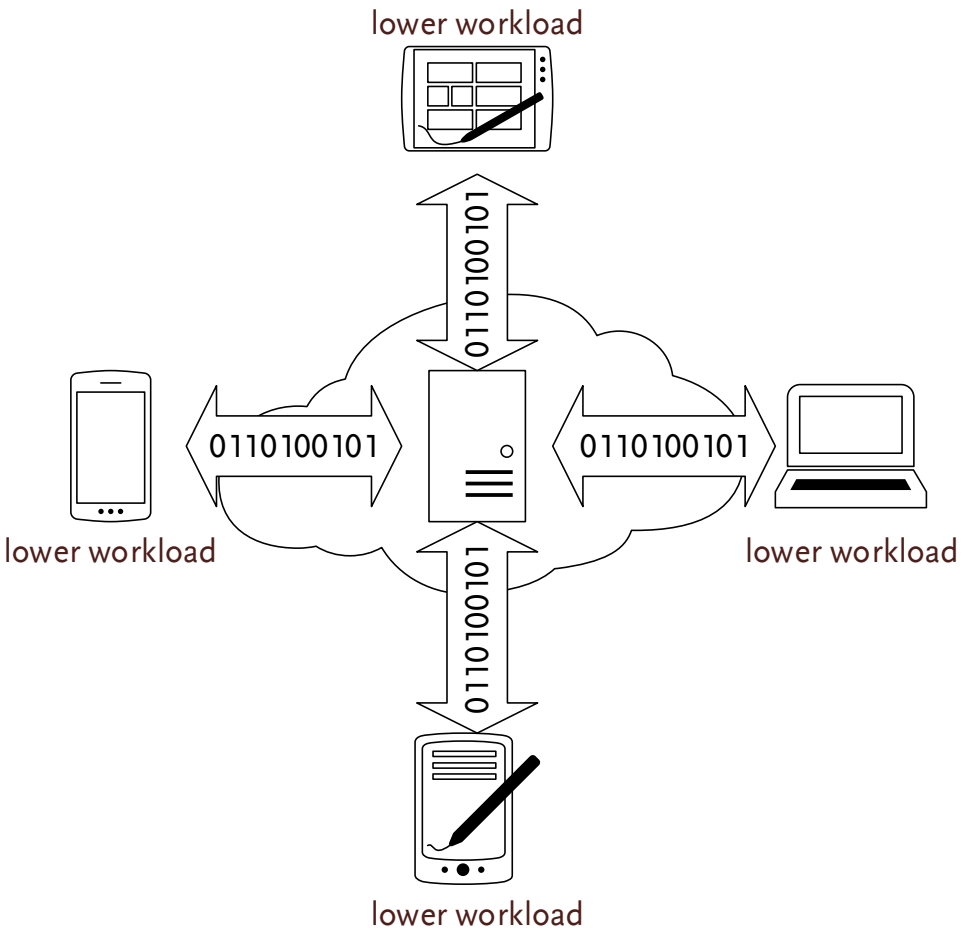


Fig. 1. The idea behind dynamic code offloading into the cloud. The server in the middle has highspeed network access, while the mobile devices around can offload user defined code onto it. This results in a lower workload of the devices.

offloading paradigms. The following approach describes, how to dynamically outsource code to be computed inside a cloud or fog architecture as depicted in Figure 1. Especially, a selection of communication protocols is analysed, if they are capable to handle the exchange of dynamic code offloading in a platform independent way. On cloud side, an actor based methodology is used, in order to provide a scalable server infrastructure. This architectural principle eliminates the problem of accessing commonly used memory; it is implemented by the open source actor framework *akka.io* [13]. The actor approach was presented by Hewitt *et al.* [6], which was adapted and implemented several times. The model is based on four principals:

- share nothing
- message transmission
- message queues
- simple scaling

Standard Java libraries for serialization are used for code transmission. Mobile classes are dynamically loaded into a Java runtime environment via a classloader approach. Therefore, the transmission technologies are evaluated to determine an approach to access the server application running on a cloud infrastructure.

1.1 Related Work

Roelof Kemp *et al.* introduced the framework *Cuckoo* [8]. With Cuckoo it is possible to offload compute intensive services by defining interfaces for them before the application is built, but not on realtime. Besides, a weakness is the concurrent usage of a single remote resource by multiple users, as the server is not solving concurrency issues by now. In *CloneCloud* [2] it is not possible to migrate native state, while this approach tries to offload classes by serializing them in realtime with their state information for execution. In our approach the execution of tasks is restricted to actor based technology that offers the ability to construct independent jobs, which can not interfere. Moreover, every job is controlled through a standardized message based methodology.

Approaches like *MAUI* [3] offload mobile code to the infrastructure, based on the whole process. In the cloud MAUI uses virtual machines, where an emulated smartphone system is executed. Thus, this results in a lack of scalability and wastes CPU resources through the virtualization of an entire mobile platform on a cloud

server. Besides, Thinkair [9] removes this burdensome emulation and runs with an Android x86 port on server side. Nevertheless, both approaches provide their scalability in the cloud by generating a new VM for each offloading device, which is a very heavyweight and less cost effective solution.

Flores *et al.* [4] propose that offloading as a service is not going to be established in those architectural proposals, because computational capabilities of the latest smartphones are comparable to some servers running in the cloud. Thus, emulation and heavyweight creation of VMs lacks the ability to scale offloading services in a cost-effective fashion.

In this paper, a finer offloading granularity is achieved by using the serialization of classes in realtime to migrate their byte representation to a cloud service. The cloud service itself is based on an actor system, where a new actor is generated for each offloading device. Moreover the actor system is integrated in a Glassfish server, which runs in a Docker container.

2 Technologies

In this section, the principles of the actor framework are described in more detail. This is followed by the question, how to perform dynamic classloading in a Java environment to be able to offload code in realtime. Existing technologies for transmission are analysed in subsection 2.4 and evaluated accordingly. Finally, the cloud infrastructure, especially the server and middleware technologies are evaluated in subsection 2.5.

2.1 The Actor Framework

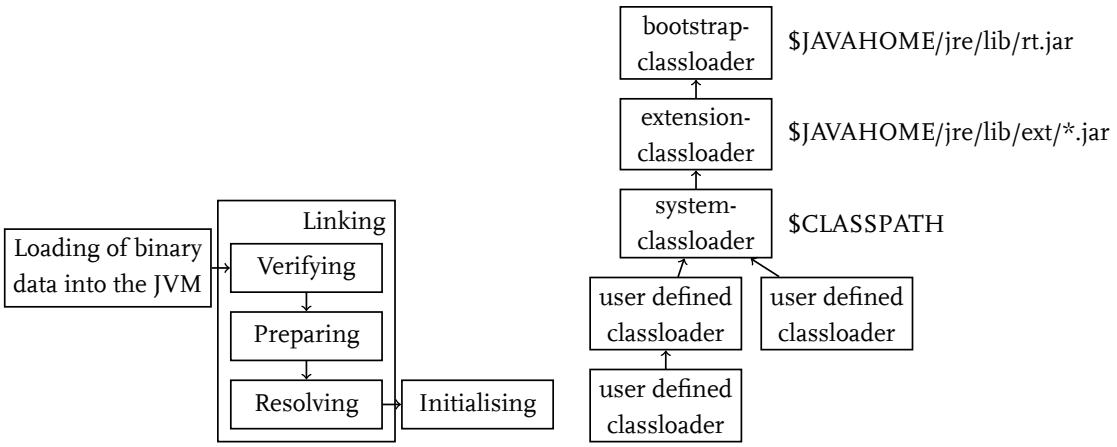
The open source actor based framework *akka* is written in Scala and targets the parallel and fault-tolerant execution of scalable applications [13]. Thus, the actor model offers scalability and concurrency, while fault tolerance is achieved by a *let it crash* model within the error handling. For the later, it is possible to define strategies on how to handle errors of actors. Akka is a distributed system, which may be extended over several *Java Virtual Machines* (JVM) and possesses an approach to realise atomic messaging as transactions. Furthermore, there are possibilities to configure the internal thread management, to manipulate the message flows and comprehensive functions to manage the actor system.

To program actors in the simplest way, a subclass of `AbstractActor` must be implemented, where the method `createReceive` is overwritten to handle received messages. With the help of the configuration class `Props`, actors can be generated from the actor class files in an actor system. Scala's integrated execution framework performs the actual execution of the actors, thus, the actor execution is solely separated from user commands.

2.2 Dynamic Classloading

While akka executes the code, one question arises, on how to transmit the code from a mobile device to the server. Investigations in akka's source code revealed that instances of `Props` are serializable, but also that a loaded actor class file is needed in the JVM for actor generation. Hence, separate dynamic reloading of class files with a self-defined classloader is required. Due to the transmission process, all of the byte code of a class and the code of individual instantiated objects must be transmitted. Therefore, the serialization is done in two steps. On the one hand, class files must be converted into byte arrays, such that a class loader can process them on the server side; this is done with the help of Java's streaming *application programming interface* (API). For serialisation and de-serialisation of instantiated objects of the memory Java offers a serialization API, which takes care of referential dependencies. To use this mechanism, the interface `Serializable` must be implemented by the class that should be serialized. It is the user's responsibility to define, if an object state is serializable. For identification purposes the `serialVersionID` is the same on both JVMs for a successful de-serialization. Nevertheless, dynamic class loading is even more complicated and must be analysed in more detail.

Classloader offer the possibility to find, load and link the binary representation of a class, such that they are usable for initialisation in the virtual machine. All classloaders inherit the abstract class `ClassLoader`. Concerning the JVM 8 specification, the point in time for classloading is differentiated either during the program start or user-defined at runtime. While the program starts up, the JVM uses its own *Bootstrap*-classloader for loading and linking of the non-array classes that are known on compile time. Arrays in Java are an internal class of the virtual machine and do not need any form of an explicit representation. Regarding the spec-



(a) . . . procedure [10]. (b) . . . hierarchy [14].

Fig. 2. Java's classloading . . .

ification this procedure including the method call is processed for a single class `Test` with the method `main` as follows:

– *Loading* of the class `Test`

The JVM tries to execute the program via the **main** method. It recognizes that there is no binary representation of the class `Test` in the memory and thus uses the classloader to find a representation in its classpath. If it succeeds, it loads this representation in the cache of the classloader, else it throws an exception. Is the classloader part of a classloader hierarchy, it delegates the loading procedure to the next higher classloader in the hierarchy. This procedure continues until no higher classloader is found anymore and the current one performs the loading. On the highest program level, this would be the system classloader. Inside the JVM two more hierarchy levels exists, the *bootstrap* and the *extension* classloader. The former is integrated into the JVM and loads the core classes from `$JAVAHOME/jre/lib/rt.jar`, while the later is responsible for extensions in `$JAVAHOME/jre/lib/ext`. The system classloader serves for loading of the program classes of the classpath of a program.

– *Linking* of `Test`

The method `main` requires an object of the class `Test`, which must be initialized first by linking the class. Linking consists of *verifying*, *preparing* and *resolving*. Verifying ensures that the code is well-formed regarding the semantics of Java and the JVM. Preparing includes the allocation of static memory and all data structures, which are internally used by the JVM, e.g., method tables. Resolving

controls the references of `Test` to other classes and interfaces and checks their correctness. Two methods are used for linking: *static* or *dynamic*. With static linking all references of a class are resolved recursively. In contrast, by dynamic linking it is done successively within the usage of a class. Java implements this dynamic approach, which allows a transmission and execution of remote code [10].

– *Initialising*
  Initialising consists of the generation of all static elements of `Test`. Even more, this means that all super classes of `Test` are also initialised. Therefore, a recursive loading and linking of the super classes must take place to build up the inheritance hierarchy in the memory. Initialization is a prerequisite, which must be fulfilled, to create object instances of a class.
– *Calling*
  After initialization of `Test` and all super classes, the method `main` can be called. Specific for Java is that `main` is declared `public`, `static` and `void`.

To influence the classloading procedure, a new classloader must be implemented, which inherits the abstract class `ClassLoader` or any other classloader implementations and serves to load classes from external sources. This is a main contribution Java has ahead of machine-oriented languages, like C or C++.

2.3 Manual Classloading

Classloaders deliver class objects, which can be instantiated. With the extension of a delegation model and resulting hierarchies, requests are delegated upwards in the hierarchy. One classloader usually delegates the loading procedure to the next upper one before it tries to load the class by itself. As a consequence, classes loaded on a higher hierarchy level can not refer to classes, which were loaded on a lower level, though it works vice versa. Threads use their own context classloader, such that all class requests inside a thread are primarily handled by this classloader. To understand the structure of the classloader itself, all important methods are described in the following:

– `Class loadClass(String name)`
  Until Java 1.2 this methods must be overwritten for an adjusted classloading procedure. With the help of the classes name with a given package structure it tries to find the class and returns a not instantiatable `Class` object.

- `protected Class defineClass(String name, byte[] b, int off, int len)`
  Method that generates a `Class` object out of a byte array by reading, controlling and converting byte code into an executable data structure.
- `protected Class findClass(String name)`
  Method used inside of `loadClass` to find the class by a given name. Adjustments of the classloader should be done by overwriting this method.
- `protected Class findLoadedClass(String name)`
  Searches the cache of the actual classloader, if the corresponding class was already loaded.
- `protected Class findSystemClass(String name)`
  Tries to load a class with the help of the system classloader.
- `protected void resolveClass (Class c)`
  Links the `Class` object, such that it is instantiatable.
- `static ClassLoader getSystemClassLoader()`
  Returns the system classloader.
- `Classloader getParent()`
  Returns the classloader of the next higher hierarchy level.

Many classloader methods are declared `protected`, such that they can be used only by implementing a subclass of `ClassLoader`. The primary method is `loadClass`, whereas all others can be used internally to design the loading procedure. Additionally, only the method `findClass` should be overwritten, which is used to define a loading procedure or a self defined source of class files, which is internally used by `loadClass`. For defining a loading procedure of class files from byte arrays, they are read in by `defineClass` and linked by `resolveClass`. Subsequently, they are available from the standard implementation of `loadClass` [7]. Moreover, the self-defined classloader should be involved in the hierarchy, such that the executing thread uses it as standard context classloader to instantiate new objects.

2.4 Server Access

Systematically, the server access is separated into synchronous and asynchronous request-response communications. This is extended by the transmission of user code and the generation of actors. In general three access structures could be used.

RESTful Webservice

Representational State Transfer (REST) is an architecture, which offers the following principles [12]. Resources are addressable via *Uniform Resource Identifiers* (URI). Resources are manipulated in a representation oriented way, which means that a representation consists of a sequence of bytes and metadata, to describe those bytes. Metadata represented with the *Hyptertext Transfer Protocol* (HTTP) can be a request type, a parameter, or a MIME type. Each message is self explanatory, such that the message content is sufficient to tell the server what to do. There is no need of an implicit session, since there is no information that is not contained in the message. Communications are stateless and independent from former requests. *Hypermedia as the engine of application state* (HATEOAS) defines that interactions are data driven and document oriented. Links are used to connect data and clients recognize from the response, how to process within the context.

The communication interfaces are self-defined HTTP requests, which are comparable to a *Remote Procedure Call* (RPC) calling methods with parameters on server side and expect a return value as HTTP response. The HTTP basis is a system independent and well supported technology. It offers a simple mechanism for sending and receiving requests through to mighty servlet technology for server development. With the *Java API for RESTful webservices* (JAX-RS) a framework is available that offers declarative annotations for designing server functions. However, designing all requests is more elaborative. Without a standardised interface, the documentation should be done in a proper way to compensate this fact. Alternatively, a *source development kit* (SDK) could be provided for the clients.

WSDL/SOAP Webservice

This type of webservice is composed of two technologies, the *webservice description language* (WSDL) as interface definition language and the *simple object access protocol* (SOAP) to encapsulate data for transmission. Both are based on the *extensible markup language* (XML) and gained a lot of interest with the hype of *service-oriented architectures* (SOA). For an implementation, Java offers several robust webservice stacks, which enable a simpler development. The standard also offers extensive functionalities for authentication, encryption, and secured data transmissions. A clear benefit of this approach is its standardised interface and communication technology with a clear definition by WSDL. Moreover, *Java API for XML webservices* (JAX-WS) is a framework for Java, which utilizes several webservice stacks.

However, extensive technology knowledge is required on developer side and the project depends on webservice stacks, which come with the issue that extending the basic functionality may lead to incompatibilities. Finally, inside of a WSDL there is no metadata included to classify binary data, such that additional documentation is necessary.

Remote Method Invocation (RMI)

This is Java's variant of the *remote procedure call* (RPC) client-server model on an object oriented basis. Method calls are performed with a proxy object on client side that represents the actual object on server side. On server side the communication object is a skeleton and regulates the access on the actual object. Two subvariants are available: with unicast each remote connection is delegated to the same object and in contrast, activatable, where a new object is created for each connection establishment.

This technology is integrated in Java APIs. A bidirectional connection is established between client and server, which results in a complicated connection setup. This approach is restricted to use the Java language. Regarding mobile devices, the handling of potential connection losses in mobile networks is uncertain, since RMI requires a constant connection.

Evaluation results

All three technologies are usable as server interfaces. On a glassfish basis it is possible to implement them as parallel access points for the actor system component. The last approach RMI, however, expects an excessive effort while developing and establishing; also the problem of connection losses in an instabile mobile context must be solved. The RESTful webservice based on JAX-RS offers all functionality, however the development of the client component is more complex, since the HTTP-requests have to be designed; in addition an interface definition is not automatically available. That is why a WSDL/SOAP webservice is the first choice for the access systematic. On server side, the functionality is configurable with annotations in the source code and the corresponding WSDL file as interface definition is generated automatically after deploying the program on a glassfish server. For the client side, it is possible to generate classes with tools of the webservice stack from a WSDL file.

2.5 Server and Middleware Technologies as Execution Environment

The execution environment should support Java and a free usage of threads. The first condition simplifies the usage with Android, while the free thread requirement is function critical for an actor system.

Cloud Computing Platforms Cloud computing platforms offer different kinds of services to third parties to use their system resources. This begins with predefined software for web applications over runtime environments to execute own programs, to complex network and storage solutions dependent on the offer of the cloud platform. For the server software the platform as a service is most promising, as it offers the possibility to run programs on cloud platforms. Therefore, the application has to be rewritten using platform dependent SDKs. In the following both platforms Windows Azure [11] and Google's AppEngine [5] with Appscale [1] are analysed, if their service supports the actor framework and the access technology. The main pitfall impacted by the architecture of the cloud platforms and the access on the SDK functionality restricts the possibility of thread generation and management. This causes malfunctions in combination with the actor system.

Google's AppEngine

This system supports Java as development language. However, extensive trials with a test implementation revealed massive restrictions when threads are generated. The Java *standard edition* (SE) functionality is replaced by a Google implementation, which enables the cloud platform to manage the threads. In the program code, the class `ThreadManager` for the generation of new threads according to the factory design pattern has to be used. A modification of the akka framework, especially the class `MonitorableThreadFactory` was identified as component for the internal thread generation and rewritten, such that no Java SE functionality is used anymore, but Google's implementation. Hence, the thread generation is properly working, new restrictions occurred; the akka framework calls the method `setDaemon` on the thread objects internally, which causes Google's security manager to throw an exception.

Windows Azure

The documentation revealed, similar to Google's AppEngine, a restricted thread usage.

Java EE with Glassfish To develop scalable distributed systems, Java *enterprise edition* (EE) comes with an API and runtime environment for a simplified development. According to the *enterprise Java bean* (EJB) API, the management of threads is done by the server component. Server internal management of instances of program components may cause an overlapping between runtime environment and user threads, when a self-defined thread is generated. User threads can be multiply created by a server internal instantiation of program components, what uses resources and disturbs a clean access of themselves. To solve this problem an architecture internal singleton EJB can be used. This variant of a session EJB is generated only once per running application by the Java EE runtime environment, thus this omits a multiple generation of user threads through internal mechanisms. However, an access synchronization on the threads is necessary. The actor framework uses Scala's own executor framework and its own thread management. In a test implementation the actor system is executed successfully in a singleton EJB. Moreover, Java's webservice APIs can be used for an efficient access on a consistent thread management. There is a fracture between actor and server threads, but both thread managements are logically separated and cause at most a not optimal scheduling of both thread groups inside the JVM. Additional trial implementations showed pitfalls by using this platform. Internally, glassfish uses a strict classloader hierarchy. It prevents that different Java EE programs gain mutual code access on each other. Furthermore, glassfish uses the executor framework of Java, which prevents to assign self-defined classloader to executing threads. In the trial environment it is not possible to add self-defined classloaders to the existing hierarchy to load classes from byte arrays; the glassfish internal loading procedure is designed to circumvent self-defined classloaders. To use self defined classes at runtime, it is necessary to copy them directly into the classpath folders of the corresponding classloaders or to deploy them together with the program on the server.

Stand-Alone-Server Implementation In this approach, no middleware would be used. The complete server component would be written manually with the help of Java's internal frameworks; this is a burdensome implementation, but the only one that results in full functionality. The thread management is completely designable, including the actor framework with configuration files. Without the restrictions of a middleware, a construction of a self-defined classloader hierarchy is

possible with this architecture, including a self-defined classloader in the execution environment of the actor framework.

Decision on Used System Finally, two types of systems are capable. The first system, which is also the current trial implementation is based on glassfish. Though, the loading procedure of remote classes is not implementable, this program enables to offload program routines from a mobile device. Therefore, the corresponding class files are deployed with the server implementation or copied into the classpath of the glassfish server. The second system would be a stand-alone-server program without middleware technology, enabling full functionality including the remote class loading procedure. The final decision must be this stand-alone-server program, but the functionality besides the remote class loading is implemented on a glassfish basis and therefore, trial programs can already be tested.

3 Architectural Design

This section lays the foundation for the first architectural design of the server in subsection 3.1. Moreover the client generation is described in subsection 3.2.

3.1 The Server Component

The server component implemented on a glassfish basis is separated into two subcomponents. The EJB component, which contains the actor system and all internal methods for accessing it. And the user access with a web component, which consists of an annotated class using the JAX-WS standard. To simplify deployment, the software is designed in a way to work with a standard setup of a glassfish server. Without a database for the trial implementation, all data is only available during the runtime of an application in the memory. However, it is important to copy the jar libraries of akka into the domain folder of the glassfish server.

The project is divided into three subprojects:

- *dynOff-EJB* contains the EJB component of the actor environment. The access on it is provided by the web component.
- *dynOff-Web* consists of a webservice for the remote access.
- *dynOff-EAR* is a project to encapsulate the previous two projects and to generate a deployable *enterprise application archive* (EAR) file.

The Class Diagram of the Server Figure 3 shows the separation into two container components, the left one for EJB and the right one for the webservices. The diagram only depicts classes and methods with self defined program logic. Libraries of Java and akka are not contained for providing a better overview.

Description of the Server Functionality Usage of the server functionality and including the akka libraries into glassfish's classpath, is the last part necessary for understanding the system. The libraries, including the Scala ones, are available on the akka.io website and must be separately downloaded to use the server component. Alternatively, the project is built with Maven and generates a Docker image, which can be deployed on a Docker enabled server. The implementation relies on akka in version 2.5.3, though no problems should occur with future versions, as long as there are no changes on the basic functionality. All library files for akka must be copied into the classpath of the glassfish server. Recommended by glassfish's documentation is to include user libraries into the directory *domain-dir*/lib. It is also possible to include the libraries into the EAR container, but it will increase the size of the container and slow down deployment. Executable code is bound to actors and controlled by messages.

Figure 4 depicts the general system usage in more detail:

1. Generation of actors with the webservice function `generateActorFromProps` or `generatePreAvailableActor`.
2. Transmission of a message to a generated actor via `sendMessage` or `dispatchAsyncJob`.
3. Reception of a message object either synchronously or asynchronously via `getAsyncJobResult`.

Actor Generation All actors must inherit the class `AbstractActor` of akka. All classes used inside the actors, including the actor and message classes, must be integrated in the classpath of the glassfish server either on application or on server level. For the generation of actors with the method `generateActorFromProps` a serialised instance of the configuration class `Props` is needed in a byte array representation and in return a string with identification features is generated. This string is used by the actor for identification. The return string of the method `getPreAvailableActor` has the same functionality. An example implementation is found in the test actor.

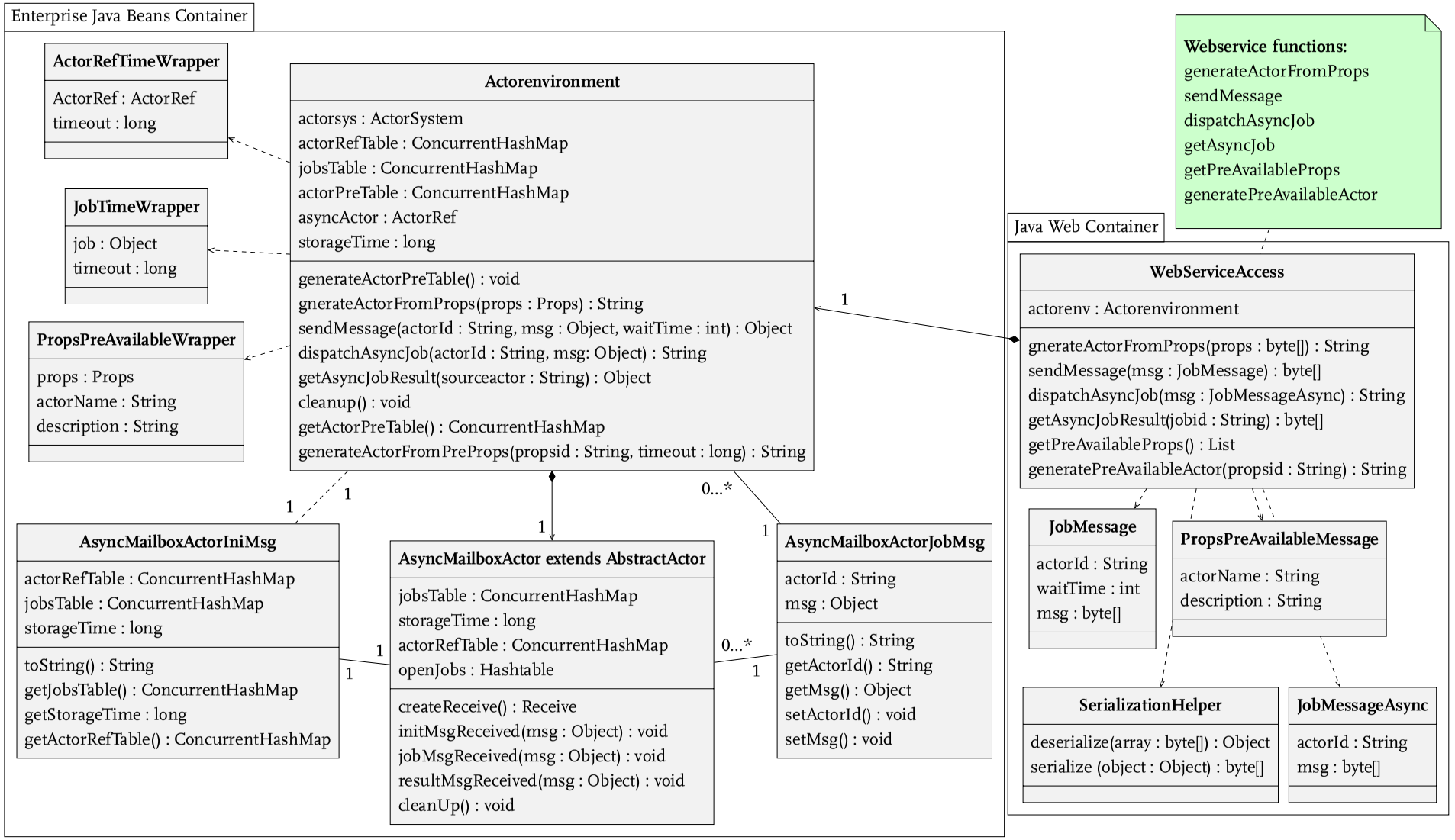


Fig. 3. An UML class diagram of the glassfish implementation. It shows the modular structure of the EAR archive, which contains two containers: the *EJB* container, which is responsible for exchanging messages with the actor environment, and the *web* container that publishes the services via web service functions described in WSDL.

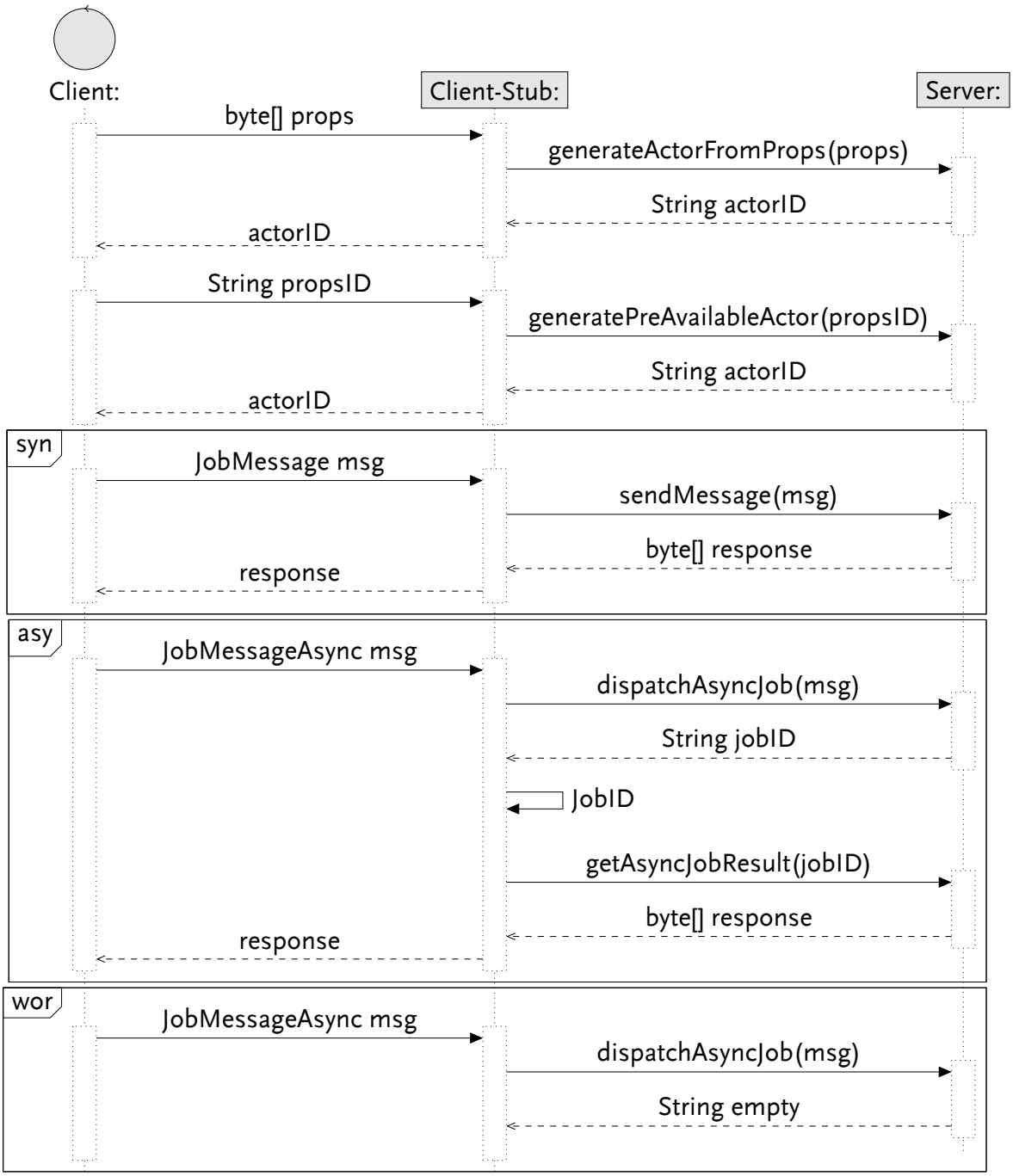


Fig. 4. Sequence diagram of communication processes between a client and a server. The job message *msg* can be transmitted with three different paradigms: either *syn*, which describes a synchronous pattern, *asy*, which shows the asynchronous communication, or *wor*, which shows an offloading without a reponse.

Message Transmission Message transmission is done by either `sendMessage` or `dispatchAsyncJob`. The first uses synchronous communication and takes a `JobMessage` object as encapsulating object for the SOAP transmission. This object contains an actor identification string, a serialised message object and a maximum response time. The second method provides an encapsulating object without a timeout and returns a job identification string, which enables a delayed message call via `getAsyncJobResult`. For the case that no response is expected, it is advisable to send an empty object as answer from your own actor.

Message Reception With a synchronous communication by `sendMessage`, a serialised response object is returned directly. If the wait time is expired a `ServerFault` exception is thrown. Using asynchronous communication the response message must be called with the job identification string with the method `getAsyncJobResult`. If a response message is not available, a `ServerFault` exception is thrown, too. For a successful deserialisation and casting into the message class, it is necessary that the corresponding class files are available on the client and on the server.

3.2 The Client Access

The implementation on client side depends on the webservice technology with its access based on WSDL/SOAP. Since the API of JAX-WS was already used for the preparation of the webservice functionality and is included into the Java SE standard libraries, the following description is based on it. After deploying the project on the glassfish server, a WSDL file is available via an URL. With the help of this file and the tool *wsimport*, classes and interfaces can be generated, which can be instantiated into a service object on runtime. On this object the service functions can be called like methods. The generated artefacts can be divided into the following categories:

– *Message artefacts*
  Classes for message transport annotated regarding the *Java architecture for XML binding* (JAXB). For each request and response as well as own classes for complex encapsulated content and exceptions. All classes are necessary for the functions.

– *Helper artefacts*
  An `ObjectFactory` class to generate classes regarding the factory pattern and a `package-info` class with annotations for setting the namespaces in the transmitted XML code. These classes are not relevant for the functions.
– *Stub artefacts*
  Consisting of an interface with JAX-WS annotated methods for using the webservice and a factory class for generating stub instances, which can be used on the former mentioned interface. Both classes are function critical.

3.3 An Actor Example

With the trial program code an example actor is delivered, which implements a simple echo call. It consists of two classes, a `TestActor`, which is the actor class and a `TestMessage` as message class, which expects an actor and transmits it. Instances of `TestMessage` contain a String that the actor sends back as response.

4 Conclusion & Future Work

The concepts of remote usage of the akka framework for dynamic code upload from a mobile device were described and a sample architectural implementation is made publicly available at the github repository whatever4711/dynOff[1]. This platform is hopefully used for dynamic code offloading experiments soon. A great advantage is the usage of the akka.io framework, which simplifies outsourcing user code, since Java libraries and resources are used, which offer a widespread area of application. Furthermore, akka with its actor model eliminates any concurrency issues when it is deployed on a server. Moreover, akka offers a great potential by configuring its execution environment and may even be distributed onto a cluster of servers to be more efficient. Thus, the offloading procedure sounds useful with the help of a server, it is also possible to run the actor system itself on the mobile device by using a background service, too. Nevertheless, a potential area of application is to avoid the bottleneck of a mobile network by outsourcing important communication structures onto a server with high-speed network access. However, security issues were not focused until now and should be analysed, especially regarding the WSDL access of the server communication. Another point

[1] https://github.com/whatever4711/dynOff

is a migration of the Java EE functions into a Java SE, since Android devices are not supporting a Java EE environment. Finally, with the stand-alone-server implementation it should be possible in future to extend it with own classloader functionalities, such that realtime code offloading is enabled.

References

1. AppScale Systems Inc. AppScale, 2013.
2. Byung-Gon Chun, Sunghwan Ihm, Petros Maniatis, Mayur Naik, and Ashwin Patti. CloneCloud: Elastic Execution Between Mobile Device and Cloud. In *Proceedings of the Sixth Conference on Computer Systems*, EuroSys '11, pages 301–314, New York, NY, USA, 2011. ACM.
3. Eduardo Cuervo, Aruna Balasubramanian, Dae-ki Cho, Alec Wolman, Stefan Saroiu, Ranveer Chandra, and Paramvir Bahl. MAUI: Making Smartphones Last Longer with Code Offload. In *Proceedings of the 8th International Conference on Mobile Systems, Applications, and Services*, MobiSys '10, pages 49–62, New York, NY, USA, 2010. ACM.
4. H. Flores, P. Hui, S. Tarkoma, Y. Li, S. Srirama, and R. Buyya. Mobile code offloading: from concept to practice and beyond. *IEEE Communications Magazine*, 53(3):80–88, March 2015.
5. Google. Google App Engine, 2013.
6. Carl Hewitt, Peter Bishop, and Richard Steiger. A universal modular ACTOR formalism for artificial intelligence. In *Proceedings of the 3rd international joint conference on Artificial intelligence*, IJCAI'73, pages 235–245, San Francisco, CA, USA, 1973. Morgan Kaufmann Publishers Inc.
7. IBM. *Understanding the Java ClassLoader*, 2011.
8. Roelof Kemp, Nicholas Palmer, Thilo Kielmann, and Henri Bal. Cuckoo: A Computation Offloading Framework for Smartphones. In Martin Gris and Guang Yang, editors, *Mobile Computing, Applications, and Services*, volume 76 of *Lecture Notes of the Institute for Computer Sciences, Social Informatics and Telecommunications Engineering*, pages 59–79. Springer Berlin Heidelberg, 2012.
9. S. Kosta, A. Aucinas, Pan Hui, R. Mortier, and Xinwen Zhang. Thinkair: Dynamic resource allocation and parallel execution in the cloud for mobile code offloading. In *2012 Proceedings IEEE INFOCOM*, pages 945–953, March 2012.
10. Tim Lindholm, Frank Yellin, Gilad Bracha, and Alex Buckley. *The Java Virtual Machine Specification - Java SE 8 Edition*, 2015.
11. Microsoft. Windows Azure, 2013.
12. Oracle. Java EE 6 Tutorial, 2013.
13. Typesafe Inc. akka.io, 2013.
14. Christian Ullenboom. *Java ist auch eine Insel Einführung, Ausbildung, Praxis*. Rheinwerk, Bonn, 2016.