nOSGi

A POSIX-Compliant Native OSGi Framework

Steffen Kächele, Jörg Domaschka, Holger Schmidt and Franz J. Hauck
Institute of Distributed Systems, Ulm University, Germany
{steffen.kaechele,joerg.domaschka,holger.schmidt,franz.hauck} @uni-ulm.de

ABSTRACT

Ubiquitous computing aims at dynamically supporting users in everyday life with applications on mobile and embedded devices in the surroundings. Component frameworks, such as OSGi, ease the dynamic management of such application software. Yet, OSGi focuses on Java, while many mobile devices only support native languages with reasonable performance. Furthermore, Java may increase the costs for devices, and even small additional costs are a relevant factor for mass-market production.

This paper presents nOSGi, the first native OSGi implementation. Our C++ prototype implements the features of the OSGi R4 specification and runs without any modifications on standard POSIX systems with support for ELF binaries. It provides the core functionality of the OSGi module, life cycle and service layer in a native C++ environment. We measured considerable improvements of performance and memory consumption in comparison to common Java OSGi frameworks. Successful tests on various platforms, such as x86, x64, Sun SPARC and ARM demonstrate the portability of nOSGi.

Categories and Subject Descriptors

D.2.8 [**Software Engineering**]: Metrics—*Performance measures*; D.3.3 [**Programming Languages**]: Language Constructs and Features—*Frameworks*; D.2.11 [**Software Engineering**]: Software Architectures—*Domain-specific architectures*

General Terms

Design, Management, Performance

Keywords

OSGi, POSIX, native, C++, components

COMSWARE '11, July 4-5 2011, Verona, Italy.

1. INTRODUCTION

Ubiquitous computing (UbiComp) envisions a future, in which people are guided through everyday life by ubiquitous applications [24]. These applications run on mobile and thus resource-limited devices in the user surroundings. On these smart devices, updating to add new functionality and bug fixing becomes more and more necessary. One platform for on-demand updating and bug fixing on desktop machines is Java OSGi [15].

OSGi is a common industry standard for a lean Java component framework. It allows runtime installation, update and uninstallation of components (so-called bundles). Bundles share functionality on the basis of Java packages that are exported and imported by means of a bundle manifest. Following the paradigm of service-oriented architectures [18], bundles provide applications in terms of OSGi services.

Although originally being designed for network-attached, resource-restricted devices, such as gateways and set-top boxes, OSGi currently represents a state-of-the-art modularisation approach for comprehensive Java software, such as Eclipse and the Glassfish application server. Thus, current framework implementations, such as Eclipse Equinox [5] and Apache Felix [1], were not designed with resource-limited devices in mind. Only Concierge [16] targets such devices but still requires the relatively heavy-weight Java ME Connected Device Configuration [21].

Java frameworks that provide OSGi mechanisms rely on an appropriate JVM. This Java environment in turn requires a lot of resources on its own. As Java is a semi-compiled language, it provides only weak performance on resource-limited devices. In order to get a satisfying runtime performance, Java requires powerful processors. Some Java virtual machines for desktop computers increase Java performance by translating Java byte code to machine code at runtime. However, this requires lots of dynamic memory which is rarely available on mobile or embedded devices. Altogether, in the area of resource-restricted devices the general purpose use of Java is difficult as the Bill Of Material (BOM) is an important concern and developers mainly have experience in C(++) developing [11]. Furthermore, for many devices there is even no appropriate Java virtual machine (JVM) available. Thus, while OSGi provides important features for small and embedded devices, such as dynamic component updates at runtime, it cannot be applied to many of these devices.

We think the dynamic OSGi environment is not necessarily limited to Java systems. To provide the same dynamics

and maintainability to native languages there should be a framework that implements most of the OSGi APIs and is able to dynamically load code. Such a dynamic environment makes it easier to reuse parts of the system, to extend and maintain software.

This paper introduces nOSGi, the first native OSGi R4 framework running on top of standard POSIX systems. Our C++ prototype uses shared objects to implement modularisation: each package (i.e., C++ namespace) is represented by a shared object file in the Executable and Linking Format (ELF). This allows implementing the dynamic import and export of packages between bundles at runtime with standard `dlopen` and ELF mechanisms. Bundles are implemented as ZIP files containing a standard OSGi manifest and shared object files representing packages. Without any imports in the manifest, shared objects of different bundles are isolated from each other. To establish visibility between shared objects, the runtime system adds ELF dependencies to the shared object files after resolving available packages according to specified imports. Thus, OSGi code-sharing functionality is provided by standard ELF mechanisms which are highly mature. Moreover, nOSGi provides OSGi life-cycle layer functionality (e.g., starting, updating and stopping of bundles). It implements the OSGi service layer with a service registry including support for RFC1960-compliant filters [8]. Finally, it supports the OSGi event and listener concept by offering framework, bundle and service listeners.

We provide an extensive evaluation that highlights the differences of nOSGi in comparison to the Java frameworks Concierge and Eclipse Equinox. Even in comparison to Concierge, nOSGi provides a much higher performance while consuming less memory. Thus, the measurements show that by using a native programming language to implement the OSGi specification, nOSGi particularly fits resource-limited devices. Finally, measurements with different system architectures, such as x86, x64, SUN SPARC, Alpha and ARM, demonstrate the portability of nOSGi with respect to POSIX-compatible systems supporting ELF binaries.

To sum up, this paper makes the following contributions. (*i*) It analyses the gap between the requirements of the OSGi specification and the features provided by the programming language C++. (*ii*) It introduces a framework that implements the features of OSGi R4 in C++ and runs on standard POSIX-compatible systems with support for ELF binaries. (*iii*) Further, it provides an extensive evaluation of nOSGi in comparison to common Java frameworks.

This paper is structured as follows. In the next section, we introduce OSGi. After analysing the requirements for a native OSGi implementation in Section 3, Section 4 shows the nOSGi approach and Section 5 demonstrates an example application. Then, we present an elaborate evaluation of nOSGi in Section 6. Finally, we discuss related work in Section 7 and conclude in Section 8.

2. OSGI BASICS

OSGi [15] is a commonly used Java component framework. It allows installing, updating and uninstalling of software components at runtime without the need to restart the entire platform. Thus, it perfectly fits dynamic application scenarios.

The OSGi platform consists of multiple layers (see Figure 1). *Hardware*, *operating system* and *JVM* are beyond

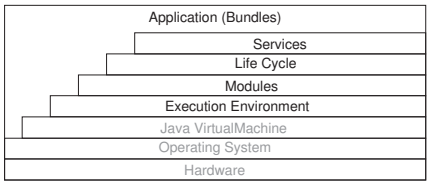

Figure 1: OSGi architecture

the OSGi specification. OSGi runs on various hardware architectures ranging from standard desktop systems to embedded devices with different operating systems. However, available JVMs for the respective operating systems highly differ regarding the provided features and interfaces. For instance, on desktop systems typically Java 1.5 or Java 1.6 is available while on mobile and embedded devices often only Java 1.2 can be used. For handling this heterogeneity, OSGi builds on top of an *Execution Environment* layer specifying multiple profiles with respect to JVM features (e.g., the *OSGi/Minimum-1.0* profile covers minimum requirements to run OSGi).

The following sections introduce the core concepts of module, life cycle and service layer in detail.

2.1 Module Layer

The module layer is the lowest logical layer of OSGi. It defines the basic module concept of the OSGi framework. In OSGi, application code is partitioned into modules called *bundles*. These can independently be installed and uninstalled in the framework. From a technical point of view, bundles are Java archive (JAR) files which include the necessary classes and resources to provide a particular functionality. Additionally, they contain a manifest file describing bundle properties, such as bundle name and version.

Initially, bundles are completely isolated from each other. Yet, OSGi allows sharing of module code among different bundles if desired. For this purpose, the bundle manifest can specify which code is to be imported from other bundles and which of its own code is accessible for other bundles. Sharing of code is implemented on a per-package basis. After having imported a package in the bundle manifest, it can be used in the Java code in a transparent way (i.e., as if it is part of the same bundle). Further, bundles can export packages that dependent on classes imported from another bundles. When a developer wants to ensure that there are no conflicting imports within the whole bundle's class space, he can specify inter-package dependencies with the "uses"-directive in the manifest-file. This requires the framework to transitively check all imports of a bundle.

OSGi framework implementations realise the isolation of bundles by assigning each bundle a separate Java class loader. Such a class loader is responsible for loading the class byte code into the JVM. There, classes are identified by both their fully-qualified name and the ID of the class loader so that loading the same byte code with different class loaders will lead to type-incompatible classes. Thus, in order to enable access to code from other bundles, OSGi connects the appropriate class loaders according to the imports in the respective bundle manifest. Hence, connected class loaders are used to load needed classes. In case a required class is not found via these class loaders, the bundle's own bundle class loader is used.

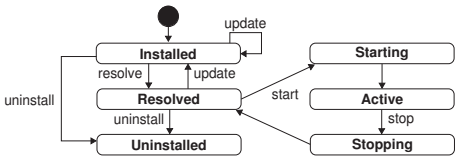

Figure 2: OSGi bundle life cycle

2.2 Life Cycle Layer

On top of the static concept of the module layer, OSGi provides the dynamic aspect of a bundle life cycle. For this purpose, the life cycle layer defines a state machine with bundle states and actions (see Figure 2). The life cycle of a bundle starts in state *installed*. Only if the framework is able to resolve all imported packages of a bundle according to Section 2.1, the bundle switches to state *resolved*. In this *wiring process*, the framework relocates each imported package to a providing bundle. Technically, the framework connects the class loader of the importing bundle with the class loader of the providing bundle. When the user updates a bundle, all dependencies have to be resolved again.

Each bundle has an activator associated with it. The activator executes bundle management tasks at bundle startup and bundle shutdown. Such tasks are for instance service registration and deregistration (see Section 2.3). When a bundle is to be started, its state changes from *installed* to *starting* and the framework calls the activator's `start()` method. Once the method has returned, the bundle state changes to *active*. For stopping a bundle its state changes to *stopping* and the framework invokes `stop()` at the activator. Bundles may register a listener in order to be notified when the state of other bundles changes

The life cycle layer is responsible for consistency and determinism of bundle dependencies. It has to ensure that all bundles with the same import statement in their manifest import the same particular package even if multiple bundles export this package (i.e., *wiring*).

2.3 Service Layer

The service layer introduces the concept of OSGi services in order to realise a service-oriented and, in consequence, loosely coupled system. The major entity of this layer is the *service registry*. It allows bundles to register services by a certain key. OSGi advises that the key be equal to the interface names the service implements. Other bundles can look up registered services at the service registry and use them to implement their required functionality. In this case, the service registry automatically ensures that service consumers only see compatible service providers. Thus, if there are different versions of the same service interface in the service registry at the same time, service consumers will only see the version they are compatible with. For looking up services, the service registry also supports LDAP-style *filters* [8]. They allow obtaining the best fitting service in case multiple services are registered with the same key.

Bundles are allowed to register and unregister services at any time. Thus, developers have to ensure that service-consuming applications are able to handle such behaviour. For this purpose, OSGi provides an event concept to notify service users about changes.

3. CHALLENGES FOR A NATIVE OSGI FRAMEWORK

The OSGi specification highly relies on features that are provided by the Java language and its runtime system. For a native implementation some of those Java features are not available. In particular, there is no unified general-purpose class loading mechanism. In the following, we analyse the gap in detail and present an overall list of requirements that build the basis for nOSGi, our native C++ implementation that is presented in Section 4.

3.1 Code Loading

OSGi offers the fundamental mechanism to install bundles at runtime without the need to stop the entire system or other bundles. For this purpose, the OSGi specification leverages the Java class loader. Furthermore, the class loader is the main mechanism for exporting and importing packages. As the C++ runtime system does not provide a class loader other means are required to add code and to enable code sharing.

Loading. The POSIX standard [22] specifies the interface `dlopen` to make executable object files available to the calling program. `dlopen` supports custom executable file formats. Yet, we focus on *Executable and Linking Format* (ELF) files as these are common in many Unix-like operating systems. ELF specifies shared objects for dynamic code loading. Dependencies between shared objects are matched with the shared object name either with or without the path. To the best of our knowledge, `dlopen` is the only standardised way to add code to a running application; thus, a native OSGi implementation has to make use of it.

Visibility. OSGi also requires that the visibility of bundles for each other be configurable. Packages of a bundle become visible for another package only if they are exported by the providing bundle and imported by the consuming bundle. In this paper, we use the term *namespace* to describe the symbols being visible by a library; we do not refer to C++ namespaces in this case. `dlopen` offers that symbols of loaded libraries are either globally visible (i.e., `RTLD_GLOBAL`) or hidden (i.e., `RTLD_LOCAL`). Neither approach sufficiently satisfies the OSGi requirements: with public symbols, bundles are not isolated, and with private symbols, sharing is impossible. Also `dlmopen`, an extension to `dlopen` that allows loading of shared libraries into different namespaces, does not solve the problem. It is not part of the POSIX specification and some implementations even restrict the number of available namespaces to 16. Consequently, the native implementation has to use mechanisms beyond `dlopen` to control bundles exports and imports.

Versioning. OSGi allows loading of different versions of a bundle at runtime and each of them can export the same package with the same name (and symbols) but at different versions. This may cause the following problems in native environments. In case both bundles share a common namespace, name clashes will occur, if there are identical symbols. If other bundles depend on the exported packages, OSGi will require that wiring be deterministic. For this purpose, all bundles importing the package have to import the same version. Neither `dlopen` nor the linker provides such a guar-

antee. Hence, our native OSGi should implement the wiring process on its own.

OO Support. The *dl-function* family is tailored towards C as programming language. Thus, it lacks support for object orientation which is required by the OSGi specification. This includes the instantiation of dynamically loaded classes. For C programs, function names are directly mapped to symbol names. By contrast, C++ method names cannot be directly mapped to symbols by the identity function due to the fact that C++ supports method overloading and overriding [9]. Instead, C++ applies a more complex mapping on the basis of class, method, parameters and return value – the name mangling mechanism. Our native OSGi framework has to be able to load and invoke object-oriented code on demand despite of name mangling.

3.2 Bundles

There are challenges concerning the management of bundles. It has to be possible to export/import code on a per-package basis. Bundles have to be deployable in a single file together with meta-information. Finally, updating and rewiring of bundles is an issue.

Modules. It is possible to put the complete bundle implementation with all packages and classes into a single library. With this approach, the whole bundle content is available for all comprised modules and classes and by removing the library the bundle can easily be removed from memory. Yet, importing and exporting is only possible on a per-bundle basis while OSGi requires dependencies on package level.

Java class code is loaded from a single file each. Mapping this approach to a native environment requires creating a shared library for each class. Yet, such an approach leads to complex dependencies between shared libraries. OSGi uses packages as modularisation unit. Hence, our native bundles should be composed of multiple shared objects containing the respective C++ namespace (i.e., package) code.

Bundling. Standard Java OSGi frameworks use JAR files to distribute OSGi bundles. These include a manifest file to describe the bundle. However, JAR files belong to Java and C++ lacks support for these files. Thus, our native OSGi framework should specify another native bundle file format.

Update. To be compliant to the OSGi specification, our native OSGi framework has to support bundle updates at runtime (i.e., replacing currently loaded and possibly bound code). Yet, updates only have to be processed immediately if none of the exported packages is imported by other bundles. Otherwise, updates may take place during next framework start. Our native implementation has to consistently map those requirements to the code loading mechanisms.

3.3 Services

For the service-layer, there are differences with respect to service method invocation and the C++ class hierarchy. Finally, the performance of the service registry is critical to overall performance.

Access. The service layer allows communication between services without having access on module layer as follows.

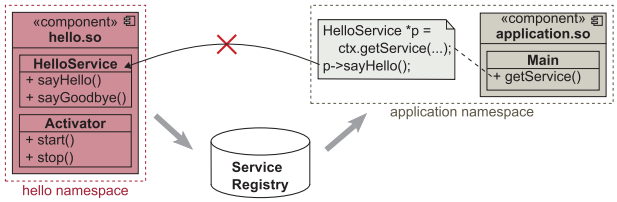

Figure 3: Isolated namespace issue: accessing the code of a service implementation from a different namespace

The service-providing bundle registers the service instance at the service registry. Then, a service-consuming bundle can retrieve the service reference.

For instance, in Figure 3 the library `application.so` uses `HelloService`. In C++, symbols are already mapped at compile time (i.e., *early binding*); at runtime, the symbol is looked up in the symbol table to get the memory address of the method call. Yet, in Figure 3, the application modules of the service implementation and the service consumer are isolated from each other. Thus, the method symbol is not visible for the consumer. A native OSGi framework should be able to handle such situations.

Filter. If multiple services are registered with the same interface, properties can be used to select the most appropriate service. These properties can be set at service registration at the service registry. Then, service discovery allows filtering with RFC 1960 filters [8]. In OSGi, properties consist of a key of type `String` and a value of type `Object`. Using the super class `Object` allows storing any type of value with a given key string. Due to the fact that C++ does not include a common super class, a native OSGi framework needs another approach to implement service properties.

At runtime, particular services can dynamically appear and disappear. To manage such dynamics, bundles can register service listeners at the service registry. These listeners are notified by an event whenever the service state changes. If a bundle registers a new service, all interested bundles are notified by the event "registered" referencing the new service. The events "unregistering" and "modified" signal deregistration and modification of services. Bundles can also restrict notification to particular services with RFC 1960 filters. If many filters are registered, the service registry performance highly depends on analysing these filters. A native OSGi framework should efficiently support service events.

3.4 Summary

This section discusses the differences between a Java and a native execution environment regarding the demands of OSGi. This leads to the following requirements for a native implementation:

1. **Loading**. A native OSGi implementation has to provide dynamic loading of code at runtime on the basis of shared object files that are loaded via `dlopen`.

2. **Visibility**. It has to offer a mechanism to control the visibility of symbols.

3. **Versioning**. There has to be a mechanism to load different versions of the same package. The behaviour

of packages relying on such a package must be deterministic.

4. **OO Support**. It has to be able to deal with libraries containing object-oriented code.

5. **Modules**. Bundles shall be deployed as a set of shared objects each of which representing a package (i.e., C++ namespace).

6. **Bundling**. Bundles have to be packaged and deployed in files with a novel JAR-like file format. It contains all classes, libraries and a manifest file.

7. **Update**. It has to be possible to update packages following the constraints defined in the OSGi specification.

8. **Access**. Bundle code has to be able to access service instances even if there is no import/export relationship on module level between provider and consumer bundle.

9. **Filter**. The implementation of filters and the associated listeners have to support values of any type even though there is no common base class. The performance of RFC 1960-style filters has to be reasonably high as it is critical for the overall system performance.

4. THE nOSGi APPROACH

This section introduces our POSIX-compliant implementation of OSGi with C++ as a native language. We discuss our approach considering all requirements analysed in Section 3. Some requirements represent constraints that do not leave space for design decisions. Thus, they are not explicitly discussed here. In particular, these are the use of `dlopen` to add code at runtime in a POSIX-compliant manner (Requirement 1) and the hence resulting modularisation of code with shared object files on a per-package level (Requirement 5). The filters addressed in Requirement 9 are evaluated with respect to performance in Section 6. Further, we provide support for inter-package dependencies by means of "uses"-constraints (see Section 2.1).

4.1 Code Loading

Visibility. In order to implement Requirement 2, we use standard dependency resolution of shared libraries: if a shared library specifies dependent libraries, its namespace is implicitly extended by the (transitive) namespace of all dependent libraries. The dependency setup is implemented with shared library dependencies on OSGi package level (i.e., nOSGi directly maps OSGi package imports to the corresponding library dependencies). We describe details of dependency mapping in the paragraph about versioning below.

A significant benefit of this solution is that the critical management of symbol visibility in different namespaces is entirely managed by common and highly mature mechanisms of `dlopen` in conjunction with the link editor. nOSGi only configures dependencies of shared libraries.

Versioning. To avoid namespace conflicts when loading multiple versions of the same OSGi package (Requirement 3), nOSGi supplies the corresponding shared libraries with unique names. For this purpose, shared library names are composed of package name following the bundle identifier (OSGi requires that the framework assigns bundles a unique identifier at runtime).

As already described, package imports are directly mapped to dependencies on shared object files. The bundle identifier (ID) being part of the shared library names poses a major challenge as the ID is not known before bundle installation. Thus, a bundle developer in fact knows the dependent packages; yet, she is not able to specify the runtime bundle ID of the bundles providing these packages. Thus, it is impossible to set up the dependencies on shared object level at compile time. These can only be specified in the bundle manifest. This leaves the task of injecting dependencies into shared object files to nOSGi.

nOSGi sets up library dependencies at runtime: when installing and resolving (i.e., wiring) a bundle, nOSGi automatically injects the required dependencies into the shared libraries. As we focus on ELF in our prototype, we use the *libelf* library to adjust the dependencies of the shared libraries at runtime. For this purpose, we modify the `.dynamic` section of the libraries containing the dependencies to other libraries (i.e. DT_NEEDED entries). We add the bundle ID of the exporting bundle to the library dependencies to specify the complete names. In case of rewiring bundles, nOSGi reloads the affected and changed shared libraries.

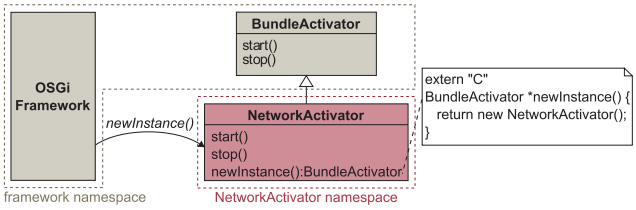

Figure 4: Bundle instantiation in the bundle activator class

OO Support. By analysing the standard OSGi workflow, we identified that Requirement 4 does not have to be solved in a general way. OSGi specifies that the framework only accesses bundles through the bundle activator interface (i.e. for starting and stopping bundles). Thus, the framework only invokes unknown object-oriented code just after a bundle has been loaded and shall be started (i.e., an instance of the bundle activator has to be created and its `start()` method has to be called). Yet, the dl-family does not support object-orientation including object instantiation. Further, as a result of the name mangling mechanism (see Section 3.1), the constructor symbol name of the activator is unknown.

As a solution, nOSGi provides a C++ macro that adds a factory method to the OSGi bundle activator (see Figure 4). To overcome the name mangling issues, the provided factory is marked as `extern "C"` to use standard C naming conversions. Thus, the factory offers a compatible starting point for `dlopen` to create an activator instance of bundles. Once an instance of the bundle activator has been created, the framework can access the `start()` and `stop()` methods by means of C++ polymorphism because the methods are defined by the `BundleActivator` prototype.

The activator class is specified in the bundle manifest.

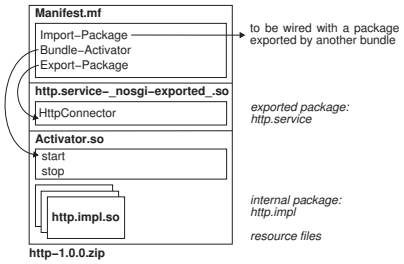

Figure 5: The structure of a bundle file

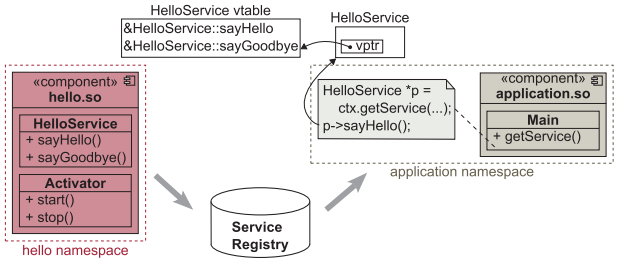

Figure 6: Solution to the isolated namespace issue: Accessing the service implementation in other bundles via vtable

nOSGi assumes that activators are located in a shared object file that only contains a single activator implementation. This shared object file has to follow naming conventions. Then, it can be loaded via `dlopen` on the basis of the class name from the manifest file. After it has been loaded, nOSGi can invoke its factory method. This only affects the activator class. There are no such restrictions for other classes.

4.2 Bundles

Bundling. To implement bundles (Requirement 6), nOSGi uses a regular ZIP file that contains a set of shared object files representing the implementation packages and a bundle manifest file (see Figure 5). nOSGi adopts the directory structure and the structure of the manifest file according to the OSGi specification. Also like many Java implementations, it uses a caching directory per bundle where information about the bundle and its state is stored. Thus, the framework can restore the state of installed bundles after stopping and starting the framework.

Yet, there are differences to Java OSGi frameworks. As `dlopen` cannot directly load shared object files from a ZIP file, nOSGi unpacks the file to the bundle directory. For this purpose, it uses the bundle ID as directory name.

The main focus of nOSGi is to provide the dynamics of OSGi such as updating, bug-fixing at runtime and the extension of existing applications to native languages. In most of these cases, the vendor provides compatible precompiled bundles for his system to update and extend functionality. Yet, such bundles are not fully portable because shared object files are tied to a specific platform. For instance, an x64-compiled bundle is not runnable on a 32-bit system. For an easy integration of third party components, such as open-source libraries, nOSGi additionally supports *source bundles*. Instead of shared libraries, these include the source code with an appropriate Makefile [20]. Developers can precompile them before deploying them to a device. Further, they can directly deploy them as source bundle to a device. An additional entry in the bundle manifest marks the bundle as source bundle (`Bundle-Content: source`). At installation time, nOSGi compiles the code. Then, the bundle can be used as if it had been precompiled. In case of highly restricted devices, this compilation can be additionally done offloaded [7] or in a previous step before deployment.

Update. Updating isolated bundles at runtime (Requirement 7) is not a severe issue. For this purpose, the bundle is stopped by calling `stop` at its bundle activator. The `stop` method allows releasing used resources. Yet, it is the task of the bundle developer to ensure that clean-up happens, e.g., running threads are stopped. Then, the bundle is removed (i.e., unloaded from memory). After updating the bundle files on disk, the new bundle is loaded into the framework. This includes restoring the prior bundle life-cycle state.

Bundles exporting packages that are imported by other bundles have to be handled differently. nOSGi updates such bundles by caching the new bundle version on disk while keeping the old version in the nOSGi working directory for current wiring. According to the OSGi standard, this stale wiring is cleaned up either during the next framework start-up, or by means of the *refresh* method. By refreshing a bundle, the framework calculates a list of transitive dependencies in order to rewire these bundles again.

4.3 Services

The OSGi service layer provides a mechanism for bundles to interact in a loosely coupled way through services (see Section 2.3). Service-provider and service-consumer bundles are generally isolated from each other (i.e. they are in different namespaces) and exchange their service through the OSGi service registry (for an example see Section 5.2). In order to prevent class cast problems, the service registry ensures that bundles can only see services that are not incompatible with them. Compatibility is given when the bundle is either not wired to a corresponding interface package, or when it is wired to an interface package that is compatible with the service.

Access. For accessing services that reside in different namespaces and are only known by their interface (Requirement 8) it is important that the method binding for the service implementation is done at runtime (see Figure 6). For this purpose, methods of the service interface (i.e., the abstract type) have to be tagged by the key word `virtual`. Consequently, method calls are not directly mapped to addresses at compile time (i.e., *early binding*) but are redirected to an additional in-memory table *vtable* (i.e., *late binding*). The *vtable* is created by the compiler and contains the memory addresses of all virtual methods of a specific type. It is accessible from all modules. Thus, virtual methods can even be accessed from modules in other namespaces although the service implementation is not visible to them.

Filter. nOSGi supports RFC 1960 compliant filters for service discovery (Requirement 9). Similar to the Concierge framework [16], the filter string is disjoint according to the priority of elements to determine the logical value. Accord-

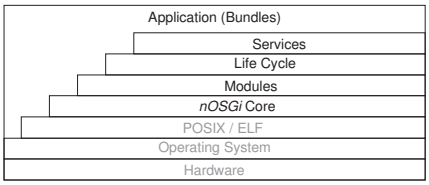

Figure 7: nOSGi architecture

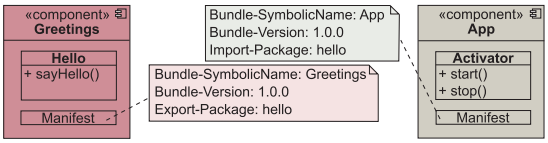

Figure 8: Basic example: bundle `App` imports the exported *Hello World* functionality of bundle `Greetings`

ing to the user preferences, the service registry either returns one or all appropriate and currently available services.

Regarding filter properties we introduce a `Property` class that is capable of holding native types as well as references to other objects. When registering a service we use `Property` where Java uses `Object`.

4.4 Summary

Figure 7 shows the overall architecture of nOSGi. POSIX and ELF abstract from the operating system and hardware in use. The nOSGi core layer contains the class loading mechanism as well as the basic functionality to patch shared object dependencies at runtime. The module layer is realised as described in Section 4.1 and 4.2. The life cycle layer is implemented as specified in the OSGi specification. Like most Java OSGi frameworks, nOSGi provides a console bundle that allows user-interaction with the life-cycle layer (e.g., starting and stopping of bundles). Finally, the service layer enables implementing service-oriented applications by means of the mechanisms introduced in Section 4.3.

5. WRITING NATIVE OSGI BUNDLES IN A NUTSHELL

In this section, we give details on writing bundle code for our nOSGi framework. With a basic example we highlight the concepts of the nOSGi module layer as described in Section 4. Figure 8 shows a scenario with two bundles. The `Greetings` bundle exports the package `hello` that provides a typical *Hello World* class. This package is imported by an `App` bundle that uses the offered *Hello World* functionality. Beside the nOSGi module layer, Section 5.2 shows a brief example for using the service layer to implement applications in a service-oriented manner. Overall, writing bundle code for nOSGi is easy and intuitive for Java OSGi developers as our application programming interface (API) is in general similar to standard OSGi. Yet, there are small differences that are highlighted in the following sections.

5.1 Exporting and Importing Packages

To provide functionality as part of a bundle, the developer has to provide an appropriate implementation. The implementation is conventional C++ code (with the example of

```
1  #include "Hello.h"
2  #include <iostream>
3
4  Hello::sayHello() {
5    std::cout << "Hello world!" << std::endl;
6  }
```

Figure 9: `Hello` code of the `Greetings` bundle

```
1  #include "Activator.h"
2  #include "Hello.h"
3
4  NOSGI_SETACTIVATOR(Activator)
5
6  void Activator::start(BundleContext &bc) {
7    Hello *h = new Hello();
8    h->sayHello();
9    delete h;
10 }
11
12 void Activator::stop(BundleContext &bc) {
13 }
```

Figure 10: `Activator` code of the `App` bundle

the `Hello` class, Figure 9 shows that nOSGi requires no special commands within the source code. Even though nOSGi delimits the interaction between code modules, it does not restrict the access to specific memory locations in C and C++, e.g., when using memory-mapped I/O). Then, the developer creates a shared object file with the name *hello_nosgi-exported_* to create the `hello` package. The nOSGi framework replaces the suffix `_nosgi-exported_` with the bundle ID at bundle installation. In order to export the package `hello`, the developer of the `Greetings` bundle adds an *Export-Package* header to the bundle manifest (see Figure 8). Through this, nOSGi adds the package to its list of exported packages with the corresponding bundle ID.

To import the `hello` package from another bundle, the developer of the `App` bundle has to insert an *Import-Package* header into the bundle manifest according to Figure 8. In this case, nOSGi searches for an appropriate exported package. If such a package is found, nOSGi injects necessary dependencies to other shared object files into all of App's shared object files (cf. Section 4.1). Otherwise, the bundle remains unresolved until the imports can be satisfied. In our example, nOSGi introduces a dependency to another shared object *hello.so* that implements the hello package. In consequence, the symbols of hello.so become visible for `App`. Thus, App can access the `Hello` class which is such a symbol. The bundle developer can access the necessary `Hello` class of the imported package in a transparent manner (see Figure 10, Line 7).

Figure 10 shows the implementation of the bundle activator. In contrast to Java OSGi frameworks, the bundle developer has to mark the class name as an activator. For this purpose, the provided macro `NOSGI_SETACTIVATOR` creates the needed methods for instantiating and accessing the class at bundle installation time according to Figure 4.

5.2 Providing and Consuming Services

nOSGi supports implementing applications in a service-oriented manner. Therefore, bundles can register OSGi ser-

```
1  #include "Activator.h"
2  #include "trlende.h"
3  #include <string>
4
5  NOSGI_SETACTIVATOR(Activator)
6
7  void Activator::start(BundleContext &bc) {
8    Trl *service = new TrlEnDe();
9    std::string srvName = "TrlEnDe";
10   bc.registerService(srvName, service);
11   ServiceReference *r =
12      bc.getServiceReference("TrlEnFr");
13   if (r) {
14     Trl *enfr = (Trl *)bc.getService(r);
15     enfr->translate("hello");
16   }
17 }
18
19 void Activator::stop(BundleContext &bc) {
20   //code to stop the bundle
21 }
```

Figure 11: `Activator` code registering and consuming an OSGi service

vices at a service registry and consume services offered by other bundles. Figure 11 shows an exemplary bundle activator that first registers an own *dictionary* service that allows translating from English to German (Line 10). Then, it discovers a service that allows translating from English to French (Line 11). The service can be transparently used (Line 15) without having to import the package on module layer that implements the service (see Section 4.3).

6. EVALUATION

We evaluated nOSGi with different setups. In addition, to standard desktop and laptop environments, we measure its behaviour on mobile, resource-limited devices.

Table 12 shows the configuration of devices for our evaluation. The devices labelled with *PC* and *Laptop* run a 32-/64-bit Arch Linux with a fully featured JVM. Our Arch Linux installation features kernel version 2.6.30.1, `glibc` 2.10.1, `gcc` 4.4.0 and `ld` 2.19.20090418. For measuring the behaviour of nOSGi on netbooks we perform our tests on the Asus EeePC PC901 running a 32-bit Ubuntu 8.04 on an additional flash memory card. As a typical mobile device with highly limited resources, we have a Nokia N810 that comes with a 32-bit RISC ARMv6 architecture. Due to the fact that the standard Sun JVM is unavailable on the N810, we equip it with two open source JVMs: *Cacao* [4] supports just-in-time (JIT)-compilation while *JamVM* [19] runs in pure interpreter mode. In order to prove the portability of nOSGi we include in the tests a *Sun Fire* running Solaris 10, an *Alpha* running a Debian Squeeze (both equipped with a Sun JVM) and a *PC* running OpenBSD with OpenJDK.

In the following, we present four performance cases. Each evaluates nOSGi in comparison to two common Java-based OSGi implementations (i.e., Equinox [5] and Concierge [16]). Concierge, however, only implements OSGi R3 which is a much simpler specification than OSGi R4 which is followed by nOSGi and Equinox. As an OSGi framework just dynamically loads the program code into the runtime, it runs application without causing additionally runtime overhead. Thus, we do not execute an application end to end comparison. We rather measure the overhead the respective OSGi platform causes at start-up. Furthermore, we analyse the memory consumption for both, resident set size and virtual memory, and the performance of the respective OSGi registry implementation. Finally, we evaluate the bundle life-cycle time.

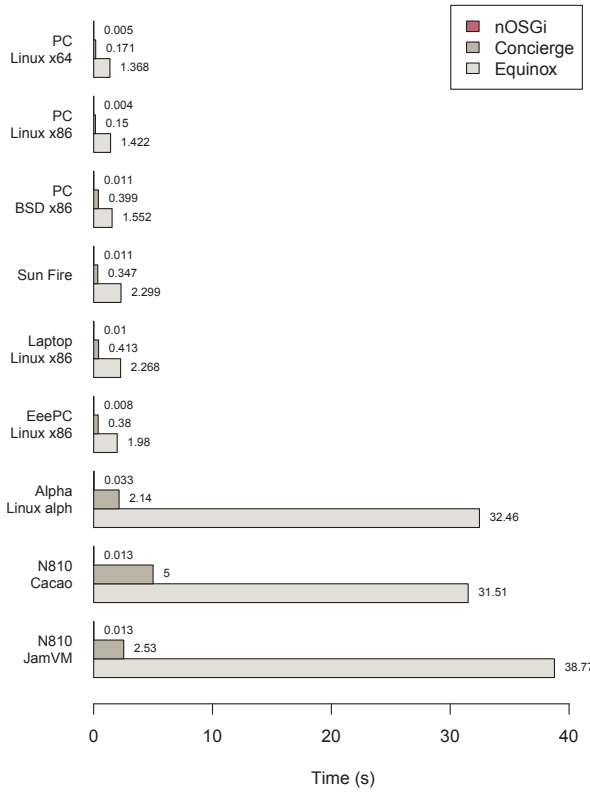

Figure 13: OSGi framework start-up time

6.1 Start-up

To identify the framework overhead, we measure the CPU time for framework start-up (i.e., the time to reach the state when developers can start interacting with bundles). For this purpose, we use the shell command `time` and also include the CPU time to load the runtime environment (i.e. libraries, JVM). As the results must not include the CPU time required for shutting down the framework, we force an abort of the process by using the signal *sigkill* after start-up.

Figure 13 shows the results of the start-up test. The values show the sum of both, CPU time in user space and in kernel space (i.e., system calls). The results show that our nOSGi framework is fastest on all architectures. Its start-up times vary only slightly in comparison to the other implementations. Furthermore, the test unveils that starting Equinox takes longer than Concierge (i.e., 5–16 times longer) and severely longer than nOSGi (i.e., up to 3,200 times longer). nOSGi starts 30–40 times faster than Concierge on standard PC-hardware, around 50 and 70 times faster on EeePC and Alpha, respectively, and 200 and 300 times faster on the N810 with Cacao and JamVM, respectively. The test also shows that for both Java implementations Cacao performs better than JamVM due to JIT compilation. Finally, the huge variance of devices shows that nOSGi runs on POSIX-

Device	CPU	RAM	HDD	OS	JVM
PC Linux x64	Intel Core2Duo 2.4 GHz	6 GB	7200 rpm	Arch Linux (64 bit)	Sun JRE 6_14b08
PC Linux x86	Intel Core2Duo 2.4 GHz	4 GB	7200 rpm	Arch Linux (32 bit)	Sun JRE 6_14b08
PC BSD	Athlon XP 2500+ 1.83 Ghz	1.5 GB	7200 rpm	OpenBSD 4.5 (32 bit)	OpenJDK 1.7.0...b00
Sun Fire	12xUltraSPARC IV+ 1.8 GHz	96 GB	Ultra3 SCSI	Solaris 10 (64 bit)	Sun JRE 1.5.0_18b02
Laptop Linux	Intel Pentium 4M 2.0 GHz	512 MB	5400 rpm	Arch Linux (32 bit)	Sun JRE 6_07b06
EeePC Linux	Intel Atom N270 1.6 GHz	1 GB	Flash	Ubuntu 8.04 (32 bit)	Sun JRE 6
Alpha	Alpha EV5 333 MHz	128 MB	5400 rpm	Debian Squeeze	OpenJDK 1.6.0_6b16
N810 Cacao	ARMv6 400 MHz	128 MB	Flash	Mameo 4.1.3	Cacao 0.99.3
N810 JamVM	ARMv6 400 MHz	128 MB	Flash	Mameo 4.1.3	JamVM 1.5.1

Figure 12: Environment and features of the used devices

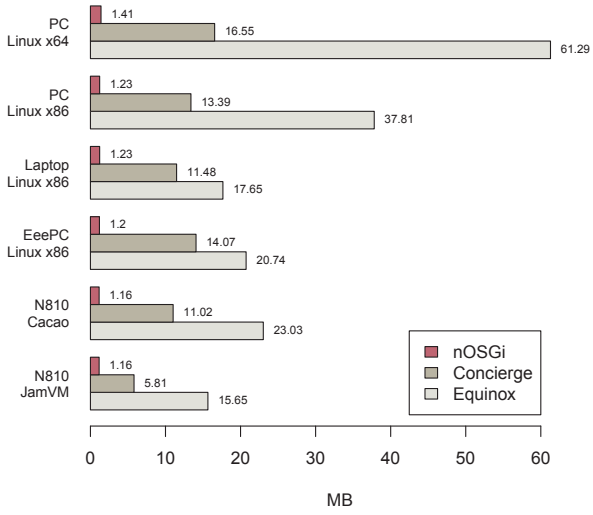

Figure 14: Memory consumption: resident set size

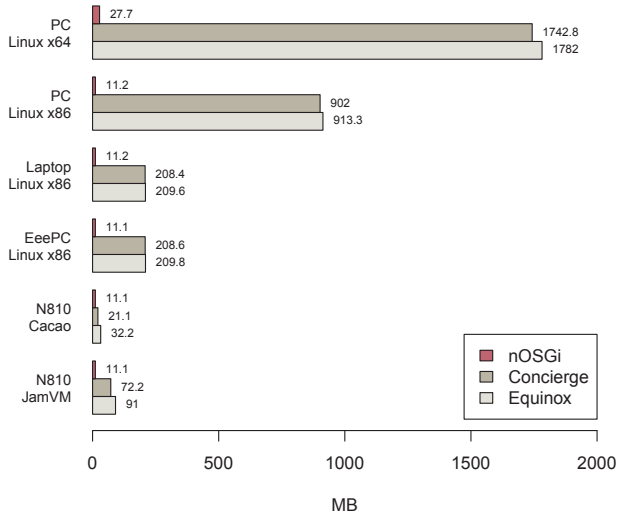

Figure 15: Memory consumption: virtual memory

compliant systems supporting ELF without changes. For the following performance cases, we omit the BSD system, the Sun Fire and the Alpha as these do not provide further insights regarding the evaluation cases.

6.2 Memory Usage

The memory consumption of a process is particularly critical on mobile and embedded devices as such devices are characterised by highly limited memory in comparison to current standard general purpose machines. In the following, we present two evaluations that investigate memory consumption. We distinguish *resident set size* memory consumption and *virtual memory* in use. The resident set size sums up all memory that is actually used by an application. It includes the program data, heap, stack and also the memory used by shared libraries. The virtual memory metrics considers the amount of memory requested from the operating system, no matter whether it is used or not. For both tests we start the framework implementations with an OSGi console bundle. Then, we measure the current memory usage.

6.2.1 Resident Set Size

Figure 14 shows the resident set size of the evaluated framework implementations. The memory usage of nOSGi is around 1.2 MB for all architectures. Thereof, 0.9 MB are due to shared C, C++ and linker libraries. Yet, these are in general also used by other processes.

On the systems with an activated JIT compiler (i.e., Sun JRE and Cacao), the Concierge implementation uses between 11 and 14.1 MB physical memory while JamVM consumes 5.9 MB (this is about half of the memory usage with JIT compilation). Equinox requires 15.6–23 MB. With 37.8 MB, the memory usage on the test system with two CPU cores is almost doubled compared to the standard PC device and with 60 MB almost quadrupled compared to the standard PC 64-bit architecture. This high increase in memory consumption is due to a change of memory management within the Sun JVMs. There, systems with at least two CPU cores and at least 2 GB RAM are automatically classified as server-class machines. If the JVM is started with standard settings (i.e., command line parameter `-client`) the memory usage is again about 20 MB.

Overall, the memory consumption of the native nOSGi implementation is about 5–10 times lower than Concierge and 20 times lower than Equinox even if the server-class machine behaviour of the Sun JVM is ignored.

6.2.2 Virtual Memory

For all Java systems, the size of the address space reflects the amount of memory the JVM requests from the operating system. Yet, it does not necessarily show how much memory the framework implementation requires. For Sun JVMs, the memory footprint is about 210 MB on the single core systems, about 900 MB on the multi-core 32-bit system and 1,700 MB on the 64-bit system (see Figure 15). For the latter two, the server-settings were switched on. The

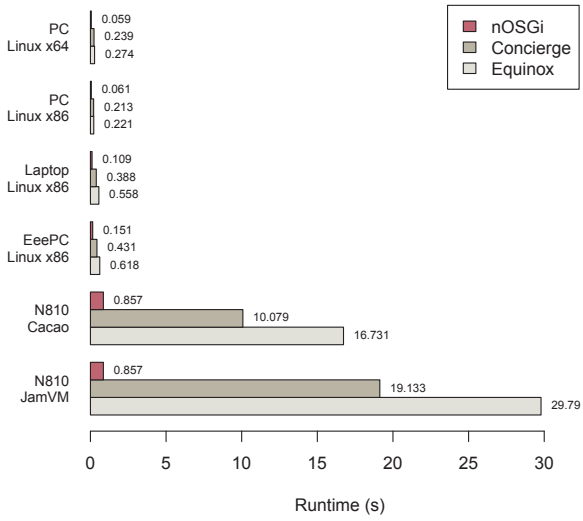

Figure 16: Knopflerfish registry performance test results

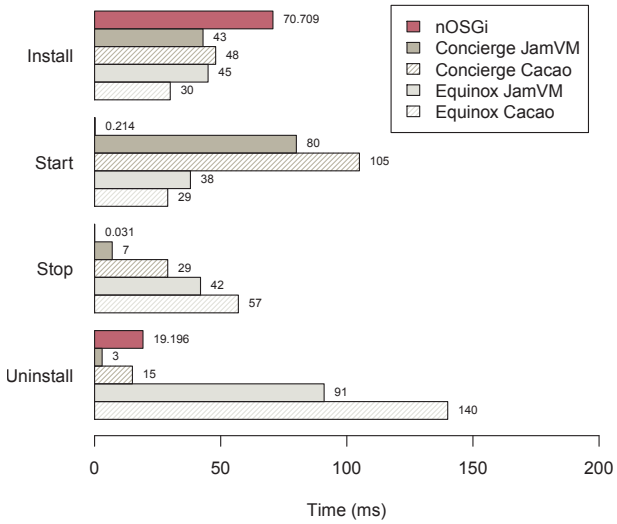

Figure 17: Bundle life cycle performance: duration to install, start, stop and uninstall a bundle on the Nokia N810

high memory consumption is a result of the JVM strategy to request memory block-wise (about 50 MB per block). Consequently, we limited heap memory to 2 MB (command-line parameter `-Xmx2m`). This leads to a decreased address space size for Java OSGi implementations (i.e., 140 and 160 MB for Concierge and Equinox, respectively).

On the Nokia N810, the virtual machines use a small address space even with the default settings. For JamVM we obtain 72.2 and 91.0 MB for Cacao 21.1 MB and 32.2 MB for Concierge and Equinox, respectively. The native nOSGi implementation uses about 11 MB of virtual memory on all considered platforms. Thereof, 8 MB are due to the console bundle thread. Figure 15 shows that the memory consumption of our native OSGi implementation is lower than all Java implementations. Even the N810 running Concierge on Cacao requires twice the memory of nOSGi.

6.3 Knopflerfish Registry Performance Test

In order to test how the system scales with typical applications, we use the *Knopflerfish regression test suite* [23]. These tests are offered as OSGi bundles and verify a certain part of the framework. The most frequently used functionality is service handling. The performance during registering, modifying and deregistering services is a severe issue for application performance. To measure this aspect, we chose the *registry performance test* from the Knopflerfish test suite. During the test, 100 listeners with corresponding filters and 1000 services are registered, modified and unregistered. To run this test on nOSGi, we ported it from Java to C++.

Figure 16 shows the results. The differences between nOSGi and the Java implementations are already significant on powerful machines (i.e., PC, Laptop, EeePC). Compared to Equinox, nOSGi is about four times faster. Compared to Concierge, the native implementation still shows a performance advantage of about three times. On the Nokia N810 (representing a typical mobile device), the performance differences are even more striking: with the native nOSGi framework, the test runs about 20 times faster than with Equinox on Cacao. In comparison to Concierge on Cacao, we measured an increase of about factor 12. With JamVM these factors even double due to the missing JIT compilation.

6.4 Life Cycle

This test case evaluates the performance of bundle life cycle operations. We implemented a reference bundle that registers a bundle listener at bundle start-up and deregisters this listener when the bundle is stopped. Through this, the bundle passes all life cycle states and allows measuring the duration of installing, starting, stopping and uninstalling the bundle.

Figure 17 shows the results for the Nokia N810. There, nOSGi is slowest with respect to installing bundles. It takes about 140% (i.e., 40 ms) longer than with the fastest Java implementation. This results from the fact that the bundle content is unpacked during installation while all other frameworks do this at start-up. Furthermore, wiring requires patching the dependencies into the shared libraries. Figure 18 shows the results for the EeePC (PC and Laptop settings are omitted as they are comparable to the EeePC results). There, nOSGi outperforms Equinox while Concierge is faster again (both running with the Sun JVM).

During the start-up phase, nOSGi profits from the preliminary work during installation. Thus, bundle start-up takes only 0.2 ms on the N810 as it only requires setting up a bundle context and executing the activator. By contrast, the Java implementations load classes from the JAR file during the start phase. This also implies extracting the archive file. Thus, starting a bundle takes 30–80 ms on the N810. The behaviour on the EeePC is comparable.

Stopping a bundle in nOSGi covers only the call of the `stop` method of the activator. Within this method, the reference bundle unsubscribes the bundle listener registered before. This takes about 0.03 ms on the N810. For Java OSGi frameworks, the time for the stop process ranges from 7 ms for Concierge on JamVM to 57 ms for Equinox on a Cacao VM (both on the N810). Again, the behaviour on the EeePC is comparable.

The fastest implementation for the uninstallation of bun-

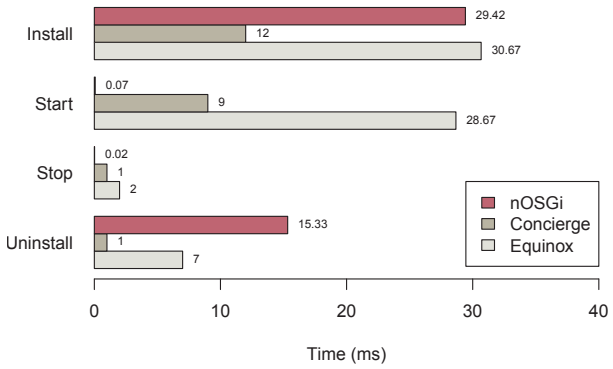

Figure 18: Bundle life cycle performance: duration to install, start, stop and uninstall a bundle on the Asus EeePC

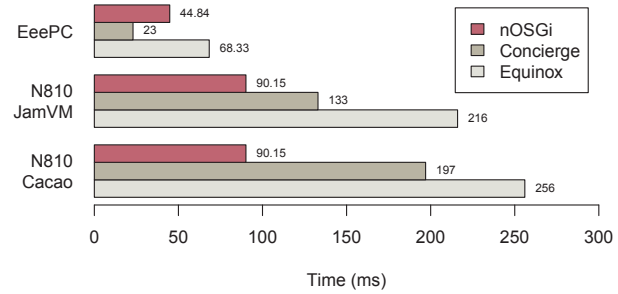

Figure 19: Overall life cycle performance results

dles is Concierge on JamVM. Yet, Concierge only removes the internal data structures of the bundle without actually deleting the bundle from the working directory. Even though nOSGi does not use such optimisations, it is almost as quick as Concierge on Cacao and outperforms Equinox by a factor of 4.5 on JamVM and 7 on Cacao (both on the N810). On the EeePC, Concierge and Equinox outperform nOSGi.

Figure 19 shows the overall time of the OSGi life cycle for all tested frameworks on the N810 and the EeePC by adding up the partial results of installing, starting, stopping and uninstalling the reference bundle. On the N810, the native implementation offers the best performance while Concierge provides the best performance on the EeePC. Yet, Concierge does not delete the bundle from the bundle directory. If uninstallation is left out, nOSGi provides the best performance on all devices.

7. RELATED WORK

Over the last years, several open source and commercial OSGi implementations were developed. Those platforms, such as Eclipse Equinox [5], Concierge [16] and Knopflerfish [23] exclusively target Java. With exception of Concierge none of them has been designed with mobile and embedded devices in mind although OSGi was originally designed for such devices. Thus, due to Java intermediate code, all of the Java frameworks have serious performance penalties compared to nOSGi. In addition, they require a JVM to run code. This is a severe issue if there is no JVM available for a specific architecture and system.

The *Celix* proposal [3] describes the idea to provide a module/component-based software solution for embedded platforms. The goal is to provide a basic implementation of some OSGi-like features to C. Compared to nOSGi there will be more differences to the OSGi specification because C is a procedural language and lacks object-orientation. The current implementation supports dynamic loading of modules and basic service layer interactions between modules. Code sharing on module layer level, however, is not supported (e.g. `export-package` and `import-package`).

Escoffier et al. [6] present an OSGi-like service platform on the basis of the Microsoft .NET framework [12]. They evaluate both ways that .NET allows dynamically loading code, i.e., on assembly-level and on application-level. Assemblies contain multiple classes and types. They are the unit at which .NET deploys and reuses code. Assemblies are only allowed to communicate via method invocation if they reside within the same application domain and cannot be individually removed from an application domain once they have been loaded. Yet, unloading of modules is possible if they are loaded on application domain-level but communication between such modules can only be implemented via inter-process communication. This is expensive and offers only weak performance. nOSGi provides a better way to allow method invocation between previously isolated entities. Additionally, it allows unloading them once they are not used anymore.

Open-Plug is a commercial component framework that is widely used by mobile phone manufacturers to easily update and maintain the firmware on such devices [14]. It allows loading binary files that have been created by the Open-Plug build-chain tools. Dependencies between components are recognised and resolved (if necessary by loading other components over the network). The build-chain as well as the life-cycle management of Open-Plug is quite similar to nOSGi. Yet, there is a difference in the way components are managed on the target device. While Open-Plug uses a proprietary technology that interprets code, nOSGi uses open standards and executes native binary code. Moreover, nOSGi uses standard method invocation for inter-component communication, whereas Open-Plug uses a message bus. Finally, Open-Plug is not compliant to the OSGi specification.

Linux kernel modules [17] can be considered related work providing a basic component framework. There, system drivers can be added to a running system. For this purpose, dependencies between modules are automatically resolved and missing modules are loaded if they are available. Similar to OSGi activators, kernel modules can provide functions for initialisation and deinitialisation. In contrast to nOSGi, kernel modules are not isolated from each other as they share a single namespace. Thus, the programmer has to follow naming conventions to avoid namespace pollution. Thus, it is impossible to load different versions of the same kernel module for multiple times if they contain the same symbols.

8. CONCLUSION AND FUTURE WORK

In this paper, we presented nOSGi, a native implementation of the OSGi specification in C++. After identifying the gaps between the requirements of the OSGi specification and the standard C++ language features, we introduced nOSGi that purely relies on POSIX features to close these gaps. nOSGi implements bundles as ZIP files that contain shared object

files (each representing a module). Dynamic loading of modules is implemented with standard dlopen mechanisms. Dependencies between modules are realised as dependencies between shared objects. They are injected during bundle installation on the basis of import-export relationships. Then, when a shared object is loaded its dependencies are automatically resolved. This enables sharing of modules between bundles while other bundles are isolated. Furthermore, we showed that developing C++ applications with nOSGi is a reasonable task, especially for OSGi developers. The evaluation of nOSGi with a multitude of devices shows that nOSGi runs on POSIX-compatible systems supporting ELF without changes. Moreover, in comparison to standard Java OSGi frameworks, nOSGi provides significant improvements regarding performance and memory consumption.

In future work, we will investigate support for POSIX-compliant systems that do not support the ELF file format, such as Mac OS and Windows. As executable file format, Mac OS uses *Mach object file format* (Mach-O) [2] while Windows uses the *Portable Executable* [13] format. Both formats specify library dependencies with plain text. Thus, we suppose that it is possible to transfer the presented nOSGi approach to Mac OS and Windows with reasonable efforts. Finally, we plan to evaluate the usage of nOSGi in distributed computing platforms such as our component-based cloud platform COSCA [10].

9. REFERENCES

[1] Apache Software Foundation. Apache Felix. http://felix.apache.org/.

[2] Apple Inc. Mac OS X ABI Mach-O file format reference. Mac OS X Reference Library, February 2009.

[3] Alexander Broekhuis. Celix proposal. http://wiki.apache.org/incubator/CelixProposal.

[4] Cacaovm.org. cacaovm. http://www.cacaovm.org/.

[5] Eclipse Foundation. Equinox. http://www.eclipse.org/equinox/.

[6] C. Escoffier, D. Donsez, and R. S. Hall. Developing an OSGi-like service platform for .NET. *CCNC '06: 3rd IEEE Consumer Communications and Networking Conference*, pages 213–217, January 2006.

[7] Xiaohui Gu, Alan Messer, Ira Greenberg, Dejan Milojicic, and Klara Nahrstedt. Adaptive offloading for pervasive computing. *IEEE Pervasive Computing*, 3:66–73, July 2004.

[8] T. Howes. A string representation of LDAP search filters. IETF RFC 1960, June 1996.

[9] ISO/IEC. Standard for programming language C++. ISO/IEC 14882:2003, October 2003.

[10] Steffen Kächele, Jörg Domaschka, and Franz J. Hauck. COSCA: an easy-to-use component-based PaaS cloud system for common applications. In *Proceedings of the First International Workshop on Cloud Computing Platforms*, CloudCP '11, pages 4:1–4:6, 2011.

[11] Peter Kriens. Minimal OSGi systems. http://www.osgi.org/blog/2010/10/minimal-osgi-systems.html.

[12] Microsoft Corporation. .NET framework developer center. http://msdn.microsoft.com/netframework/.

[13] Microsoft Corporation. Microsoft portable executable and common object file format specification. Windows Hardware Developer Central, March 2008.

[14] Open-Plug. Component based software development for mobile devices. *Whitepaper*, February 2008.

[15] OSGi Alliance. *OSGi Service Plattform Core Specification, Version 4.0.1*, August 2005.

[16] J. S. Rellermeyer and G. Alonso. Concierge: a service platform for resource-constrained devices. *ACM SIGOPS Operating Systems Review*, 41(3):245–258, March 2007.

[17] P. J. Salzman. The Linux kernel module programming guide. Linux Documentation Project, May 2007.

[18] W. Schulte and Y. V. Natis. 'Service oriented' architectures, part 1. SSA research note SPA-401-068, Gartner, April 1996.

[19] Sourceforge.net. JamVM – a compact Java virtual machine. http://jamvm.sourceforge.net/.

[20] Richard M. Stallman, Roland McGrath, and Paul D. Smith. *GNU Make – A Program for Directing Recompilation*. Free Software Foundation, July 2010.

[21] Sun Microsystems. Connected device configuration 1.1.2. JSR 218, August 2006.

[22] The IEEE and The Open Group. IEEE std 1003.1-2008, 2008.

[23] The Knopflerfish Project. Knopflerfish OSGi. http://www.knopflerfish.org/.

[24] M. Weiser. The computer for the 21st Century. *Scientific American*, 265(3):66–75, February 1991.