Concierge

A service platform for resource-constrained devices

Rellermeyer, Jan S.; Alonso, Gustavo

DOI
10.1145/1272996.1273022

Publication date
2007

Document Version
Accepted author manuscript

Published in
Operating Systems Review - Proceedings of the 2007 EuroSys Conference

Concierge: A Service Platform for Resource-Constrained Devices *

Jan S. Rellermeyer Gustavo Alonso
Department of Computer Science
ETH Zurich
8092 Zurich, Switzerland
{rellermeyer, alonso}@inf.ethz.ch

ABSTRACT

As mobile and embedded devices become widespread, the management and configuration of the software in the devices is increasingly turning into a critical issue. OSGi is a business standard for the life cycle management of Java software components. It is based on a service oriented architecture where functional units are decoupled and components can be managed independently of each other. However, the focus continuously shifts from the originally intended area of small and embedded devices towards large-scaled enterprise systems. As a result, implementations of the OSGi framework are increasingly becoming more heavyweight and less suitable for smaller computing devices. In this paper, we describe the experience gathered during the design of Concierge, an implementation of the OSGi specification tailored to resource-constrained devices. Comprehensive benchmarks show that Concierge performs better than existing implementations and consumes less resources.

Categories and Subject Descriptors

D.3.3 [**Programming Language**]: Language Constructs and Features—*Frameworks*; D.2.8 [**Software Engineering**]: Metrics—*Performance measures*; H.4 [**Information Systems Applications**]: Miscellaneous

General Terms

Design, Management, Measurement, Performance

Keywords

Resource-constrained Devices, Service Oriented Architecture, OSGi, Concierge, Average Bundle

*The work presented in this paper was supported (in part) by the National Competence Center in Research on Mobile Information and Communication Systems (NCCR-MICS), a center supported by the Swiss National Science Foundation under grant number 5005-67322.

1. INTRODUCTION

In the last few years a clear trend has developed towards the widespread use of mobile and embedded devices. As a result, the problem of software life cycle management on these devices has become more acute. This problem is compounded by the special characteristics of ubiquitous computing. For instance, in cars, the life time of the hardware is far longer than the life cycle of a single software release. Mobility also requires constant adaptation through addition and replacement of software modules. Furthermore, the user experience is very important, and it can not be expected that the user manually configures all the devices. Until now, these tasks are performed ad-hoc and largely based on proprietary systems. In response to the demand for a standardized software management platform, the Open Service Gateway Initiative (OSGi) [24] has released an open business standard for the management of dynamic modules in Java. The OSGi platform is a service-oriented architecture that allows to build applications out of loosely coupled software modules to facilitate the addition, removal, and update of each module independently.

1.1 Software management systems

The originally intended area of OSGi was the management of home communication gateways like multimedia systems or set-top boxes for digital television. The first release of the OSGi specifications in 1999 had a strong focus on resource-constrained devices. OSGi has been very successful as it addresses an increasingly important problem that is not exclusive to resource-constrained devices. Today, OSGi is widely used in large applications that need to be extensible at runtime, and to facilitate software management and updates without taking an application down. One example is Equinox [8], an OSGi implementation that forms the extensible plugin system of the Rich Client Platform [9] and the Eclipse IDE. Projects like the J2EE container Jonas [7] or the Apache Directory Server [2] combine OSGi with enterprise server architectures. Additional implementations have been provided by several members of the OSGi Alliance, e.g., Sun, IBM, Siemens, or Samsung. Some of them, for instance the IBM Service Management Framework (SMF) [12] and the ProSyst mBedded Server [26], are available for developers and end-users as commercial products. Currently, two open source implementations of the OSGi Specification Release 3 (R3) exist, Oscar [10, 23] by Richard S. Hall from University of Grenoble and Knopflerfish [17] by Gatespace Telematics. Oscar has been discontinued and succeeded by

the Apache Felix project [3], which implements the OSGi R4 specifications. Eclipse Equinox also implements R4. All these implementations focus on server-side applications, are tailored to full-sized computers, and tend to be very liberal with the use of resources.

There are several alternative proposals to implement software management functionality for small devices. One of them is MIDlets, which follows an orthogonal approach to OSGi. MIDlets are part of the CLDC MIDP Profile [14] and are used in Java-enabled cell-phones. MIDlets provide some basic life cycle management but support only isolated components and do not allow interaction across MIDlets. Hence, they are inadequate for larger software applications. Another example is the Multimedia Home Platform (MHP) [21] and its US counterpart, the Open Cable Application Platform (OCAP) [22]. These are de-facto standards for set-top boxes for digital television. The software stack is based on a J2ME CDC Foundation Profile [15, 16] and allows network operators to distribute content and Java applications in the form of Xlets to the user's Digital Video Broadcasting (DVB) terminal equipment. Technically, Xlets provide a similar basic life cycle interface as MIDlets. Although MHP has the concept of services, this denotes services of the terminal device and cannot be dynamically extended. The only way for Xlets to interact is through Remote Method Invocation (RMI) calls outside of the control of the management platform.

There is no doubt that OSGi offers a more complete platform than either MIDlets or Xlets. Unfortunately, existing OSGi implementations are hardly suitable for devices other than full scale computers. Nevertheless, the functionality of OSGi is a key element for making mobile and ubiquitous computing a reality. The question is whether it is possible to implement the OSGi functionality in a platform adequate for smaller devices without loosing any of the advantages of the OSGi idea.

1.2 Contributions

In this paper, we answer this question by describing the experience gathered during the development of Concierge, a complete implementation of the OSGi framework. Concierge has been optimized for small[1] and embedded devices. Yet, it provides all the functionality available on the OSGi R3 specification. An important contribution of the architecture of Concierge is that it has a considerably smaller file and memory footprint than existing implementations. For instance, Oscar and Knopflerfish have a file footprint of over 200 kBytes, while Concierge takes only 75 kBytes. Another important contribution of the design of Concierge is that, in despite of being far more compact than existing implementations, the evaluations included in the paper show that it is more efficient and has better performance than larger systems. On a notebook, Concierge provides 50% performance improvement over Knopflerfish and over 60% performance improvements over IBM's SMF on the Knopflerfish Regression Test benchmarks. In smaller devices, such as the PDA Sharp Zaurus with the Sun cvm, Concierge is roughly ten times faster than Oscar. The performance gains are important for the user's experience point of view as well as in terms of more efficient energy consumption. In the paper, we provide a detailed analysis of optimization strategies for Java on small VMs. We then show how these optimizations have been used in the design of Concierge. An important aspect of these optimizations is that they are not restricted to Concierge but apply to any Java software on small and embedded devices. Statistical measurements of the behavior of existing OSGi applications are used to complete the picture and assess the concrete influences of each optimization. As a further contribution, from these measurements, we derive the notion of the *Average Bundle* which characterizes the behavior of a typical OSGi software module.

[1]Throughout this paper, the term "small" device is used as a synonym for "resource-constrained".

2. OSGI

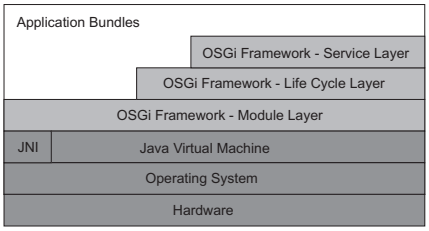

Figure 1: OSGi Architecture

2.1 Overview

OSGi is a platform for managing software modules, which in OSGi are called *Bundles*. What OSGi provides is support for adding, removing, and replacing bundles at runtime while maintaining the relations and dependencies between the bundles. OSGi also provides a service-oriented architecture where bundles can publish and use services through an event-based publish-subscribe system. An OSGi implementation has to provide all the necessary infrastructure to manage bundles, maintain consistency across import/export dependencies of the bundles, and implement a services and event infrastructure including service registry and forwarding mechanisms, as well as the publish-subscribe system.

The advantage of using OSGi is that the always complex problem of module control and consistency in larger applications is done automatically. The facilities provided by OSGi also make it possible to dynamically replace parts of an application that has been built as a collection of bundles without interrupting the application. This is what makes OSGi so attractive on the server side and is one of the reasons why the focus of most implementations tends to be larger installations rather than small devices (the other reason being that until recently there was not enough critical mass around the efforts on small devices to justify focusing on them). With OSGi, it becomes possible to use service-oriented architecture principles to build applications out of independent, loosely-coupled software modules. Through the service-oriented approach, OSGi clearly separates implementation and interface, thereby making it possible to change the implementation of a service at runtime.

2.2 Bundles

Bundles in OSGi are the concrete realization of the concept of software modules. From a technical point of view, they are ordinary JAR files containing class files and other resources and having additional OSGi-specific attributes in

the JAR manifest. The management of bundles requires a bundle registry where the system can keep track of what bundles are around and what their status is. When a bundle is inserted into the system, it is first persistently stored. Then, the system builds a class loader for the bundle. The point when the bundle is resolved depends on the concrete OSGi implementation. Resolving the bundle involves checking its internal class path, fulfilling the import declarations, and resolving the bundle's activator if it has one. After being resolved, the bundle is ready to be started when invoked. Removing bundles is more involved. If the bundle is active, it has to be stopped. All registered services have to be unregistered and listeners have to be deactivated. Finally, the bundle's class loader is disposed of to remove knowledge of the class types from the system (at which point is as if the bundle would have never been there).

What makes OSGi interesting is the support for composing bundles by managing the dependencies between them. There are two different types of dependencies between bundles. The first one involves the sharing of Java packages between bundles so that one bundle exports the package and one or more other bundles import it. This is a strong dependency, since the presence of the package is required for the importing bundles to work. The second form of dependency is intended to support loosely-coupled interactions so that bundles can be installed or uninstalled at any time. These weaker dependencies are implemented through services.

2.3 Services

A service is an object with one or more interfaces plus an optional set of properties. Services are registered by bundles. The OSGi framework maintains a service registry where bundles can register and provide services. Services are by design not required for the resolving and the starting of a bundle. Every bundle can request a service from the registry and has to take own provisions for the case that the service is currently not available. Service consumers are only dependent on the interface but not on the concrete implementations, i.e., they do not depend on the service object. It is thus possible to replace the service implementation at runtime. The goal is to allow bundles to interact in a loosely-coupled manner. For instance, a command-line shell bundle listens to application bundles that register services. These application services tell the shell how to interact with these applications. In this way, the applications can extend the functionality of the shell at runtime without creating a strong dependency between them. Applications can come and go while the shell is always able to interact with all applications that are currently running.

2.4 Events

Information about changes in the state of bundles and/or services is distributed across the system via events. These events are managed through a *Publish-Subscribe* infrastructure. A bundle can register a `ServiceListener` if a certain service is not available, and wait until another bundle registers this service. When the services appears, an event is generated and forwarded to subscribed listeners. Similarly, `BundleListeners` get informed whenever the state of a bundle changes and `FrameworkEvents` are fired in case of changes in the framework's own internal state. Bundles often communicate using the more sophisticated *Whiteboard Pattern* [25]. Instead of requiring all bundles to subscribe to the event source, bundles can register the listeners as services, and the source bundle calls all listeners that have an entry in the service registry. This is more flexible since it decouples event source and listener.

2.5 Framework

An OSGi platform has a layered architecture (Figure 1). The lowest three layers are not really part of the system but are very important in determining how it works, especially the JVM layer. The lowest layer of a running OSGi framework is the hardware itself, that can range from a standard x86-based PC to mobile devices down to highly specialized embedded systems. For the experiments in this paper, we use a wide variety of hardware and software platforms. The complete list and their characteristics can be found in the Appendix. The next layer is the operating system. In our tests, we have used several flavors of Linux, Windows XP, Windows PocketPC, and SymbianOS. Which one has been used in which device is also summarized in the Appendix.

The Java Virtual Machine layer deserves special attention. The problem is that not all JVMs are the same and this is particularly true in resource-constrained devices. On the one hand, the evolution of Java SE did not proceed equally on all platforms. Whereas desktop and server range machines typically have Java 1.5 and are heading towards Java 1.6, small and embedded devices often still run on 1.2 or 1.3 compliant VMs. The differences in terms of features are remarkable, for instance, the total number of classes and interfaces in Java 1.2 is 1520, while it is over 12000 in Java 1.5 with not only new classes but also with important changes to very basic and frequently used classes. On the other hand, J2ME profiles feature by design only a subset of standard Java. To overcome the pitfalls of compatibility, OSGi has specified two different execution environments. The first is *OSGi/Minimum-1.0*, the minimal subset of standard Java that is required to have a fully functional OSGi framework. The second execution environment is the *CDC-1.0/Foundation*, which is derived from the J2ME CDC Foundation Profile and defines the intersection between the Minimum execution environment and the Foundation Profile. J2ME CLDC profiles are not valid execution environments and the underlying KVM [19] lacks basic concepts like user-defined class loading that are crucial for OSGi. To be fully compatible, bundles should not rely on any class beyond the minimum execution environment.

The module layer handles the class loading of OSGi bundles and the import and export of packages. Java class loaders consume arrays of bytes representing the Java byte code of a class. For each loaded class, a `Class` Object and a supporting data structure in the method area is created that forms the type of the class. In OSGi, every bundle is loaded by a separate class loader. Package imports therefore have to be delegated from the importing to the exporting class loader. The purpose of the module layer is to handle the class loading and the delegations between bundles.

The life cycle layer provides an API for managing the life cycle of the installed bundles. Since update and removal of running bundles has implications on other bundles that have imported packages from the affected bundles, the life cycle layer has to ensure consistency. For example, all bundles that want to import a certain package have to get the same

package, even if more than one bundle offer to export this particular package. The handling of dependencies is also performed by the life cycle layer. Every bundle can provide a class that implements the `BundleActivator` interface with a start and a stop method. Whenever one of these two operations are performed on the bundle, the corresponding method of the `BundleActivator` is called. In this sense, the start method of the `BundleActivator` replaces the `main` method of traditional Java applications.

The service layer contains the service registry. Bundles can register their service objects under the name of one or more interfaces and together with a set of properties. The properties are described by key/value pairs. Bundles interested in services can retrieve `ServiceReferences` for matching services, optionally by providing an RFC 1960 LDAP filter [11] that is matched against the registered properties. The service reference gives access to the full set of properties of the service and allows the bundle to fetch the actual service object from the registry.

3. THE CONCIERGE DESIGN

Implementing the OSGi specification for resource constrained devices involves many different trade-offs. First, it is important to save CPU cycles, especially on mobile devices. Second, memory has to be used carefully and treated as a very scarce resource. The challenge here is that there is no set of design rules on how to optimize code for small devices. To make matters worse, smaller JVMs are far more heterogeneous and less optimized than their larger counterparts. Hence, any design choice has to be tested on different hardware platforms and different JVMs. Before describing the implementation of the different OSGi layers in Concierge, in this section we discuss general design principles behind the Concierge architecture.

3.1 Consistent behavior across devices

The Java Virtual Machine Specification [20] defines the general behavior of VM implementations but does not enforce particular implementation rules. As long as a system behaves to the outside world like a Java VM, internally almost everything is possible. Even well-known techniques like method tables or garbage collection are not compulsory. As [28] accentuates, it is only mandatory to have some kind of memory management. Therefore, internally, JVMs widely differ. On full-sized machines, this does not matter that much. First, because not many different Java VM implementations are widely used. Second, all these VMs have been optimized to roughly equivalent levels of performance. On small devices, however, none of this applies. There are many different VMs and they are optimized for size rather than performance. A typical example is just-in-time compilation often not available in small VMs. In addition, optimizations tend to be hardware-specific, and the behavior of a VM can radically change when it runs on, e.g., a PDA and a mobile phone. Thus, Concierge has been designed without taking advantage of specific platform characteristics. It uses only generic optimizations based on principles that apply to all potential hardware and software platforms. A key challenge in the design of Concierge is to achieve, as much as it is possible, a consistent behavior across platforms. This is crucial when developing applications that must run on different devices.

3.2 Code optimizations

One of the key problems when optimizing performance is how the code is organized. In this context, the `final` modifier is a still underestimated element in Java. Besides the software-engineering and code-style aspects of the `final` keyword, here we are interested in its performance implications. Generally, final methods use static invocation instead of virtual method dispatching, which is considerably slower. Static and private methods are implicitly final and do not have to be explicitly marked as such. Just-in-time compilers take advantage of final methods to, e.g., inline them. Newer virtual machines can detect at runtime if a method can be treated as final, small virtual machines are unlikely to have these sophisticated optimizations. In the context of resource-constrained devices, it is advisable to explicitly use the final keyword if possible and preferably mark whole classes as final, if they are not explicitly designed for inheritance.

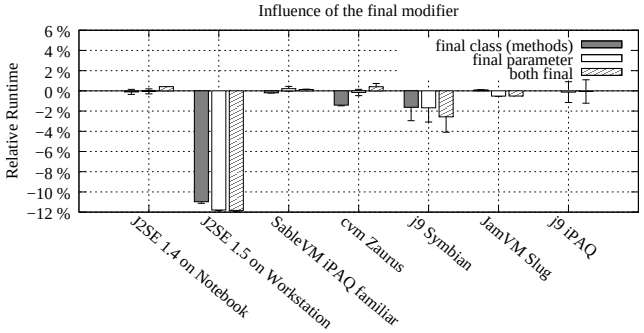

Figure 2: Influence of the final modifier

The impact on performance between using and not using `final` is shown in Figure 2. In this experiment, the average execution time of methods with different arguments are measured when invoked as a normal method, with the method as final, with the parameters as final, and with both the method and the parameters as final. Figure 2 shows the relative performance gain with respect to the execution time of the normal methods. The experiments are run on seven different platforms, from a mobile phone to a workstation. In most cases, a small performance benefit can be observed. Surprisingly, the biggest gains are for Sun Java 1.5 (the workstation) which points out to a problem in how `final` is treated in Java 1.5, since this behavior is not observed in 1.4 on the notebook. As it will happen with several of the measurements discussed in this section, the gains are apparently small. This, however, is deceiving. First, the tests are done on very simplified code so as to be able to determine precisely the gain per method. In real applications, these gains have an important cumulative effect, not only by themselves but also when taking into consideration all the aspects discussed in this section. Second, often these effects result in radically different behaviors across platforms. To ensure consistent behavior, these effects must be taken into account so that applications built on top of Concierge can have a behavior that is predictable across different devices.

3.3 Class structure

The importance of a method being final or not becomes more clear when this is considered in relation to whether the

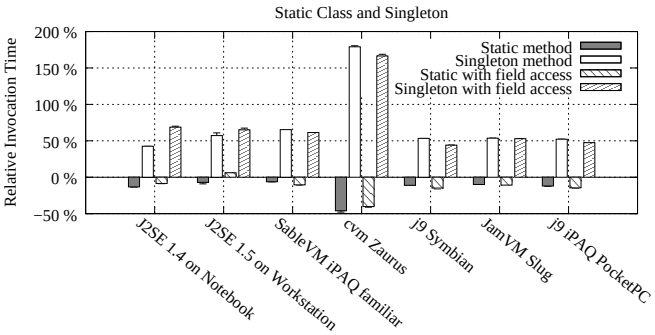

Figure 3: Static and Singleton Methods

class is static or not. Static classes are implicitly final. In OSGi, there is a central *framework class* that sits in the middle of all performance-critical paths. This is because bundles are constantly talking to this class through indirection (e.g., bundle context, service reference, event listeners, etc.). This class is by nature a singleton entity and the vast majority of existing systems implement the framework class as some flavor of the singleton pattern. Again, in large installations, this might not play a major role. In small devices, it is a huge bottleneck.

When implemented as a true singleton, prior to every method call, the singleton getter method has to be called to receive the instance. Sometimes, the singleton pattern is hidden. Instead of calling the singleton getter method, the singleton class passes a reference to itself whenever it creates a class that requires access to it. In this case, the overhead is in memory consumption since every class has to have a reference to the singleton class. Using the singleton pattern still has the advantage that accessing the fields of the class is faster than if the fields were to be made static. Nevertheless, the overhead of getting the instance dominates. This is shown in Figure 3. The Figure shows, for seven different hardware platforms, the relative performance of calling a method of a static class, calling a method of a singleton class, calling a static method with field access, and calling a singleton method with field access. As baseline, we use the cost of calling a normal class method that is neither static nor singleton. In all cases, the singleton method results in significant overhead, close to 50% in most cases and over 150% on the Sharp Zaurus with cvm. The same can be said for singleton method invocation with field access. This overhead has to be paid for every access to the framework class, which, as pointed out, happens very frequently in OSGi.

The conclusions from this experiments are obvious. In Concierge, unlike all other OSGi implementations we are aware of, the framework class is implemented as a static class. This has the drawback that field accesses are more expensive than for singleton fields but the overall performance gains make this an acceptable trade-off. It also has the drawback that static classes cannot be subtyped and extended. But since the framework itself is a monolithic and closed entity, this is not an issue. Furthermore, one can extrapolate from the results of the singleton pattern that common design pattern which increase the number of method calls can severely slow down the execution of code on small devices. Most often, this effect is underestimated.

3.4 Avoiding Indirection

The program fragmentation that results from using object-oriented programming languages is typically hidden behind powerful hardware and very advanced VMs. These resources are not available in small devices. To quantify the impact of code fragmentation, we have analyzed the execution time of a local field access, the use of a local getter method, a field access on an external class, and a getter method on an external class. The measured time for 1.000.000 invocations are shown in Figure 4. These results are interesting, because in the notebook and the workstation, the overhead is small and almost independent of what mechanism is used. This is also true for some of the small platforms that have J9, a more optimized small JVM. For the other devices, the difference is very substantial and can reach a factor of 6 for every field access, even if the access is local.

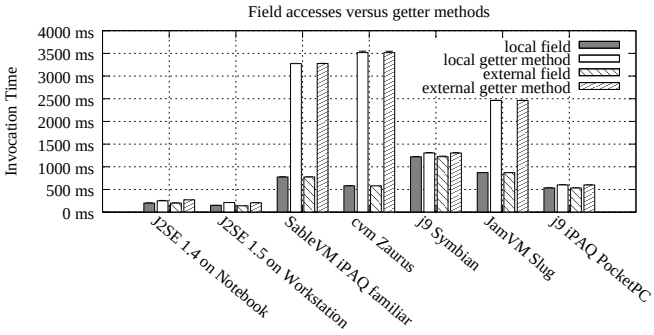

Figure 4: Invocation time of field access

Frameworks in general, and OSGi in particular, tend to be implemented in an extensively object oriented way. For instance, Knopflerfish has 51 classes plus 31 inner classes and Oscar has 79 classes plus 39 inner classes. To make sure that Concierge has optimal behavior on all platforms, even those that are less optimized, Concierge has been designed in a total of 6 classes plus 12 inner classes, all of them final. The structure of the core classes is shown in Figure 5. The experiment also shows that getter methods should be avoided in resource-constrained devices. Generally, the costs of method calls can have a vital impact on the runtime of an application. If possible, code hot spots should not be divided into many methods, since not all platforms are able to compensate the costs of method calls using the corresponding optimizations. Hence, in Concierge, no getter and setter methods are used. The purpose of these accessor methods is usually to provide an abstraction from the data fields of the class and additionally have the possibility to perform sanity checks on access. However, this is only necessary if external classes can access the data. In an OSGi

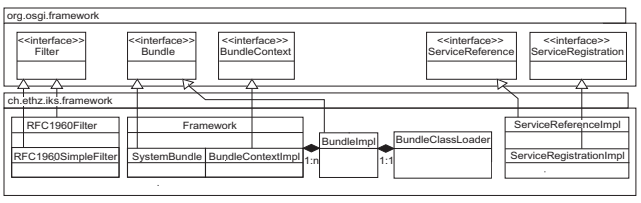

Figure 5: Concierge core classes

framework, the external access is restricted to the API. In this sense, we can afford to let the different classes of the framework directly access the fields to save the overhead of method calls. As before, the importance of this design decision lies both on the performance gains (up to a factor of 6 in every call) as well as on the level of consistent behavior that can be achieved.

3.5 The Average Bundle

Some optimizations are valid under all circumstances. Issues like choosing the best suitable data structure for a certain task however requires some knowledge about the frequency and the kind of access to that data structure. In OSGi, all activities of the framework are triggered by bundles. It is therefore very important to characterize the behavior of an average bundle on which to base optimizations. However, it is impossible to postulate an *a priori* knowledge about a general bundle. In the style of Sartre's fundamental theorem of existentialism [27], *existence precedes essence*, it is hence inevitable to observe the behavior of existing bundles to get a rough idea of how bundles operate.

As part of the work on Concierge, we introduce the notion of the *Average Bundle* to incorporate exactly this idea. We have collected from a variety of sources 107 publicly available bundles written for OSGi. These bundles have been installed on an instrumented version of Concierge. In this *profiling edition* of Concierge, all relevant activities have been logged. Then we have analyzed the log to establish a profile of what these bundles do on average. The resulting behavior is what we call the *Average Bundle*. For reasons of space, we can not describe the complete behavior of this *Average Bundle*. We will refer to concrete aspects of it as we discuss different design decisions in the architecture of Concierge.

The *Average Bundle* captures only the application independent behavior of bundles. It does not consider user interaction effects and it does not include the effects of incoming calls from remote machines (it does however include the effects of outgoing calls). The reason is that it is difficult to characterize user interaction and remote access patterns. Both are highly application dependent and in here we are only interested in platform relevant optimization. Eventually, the *Average Bundle* could contribute to solve the problem of adequate performance benchmarks for OSGi. So far, no real application benchmark for OSGi frameworks exists. The OSGi Alliance publishes test cases to test compliance, not performance. Knopflerfish has a regression test suite [18] that we use for benchmarking in this paper. However, in terms of benchmarking, these tests are only synthetic benchmarks. They reflect the Knopflerfish developers' opinion of useful and important framework features but cannot approximate the real behavior of bundles. The *Average Bundle* could be used in future work to create a meaningful application benchmark for OSGi frameworks.

3.6 Concierge design constraints

The discussion above document some of the key design decisions around Concierge. Although some of them may appear to be small optimizations, the results are cumulative. As the performance evaluation will show, these optimizations are what makes Concierge far more efficient than existing implementations. Our goal with Concierge is to have a real working platform for small devices. In those devices, it is impossible to get around such small level details because they are not hidden by ever increasing hardware capacity and a sophisticated VM with JIT compiler optimizations. Not tackling these issues at the level of Concierge would mean that they will become visible to the application developer. This is not acceptable according to our design requirements because it would defeat the purpose of using an OSGi platform. Given the current state of the art and development in hardware and software for small devices, it is unlikely that those issues will go away soon. Whereas for Java SE, the newest version 1.6 has just been released, the VM implementations for small devices still remain on the level of Java 1.4 or even earlier versions. The design of Concierge shows that being aware of these issues leads to more efficient systems without compromising code quality or maintenance. This same design points out what needs to be done at the OS and JVM level to provide a more consistent support across small computing devices. Until that happens, Concierge provides the necessary homogeneity to be able to develop applications that run on a multitude of devices and still exhibit consistent behavior across all of them. In what follows we discuss each OSGi layer of the Concierge implementations and how the ideas discussed in this section are applied in practice.

4. CONCIERGE MODULE LAYER

The module layer is the base layer of the Concierge implementation. As indicated, it is in charge of loading bundles.

4.1 Bundles Overview

As a first step to understand the difference between Concierge and existing implementations, we need to review what happens when a bundle is loaded. As mentioned before, bundles are JAR files. JARs are from a technical point of view ZIP compressed archives with one particular element called Manifest. The manifest is stored in the file `/META-INF/MANIFEST.MF` and contains meta data about the JAR, for example the vendor of the application or the application's main class. OSGi defines additional attribute elements for the manifest main section.

Instead of a main class, a bundle can define a `Bundle-Activator` that is the interface to the bundle's life-cycle control. When a bundle is loaded in the OSGi framework, the manifest has to be processed. Specific attributes like `Import-Package` or `Bundle-ClassPath` are relevant for the loading and resolving of the bundles. All attribute/value pairs are preserved in the properties of the Bundle object and accessible to other bundles, if they have appropriate permissions.

Bundles can be loaded from any external location that is expressible as a valid URL, i.e., a well-formed URL that maps to an `URLStreamHandler` known to the Java VM. Alternatively, an `InputStream` to a JAR can be directly provided. This includes data sources other than files, for example network streams or database entries. OSGi maintains persistent state between restarts of the framework. In order to do so, all Bundles are stored on persistent storage.

4.2 Dealing with compressed bundles

From a performance point of view, the fact that the bundles are compressed requires careful treatment because there is a significant performance difference between decompress-

ing bundles or operating on the compressed content. For small devices, it is clearly preferable to keep bundles compressed as much as possible. Unfortunately, this is not that straightforward because it depends on the content of the bundle. In this regard, the main problem is that the JAR file of the bundle might contain additional, embedded JAR files. These embedded JAR files can be accessed either by their content, if they are part of the bundle-internal class path, or the JAR file itself can be accessed as a normal resource in the bundle.

A possible strategy is to completely decompress the bundle, including (recursively decompress) embedded JARs. Obviously, this does not work for small devices, since in addition to the space that the bundle takes, the embedded JAR files have to be stored twice. A compressed copy is needed in case a bundle accesses the embedded JAR as a resource. A decompressed copy is needed for all other cases. A better strategy is to decompress the bundle but do not decompress the embedded JARs (we refer to this strategy as *Decompress Bundle*). This strategy incurs the runtime penalty of decompression at the time when the `Bundle` object is created. Although intuitively this strategy pays a one-time cost for decompression and performs all successive accesses directly on the uncompressed files, it also has drawbacks. The main one is that JAR archives contain a large number of small files and the cost of writing them to the file system turns out to be quite high.

This first strategy is based on the assumption that accessing decompressed data is faster. This is true only if there are repeated accesses. If information is accessed rarely, the overhead of decompression becomes less relevant. In fact, the overhead of accessing compressed bundles is lower than one would expect. ZIP archives compress each file individually, so it is possible to access only what is needed instead of having to decompress the whole bundle. Since they have an internal index structure consisting of a central directory header and file headers, the overhead of locating a file within a ZIP archive is low. In addition, bundles are not accessed that often. The vast majority of content in the bundle is in the form of class files. And, by design, class loaders access classes only once. This is discussed in more detail in Section 4.4. Multiple accesses to the same file can only happen in the case of resource files. We have used the *Average Bundle* to test the hypothesis that repeated access to the compressed bundle rarely happens. Table 1 contains a list of relevant events (left column), how often the the event takes place for the 107 bundles analyzed (center column), and how often the event happens in the *Average Bundle* (right column). The relevant events that access the bundle are the class and resource requests. As the table shows, the *Average Bundle* requests the loading of 42 classes. Of these 42 classes, 27.24 are from the bootstrap class loader. These are classes that do not reside in bundles and do not require decompression. The remainder of 13.88 class loads are to its own classes. Each class can be decompressed individually and once loaded, it is never accessed again (from the bundle). For these classes, it does not matter if the bundle is decompressed or compressed, the overhead is the same. The *Average Bundle* also accesses 2.11 times its own resources. This is a very small number and does not justify the space overhead of decompressing everything. Hence, overall, the numbers for the *Average Bundle* show that the bundle is not accessed repeatedly and it could pay off to keep it compressed.

Event	**total**	**avg**
Classes requested	4011	42.22
Classes loaded from bootstrap cl	2588	27.24
Classes loaded from import delegation	68	0.72
Classes loaded from own bundle	1319	13.88
Classes not found	36	0.38
Resources requested	223	2.45
Resources found by own bundle	200	2.11
Resources found by import delegation	0	0
Resources not found	23	0.24
Multiple resources requested	15	0.16
Multiple resources not found	14	0.15
Resolved bundles	95	1
Bundle has embedded jar	22	0.23
Bundle jar accesses	2386	25.12
Bundle jar reads	1022	10.76
Bundle embedded jar reads	499	5.25
Package mappings created	201	2.12

Table 1: Classloading behavior of the average bundle

The question that remains is what to do with embedded JARs, whether to extract them into separate files or to keep them inside the bundle. There are two possible strategies. The first one (called *StoreBundle*) leaves the embedded JAR files inside the bundle. The drawback here is that the runtime penalty for accessing classes and resources in embedded JARs is higher than accessing the compressed bundle itself. Embedded JARs are not accessible as files, they have to be located within the bundle JAR first and then accessed as `JarInputStream`. These streams cannot be searched for an entry in constant time, since the end of an input stream is not known at runtime and therefore no index exists. The second strategy (called *ExtractEmbeddedJAR*) is a hybrid solution between *Decompress Bundle* and *Store Bundle*. The embedded JARs are extracted from the bundle and stored separately as (non-embedded) JAR files. To avoid duplication, the embedded JARs are removed from the bundle. Since Java does not allow the deletion from files in archives, this involves decompressing the entire bundle, extracting the embedded JARs, and recompressing what is left of the bundle. Figure 6 compares the performance of these three strategies (*DecompressBundle*, *ExtractEmbeddedJAR*, and *StoreBundle*). The experiment uses a scenario where a bundle with a size of 141 kBytes containing an embedded Jar with a compressed size of 105 kBytes is loaded and accessed. The runtime of each strategy is divided into three phases: *init* puts the JAR into the initial state, which includes, depending on the strategy, copying, decompression or recompression of the bundle; *load* loads classes from the original bundle JAR; *load embedded* loads classes from the embedded JAR.

The interpretation of the results of the Figure depend on whether embedded JAR files are present or not. If there are embedded JAR files, the *DecompressBundle* strategy is clearly preferable in all platforms in terms of runtime. However, this is the most expensive strategy in terms of space consumption, a critical parameter in small devices. If there are no embedded JARs, the *StoreBundle* strategy is the best.

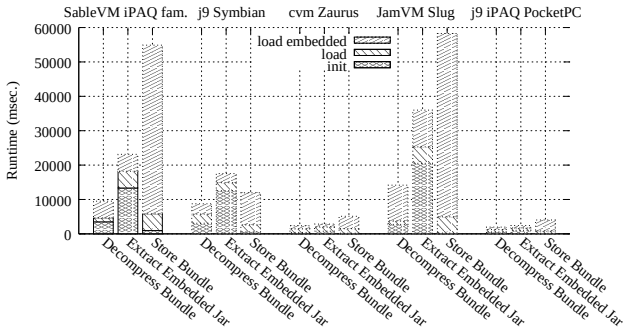

Figure 6: Bundle JAR loading variants

Unfortunately, in some platforms, this strategy pays a huge penalty when embedded JARs are present. Fortunately, it is known in advance if a bundle contains embedded JAR files. In addition, according to the Average Bundle, only 23% of all bundles have embedded JARs.

Based on this, Concierge implements both the *StoreBundle* and the *ExtractEmbeddedJAR* strategies. By default, it uses *StoreBundle*. A system property can be set that will force Concierge to use the *ExtractEmbeddedJAR* strategy whenever a bundle contains embedded JARs. The reason to use the flag is that the performance difference between the two strategies is highly platform dependent. For instance, in the platforms with J9, the strategy does not pay off. In which platforms it might pay off to set the system property is shown in Figure 7. In this experiment, we run the Knopflerfish Regression Tests once with the system property set and once with the default behavior. The Figure shows the relative gain of setting the property over using the default behavior. The reason to use these tests is that they do not contain many bundles with embedded JARs. Once more, the performance gains might seem small, however, this performance difference constitutes a lower bound to the potential gain of switching the strategies. For more general loads, the gain will be much larger. The upper bound of the gain between the strategies corresponds to the values in Figure 6.

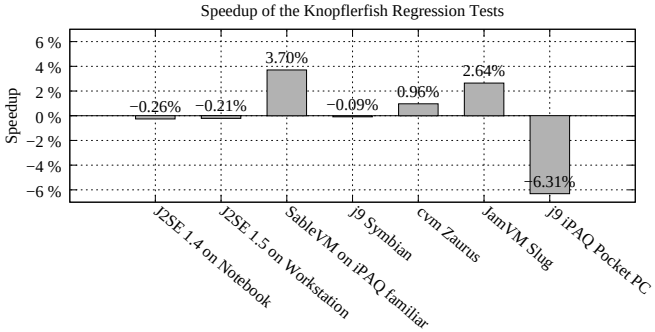

Figure 7: Speedup of the knopflerfish regression tests by decompression of embedded JAR files

For completeness, it must be mentioned that there is one case in which parts of the bundle have to be decompressed no matter what. By design, Java class loaders cannot load native code libraries from input streams. If a bundle contains native code libraries, these libraries have to be extracted. Native code libraries are not frequently used in OSGi bundles, although the OSGi specification features a sophisticated matching algorithm that allows to embed libraries for different platforms. Only three of the 107 bundles that contribute to the *Average Bundle* use native code.

4.3 Resolution

In smaller devices, the possibility for interaction are often limited. Thus, operations that impose too many constraints should be avoided, as these operations often end up increasing the number of interactions required. This is particularly true for software installation and especially delicate when it involves users that are not computer experts. In this regard, existing OSGi implementations suffer from the problem that for applications to work, some bundles have to be loaded in a very specific order. Failure to do so results in applications not working in ways that are not easy to identify. Concierge makes here an important contribution since it removes this constraint as it allows bundles to be loaded in whatever order, independently of the dependencies among them. We expect this to be a very important feature for hand-held devices and mobile computing.

To explain the problem in detail, we need to dwelve in how bundles are resolved. Prior to any package import/export or class loading, every bundle has to be resolved. First of all, the bundle's class path mentioned in the manifest's `Bundle-ClassPath` attribute has to be resolved by ensuring that every entry in the classpath list actually maps to an existing embedded JAR in the bundle. Embedded JAR files that do not occur in the internal classpath are possible, but they cannot contribute code and act like ordinary resources. Next, the `Import-Package` declaration is resolved. For every entry in the declaration list, a matching package has to be found. The framework maintains a mapping of all exported packages and the corresponding exporting bundles. Entries in the mapping are made only when the exporting bundle has been resolved. If this bundle imports from a second bundle, the second bundle has to be resolved before the first bundle can be resolved. This makes the correct startup order of bundles crucial and circular dependencies between bundles potentially challenging. However, the specification does not enforce a specific point in time where a bundle is to be resolved. The only constraint is that bundles have to be resolved before they can be started. When installing applications, these dependencies are far from trivial and can be resolved only by experts familiar with the code.

Concierge eliminates the problem entirely by adopting a two level resolution strategy. Whenever a bundle is installed, the framework tries to directly resolve it. If this is successful, all exported packages are registered. Otherwise, all exported packages are registered as *potential exports*. The first level, *non-critical* resolution of a bundle will try to make use of potential (=unresolved) exports of other bundles and trigger their resolution but without recursion. As a result, required potential exports are themselves not allowed to trigger the resolution of other potential exports. When a bundle is started, the resolution is *critical* because the requested operation cannot be performed otherwise. In this case, potential exports are tried to be recursively resolved with infinite recursion depth. Cycles in the

dependency graph are suppressed. The combination of the inexpensive but limited eager resolution attempt and the exhaustive lazy resolution in critical situations, make Concierge far more flexible then existing OSGi implementations. Cyclic dependencies and lately installed package imports are no longer harmful. While this feature of Concierge does not affect performance, it goes a long way to simplify application development, testing, and maintenance.

4.4 Class Loading

OSGi makes heavy use of user-defined class loading to allow for dynamic adding and removal of bundles. Since every creation of a new class involves class loading, the performance of the class loading subsystem is crucial for the entire OSGi system and the applications running on it. On the one hand, the design of OSGi allows the independent loading and unloading of bundles which requires every bundle to be loaded by a separate class loader. On the other hand, packages of a bundle can be imported by other bundles, which requires a flexibility in the delegation between class loader that exceeds the possibilities of standard Java. For our target area, it is hence of extraordinary importance to provide a robust and efficient way of class loading for bundles.

Class loading is frequently considered to be one of the most cryptic concepts behind Java. First, a class loader builds up a namespace of all classes loaded by itself. Instances of the same class but loaded by different class loaders are considered to be different and instances of these classes are incompatible, although they have the same qualified name. Each class loader has to keep a reference to all class types that have been loaded by it. This is necessary to ensure that a class is never loaded more than once by the same class loader. Otherwise, the type system would become inconsistent since two instances of the same type could be incompatible. Whenever a class is requested by the runtime system, the class loader is obliged to first check if it has already loaded the class before. Class types are thus referenced by the class loader and are not garbage collected as long as the defining class loader instance exists. Removing class types from the system hence requires to unload the entire defining class loader. For an OSGi system, that means that for each bundle, a separate class loader is required to allow the removal of individual bundles.

Conventional Java applications typically use a single class loader for all classes: The system class loader. This class loader has access to every class file that is in the class path. Java uses a delegation model between class loaders. With version 1.2, every class loader can be assigned to be parent of another class loader. In conventional Java applications, the parent of the system class loader, that loads the application's code, is the bootstrap class loader, that loads all system classes, such as those in `java.*` packages. The parent of the bootstrap class loader is always set to `null`. Every standard class loader first delegates the call to the parent class loader and only if the class is not known by the parent, it takes up own efforts to load the class.

User-defined class loading is the core of most component-based extensible Java application frameworks. It is not only possible to remove types at runtime by disposing the defining class loader, customized class loader can be used to locate and load classes from locations different from the file systems. The most popular example of a network-based class loader is the Applet [13] platform, which is a user-defined class loader able to load classes by using the HTTP protocol and equipped with special security constraints. Another example is the Tomcat [4] web container. Similar to OSGi, it needs to add and remove entities (in this case web applications) at runtime. Hence, these web applications have to be loaded by separate class loaders, as depicted in Figure 8. Furthermore, different web application must be isolated so that a malicious web application cannot affect other web applications or the runtime system. This is also an intrinsic property of the class loader design since each web application is only able to access code of its own class loader or of those in the delegation path. The same applies to OSGi and is another reason why OSGi support is important in mobile devices.

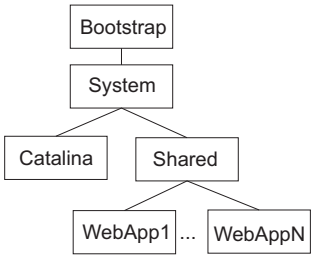

Figure 8: Class loader schema in Tomcat

Class loading in OSGi is far more complex. Instead of a so-called *flat* delegation schema—typically a tree as in Tomcat—OSGi bundles form a partly-connected delegation graph. The import and export of packages are the directed edges of the delegation graph, whereas the class loaders of the bundles are the vertices.

The OSGi specification requires the following search order for class loading: First, the system class loader, then the class loader that exports the shared package, if the class could belong to an imported package, and finally the bundle's own class loader. Since the parent delegation model of Java does not support *N to N* delegations as necessary in OSGi, package imports have to be handled outside the standard Java behavior. As a first step, the class loader has to determine if a requested class potentially belongs to an imported package and explicitly call the corresponding class loader prior to any attempt to load the class. Owing to the impact of class loading on performance, Concierge uses a `HashMap` to store a mapping between package name and the exporting class loaders. This map is initially filled during resolution of the bundle and allows to decide in constant time, if a class is a candidate for delegation and getting the exporter in the same step. If the result of the hash map lookup is `null`, the class cannot be imported and the class loader is thus allowed to be loaded from the own bundle. Otherwise, the request to load the class is delegated to the exporting class loader. This involves, depending on the design of the class loader structure, several method calls. As discussed in Section 3.4, method calls are very expensive in smaller devices. Unlike other existing OSGi implementations, Concierge has implemented the class loader subsystem in such a way that method calls for delegation of classes

are reduced to a minimum. Since the particular impact of method calls differs across the platforms, this is also important for achieving consistent behavior on all platforms.

If the class is not imported, the `BundleClassLoader` tries to locate the class file in the bundle JAR file and in all embedded JARs that are in the bundle's internal class path. The actual loading of the class is performed by reading the corresponding byte array and feeding it into the *defineClass* method, which returns a `Class` object. This object is then returned through the delegation path to the original requester.

4.5 Bundle Registry

The framework has to maintain the information about all registered bundles. Many operations act on the bundle ID, a `long` value assigned to the bundle during installation. In database terminology, the bundle ID is a primary key of the bundle. It is unique, ascending, preserves the installation order and must not be changed during the lifetime of the bundle. A once assigned bundle ID must not be reused for a different bundle, even after restarts of the framework. To allow for fast lookup of the bundle ID–bundle reference, a hashed data structure like `HashMap` is most suitable. However, in some situations, the order of the bundles is important. It is expected that bundles start in the order in which they were installed, i.e., the ascending order of bundle IDs. Since the `HashMap` does not guarantee any iteration order, the `LinkedHashMap` would be ideal for that purpose. Such data structure has been introduced with Java 1.4 and is not available, for instance, on J2ME CDC profiles. Concierge instead uses an additional `ArrayList` to keep the order in an efficient way. The overhead compared to a reimplementation of the `LinkedHashMap` is 8 bytes per entry, since a `HashMap` entry consumes 24 bytes, an `ArrayList` element is an `Object` reference of 16 bytes and a `LinkedHashMap` entry has only 32 bytes. Measurements have shown that not only the code size is more compact for the `ArrayList` variant, but also the performance is slightly better, even compared to the Java 1.4 `LinkedHashMap` implementation.

5. LIFE CYCLE LAYER

Besides the bundle operations *install*, *start*, *stop* and *uninstall*, the life cycle layer offers the possibility to update bundles. Updates can have implications on the imports of packages and on services used by other bundles. The OSGi specifications emphasize the consistency of the framework at any time. Packages that are imported by other bundles are therefore unaffected by bundle updates since this could lead to an inconsistent framework state. All other packages, however, have to be updated immediately. In terms of performance and consistency in our heterogeneous set of devices, the life cycle layer is not too relevant. The optimization is to make reasonable trade-offs to assure that important operations are well supported by appropriate data structures while keeping memory consumption to a reasonable minimum.

Concierge accommodates the perpetuation of consistency among package exports by preserving the exporting class loader when bundle updates occur. Existing delegations between other bundles and the updated bundle's class loader are redirected to the originally exporting class loader, unless a full update of the bundle is triggered by the `PackageAdmin-Service` (an optional standard OSGi service) or by a restart of the framework. In both cases, affected bundles have to be restarted to access the new version by import delegation. Uninstallation of bundles or failed registrations require an entire cleanup of all resources associated with the affected bundle. This includes the deregistration of registered services, the disposal of exported packages and a notification to all subscribed listeners. Own registered listeners have to be unregistered as well. The used data structures do not in every case allow a fast lookup of the entities that belong to the bundle. For example, the service registry does not preserve the information of bundles that have registered the particular service. The `BundleImpl` object therefore contains references to certain registered entities to allow quick and thorough cleanup.

6. SERVICE LAYER

Whenever a bundle wants to use a service, it requests a service reference via `BundleContext` that matches the requested interface name and, optionally, a provided filter. The result of this service request is either a valid `ServiceReference` object or `null`, if no matching service is available. It is possible to retrieve more than one service reference at once, for more than one implementation of the same service interface or for different services matching the same filter. The performance of the service layer is a potentially limiting factor for any OSGi implementation. In the following, we present Concierge's optimized service management subsystem. Different from the optimizations discussed so far, the performance of operations on this layer cannot be easily isolated, since the variety in which they can appear is too large and the operations depend too much on the composition of bundles and services. However, a good indicator of the service layer's overall performance is the Performance Registry Test Suite of the Knopflerfish Regression Tests, that we discuss as part of the evaluation of Concierge.

6.1 Filters

Among all entities, services typically appear in the largest quantity. It is hence often necessary to limit the results of a service request and the set of events for which a service listener gets informed to a defined subset. OSGi supports RFC 1960 [11] LDAP String Filters for this purpose. The filter implementation is very important for the performance of the whole framework. Not only that service requests are on average the most frequent framework operation at runtime and about ten percent of them are filtered, also service listeners can be registered with filters to get only informed if a matching service arrives, the properties have been modified, or the service has disappeared. Some OSGi implementations try to improve performance by, e.g., caching intermediate results of filter evaluations. This is not feasible in Concierge because of the tighter memory constraints on small devices. Instead of caching, Concierge provides a highly optimized RFC 1960 filter implementation that performs well even when resources are scarce.

Filters can consist of several well-formed braced literals combined by boolean operators. This structure has to be parsed and modeled by appropriate Java data structures. For instance, the filter

```
(&(service.id<=10)(|(requires_screen=false)
(!(presentation=*))))
```

is modeled in Concierge by the following data structure:

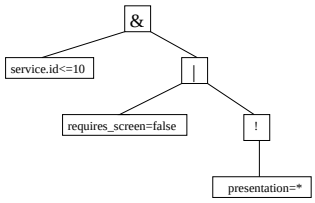

Figure 9: Structure of a filter in Concierge

Typically, OSGi implementations do the lexical analysis of the literals and recursively parse them to build a tree of filter parts. However, recursion creates a severe performance problem on resource-constrained devices, since it leads to large stack frames and typically involves a large amount of method calls.

Since LDAP filters are described by a context-free grammar in a prefix notation, Concierge uses an innovative one-pass stack-based LALR [1, 6] parsing approach with a lookahead of length one. The detection of a left parenthesis leads to a lookahead whether the following character signals a boolean concatenation of literals or if a singular literal follows. In case of a concatenated expression, a new filter part object with the according boolean operator is pushed onto the stack. All following symbols are linked to this element and either also pushed to the stack, or, if they are single literals, only parsed and linked. Every terminal literal is of the form *key operator value*, except literals using the *present* ('=*') operator, that requires no value (in our example, the presentation attribute). The operators can also be parsed with a lookahead of one. Whenever a right parenthesis is read, the topmost stack entry is removed. At the end of the parsing, a tree of filter part objects is formed that exactly reflects the structure and logic of the filter string.

Once parsed, the filter object can be used for matchings against property dictionaries. For every literal, a corresponding key in the service property dictionary has to be found by case insensitive matching. This complicates the implementation of matching since `Hashtables`, the typical implementation of the `Dictionary` interface, by nature only supports case sensitive key lookup. Variants of the key have different hash values and therefore do not lead to matching. It would be possible to initially transform all keys of a dictionary to equal case, and do the same for the filter's attribute key on every request. However, this is often unnecessary overhead since most attribute keys used in literals are actually constants defined by the OSGi API, like `service.id` or `objectClass`. The filter statistics of the *Average Bundle* summarized in Table 2 show that 40% of the filter attribute keys were well-defined constants. Concierge therefore first tries to find a key by case sensitive matching, then checks the lowercase variant of the filter key and as final fallback tests every key in the dictionary. This preserves the performance of the `Hashtable` for all constant keys and otherwise performs as custom array-based case-insensitive implementations of the dictionary interface such as those used, e.g., in Knopflerfish.

According to the type of the value in the dictionary, the value in the filter has to be transformed into the same type to be compared. A naive approach would use reflection and try to find a constructor that takes the string as argument. However, reflection is a serious performance bottleneck. Thus, this approach is only used when all optimizations fail. Concierge has optimized matchings for all primitive types and all boxed types[2] as well as Strings, arrays, and vectors.

filter count	42	0.44 per bundle
simple filters	25	60% of all filters
complex filters	17	40% of all filters
number of literals	70	4.12 literals per complex filter
total literal count	95	2.26 literals per filter
constant attr keys	38	40% of all literals

Table 2: Filters of the *Average Bundle*

6.2 Service Registry

Every service registered with the OSGi framework has to be stored in a service registry to allow lookup, modification of the properties, and removal. Each service can be registered under 1 to n service interface names, every service interface can be implemented by 0 to m services. This is, from a database point of view, an N to M relationship, which is not trivial to model efficiently in Java. Due to the special importance of services for the overall performance, an algorithmic optimization of the service registry is very important for the runtime performance of applications running on the OSGi platform.

Concierge uses a `HashMap` with the interface name as key and an `ArrayList` of `ServiceReference` objects as value (Figure 10). This is the optimal support for the most frequent lookup operation where one or all services implementing a certain interface have to be found and their references have to be returned. Modification and removal of registered services uses the `ServiceRegistration` object that is returned by the framework as result of a registration. In some sense, `ServiceReference` and `ServiceRegistration` are just two different facets of the same entity: the service. But the two interfaces cannot be merged into one implementing object since this would cause security problems. Not every service that has permissions to retrieve a service reference is also allowed to modify or unregister the service. In Concierge, `ServiceRegistrationImpl` is a private inner class of `ServiceReferenceImpl` to prevent this. Only the entity that received the object by registering the service has access to it but registrations do not have to be stored separatly.

Most other OSGi implementations use the opposite approach and store registrations in the registry with a field pointing to the service reference. This is unnecessary since the registration is never returned as the result of a query. A problem of the design with the `Map` containing `Lists` is the removal of services. Whenever a service is removed, it might have been registered under more than one service interface and potentially every `ArrayList` in the `HashMap` would have to be checked for the service reference. However, the information under which interfaces the service has been registered is available in the service properties as `objectClass`. Thus, the removal can be done in deterministic time. For

[2]except `BigInteger` and `BigDecimal` that are not supported by every VM and hardly occur on small devices

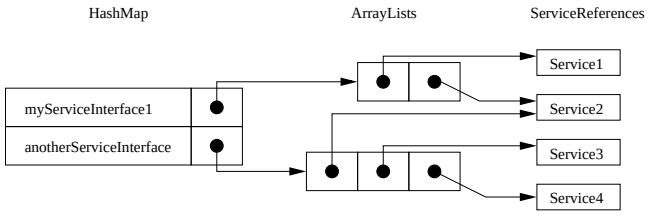

Figure 10: Service Registry Data Structures

some operations, all registered services have to be traversed. To support this, an additional `ArrayList` of all registered services is maintained, as it is done for the bundle registry.

7. SERVICES

The OSGi specification defines a couple of standard services for OSGi platforms that can optionally be implemented. Normally, services should be provided as independent bundles. Not every setup of application bundles requires all of the standard services, so they can be added on demand. For a small number of services however, there are good reasons for implementing them as part of the framework. Services like the `PackageAdminService` are so heavily dependent on internals of the framework that they are hard to implement efficiently as stand-alone bundles. And since they provide important extensions of the framework's abilities, they are most likely to be used by other bundles. The feature matrix in Table 3 lists the open source OSGi distributions and the services they provide within the framework.

Service	Knopflerf.	Oscar	Concierge
StartLevel	Framework	Framework	Framework
PackageAdmin	Framework	Framework	Framework
Log	Bundle	Bundle	Framework
ServiceTracker	Framework	Framework	Bundle
PermissionAdmin	Framework	Bundle	Bundle
URL	Framework	Bundle	Bundle

Table 3: OSGi Implementations Feature Matrix

Although Concierge is designed with a focus on footprint, we have implemented the `StartLevelService` and `PackageAdminService` as part of the framework's system bundle for reasons of performance and simplicity. The `StartLevelService` adds the possibility to group bundles into start levels, which can be started and stopped simultaneously. The `PackageAdminService` allows to find out about exported packages, which bundle has exported them and where they are in use. And, most important, it allows to force the replacement of all exported packages that have been updated. Both services increase the total footprint of Concierge only marginally but provide adequate benefits to the user. The `LogService` is normally independent of the framework. It allows bundles to log messages with certain log levels and `LogReader` instances to get the logged messages for evaluating or displaying them. Since logging is of special importance in embedded systems which sometimes are "headless" and feature no screen device, Concierge has implemented a minimal log service in the framework to allow bundles and the framework itself to use this service.

So far, no other standard services have been implemented. It is up to future work to build some of the standard services with the "Concierge philosophy". The usual way of other distributions to provide services like the `HttpService` is to embed a full-blown Java implementation like Jetty into a bundle and to write a technology adapter for the specific OSGi service API. These services can easily require a multiple of the framework's own footprint and are thus certainly not suitable for resource-constrained devices. Providing such services will mean in many cases writing new custom implementations tailored to small devices.

8. EVALUATION

Concierge is a complete OSGi R3 implementation and thus compares with Knopflerfish 1.3.5 or Oscar 1.0.5. The supported startup semantics include Knopflerfish *init.xargs* files as well as properties in the style of Oscar. Additionally, Concierge allows to explicitly define profiles in the startup file. These profiles allow to start the framework from shell scripts using different configurations of bundles and different storage locations without the user interaction needed in systems like Oscar.

8.1 Setup

Two different aspects were considered during the implementation of Concierge: Runtime performance and resource consumption. Lacking a standard benchmark, for evaluation of the performance, the Knopflerfish Regression Tests have been used on the several test platforms. These tests consist of several JUnit test suites which cover different functional aspects of an OSGi framework. Although the main intention of the test is to ensure the correctness of the functionalities described in the OSGi specifications, they also measure the total time for each test suite. This allows to use them for benchmarking purposes. The *ConstantTestSuite* is of little interest in terms of performance. It only checks that all constants mentioned in the OSGi specifications are set to the correct value and are accessible for applications. The *FilterTestSuite* creates a large set of LDAP-style filters as they are used in service reference requests. The *FrameworkTestSuite* checks a large range of framework features and is thus a good indicator how the framework as a whole performs. It is close to the kinds of operations that occur during the startup of an application. The *PackageAdminTestSuite* performs updates on bundles and checks, if the exported packages are still consistent. The performance of this test is a good indicator how the class loading performs. For estimating the runtime performance, the *PeformanceRegistryTestSuite* has a high relevance. This test registers 1000 services and 100 service listeners, which is a good indicator of how a system scales. Furthermore, operations on services are the most frequent ones during the runtime of application bundles. The other tests are more functional and less relevant in the context of performance.

Not all tests ran successfully on every platform. Oscar had problems with matching certain filters on SableVM and both Oscar and Knopflerfish threw an exception during the PackageAdmin test on JamVM. A few of the tests had to be patched because they were too much tailored to Knopflerfish's special behavior. For example the order of certain events during installation of bundles depends on the resolution strategy.

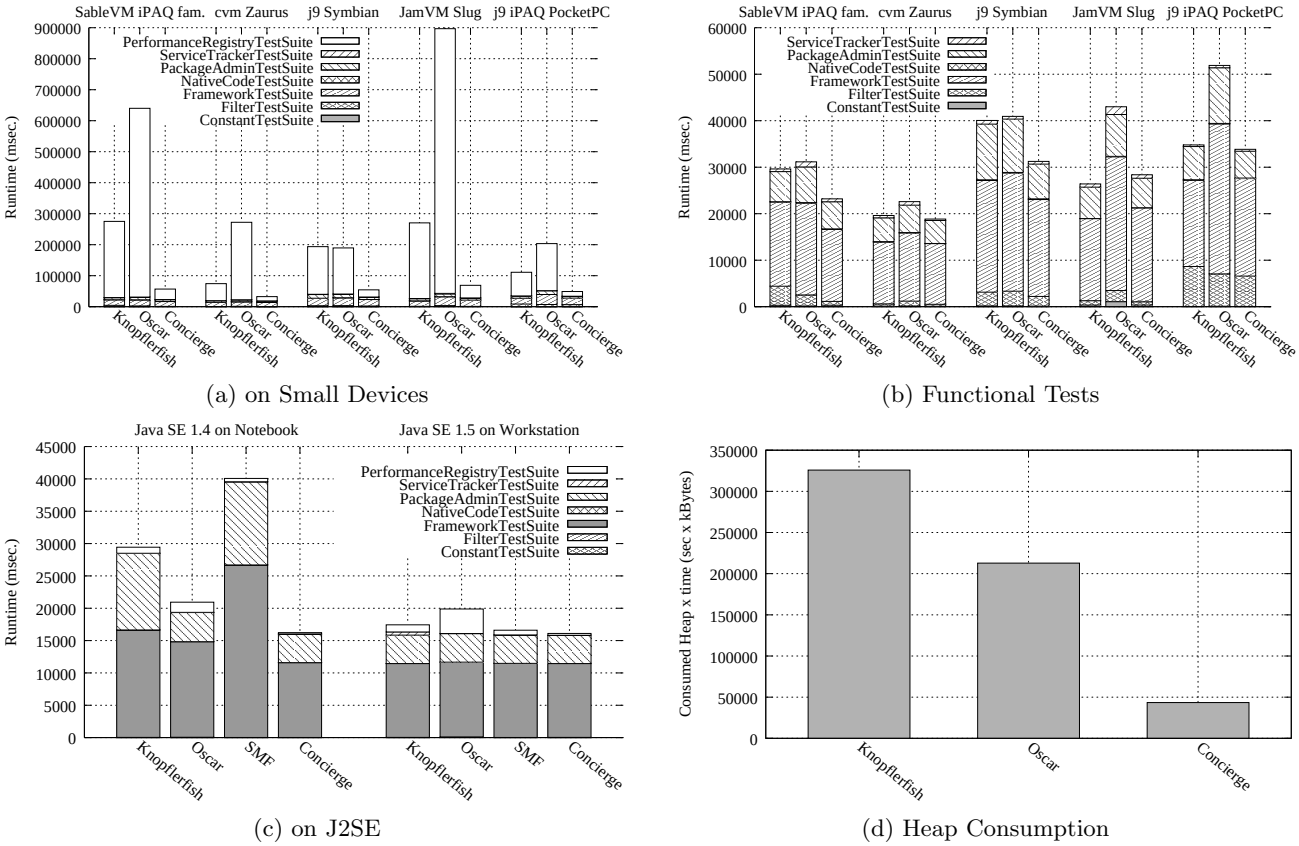

Figure 11: Benchmarks

8.2 Performance

Figure 11(a) shows the detailed runtime of the regression tests on several small devices. Concierge performs substantially better than the other open source implementations. Although the selected platforms are heterogeneous, Concierge outperforms the other implementations on all platforms. The speedup ranges from about 3.2 in comparison with Knopflerfish on cvs Zaurus up to about 13 times better than Oscar on the Slug. The main performance improvement is the service layer that is benchmarked in the PerformanceRegistryTestSuite. The service layer is a major performance factor during the runtime of an OSGi framework, but it also interesting to analyze the performance of the other layers. Figure 11(b) shows only the more functional tests in a larger scale than in the previous Figure. Concierge performs better on most platforms or at least competitive in the case of JamVM on the Slug. But not only the intended domain of resource constrained devices can profit from the optimizations within Concierge. Figure 11(c) shows the runtime of the regression tests on a Notebook with J2SE 1.4 and on a Workstation running J2SE 1.5. Even on these devices where resources are not a limiting factor and the Java VMs are highly optimized, OSGi applications can gain some performance when running on Concierge. With more complicated setups, for instance, a large number of installed bundles and a high frequency of framework operations, the speedup of Concierge is also noticeable. In this situation, the optimizations on the module layer lead to the improvement.

8.3 Resource consumption

Concierge has a file footprint of under 75 kBytes whereas Knopflerfish and Oscar are both slightly above 200 kBytes, ProSyst's mBedded Server needs about 275 kBytes and IBM's SMF almost 440 kBytes. Concierge also requires less memory for maintaining the framework and the registries. The traces depicted in Figure 11(d) show the integral of the heap consumption during the regression tests (measured on a notebook). The VM is configured with an initial heap size of 1 MB and a maximum size of 2 MB. Concierce allocates less memory on the heap during the tests. Therefore, the garbage collector is not so frequently invoked, especially not in critical cases that slow down the execution of the program. The overall runtime of the tests under these conditions are 3.5 minutes Oscar, 2.4 minutes for Knopflerfish and 38 seconds on Concierge. In conclusion, when running with low resources, the performance of Concierge is significantly better even on powerful machines.

8.4 Consistency across platforms

One important goal of Concierge is to provide a certain level of consistency on all platforms, although both the hardware and the Java VM are heterogenous. In Figure 11(a), we have compared the runtime of different OSGi implementations on the test platforms. Considering just the results for Concierge, it can be said that this goal has been reached. For instance, the SableVM iPAQ platform is a PDA with a generic, mainly unoptimized open source VM, wheras J9

Symbian is a Nokia Smartphone with a custom-tailored and optimized J9 port. Unlike Oscar, Concierge is able to provide almost equal performance, although the setups are completely different. A gap that Concierge cannot level are performance differences that result from fundamentally different levels of optimized VMs. The JIT-enabled cvm on Sharp Zaurus remains two times faster than the SableVM iPAQ, although the hardware setup is almost identical.

8.5 Even smaller devices

As an additional test on "physically small devices", we have benchmarked Concierge on an Intel iMote2 sensor network board (ARM5-TE compliant XScale PXA27x, 32 MB RAM, 32 MB Flash). With a linux OS for arm-embedded and a JamVM installed, Knopflerfish and Concierge are able to run, although Knopflerfish has some erroneous test cases. Oscar does not run properly, many test cases fail. The final runtime of the regression tests for Knopflerfish is 5 minutes, Concierge needs only 70 seconds.

9. CONCLUSIONS

In this paper we have presented Concierge, a full implementation of the OSGi R3 specification. Concierge targets small devices. The experiments in the paper show that the design of Concierge leads to better and more consistent performance across devices than with existing OSGi implementations. In the paper we have addressed in detail the challenge of designing an OSGi platform in resource constrained devices and how to take advantage of the insights gathered through extensive experimentation. The result is an OSGi implementation that performs significantly better than existing implementations and thereby extend the range of applications for OSGi. Future work around Concierge includes a distributed version and a fluid computing platform [5] implemented on top of Concierge. More details and downloads are provided on `http://www.flowsgi.inf.ethz.ch`.

10. REFERENCES

[1] A. V. Aho, R. Sethi, and J. D. Ullman. *Compilers: Principles, Techniques, and Tools*. Addison-Wesley, 1986.

[2] Apache Directory Server. `http://directory.apache.org`.

[3] Apache Felix. `http://incubator.apache.org/felix/`.

[4] Apache Tomcat. `http://tomcat.apache.org`.

[5] D. Bourges-Waldegg, Y. Duponchel, M. Graf, and M. Moser. The Fluid Computing Middleware: Bringing Application Fluidity to the Mobile Internet. In *SAINT 05: Proceedings of the 2005 Symposium on Applications and the Internet, Trento, Italy*, 2005.

[6] F. DeRemer and T. Pennello. Efficient computation of lalr(1) look-ahead sets. *ACM Transactions on Programming Language Systems*, 4(4):615–649, 1982.

[7] M. Desertot, C. Escoffier, and D. Donsez. Autonomic management of J2EE edge servers. In *MGC '05: Proceedings of the 3rd international workshop on Middleware for grid computing*, pages 1–6, 2005.

[8] Eclipse Equinox. `www.eclipse.org/equinox/`.

[9] Eclipse Rich Client Platform. `www.eclipse.org/rcp/`.

[10] R. S. Hall and H. Cervantes. An osgi implementation and experience report. In *CCNC 2004: Proceedings of the First IEEE Consumer Communications and Networking Conference*, pages 394–399, 2004.

[11] T. Howes. A String Representation of LDAP Search Filters. *RFC 1960*, June 1996.

[12] IBM Service Management Framework. `www.ibm.com/software/wireless/smf/`.

[13] Java Applets. `http://java.sun.com/applets/`.

[14] Java Mobile Information Device Profile. `http://java.sun.com/products/midp/`.

[15] JSR 218: Connected Device Configuration (CDC) 1.1. `http://jcp.org/en/jsr/detail?id=218`.

[16] JSR 219: Foundation Profile 1.1. `http://jcp.org/en/jsr/detail?id=219`.

[17] Knopflerfish OSGi. `www.knopflerfish.org`.

[18] Knopflerfish Regression Tests. `https://www.knopflerfish.org/svn/knopflerfish.org/tags/1.3.5/osgi/bundles_test/regression_tests/`.

[19] The K Virtual Machine (KVM). `http://java.sun.com/products/cldc/wp/`.

[20] T. Lindholm and F. Yellin. *The Java Virtual Machine Specification*. Addison-Wesley, 2nd edition, 1999.

[21] Multimedia Home Platform. `www.mhp.org`.

[22] Open Cable Application Platform. `www.opencable.com/ocap/`.

[23] Oscar OSGi. `http://oscar.objectweb.org`.

[24] Open Service Gateway Initiative. `www.osgi.org`.

[25] OSGi Alliance. Listeners Considered Harmful: The "Whiteboard" Pattern. `www.osgi.org/documents/osgi_technology/whiteboard.pdf`, 2004.

[26] ProSyst mBedded Server. `www.prosyst.com/products/osgi_framework.html`.

[27] J.-P. Sartre. *Being and Nothingness*. 1943.

[28] B. Venners. *Inside the Java 2 Virtual Machine*. McGraw Hill, 1999.

APPENDIX

- **J2SE 1.4 Notebook:** *Type:* Notebook, *Manufacturer:* Dell, *Model:* Inspiron 9300, *CPU:* Intel Pentium M 750 1.8 GHz, *BogoMIPS:* 1576.09, *RAM:* 1024 MB, *OS:* Windows XP SP 2, *Java VM:* Sun JDK 1.4.2_12, *Compliance:* J2SE 1.4, JIT
- **J2SE 1.5 Workstation:** *Type:* Workstation, *Manufacturer:* Dalco, *Model:* D865 Workstation, *CPU:* 2x Intel Pentium 4 3.0 GHz, *BogoMIPS:* 5985.04, *RAM:* 2048 MB, *OS:* Linux Fedora 2.6.17_smp, *Java VM:* Sun JDK 1.5.0_06-b05, *Compliance:* J2SE 1.5, JIT
- **SableVM iPAQ Familiar:** *Type:* PDA, *Manufacturer:* Compaq, *Model:* iPAQ 3870, *CPU:* StrongARM SA-1110 206 MHz, *BogoMIPS:* 137.21, *RAM:* 64 MB, *OS:* Linux familiar 2.4.19-rmk6-hh37, *Java VM:* SableVM 1.11.3, *Compliance:* J2SE 1.4
- **cvm Zaurus:** *Type:* PDA, *Manufacturer:* Sharp, *Model:* Zaurus 5500 G, *CPU:* StrongARM SA-1110 206 MHz, *BogoMIPS:* 137.21, *RAM:* 64 MB, *OS:* Linux Embedix 2.4.6-rmk1-np2, *Java VM:* Sun cvm 1.0, JIT, *Compliance:* J2ME CDC Personal Profile
- **J9 Symbian:** *Type:* Smart Phone, *Manufacturer:* Nokia, *Model:* 9300i, *CPU:* TI OMAP 1510 150 MHz, *BogoMIPS:* 125, *RAM:* 80 MB, *OS:* SymbianOS, *Java VM:* IBM J9 2.2, no JIT, *Compliance:* J2ME CDC Foundation
- **JamVM Slug:** *Type:* Embedded Linux Network Storage Link, *Manufacturer:* LinkSys, *Model:* NSLU2, *CPU:* Intel XScale IXP420 133 MHz, *BogoMIPS:* 131.48, *RAM:* 32 MB, *OS:* Linux NAS 2.4.22-xfs, *Java VM:* JamVM 1.4.3, *Compliance:* J2SE 1.4
- **J9 iPAQ PocketPC:** *Type:* PDA, *Manufacturer:* HP, *Model:* iPAQ 5550, *CPU:* Intel XScale PXA255 400 MHz, *BogoMIPS:* 397.31, *RAM:* 128 MB, *OS:* Windows PocketPC 4.20, *Java VM:* IBM J9 2.2, no JIT, *Compliance:* J2ME CDC Personal Profile