Abstract

Computing systems are increasingly becoming dynamic. One example is cloud computing and its property of elasticity. On cloud platforms, resources in the form of additional nodes can be added and removed at any time. Software systems expected to run in such environments, on the other hand, are not nearly as elastic and flexible. Another example is the clearly visible trend towards incorporating an increasing number of processor cores into modern computer systems. In the future, it is likely that not all of these cores will have a uniform instruction set anymore but specialized units are used to accelerate certain tasks. At the same time, however, the power envelope of computer systems is increasingly becoming an issue so that probably not all cores can run at the same time anymore. Software written for such systems is thereby required to adapt to a changing pool of resources, a situation that today’s software is hardly prepared for.

This thesis contributes towards understanding how to build software systems that are able to reflect such a degree of flexibility. The fundamental observation underlying this work is that in order to respond to the emerging dynamism in the platforms, software has to become equally flexible in its design. This requires a segregation of the software into smaller units. In the early days of computer science similar challenges in the development process of complex software have catalyzed the concept of software modularity. The premise of this thesis is that the same kind of modularization—when not only applied on a logical level to the source code but also in a physical form and preserved until runtime—results in the required degrees of freedom in the design of software systems. In combination with a smart runtime system, this approach turns software into flexible, fluid entities able to adapt to a dynamic environment.

Three systems are presented in this thesis which are based upon the OSGi standard for dynamic modules in the Java language. They enhance the standard with different degrees of flexibility. The Juggle system co-manages software modules and the binaries for reprogramming an FPGA device so that applications can be selectively accelerated on demand and without interrupting running operations. The CPU/FPGA board serves as an example of a computer system that is already dynamic and reprogrammable today.
It requires software systems to adapt to and actively manage the resources. R-OSGi turns modularity into a programming model for distributed systems by transparently turning OSGi services into remote services. At the same time, failures arising from network unreliabilities are mapped to module and service unload events which applications written for OSGi are already prepared for to handle gracefully. This permits developers to focus on the functional decomposition of the application into modules and defer the decision where to put the boundaries of distribution to deploy-time or even runtime. The Cirrostratus module runtime system for the cloud drives the idea of location-transparency for software modules even further by providing a single system image on top of a variable set of machines in the cloud. Modules installed on one machine are uniformly visible and accessible on all machines. The system is able to provide the communication facilities to invoke services crossing machine boundaries and to create replicas of services on demand while keeping the internal state of modules consistent across the entire system. Without requiring adaptations to the code, modular software running on Cirrostratus has become completely fluid within the constraints of the modular decomposition.

The three contributions indicate how software modularity—not only applied to the source code as traditionally but understood as a runtime concern—is the system design principle required to cope with the challenges of increasingly dynamic environments. Most importantly, it does so in a generic way and beyond ad-hoc solutions.
Zusammenfassung


Drei Systeme werden in dieser Dissertation präsentiert, die auf dem OSGi Standard für dynamische Module in der Java-Programmiersprache basieren. Sie erweit-

Acknowledgments

I started my PhD program in a time where people talked a lot about work-life-balance. Thus, I balanced day by day, sometimes stumbled, and most of the time landed on the work side of life.

I had the privilege to be part of a truly great group, the IKS group at ETH Zurich. I would like to thank the members of the group for supporting me in my enthusiasm for research. In particular, I owe my gratitude to my advisor, Prof. Gustavo Alonso, for his guidance and advice but also for having the confidence in me and letting me explore my ways when I needed to. I further want to thank Michael Duller for being a great friend and for all the discussions that often triggered my creativity and in the end lead to new research ideas. When my group joined forces with the database and operating systems people to become the Systems Group, I could profit from the fantastic opportunity to work in a setup with a diverse set of talented people of different backgrounds and learn about their mindsets. This has encouraged me various times to step out of my comfort zone and challenge myself, which I greatly appreciated.

In addition, I would like to show my gratitude to the members of the OSGi Alliance for giving me the opportunity to actively participate in the standardization efforts as an Invited Researcher and to work with true experts and at the same time wonderful people. Finally, I would like to thank the Swiss National Science Foundation (SNF) for supporting some of my projects through the NCCR MICS initiative and the ProDoc Enterprise Computing program as well as Microsoft for supporting one of my projects through the Innovation Cluster for Embedded Software (ICES).

Fortunately, I still also had a life during my PhD time. My work would not have been possible without the loving support of my wife Anita and our little son Marc. The same is true for the support of my family back in Germany over all the years, and my family in-law in Switzerland. It has been a long way and it is a pleasure and honor for me to share the fruits of the work with all who supported me.

Austin, TX. May 2011.
Contents

1 Introduction 1
   1.1 Motivation ................................................. 1
   1.2 Contributions ........................................... 5
   1.3 Structure of the Thesis ................................. 6

2 Software Modules 9
   2.1 Principles of Software Modularity ...................... 9
   2.2 Historical Context ....................................... 11
      2.2.1 The Software Crisis ............................... 12
      2.2.2 Structured Programming .......................... 13
      2.2.3 Abstract Data Types ............................... 14
      2.2.4 Object Oriented Programming ..................... 14
      2.2.5 Pipes ................................................. 15
      2.2.6 RPC, CORBA, and RMI ............................... 16
   2.3 Logical Modularity ....................................... 17
      2.3.1 Ada .................................................. 18
      2.3.2 Modula-2 ............................................ 19
      2.3.3 Haskell ............................................. 21
   2.4 Physical Modularity ...................................... 21
      2.4.1 Shared Libraries .................................... 22
      2.4.2 Component Object Model (COM) .................... 23
      2.4.3 Java JARs ............................................ 25
      2.4.4 .NET Assemblies .................................... 26
      2.4.5 OSGi ................................................ 27
   2.5 Summary and Discussion .................................. 28

3 Juggle: Co-Managing Software and Hardware Modules 31
   3.1 Introduction .............................................. 31
<table>
<thead>
<tr>
<th>Section</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>6.2</td>
<td>Virtualizing an OSGi Framework</td>
<td>133</td>
</tr>
<tr>
<td>6.2.1</td>
<td>Local and Global Scope</td>
<td>134</td>
</tr>
<tr>
<td>6.2.2</td>
<td>Bundle Start and Meta-Resolving</td>
<td>135</td>
</tr>
<tr>
<td>6.2.3</td>
<td>Achieving Location Transparency</td>
<td>136</td>
</tr>
<tr>
<td>6.2.4</td>
<td>Providing and Consuming Services</td>
<td>137</td>
</tr>
<tr>
<td>6.3</td>
<td>Managing the State of Modules</td>
<td>138</td>
</tr>
<tr>
<td>6.3.1</td>
<td>Symbolic Execution</td>
<td>139</td>
</tr>
<tr>
<td>6.3.2</td>
<td>Adapting the Code</td>
<td>141</td>
</tr>
<tr>
<td>6.4</td>
<td>Managing elastic modules</td>
<td>146</td>
</tr>
<tr>
<td>6.5</td>
<td>Evaluation of the Scalability</td>
<td>147</td>
</tr>
<tr>
<td>6.5.1</td>
<td>Cirrostratus on a Cluster</td>
<td>148</td>
</tr>
<tr>
<td>6.5.2</td>
<td>Cirrostratus on PlanetLab</td>
<td>150</td>
</tr>
<tr>
<td>6.6</td>
<td>Performance Evaluation</td>
<td>155</td>
</tr>
<tr>
<td>6.6.1</td>
<td>Dacapo Xalan Benchmark</td>
<td>155</td>
</tr>
<tr>
<td>6.6.2</td>
<td>The Effectiveness of Elasticity</td>
<td>157</td>
</tr>
<tr>
<td>6.7</td>
<td>Use Case: Stendhal</td>
<td>160</td>
</tr>
<tr>
<td>6.8</td>
<td>Related Work</td>
<td>163</td>
</tr>
<tr>
<td>6.9</td>
<td>Summary and Discussion</td>
<td>164</td>
</tr>
</tbody>
</table>

7 Conclusions 167

7.1 Modular Systems 167

7.2 Directions for Future Work 170

Bibliography 171
Chapter 1

Introduction

1.1 Motivation

The environment for which software systems are designed and developed is changing towards increasingly dynamic settings. One example for this trend is the increasing importance of distributed systems. Since nodes can fail at any time and even the hardware driving the network can be unreliable or congested, every distributed system is inevitably a dynamic system. Besides the cases in which the hardware operates outside of its specifications, there are also other forms of impact on distributed systems which require the software to adapt. With the proliferation of cloud or utility computing, it is no longer reasonable to assume a static resource assignment to a deployment of a software system. For instance, in the context of cloud computing, the maintainer of a system might decide to add or remove resources at runtime to optimize for the current load of the system. This elasticity of the resource pool requires the software to be prepared for operating on a changing set of nodes and adapt its deployment.

Another example of a dynamic resource pool is the computer itself. With the entrance into the multi-core era with non-uniform memory local to the cores and high-bandwidth interconnect between the cores, computers have become akin to micro-scale distributed systems. Future computing devices are likely to have an even higher number of cores, including cores specialized for a specific purpose, and an increased number of possible configurations to run in. Since energy efficiency is becoming an issue in the system design, it is likely that not all cores will be permanently available any more but instead only enabled temporarily when there is a substantial demand from the software. As a result, future software is expected to adapt in a much higher degree to the no longer static underlying hardware and respond to the increasing dynamism of the platform with increased flexibility and the ability to adapt.
Clearly, software written for a single CPU and a single, contiguous, uniform address space is not feasible to be operated successfully in such an environment. Instead, it is necessary to segregate the system into discrete units which can operate much more independently of each other and, instead of communicating implicitly and arbitrarily through shared memory, design explicit interfaces between these discrete units. In software engineering, similar challenges in writing and maintaining source code resulted in the idea of software modules, which is by now well-established and pervasively used. Instead of permitting arbitrary interaction between code units, the concept of modularity mandates the encapsulation of code and the exposure of well-designed interfaces through which modules interact in a controlled way.

The broader vision expressed in this thesis is a principle enabling the creation of fluid systems, i.e., systems which can react to changes in their environment with seamless adaptation so that, without any further intervention from outside, they always operate in an effective manner. This is analogue to a liquid which can take any form to adapt to the shape of its container. In a liquid, this behavior is the result of a firm but not rigid binding between the molecules. The prerequisites for a software system to mirror this behavior are a discretization of the system into functional units of a useful granularity (where useful qualitatively means in alignment with the degrees of freedom of the platform, e.g., the number of nodes), and a binding between the functional units loose enough to
permit a flexible allocation of functional units to hardware resources and general enough to enable a variety of communication protocols to be used.

The granularity of modularization yielding the highest degree of freedom is not the most fine-granular level possible, like, e.g., single objects. Figure 1.1 shows a class dependency diagram of Apache Tomcat, a widely-used implementation of an application server. The illustrations has been created with a dependency analysis tool developed as part of a student project [Meier2009]. Class names are arranged in a circle. The dark area in the periphery of the picture is the result of the names of the numerous class names overlapping. Dependencies between classes are represented through a line between the class names. The red area in the middle of the picture is the result of all the dependencies overlapping. As this illustration indicates, single classes as a form of modularity are not predestined as building bocks for flexible systems due to their usually high degree of coupling, e.g., through inheritance and cross-referencing.

Figure 1.2 shows a more coarse-granular modular view of Apache Tomcat, the black names now represent Java package names. Each package represents groups of classes which have a functional cohesion. Clearly, the complexity has been reduced by now dealing with fewer and less intertwined entities. However, it is not clear that this structure allows for a factual decomposition of the system into functional units. The ideal decomposition envisioned is the one that yields a high cohesion of the content of the
modules while keeping the coupling between modules low. In this regard, the notion of modules for building flexible systems is identical to what software quality metrics in software engineering mandate. In the analogy of liquids, it is not the smallest unit which forms the building blocks for a successful modular decomposition. A fluid exists because of weak interactions on the inter-molecular level whereas the interactions on the level of single atoms or even elementary particles are, in relation, stronger.

It is important to understand that implementing a modular design effectively means to comply with certain restrictions imposed by the modularization. For instance, modules cannot interact arbitrarily but only through well-defined interfaces. The interface, however, is a partition to both the module that is binding and the module to which it binds to. As a direct consequence, a module should not make any assumptions about the presence or absence of a particular other module implementation in the system, in particular not about the modules it is bound to, and only act locally. The advantage of source code modularization in return is the increased flexibility of software modules in terms of composition and code re-use. This form of logical modularization, however, is eliminated during the compilation and linking process and the resulting binary is again monolithic and inflexible in terms of deployment. The theme of this thesis is that software modularity, when applied to the physical deployment of software systems and carefully enough to result in a loose coupling of modules, is able to provide an equivalent degree of flexibility to the deployment of a system at runtime. Thereby, the focus of modularization shifts from a programming language concern towards a runtime system concern.

As the example systems presented in the thesis show, this simple approach is powerful enough to provide location-transparency, which, e.g., permits that modules change their physical location without requiring changes to the code or interrupting the rest of
Chapter 1. Introduction

a running system. For instance, Module C in Figure 1.3 could reside on a different machine than Modules A and B. It also permits the substitution of different implementations at runtime and thereby simplifies the integration of hardware accelerators into software systems, illustrated by Modules D1 and D2 in Figure 1.3. Finally, modularization exposed and understood by an intelligent runtime systems enables a fluid behavior of software modules, i.e., the modules of an application can seamlessly adapt to changes of an elastic resource pool.

1.2 Contributions

This thesis envisions physical modularity as the distinctive design principle for building flexible software systems for future platforms. In detail, the thesis makes the following contributions:

A Characterization of Software Modularity. As with many ideas that evolved over a longer period of time, there is no clearly authoritative characterization or even definition of software modularity. Based on several informal descriptions of the properties of software modules and the discussion of various implementations of modularity, we develop a general notion and characterization of software modules on which the remainder of the thesis is based upon.

Modularity as a Programming Model for Distributed Systems. We introduce the idea of using the module boundaries as potential distribution boundaries. Thereby, a system can be developed in its entirety and tested without distribution involved. Concrete deployments can be created by placing modules onto different machines and using a module-aware runtime system to take care of the communication. Potential faults arising from network communication can be mapped to module unload events, which applications developed for dynamic module systems are already prepared to handle gracefully.

Software Modules as the Foundation for Elastic Systems. Elasticity refers to the ability to dynamically provision and discard resources as need dictates, typically from a pool of resources shared among potential users. Hardware elasticity is often achieved through virtualization. Software elasticity—the ability of a software system to run on an elastic hardware platform—is, unfortunately, more difficult and remains an open issue in its more general form. In this thesis, we show how software modules which expose
their structure to an intelligent runtime system can form the foundation for implement-
ing the same degree of elasticity in software known from cloud platforms.

1.3 Structure of the Thesis

The remainder of the thesis is structured as follows:

Chapter 2 introduces the concept of modularity. It describes a brief history of modularity in programming languages and systems and introduces a characterization of software modules. It furthermore motivates a shift of focus from modularity as a language construct, as it was traditionally discussed, towards modularity as a runtime concern.

Chapter 3 presents an outlook into designing software for dynamic computer systems in which hardware resources are dynamic and require an active management from the application. The prototype system builds atop an FPGA board—a dynamic hardware platform already available today—in which the reprogrammable hardware can be used to selectively accelerate the modular application without requiring any changes to existing code. Our approach uniformly manages software modules and the binary images used for dynamically reprogramming the hardware through the same module management system. The resulting system, Juggle, is evaluated using a TripleDES encryption co-module for Java and a Xilinx Virtex-II Pro FPGA.

Chapter 4 describes the challenges of turning OSGi into a programming model for distributed systems. The result of this effort is R-OSGi, a conceptual extension of the OSGi standard for using services as building blocks for distributed systems. R-OSGi can seamlessly turn OSGi services into remote services while, different to other remote component systems, it does not attempt to hide faults arising from network communication errors but passes them to the application as regular service and module unload events.

Chapter 5 presents three applications built atop R-OSGi and taking advantage of its capabilities to implement advanced functionalities in distributed systems. The first application is a tool for providing drag-and-drop distribution of software modules without any need to change the code of the modules. The second application provides a practical solution to the problem of updates in distributed systems by extending the OSGi
concept of versioning. The last application, AlfredO, introduces a more coarse-granular but semantically richer distribution model in the form of detachable tiers atop R-OSGi.

Chapter 6 covers the connection between software modules and cloud computing. We present a cloud computing runtime, Cirrostratus, which turns conventional software systems into elastic applications and manages their dynamic deployment by exploiting their inherent modular structure. The approach only requires the programmer to structure the application into modules. The Cirrostratus runtime system then treats the modules as the units that can be dynamically provisioned, distributed, migrated, and replicated.

Chapter 7 concludes the thesis and puts the insights gained from building the different systems into the context of the original vision of fluid systems. Furthermore, it discusses directions for possible future work.
1.3. Structure of the Thesis
Chapter 2

Software Modules

2.1 Principles of Software Modularity

A software module is primarily a self-contained functional unit which consists of data and functions, objects, or other entities of the programming language. The content of a module is expected to be not arbitrarily chosen but closely related and fulfilling a common purpose. The self-containment property, however, does not automatically mean isolation. In many applications of module technology, it is sufficient for modules to be declaratively self-contained, i.e., in addition to their own content, they are only allowed to use entities for which they explicitly declare their dependencies. Modules are units of encapsulation. This means, not all content of a module is exposed to the outside. Instead, the principle of information hiding demands that only contractual interfaces are exposed whereas as much as possible the concrete implementation is hidden.

Designing modular systems is the art of decomposition and composition. The decomposition is the result of a top-down approach of taking apart a problem to form a set of smaller and less complex subproblems. Originally, David Parnas described modules to be the result of a responsibility assignment [Parnas1972]. Therefore, software modularity can be considered an implementation of what was later called separation of concerns [Dijkstra1974]. The resulting modules which encapsulate the subproblems are expected to largely be able to operate independently of each other.

Ideally, the correctness of a modular system depends only on the correctness of its modules and the correctness of a single module does not depend on other modules. For instance, the effects of faults occurring in a single module should ideally be restricted to the scope of the module in which the fault occurs. Due to sharing through dependencies, this cannot always be guaranteed. However, it is expected that the effects are at least restricted to the module and its dependents and under no circumstances affect unrelated
2.1. Principles of Software Modularity

parts of the system. In the same spirit, the effect of small changes to the problem specification should only affect a single module or a small number of modules. Bertrand Meyer called the first criteria the Modular Protection and the second the Modular Continuity [Meyer1988]. In practice, both are closely related. The mandate that a small change only affects a single module is a direct consequence of the separation of concerns. As long as the change does not affect more than one concern simultaneously, in a well-designed modular system the immediate change can only affect a single module. What remains is the common principle of localized reasoning and behavior. A module should not reason about and should not be affected by parts of the system to which it does not have a declared dependency. This means in particular that there is no implicit sharing hidden from the module layer. In practice, this assumption cannot always be made, for instance, where different modules depend on the same physical resource.

The composition is the result of a bottom-up approach of combining modules into new systems. This means that modules should be designed for reuse in systems different from the purpose that drove their original creation. In practice, composition requires the designer of a system to limit the dependencies. As David Boundy expressed it in a Software Engineering Note [Boundy1991]:

> The good programmer writes software that can be reused, even if he doesn’t see a reuse immediately on the horizon. His experience with things seen and faith in things not seen assures him that someone from the next hallway, reasonably soon, will be asking, "Do you have a module that ..." He also knows that writing reusably will force him to define clean interfaces.

The quality of a module is traditionally assessed in terms of coupling and cohesion [Stevens1974]. Coupling determines to which extent and through which mechanisms one module depends on other modules. An example of high coupling is, for instance, content coupling in which one module depends directly on internals of another module. Low coupling can be implemented, e.g., through indirect invocation patterns like events or message passing. Instead of directly depending on other modules, the module posts events (which constitute independent entities and can be defined by an unrelated third module) and no physical coupling exists between the sender and the receiver or an event. It is generally advised to avoid high coupling since it defeats the purpose of modularity. If two modules are tightly coupled, one cannot be changed without the other, reused or deployed independently, and the behavior of one depends inherently on the implementation of the other. This can either be a result of poor programming (accidental coupling) or have systematic reasons (essential coupling) originating from the nature of the problem. In the latter case, however, it can be argued that the two
concerns that under no circumstances be implemented independently are actually two ends of the same concern and should consequently better be implemented in the same module.

The degree to which the functionality exposed by the content of a module is related is determined by cohesion. It is generally advised to design modules to be highly cohesive, ideally implementing only a single functionality (or concern) to improve the maintainability and understandability of the module. If a module exhibits a low cohesion, e.g., when entirely unrelated functions are grouped into a single utility module, the reuse of one function draws in a large number of unwanted functions, which can be considered as an unnecessary source of entropy.

In practice, there are programming paradigms which help to both increase the cohesion of the module while keeping the coupling low. For instance, the actor model [Hewitt1973] is often considered to be such a paradigm due to its fine-granular modularization and exclusively indirect invocation. However, there are also cases in which non-functional requirements require a concession in one or both of the two dimensions. The most intuitive example here is performance, e.g., in the design of operating systems. The micro-kernel architecture [Liedtke1995] exhibits all the benefits of modularity and the coupling between kernel and modules is low since it only relies on system calls, which can be considered messages. Still, the monolithic kernel design is still the dominating form due to the communication overhead between the micro-kernel and the user-space modules. If modularity is applied at all in mainstream operating systems, as it is the case in the Linux kernel, it is with a closer form of coupling where kernel and modules reside in the same address space, can share entire data structures, and thus communicate much more efficiently.

### 2.2 Historical Context

The software crisis of the 1960s clearly was a main driver for paradigms like modularity to gain traction. Other ideas like structured programming, abstract data types, and object-oriented programming emerged in parallel with modules (Figure 2.1) and are discussed in the following sections. Even though these ideas share some properties with modules, the different concepts largely co-existed and still do. It became clear that none of the approaches was the silver bullet to defeat the software crisis and such a thing is unlikely to even exist due to the inherent essential complexity in software development [Brooks1987].
2.2.1 The Software Crisis

In the end of the 1960s a fundamental frustration with the state of the art in software programming emerged, which was finally proclaimed to be a software crisis. The observations of this time were that the overall quality of software too often fell short with its ambition. On the one hand, this applied to the productivity of the software programmers. Projects were often over budgeted time and cost limits or, even worse, the results were not even delivered to the stakeholders. On the other hand, also the quality of the delivered software often did not meet the expectations. Software was poorly designed, not delivering the intended functionality, error prone, and difficult to maintain.

The reasons for this phenomenon were not well understood at that time. However, it was presumed that the complexity of hardware and software and the size of the state space in software systems and the resulting difficulties to do effective testing had a strong influence. As Edsger Dijkstra stated in his 1972 Turing Award lecture [Dijkstra1979a]:

> The major cause of the software crisis is that the machines have become several orders of magnitude more powerful! To put it quite bluntly: as long as there were no machines, programming was no problem at all; when we had a few weak computers, programming became a mild problem, and now we have gigantic computers, programming has become an equally gigantic problem [...] Is it a wonder that we found ourselves in a software crisis?

At the same time, it was inevitable that another strong factor was the lack of widely accepted programming methodologies or even tractable standards and poor project and change management. In 1968, a NATO-sponsored conference in the German city of
Garmisch was held that provocatively dealt with the question of how to do *Software Engineering* [Naur1968], thereby assessing the need of turning the formally mostly undefined process of software development into a dependable engineering discipline. In the course of the conference, Douglas McIlroy introduced the notion of *software components*, which, in his formulation, correspond to the idea of software libraries and therefore modularity in that they try to improve the quality of software through encapsulation and reuse.

### 2.2.2 Structured Programming

The paradigm of structured programming encourages the developer to plan and enforce a logical structure for a program that is written. This logical structure can be visualized in a flow chart and corresponds to the control flow in the program. Structured programming strongly opposes the usage of branch or jump instructions on which early versions of Fortran, Cobol, or Basic were based upon as the only way to deal with control flow. This culminated in the discussion about the elimination of the goto statement [Dijkstra1979]. The claim was that structured programming would result in more readable and understandable code, which in turn leads to a higher quality and better maintainability.

The theory behind structured programming is that every computable function can be implemented in a programming language that combines subprograms using sequence, selection, and repetition [Boehm1966]. Sequence means the ordered execution of statements. Repetition means that a statement is executed repeatedly until the program has reached a certain defined state. The concept is equivalent to todays conditional loops. Selection means that statements are conditionally executed depending on a certain defined state of the program. Manifestations of selection in todays programming languages are, e.g., if-clauses and switch statements. Structured programming is nowadays a widely accepted principle and most programming languages apply it. The most notable deviation from the principle is exception handling when implemented through try/catch clauses, which still correspond to jumps.

Frequently, structured programming is also associated with the rule that code sequences may only have a single, well defined entry point. This leads to the concept of procedural programming and subroutines, which already can be considered as a weak form of modularization. Modularity to a great extent relies on structuring programs, even though the focus is not on control flow.
2.2.3 Abstract Data Types

Abstract data types (ADT) [Liskov1974] are a way of decoupling a more abstract behavior from a concrete implementation. ADTs are centered around a data structure which is not directly defined but indirectly through the operation on it. For instance, an ADT can be an abstract collection type representing all concrete data structures with support for addition and removal of elements. Through the common abstraction, the different concrete implementations become substitutable. Programs requiring support for such a data structure can hence be written against the abstract concept and the choice of a concrete implementation can be taken and altered later on without affecting the rest of the program code.

Typically, the compiler does type checking between the abstract data type and the concrete implementations and enforces compliance. Conceptually, the idea of interfaces is congruent with the idea of abstract data types, even though the notion of ADTs appears more frequently in the context of data type libraries. In many modern languages, interfaces are used as a more general form of abstraction from concrete objects whereas ADTs are usually associated with traditional data abstraction.

2.2.4 Object Oriented Programming

The first occurrence of objects as a formal concept for programming was in Simula 67. Since Simula was primarily designed for discrete simulation of real-world phenomena, the introduction of classes as blueprints for creating object instances has the purpose of better reflecting the behavior of real-world objects.

The are plenty of different formal definition in the literature of what the fundamental and transcendental properties of an object are. Grady Booch uses the definition [Booch1991]:

An object has state, behavior, and identity; the structure and behavior of similar objects are defined in their common class; the terms instance and object are interchangeable

The idea here is to understand the object as an encapsulation of data (state) and a collection of subroutines or methods operating on the data (behavior). In this, object-oriented programming is primarily a data-centric approach. The postulate of identity is not formally fulfilled in all object-oriented programming languages.

The internal state of an object is typically hidden and only accessible through subroutines (methods). Besides this heritage from abstract data types, objects also implement the idea of hierarchical types. Classes can be derived from other classes and create
specialized subtypes thereof. This enables a specific pattern of code reuse in which common functionality is implemented in (possibly abstract) base classes and reused in all derived subclasses.

Frequently, objects are characterized by a different set of three properties: inheritance, encapsulation, and (subtype-) polymorphism. However, the *conditio sine qua non* for polymorphism is object identity.

Object-oriented programming is a modular approach in the sense that classes in general can be written and maintained independently of each other. However, classes are a fine-granular representative of modularity. In practice, the manifold dependencies created through referenced or inherited types limit the reuse of single objects. Therefore, some object-oriented languages support coarse-grained modularity in the form of collection of classes.

### 2.2.5 Pipes

Pipes were originally developed for the Unix command shell by Douglas McIlroy, even though the underlying design principles are similar to the communication files used in the earlier Dartmouth Time-Sharing System. The pattern distinguished between pipes, which are character streams turning the output of the previous process into the input of another process, and the processes themselves which were later called *filters*. A linear composition of filters and pipes, as usually created from the command line shell, is called a *pipeline*. A filter has to expect that its output becomes the input of a different filter and hence not make any assumptions about its environment. In particular, its behavior should not depend on any order in the pipeline or the presence or absence of a specific other filter in its up- or downstream neighbors. Filters cannot share any state but only operate on streams of data. As McIlroy is cited in [Salus1994]:

> This is the Unix philosophy: Write programs that do one thing and do it well. Write programs to work together. Write programs to handle text streams, because that is a universal interface.

From an architectural point of view, the pipes and filters pattern is a specific form of co-routine with a particular emphasis on non-hierarchical control flow. Filters are modules and are strongly based on the principles of encapsulation, composable, and locality. As a result, filters are easy to reuse; many of the original Unix tools have stood the test of time and are still frequently used. Composition is easy and can happen either declaratively through the command shell syntax, or imperatively in other programs. Finally, filters are a good example of modules for which the behavior of more complex
compositions thereof can often be understood by observing the behavior of the single modules. However, the simplicity in composition comes at the cost of the generality of the interface between filters. The character stream can be considered as the lowest common denominator, which often lead to additional parsing effort for the filters. Furthermore, there are cases in which filters want to know some information about their environment and in practice infer this information. For instance, in modern Linux distributions the output of the `ls` command in fact depends on the context in which the command is used and can lead to differently formatted output.

To overcome such limitations, more recent implementations of the pipes and filters pattern like the Windows Power Shell exchange semantically rich objects instead of plain characters through the pipes. This allows for more specialized behavior of commands without violating the principle of local reasoning.

### 2.2.6 RPC, CORBA, and RMI

The shift from unstructured to structured programming and procedural programming introduced a notion of modularity through procedure or function definitions. The functions here do not only serve the purpose of making the control flow of an application easier to understand but also provide a very basic level of encapsulation and isolation since in a clean functional or procedural design the functions cannot share arbitrary data but have to pass it explicitly through arguments. With remote procedure calls (RPC) [Birrell1984], this property is exploited to turn procedures into a programming abstraction for distributed systems to allow the modular design.

The communication in RPC is treated as an orthogonal concern to the decomposition of functionality through procedural boundaries. What was designed to be a local function call is transparently turned into a remote function call by replacing the actual function through a stub. This stub is used to captures the arguments and transforming them into a wire format (marshalling). After the arguments have been transmitted over the network and unmarshalled, the function call is performed on the server on behalf of the client. The result runs through the stack in reverse order and is eventually returned by the stub. The inherent coupling between client and remote procedure is often made explicit through the use of an interface written in a language-neutral interface definition language (IDL). For many RPC systems, there are interface compilers to generate client stub and a server-side skeleton from an IDL.

The Common Object Request Broker Architecture (CORBA) [OMG1995] introduced an object-oriented version of the RPC concept in which calls to remote objects are mediated through an object request broker (ORB). Thereby, objects as a more coarse-
granular unit than single procedures became the unit of design for distributed systems. At the same time, the focus shifted towards a middleware infrastructure that not only enables the interoperability and remote invocation between distributed modules but also provided common system-wide facilities such as transactions or object persistence.

Remote Method Invocation (RMI) [Sun2004b] implements an approach comparable for methods of objects in the Java language. RMI prior to Java 1.4 required the explicit creation of stubs by defining remote object interfaces and running them through an RMI compiler. In later versions, this mechanism was obsoleted through the use of dynamically generated client-side proxies.

A often perceived disadvantage of RPC-like mechanisms, however, is the inherent failure transparency of the approach [Waldo1994]. Since local functions are designed to always perform reliably and with a predictable latency, the substitution of local with remote procedure calls potentially violates the semantics of the application by introducing new failure patterns not experienced in a local setup. RMI technically avoids this problem by mandating that remote objects not only are only reachable through facets which extend a specific marker interface (java.rmi.Remote) but all remotely accessible methods must additionally declare to throw java.rmi.RemoteException. Thereby, callers of remote objects necessarily become aware that the invocation could fail, e.g., due to communication errors.

## 2.3 Logical Modularity

Modular software design was originally motivated by software engineering challenges. Developing complex systems in an unstructured, monolithic process turned out to be prohibitive. As a consequence, language designers started to introduce support for partitioning the code into logical units, into modules. The following sections discuss the module support of three concrete languages which are representative for many other language designs: Ada, Modula-2, and Haskell.

Logical modularity or *namespace modularity* operates entirely in the implementation phase of an application as a mechanism to facilitate the re-use of code while at the same time applying a structure (e.g., a hierarchy of modules) to avoid namespace collisions. In contrast to physical modularity, logical modularization is eliminated in the linking process and the resulting binary is no longer modular. Therefore, only static composition of logical modules is possible, the binary resembles the choice of a single composition and does not further compose. The original separation into namespaces can be completely eliminated. In some languages, the namespaces are still observable
package MyAlgorithms is

    type Integer_Type is range 0 .. 999_999_999_999_999_999

function Fib (n : Integer_Type) return Integer_Type is
begin
    if n = 0 then
        return 0;
    elsif n = 1 then
        return 1;
    else
        return Fib (n - 1) + Fib (n - 2);
    end if;
end Fib;
end package;

Listing 2.2: Modules in Ada

from the binary, e.g., through mangling of the symbols, but no longer fulfill any purpose other than avoiding name clashes.

2.3.1 Ada

Ada is a procedural, static and strongly typed programming language with a focus on safety and reliability through compile-time checks. It was approved as an ANSI standard [DoD1983] in 1983 and later also as an ISO standard ISO-8652 in 1987. The 1995 revision of the Ada language introduced object-oriented programming and dynamic dispatch.

The module system in Ada is a typical representative of namespace modularity. Modules in Ada are declared with the package keyword which acts as a scope and is terminated by the end keyword, as shown in Listing 2.2. Types, functions, and procedures declared in the package scope become content of the module and visible to importers unless the visibility is explicitly restricted, e.g., to be private to the module. Therefore, Ada implements information hiding through restriction of visibility and not through permission.

Modules can be made available to the current namespace using the with keyword.
In this case, all accesses to included types, procedures, etc. (entities) have to be fully qualified with the name of the module. Another option is the use keyword. In this case, the module namespace is inlined into the current namespace. Included entities become indistinguishable from those declared in the current namespace. However, used packages cannot hide (shadow) entities declared in the current namespace. Access to conflicting entities still need to be fully qualified. A third option is aliasing through the package X renames Y construct, usually within restricted scopes, e.g., within procedures.

Ada provides a declaratively self-contained module system which composes through namespace inclusion. The compilation of Ada modules happens separately (one module can be compiled without also compiling all other modules it depends on) but not independently (one module can be compiled without even knowing the content of the modules it depends on) as in languages like C or C++. As a result, Ada is able to preserve its strict type checking across the boundaries of modules.

### 2.3.2 Modula-2

Modula-2 is a statically strongly typed procedural programming language developed by Niklaus Wirth in 1978. The purpose of Modula-2 was to be the sole language for programming the Lilith [Wirth1981] personal workstation. As the name suggests, modularity was a key design principle of the language. Modules in Modula-2 are strictly separated into two parts, the definition module which fulfills the purpose of an externally visible interface, and the implementation module which contains the code internal to the module. In this regard, the motivation of the module system of Modula-2 is closely related to the one of abstract data types. However, Modula-2 cannot support multiple implementations for the same definition (in the same compilation process) since the two entities are linked by having the same name.

Listing 2.3 shows the definition and the implementation of a module. Due to the strict separation and the abstraction, modules can also be compiled separately and without making a concrete choice of implementation to compile against. Therefore, implementations of the same definition can be exchanged in between compilation processes. A module can by construction only have tight couplings with other definition modules but never with implementation modules. The implementation module is a stringent unit of encapsulation and its scope is entirely private except for entities exposed through the definition module\(^1\). It therefore implements information hiding through granting

\(^1\)The variant of the language described in the second edition of Programming in Modula-2 [Wirth1983] required an explicit EXPORT clause in definitions, later versions dropped this requirement
Listing 2.3: Definition and implementation module in Modula-2

exposure and not through restricting visibility.

Modules are groupings of variables, constants, types, and procedures, here collectively referred to as entities. Importing entities from modules happens selectively through the FROM Module IMPORT a,b,... construct. To avoid name clashes, exports can be marked as qualified, which has the result that importers always have to use the fully qualified name (e.g., Module.a).

Besides global modules which can fulfill the purpose of libraries, Modula-2 also supports local modules, which are modules nested inside other modules. These local modules do not have a definition since they can anyway only export entities to the surrounding module and do so through an EXPORT declaration. Imports in turn are restricted to entities of the outer module and everything what this module itself has imported.

due to its redundancy
module MyAlgorithms (fib) where

fib :: Integer -> Integer
    fib 0 = 0
    fib 1 = 1
    fib n = fib (n - 2) + fib (n - 1)

Listing 2.4: Module in Haskell

2.3.3 Haskell

Haskell is a representative of a purely functional language with module support. Module declarations (Listing 2.4) can either expose all entities, or specify a list of entities to be exported. Entities can be values, field names, class methods, operators (need to be parenthesized), algebraic data types, or type synonyms. Additionally, other modules can be exported, which means that Haskell is one of the few examples of a language that supports re-exporting of entities that are imported from other modules. It is more common to have the requirement that modules are only permitted to expose their own content, which makes the coupling between modules explicit and giving each module the individual freedom to bind to a provider of a declared dependency. The Haskell approach of re-export, however, can be used to enforce the uniform use of the same dependencies among a group of modules.

Imports are declared with the import declaration that can occur in different variations. By default, a module import includes all exported entities of the imported module into the current namespace. This behavior can be changed by explicitly stating a list of entities to be exported. A particularity of the Haskell module system is that also an explicit exclude of entities is supported in the form of the hiding directive.

When the import is declared as qualified, all accesses to imported entities have to be fully qualified. The last variant support local aliasing through the import qualified X as Y form. It can be used to abbreviate qualified accesses while still preventing name clashes with entities of the importing module.

2.4 Physical Modularity

Logical modularity requires the developer to make a choice and pass a specific composition of modules to the compiler and later to the linker to create a single monolithic binary. In contrast, physical modularity preserves the module boundaries. Since the
modules can then be individually deployed, physical modularity is also referred to as 
development modularity. The advantage of physical modules is that the act of composi-
tion can be deferred to the load- or even runtime of an application. This enables, e.g.,
extensibility of applications through **plugins**.

A specific challenge of physical modules is versioning and binary compatibility. Since in essence every user can compose an application out of different modules, it is of particular importance to be able to identify if two modules which declaratively match also can factually inter-operate at runtime.

In the following sections, different concrete implementations of physical modularity are discussed.

### 2.4.1 Shared Libraries

In many languages like C, C++, or Pascal, the compiler translates the source code into machine code (or an intermediate format) and creates an object file from the trans-
luated code. The source files can be compiled independently (leaving dependencies un-
resolved) or separately (resolving against concrete providers of the dependencies but not requiring the compilation of the dependencies as well). The latter requires the compiler to understanding the modularity of the sources. In isolated compilation, addresses for jumps and calls are left relative since the final addresses cannot be assigned in isolation but need coordination between the modules that form the final application. It is then the responsibility of the linker in a second step to combine and link several objects into a binary module, which can be an executable or a library. In this step, the linker resolves all previously unresolved references and assigns fixed or relocatable addresses, the latter are addresses relative to a common base which can be adjusted at runtime.

A shared library in essence is a module exporting a set of symbols (i.e., global vari-
ables, functions, etc.) which have been written once to be re-used in new applications. Libraries can be used in different phases. Static shared libraries are not much more than an archive of one or more object files equipped with a single shared symbol table. In the linking phase, the libraries are statically linked into the resulting binary just like any other object file. This facilitates reuse in a logical sense since the same library can be linked into many applications but once linked the code is physically duplicated. In particular, two applications based on the same library at runtime load the same (duplicated) code so that it exists twice in memory.

In contrast, dynamically linked shared libraries are singleton entities and the same library can be linked at runtime into multiple applications. The linking process is similar to using static libraries except that the linker does not link the dynamic library into the
Chapter 2. Software Modules

binary but allows the references to remain unresolved. The shared library needs to provide position-independent code, which means that it can be loaded into any address space. Global variables defined by a dynamic link library are not allowed to be accessed directly but indirectly through a global offset table (GOT). Functions are not called directly either but through a Procedure Linkage Table (PLT). These indirection tables are created at runtime when the dynamic linking takes place. The GOT is always fully initialized and populated whereas the PLT allows for lazy loading.

In the case of ELF [TIS1995] binaries, a program using dynamic shared libraries contains an entry in the INTERPRETER section which points to the dynamic linker. The operating system loads the binary into memory and then passes the context to this program which performs the necessary steps of linking. The required shared libraries are referenced by their name and looked up by the loader from the library path. Alternatively, the application can also explicitly use dynamic shared libraries by calling `dl_open` in unix.

Shared libraries in Unix are in general versioned through their file names. For instance, in Linux and BSD the version number is appended to the name of the shared object file, e.g., `libc.so.2.2`. ELF systems ignore the minor version number and consider all minor revisions to be binary-compatible. Major revisions are considered incompatible. Hence, it is often a workaround to create symbolic links in the file system which map a certain library to older versions of the library to which it is backwards-compatible. HP-UX additionally supports intra-library versioning through standardized symbols in the libraries which contain version information [HP1997]. Thus, it can be detected if an application tries to be linked against a version of a library which is older than the one it was compiled against.

### 2.4.2 Component Object Model (COM)

COM [Box1997] is a binary interface standard developed by Microsoft to enable the communication and dynamic object creation between a range of different applications on the Windows platform. It is language neutral and supports interaction between components written in different languages. The communication follows a client-server pattern, the COM server is a dynamic link library (DLL) or executable offering functionality through one or more COM interfaces. Components in COM have a globally unique identifier, the Class Identifiers (CLSIDs) and each of the interfaces exposed have their globally unique Interface Identifier (IID).

A COM component has to implement and provide at least the `IUnknown` interface. This interface provides methods for controlling the life cycle of the component through
reference counting (AddRef and Release) and a method QueryInterface for accessing other interfaces provided by the component. The latter, however, does not fulfill the purpose of a reflection on the component in that it would return a list of all implemented interfaces. Instead, it operates with semantics comparable to a dynamic case. Clients pass an Interface ID to QueryInterface and in return either get NULL if the requested interface is not provided, or a reference to the interface. In this sense, a client has to know in advance which interfaces it expects from a COM component.

The reference passed back in case of a successful binding to the facet of the component is a virtual function table from which the functions of the interface can be directly invoked. This keeps the overhead low for invocations to components residing in the same address space while still providing an indirection for the case that the component resides in a different address space. With DCOM (Distributed COM) [Brown1998], this can even be a different machine. Notably, the memory layout of the virtual function table equals a purely virtual class in C++ so that COM components can be seamlessly embedded into the language in which many of the Microsoft applications are developed. However, COM can also inter-operate with languages that do not provide support for function pointers. For instance, if COM components are called from (pre-.NET era) Visual Basic the calls take the path through the IDispatch.invoke interface that components are encouraged to implement. This type of call is similar to reflective method calls in languages like Java or C#. Even though IDispatch.invoke is slower, it sometimes is the preferred way of calling even from C++ when lazy binding is favorable over the eager binding through QueryInterface.

Since the binding between components relies on the interface IDs, it provides a simple form of versioning by avoiding that clients and servers with different versions of an interface can bind at all. It formally also implements strong typing since interfaces are immutable. The interfaces of a COM component can be described to MIDL, an extension to the Interface Description Language (IDL) [Exton1997]. Since COM is a binary standard, it is however not mandatory to provide an MIDL description with the component. More often, the component is described by binary interface definitions compiled from the IDLs and provided through type libraries. These type libraries additionally provide other meta-data about the component and can either exist as independent files, or embedded into the component file. If a type library is provided, tools can generate language bindings in different languages for the development of new client applications.

In Windows, COM classes, interfaces, and type libraries are registered in the Windows Registry and can be looked up by their globally unique identifiers. COM uses the registry information to either locate the correct local library for each COM object, or find the network location for a remote server. If the server is remote, the COM frame-
work takes care of the marshalling of arguments and transparently redirects the call to the remote machine. For optimization, components aware of distribution are facilitated to provide their own custom marshalling.

### 2.4.3 Java JARs

The Java virtual machine does not only support logical modularity in the form of packages but also physical modules in the form of Java Archive (JAR) files [Sun1994a]. By default, the JVM is launched with the name of a main class as an argument and expects to find this and all dependency classes in the class path. Without any further intervention, the class path is the directory from which the JVM is invoked. In Java, the logical modularity is strictly required to be reflected by the structure in which the class files are deployed. More precisely, the package structure (recommended to be in reverse domain name notation) is mapped to a hierarchy of folders on the file system. Classes not adhering to this convention will either fail to be found in the class path, or fail to load. A JAR is essentially a ZIP archive [Katz1989] (compression is optional) of a file system tree, optionally with additional meta-data in the form of a manifest file. It can not only contain class files but also other resources and, if declared in the manifest through the JAR-internal class path, even nested JAR files. JARs can be put into the class path and play the same role as a directory entry. The manifest file can contain arbitrary key-value pairs in different sections but only the signatures for signed JAR files are checked and enforced at load time. All other attributes have informative character. Most notably, the JAR file does not support any form of versioning. Furthermore, there is no mechanism to declare dependencies between JAR files. It is up to the entity launching the virtual machine to ensure that the class path is resolvable.

In essence, JAR files lack many properties which qualify module systems. It is therefore not possible to effectively manage JAR files in any form. One particular disadvantage of the JAR file concept and the class path is that the order of entries in the class path is significant. When the class loader tries to lookup a class from the class path for the first time, it linearly scans through the class path and loads the first matching class found. If a class of the same fully qualified name is provided in more than one module, the first one wins, which can lead to different results if the order of the elements in the class path is altered. Furthermore, there is no support inside the virtual machine to load the content of JAR files in a controlled way at runtime. Therefore, applications which require runtime-extensibility often either implement their own ad-hoc solution for loading modules or resort to more capable module management systems like OSGi.
2.4.4 .NET Assemblies

.NET Assemblies [Microsoft2011] are the physical modules for the Microsoft Common Language Runtime (CLR), which is a stack-based virtual machine. Interoperability between different languages was a major design goal for the CLR and in this regard it can be considered the conceptual successor of COM while at the same time providing a complement to the Java platform. The languages available on the CLR do not only involve object-oriented languages like C#, but also procedural or functional languages. Therefore, unlike the Java virtual machine, the CLR virtual machine provides support for functions outside of class definitions. The CLR typically compiles managed code (bytecode) into native code through just-in-time compilation but also supports ahead-of-time compiled code in Assemblies. Managed code can call into native code through the Platform Invocation Service (P/Invoke), e.g., to do Windows system calls.

All code running on the CLR needs to be provided by Assemblies, which are files on the file system. Assemblies contain a manifest which contains a name and version information, and explicitly declares all dependencies to other assemblies by their names. If an Assembly manifest also contains a secure hash value, it is called a strongly-named assembly. Applications providing their own private Assemblies can store them in their working directory where they remain hidden from other applications. Shared Assemblies can be stored centrally in a Global Assembly Cache (GAC). Since .NET Assemblies are self-describing through their manifest, they do not need to be registered with their meta-data in the Windows Registry like COM components. This solves many of the problems historically encountered with maintaining different versions of components on the same system (sometimes referred to as the DLL-hell). The GAC is able to store different versions of the same Assembly, which can be different localizations of the same functionality, different release versions, or ahead-of-time compiled code for different architectures. To avoid name collisions, shared .NET Assemblies are required to be strongly-named. However, there can still be name collisions in terms of file names. Therefore, the GAC is implemented as a hierarchy of folders on the file system to allow for physical separation of Assemblies even though the Windows Explorer and the Windows Command Shell present a flattened view of the GAC to the user.

Assemblies are physical modules with a high degree of management support. Strong naming does not only ensure the integrity of the modules but also compatibility when composed. However, assemblies by themselves only provide a tightly-coupled way of interaction. Protocols like .NET Remoting [McLean2002] can be used to implement loosely-coupled communication between modules but these implicit dependencies remain invisible to the runtime system and cannot be actively managed.
2.4.5 OSGi

OSGi [OSGiAlliance2010] is an open standard for dynamic modules for the Java language. It is widely used as a plugin and extension mechanism for embedded systems or extensible desktop applications like the Eclipse IDE. Recently, OSGi has gained traction on the server side and many vendors of application server software are about to release new versions of their products building up on OSGi.

Modules in OSGi are called Bundles. A bundle is a traditional Java JAR file with additional meta-data, most importantly version information and an explicit declaration of dependencies. Bundles are loaded by the OSGi runtime, called framework, which provides a sophisticated management API to control the life cycle of individual bundles. For instance, a bundle can be updated to a newer version at runtime or uninstalled and the associated resources released. This is achieved by loading every bundle through its own class loader. Thereby, OSGi is different to standard Java, which has only the bootstrap- and the application class loader, and then burdens the application to deal with their extensibility requirements through user-defined class loaders.

By default, modules are completely encapsulated and do not share any code\(^2\). Java packages defined in a bundle can be made available to other modules by declaring a package export in the manifest meta-data. Hence, the physical modularity of OSGi embraces and extends the logical modularity of Java packages. Bundles which depend on a certain package have to consequently declare a package import. In contrast to, e.g., .NET Assemblies, OSGi uses a non-imperative way of dealing with dependencies. Bundles declare which packages they depend on but not from where to get the packages. As a result, multiple providers of a package can co-exist in the same deployment of OSGi. The runtime system ensures that clusters of modules coupled through dependencies only see a single version of each package to maintain type consistency within the VM.

During the resolving of a bundle the OSGi runtime system creates static bindings (or wirings) with a single provider of each package dependency. Declared dependencies can influence the resolving process by declaring acceptable version ranges for imports or other constraints. At the latest when a class is first used, it is linked by the class loader against classes potentially imported from other bundles. This, however, leads to a tight coupling between dependent bundles. A bundle can only resolve when all of its dependencies are resolvable.

To overcome this limitation and to allow for more dynamic compositions, OSGi facilitates the loose coupling of modules through services. Every plain Java object can

\(^2\)other than with the JVM through an implicit default delegation to the bootstrap class loader for the java.* packages
become an OSGi service when it is published under one or more service interfaces. The OSGi framework maintains a central service registry from which other bundles can retrieve the service. The retrieval can use the name of a service interface as a key for lookup and optionally an LDAP filter string [Howes1996] which is then matched against service properties that providers can attach to their services. Different to systems like Enterprise Java Beans [JSR220], a client acquiring an OSGi service receives the service object so that subsequent uses of the service do not differ from the behavior of any standard Java object. OSGi embraces the inherent dynamism of module compositions in that changes in the setup are signaled to listening bundles and it is expected that they react accordingly. For instance, if a service becomes unavailable, an event is sent and the expectation is that the bundle releases all references to the service. There is, however, no mechanism in OSGi to enforce this. Since OSGi originally emerged from the embedded systems world, the general spirit is to expect modules to behave well and thereby avoiding the overhead of indirections through which the runtime system unilaterally withdraws resources.

Traditionally, OSGi only deals with services on a single JVM (in-VM SOA). Recent additions to the standards in the form the OSGi Release 4.2 Enterprise Specifications [OSGiAlliance2010], which were in parts motivated by work presented in Chapter 4 of this thesis, involve remote access to services through distribution providers.

### 2.5 Summary and Discussion

Like other pieces of technology which evolved over a longer period of time, there is no concise yet universally accepted definition of what the fundamental properties of software modules are. Based on the informal description of modules of the first section of this chapter and the reflection on different implementations of modularity, for the remainder of the thesis we define the following six fundamental properties of software modules:

- **declaratively self-contained** a module is self-contained with regard to its own content and its declared dependencies.

- **encapsulated** a module exposes its content solely through well-defined interfaces \(^3\).

- **decomposed** modules are created by segregating a larger problem into smaller sub-problem so that a module ideally only deals with a single, not further separable

\(^3\)Interface here does not strictly mandate an interface of the programming language but can be a declarative interface in the module system that is not visible to the language
concern and the content of the module is highly cohesive.

**composable** modules are created for reuse in different applications and can be composed into new applications. Declared dependencies and declared interfaces put constraints on the validity of compositions. Ideally, the degree of coupling between modules is low so that composition is facilitated and not prohibited.

**substitutable** two modules (or sets of modules) providing the same interfaces can be exchanged for one another.

**localized behavior** modules are designed to behave locally, i.e., the effect of the code is restricted to the content of the module or its declared dependencies. A module should not make any assumptions about its dependents other than the ones expressed through the declared dependencies.

These properties apply to all systems and languages discussed in this chapter except for Java JAR files. In the remainder of the thesis, we will use OSGi as a concrete example for physical modularity. OSGi is the only system discussed which has a clear separation between tightly-coupled dependencies through package imports and loosely-coupled interaction through services, which often makes it easier to reason about the degrees of freedom offered versus the constraints which have to be fulfilled. However, the approaches discussed are general enough to apply to every other implementation of modularity complying with the before mentioned properties.
2.5. Summary and Discussion
Chapter 3

Juggle: Co-Managing Software and Hardware Modules

3.1 Introduction

The increasing degree of dynamism and the resulting need for more flexible software becomes especially apparent for mobile devices. Traditionally, mobile devices implement much of their performance-critical functionality in ASICs, application-specific integrated circuits with a fixed implementation. Once manufactured and implemented in a mobile device the ASIC cannot be changed, e.g., for extending the functionality of the device or for applying critical updates. FPGAs (field-programmable gate array), in contrast, are known for their reconfiguration support and their ability change the implementation of their functionality. With the technological advances in FPGAs, partially reconfigurable chips have been developed which can alter parts of their fabric at runtime.

An example of a mobile device that already makes use of reprogrammable hardware for on-demand acceleration is the Sony Playstation Portable and its Virtual Mobile Engine [SonyVME]. One of the challenges, however, is the integration of FPGAs into applications. Whereas FPGAs are programmed in low-level hardware description languages like VHDL [IEEE1994] or Verilog [IEEE2009], the application software on mobile devices is often developed in high-productivity languages like Java, JavaScript, Python, or Objective C. Bridging the gap between these two worlds is far away from being trivial, especially designing the communication interfaces between applications and FPGAs and effectively managing both the reprogrammable hardware and the software side of applications.

A popular approach to deal with the problem is hardware-software co-design. The
idea behind co-design is to simultaneously consider both the hardware and the software during the design phase and develop a single coherent design. An advantage of this holistic approach is that the product in its entirety can be formally verified (co-verification) and tested. Hardware-software co-design, however, rarely deals with dynamism, e.g., evolution, extensibility, or dynamic composition. For many applications on mobile devices, however, dynamism and the possibility to reconfigure its functionality is inevitable. When using implementing critical parts of such applications in hardware, this problem is no longer a pure software problem but now also requires the management and maintenance of the reconfigurable hardware device.

Mobile phones, for instance, are naturally constrained in the way humans can interact with them due to their form factor. For a systems design, however, this means that most of the time a mobile phone is used for exactly one interactive (foreground) task whereas the remaining tasks are running in the background and have a lower priority. For instance, when the user receives a phone call, the web browser functionality of the device becomes secondary. Ideally, a system could exploit this by using the reconfigurable hardware for always accelerating the interactive task. In the example of the phone call, this would be the audio encoding and decoding. Since the hardware is reconfigurable, the system can keep the invariant of accelerating the most critical interactive task even when the user switches from one application to another.

Implementing such systems, however, requires the developer to overcome the impedance mismatch between hardware and software and an active handling of the inherent dynamism of the problem, which typically results in ad-hoc solutions. In this work, we are trying to change this by introducing an equivalence between software and hardware functionality by both treating them as modules and effectively managing them. In terms of modularity, there are several concrete challenges to be solved:

- handling and managing both software and hardware modules where the latter is the different binary images (bitstreams) used to reconfigure the hardware.
- the ability to substitute one implementation of a functionality with another, e.g., a software module with a hardware module
- making decisions when to do substitutions, given that the hardware resource is constrained so that typically not all tasks can run in hardware.

Juggle is a system that responds to these challenges by treating software and hardware modules in a uniform way and applying the management facilities known from software module runtime systems. It takes advantage of OSGi for dealing with the life-cycle of software modules and introduces support for functionality implemented
in reconfigurable hardware, thereby applying the identical module management facilities to both worlds. At the same time, the approach permits applications to be written against common module interfaces while at runtime certain functionality can be accelerated through reconfigurable hardware. In contrast to approaches like, e.g., Liquid Metal [Huang2008], Juggle does not attempt to apply a unified design strategy for hardware and software in the small but instead focuses on the composition in the large. Juggle then takes care of the co-existence of software and hardware implementations and provides a unified model of communication through loosely-coupled services. Based on application-dependent policies, the system can thus dynamically switch between hardware and software implementation to accelerate most critical tasks without interrupting the system.

3.2 Background

Reprogrammable hardware like Field-Programmable Gate Arrays (FPGAs) is increasingly becoming powerful and affordable. Modern FPGA chips can be reprogrammed at runtime and with low latency which makes them attractive to be used as a dynamic resource in systems. For instance, on mobile devices FPGAs can help to increase the performance of critical tasks and at the same time increase the energy-efficiency of the device. The following sections provide background information about FPGAs and their reconfiguration, to the extent necessary for understanding the Juggle approach.

3.2.1 Field-Programmable Gate Arrays (FPGAs)

Traditional integrated circuits are the result of a manufacturing process; once they are manufactured they cannot be altered any more. FPGAs, in contrast, are integrated circuits with the ability to be reconfigured after manufacturing by either the designer itself or the customer. This advantage especially comes into effect when the implemented functionality undergoes changes—one-time changes as in product line customization or continuous changes as through frequent upgrades.

Internally, an FPGA is structured into three main parts: a set of programmable logic cells (PLC) or configuration logic blocks (CLB), a programmable interconnection network between the blocks, and a set of input and output cells around the device. The actual implementation of a configuration logic block (or basic block) can vary and depends on the concrete FPGA chip used.

Juggle has been prototyped on a Virtex-II Pro chip, which consists of groups of four slices, each containing two actual basic blocks. A basic block consists of a lookup table
3.2. Background

Figure 3.1: Virtex-II Pro chip layout

(LUT) with 4 inputs and an output, a set of multiplexers, arithmetic logic and a storage element. The LUT is a group of memory cells which contain all the possible results of a given function for a given set of input values. It can therefore implement arbitrary mappings between input and output ports. Altering the content of a LUT through a configuration consequently changes the behavior of the basic block.

The FPGA chip is a 2-dimensional array with the CLB as the smallest element. A full system design can be seen as a set of modules, which are reusable components that can be instantiated and interconnected through signals. Modules are organized as rectangular areas which have to be positioned on the array so that the module areas do not overlap and inter-communication timing constraints between all modules are met. The precise layout of the FPGA structure such as the arrangement of the logic blocks and the interconnection paradigm of the logic blocks is vendor-dependent. Figure 3.1 shows a schematic picture of the Virtex-II Pro FPGA.

The interconnect fabric is generally a network of vertical and horizontal wires ar-
ranged in a mesh topology. At the intersection points, there are programmable multi-
plexers facilitating the routing inside the FPGA fabric. The concrete topology can be
classified into four categories ([Bobda2007]): symmetrical array, row-based, hierarchy-
based, or a sea of gates where the cells are dense and the routing layer lies top of the
cells. Around the periphery of the FPGA chip are the I/O components used for commu-
nication with off-chip components. Those I/O components are programmable just like
the CLBs and can use as input, output, or bidirectional gates.

The programming of an FPGA typically starts with a design of the required func-
tionality in a hardware description language like VHDL or Verilog. This design can
be considered as an abstract description without taking a particular technology into ac-
count. It operates on the register transfer level (RTL), a behavioral description in terms
of signal flows between hardware registers. The mapping to a concrete technology is the
task of an electronic design automation (EDA) tool, which synthesizes a netlist from the
HDL code. This netlist now describes concrete gates and could be implemented in ac-
tual hardware. However, netlists only describe instances of gates, ports, and the wiring
in between but not a concrete topology. It is the task of a place-and-route tool to create
an instance of the template-like netlist which resembles a concrete layout of a digital
circuit. In the case of an FPGA, it represents a configuration of the FPGA fabric that
implements the designed functionality.

3.2.2 Reconfiguration

The data to configure an FPGA is called a bitstream. Bitstreams can be downloaded to
the device via several configuration ports, e.g., a JTAG interface or a USB cable. This
has the effect that the LUTs and the routing fabric of the FPGA is changed and hence the
behavior is altered. Full reconfiguration—the process of rewriting the complete design
of the FPGA chip—however, requires a reconfiguration-time linear to the size of the
bitstream to write. In addition, the entire chip is inoperable during the reconfiguration
and running processes are interrupted.

Many chips therefore support the rewriting of only a part of the fabric. Specific
regions on the chip are marked as reconfigurable and can at runtime be reconfigures
through a partial bitstream while the remainder of the fabric can continue to operate.
In order to support partial reconfiguration in a design, the FPGA fabric is partitioned in
into a static region holding the functionality critical for the running of the system, e.g.,
the bus systems, and one or more regions that are partially reconfigurable (PRRs).

Functional tasks (reconfigurable modules or PRMs) can be mapped into individual
PRRs (space multiplexing). If the tasks are mutually independent, they can also be
mapped into the same PRMs (time multiplexing) to reduce the required FPGA real estate but at the same time introducing reconfiguration latency into the system.

However, partial reconfiguration does not automatically mean that the board continues operation during reconfiguration. Depending on the hardware it can be the case that the reconfiguration requires the board to be in an inactive state. The ability of a chip to be reconfigured during runtime without interruption of the system is called dynamic partial reconfiguration.

One example of such a chip is the Xilinx Virtex-II Pro which was used to prototype Juggle. A complete discussion of the prototype system follows in Section 3.5. Partial reconfiguration requires the designer of a system to explicitly mark areas of the chip as reconfigurable. The place-and-route software has to take care that no signal lines are crossing these areas so that dynamic reconfiguration becomes possible. Otherwise the static part of the chip could encounter malfunctions during reconfiguration or even be short-circuit.

3.3 Management and Substitution of Modules

Introducing an FPGA into a mobile or embedded system enables applications to implement parts of their performance-critical functionality in hardware. These hardware modules are physically handled as bitstream files. Reprogramming the device requires to write a bitstream to a reconfiguration device embedded into the system.

The first step in making FPGAs easier to use in applications is to provide management of the bitstreams as seamless as for software modules. In a system like OSGi, this gives both the application itself and an operator the possibility to explicitly control the composition of an application and the life-cycle of the individual components. However, this alone does not solve the integration problem.

Whereas software modules can be seamlessly used in the programming languages (e.g., in OSGi through package imports), FPGAs are hardware and by nature have much more low-level interfaces like memory-mapped I/O ports, registers, or interrupts. Mixing hardware and software modules without any further integration effort partly defeats the purpose of modularity since it would not be possible to substitute a software implementation with a hardware implementation of a certain functionality due to the mismatch of interfaces.

In order to preserve the full flexibility of modularity, the interfaces between a software and the corresponding hardware module have to be uniform. In practice, this means that the representation of the FPGA functionality cannot only be the bitstream
but need to be a device driver which embeds it into the host programming language.

Such a driver, however, is beyond a straightforward mapping of functions in the language to the hardware. It acts as glue code and sometimes needs to adapt the input and output data to match the formats or even keep additional state in software. From a design point of view, it is important to realize that modularity does not replace or obsolete the co-design strategy; the device driver is the piece of code that in fact requires software-hardware co-design. The advantages of a uniform model of modularity are that the low-level communication with the hardware no longer taints the entire application and that substitution of hardware and software functionality is enabled.

### 3.3.1 Representing Hardware Modules in OSGi

In OSGi, there are two different ways for modules to interact and both are generally feasible approaches for making hardware modules available to applications.

The first way (1 in Figure 3.2) of representing hardware modules is through package exports and imports. A hardware module has to export a corresponding package containing the code around the functionality to be implemented in the FPGA reprogrammable region. Other modules can import this package and access the wrapper code the same way as they access regular Java code. A caveat of this approach is that interaction through package exports is tightly coupled, thereby limiting the flexibility. Substitution of a software module with a hardware module is generally possible but requires a package refresh. This means, the system has to be brought into a state in which the package of the new target module has higher precedence than the currently used package (e.g., a higher version) or the bundle providing the current package needs to be uninstalled prior to the package refresh. Furthermore, a refresh interrupts the running modules that need to be re-wired.

The second and more flexible way of interaction is through loosely coupled services (2 and 3 in Figure 3.2). Different instances of services can co-exist in the same OSGi application and the application is always free to switch. For the hardware module, it means that the implemented functionality needs to be registered as a service with the OSGi service registry under the same interface as the other (software) implementation which it wants to be able to substitute. If the application is coded in a way that it always uses the service with the highest rank from the service registry (e.g., by having a service tracker in place), the switch from a software to a hardware implementation is seamless and works without interrupting the running application. Falling back to a software implementation is the same when the hardware module unregisters its service. This is reasonable in the case where the hardware-accelerated service is swapped out of
the FPGA fabric to free the space for a different hardware-accelerated service.

There might, however, be cases in which certain consumer of a service should get accelerated by a hardware implementation while others should continue to run against a software implementation. Such a pattern of interaction is far away from being trivial to implement in OSGi since in general the application chooses the service and not the service the application. The solution to still support such a case is the approach of Co-Modules, modules providing both a software and a hardware implementation of the service at the same time (see 3 in Figure 3.2). Such modules can register a common proxy service as an indirection in between the service exposed to applications and the back-end implementation. Thereby, co-modules can seamlessly switch between either of the two implementations (if the hardware resource is available) and provide dynamic acceleration. If the service is implemented as an OSGi ServiceFactory, it can even selectively accelerate the service only for certain consumer and serve requests from other bundles through the software implementation.

All three approaches require the application developer to explicitly design OSGi bundles around the FPGA bitstream and the wrapper code. The last two more flexible approaches introduce services to represent the hardware functionality. Unfortunately, they thereby also require the developer to write the boilerplate code to register these hardware-accelerated services with the OSGi service registry. To some extent, this is a known problem with OSGi and there is a solution to it in the form of the Extender
Pattern which is also of great value to simplify the development of hardware modules, as shown in Section 3.3.3.

3.3.2 Hardware-Accelerated Services

A hardware module in the first place consists of a partial bitstream designed to configure the core functionality of the service into a partial configuration region of the FPGA. The bitstream is embedded into the OSGi bundle as a file. Once this bitstream is applied to the FPGA, the hardware is ready to be used but still not accessible from Java. The virtual machine approach prevents Java code from accessing the underlying physical machine. Hence, the device driver is typically coded in C and against the Java Native Interface (JNI) to bridge between Java and the hardware. From the point of view of the Java OSGi application, the device driver is represented through a Java class in which all critical methods map to JNI native code methods.

When loaded, the JNI code initializes by mapping the hardware addresses into the virtual memory of the JVM process as well as registering handlers for interrupts or initializing DMA. For each service method, there is a piece of code turning the service call into one or more interactions with the FPGA. Usually, this involves a mangling of the arguments and selective stores and loads of portions of the arguments into memory, waiting for a result to become available and then factoring returnable Java data out of the raw bytes.

Even though writing the driver is still a challenge that requires a skilled programmer, most of this can be done in a more declarative way that takes full advantage of having a clear specification of the hardware interface on the one hand and the high-level service interface on the other hand. For instance, the driver code could be generated from the domain-specific language (like, e.g., in Devil [Merillon2000]). The pair of driver and bitstream is the foundation for the hardware service and instantiated at runtime. It is dual to the Java software service implementation of the same service and can replace the latter on demand. An overview of the structure is depicted in Figure 3.3.

3.3.3 The FPGA Bundle Extender

The basic unit of modularity in OSGi is the bundle. Even though there are only very few requirements for a bundle to participate in an OSGi application, in practice there is a small piece of code in the bundle which interacts with the runtime and registers or consumes services\(^1\). This code is specific to OSGi whereas most of the remaining

\(^1\)This is either directly or indirectly the BundleActivator interacting through the BundleContext
bundle code is standard Java. For some applications, however, this OSGi-specific code is a liability. For instance, taking an web application server based on OSGi, every web application is preferably a bundle and registers its servlets as services. In traditional Java EE, however, web applications are packaged in WAR files, which are JAR files with a set of specific XML configuration files. The requirement to write the boilerplate code so that the servlets are discovered from the *web.xml* file and registered as services so that the server engine becomes aware of their existence is a burden for the adaption of the OSGi model for web applications. The solution to the problem is the extender pattern.

In the extender pattern, there is a singleton entity—called extender—in the system that listens for newly installed bundles. Whenever a new bundle is installed, it scans the content of the bundle for the existence of a specific configuration file. If such file is present, the extender interprets this file and extends the bundle by, for instance, registering services on behalf of the bundle. Thereby, in principle plain WAR files can be used within an OSGi deployment; the extender takes care of integrating the content of the file into the application server.

For Juggle, a similar approach is taken. An FPGA extender listens for new bundles containing a configuration file for hardware-accelerated services and then registers the service on behalf of the bundle. Listing 3.4 shows an example of a configuration file. Each bundle can contain arbitrarily many hardware-accelerated services. As for traditional OSGi services there can be properties attached to the service which permit clients to filter their request.

Instead of selecting either the Java or the FPGA-based service, the FPGA extender
<?xml version="1.0"?>
<hw-module xmlns="http://flowsgi.inf.ethz.ch/hwmodule">
  <hwa-services>
    <hwa-service>
      <javadservice>math.MathAddImpl</javadservice>
      <hwservice>
        <driver>adder_driver.so</driver>
        <bitstream>opb_prr_0_adder_partial.bit</bitstream>
      </hwservice>
    </hwa-service>
  </hwa-services>
  <properties>
    <property key="description", value="A hw-accelerated service for adding two int numbers"/>
    <property key="version", value="1.0.0"/>
    <property key="foo", value="bar"/>
  </properties>
</hw-module>

Listing 3.4: Juggle service descriptor example
generates a service proxy from the service interface. The purpose of the proxy is to provide the system with an interception point located between the caller and the service. This enables the system to seamlessly switch between a software service and a hardware service as well as tracing service invocations to derive performance information.

3.4 Juggling Software and Hardware-Accelerated Services

Not only is the FPGA a singleton entity in the system, the resources of the FPGA in terms of logic gates are also limited and permit—depending on the complexity of the service—just one or a small number of hardware-accelerated services to co-exist at any given time. It is hence inevitable to make resource scheduling decisions and set priorities. For this purpose, Juggle continuously traces service invocations and assembles statistical data to make decisions which services can run in software and which can profit from hardware acceleration.

Deciding which service of a single application to swap into hardware is a policy decision and can thus be best made by the application itself. On an OSGi runtime and particularly on mobile devices, however, it is not unusual to run multiple applications simultaneously. Deciding in favor of a specific application hence requires coordination. However, global knowledge about the setup is against the principle of modularity; a module should only reason locally and not require knowledge about other modules installed on the same system, especially when there is no interaction with these other modules.

Juggle deliberately avoids implementing policies and instead expects the platform to implement a controller defining the criteria for a reconfiguration. For instance, such a policy could be that the application currently running in the foreground and having the focus of the user is prioritized over the background applications. The information which service to prioritize could therefore come from the window manager, which by definition is an entity with full knowledge about all modules currently using its services. What the system has to provide is access to the basic collected performance data of each service, such as invocation frequency, average duration of the invocations, etc. and a simple imperative command interface to turn a software into a hardware-accelerated service and vice versa. This command interface consists of a single primitive: the _juggle_ operation. As arguments, the _juggle_ operation takes the service id of a service to turn into a hardware-accelerated service as well as the ids of previously hardware-accelerated service to turn back into software services.
3.4.1 Reprogramming the FPGA

When the controller has issued a juggle operation, the system first does a sanity check, e.g., if the freed slots are adjacent and if the reclaimed FPGA space is sufficient to accommodate the new service. If the check passes, the hardware can be reprogrammed. The Xilinx FPGAs are able to perform a \textit{glitchless} dynamic reconfiguration. This means, if a resource on the chip—despite being in the reconfigured area—is not affected by the reconfiguration it can be accessed without interruption. There can, however, be problems in the design phase of hardware service implementations. If the place-and-route tool does not have to meet any communication constraints between two hardware services the signal will much likely cross the boundaries of the partial reconfiguration area where the best timing can be afforded. The routing can be different for every hardware service implementation.

The solution is the usage of bus macros [Huebner2004] which can be seen as fixed data paths for signals going between PRRs. Bus macros serve the purpose of a socket where the corresponding hardware service can be plugged into the system. Hence, they provide the means of locking the routing between hardware services and the static part, making the modules pin compatible with the base design. In addition to locking the routing path, bus macros also serve as switches to enable and disable the transmission of signals. The signal propagation has to be disabled while reconfiguration, and enabled after, to avoid bus congestion or even a corruption of the bus during the reconfiguration process.

In the Juggle design, the bus macros for PRRs are encapsulated into their own IP core (an encapsulated reusable unit in the hardware design process), the socket bridge. In our prototype system, the socket bridge is controlled over the Device Control Register (DCR) bus (Figure 3.5). This bus bypasses the standard memory bus and bus controller for low latency and implements a daisy-chain architecture propagating the signals to all attached cores.

The communication between the runtime system and this IP core happens through a Linux kernel driver. This driver registers a character device which accepts control words to open or close the socket bridge and which read can be read from to retrieve status information. Since the character device is represented by an ordinary file in Linux, the Java VM can access it as a RandomAccessFile.

The actual reconfiguration happens through the internal configuration access port (ICAP) interface. The ICAP device is supported under Linux by a driver in the patched Xilinx kernel and can therefore also be accessed from Java. The system has to open the socket bridge for the RPP, retrieve the partial bitstream from the bundle, write it to
the /dev/icap character device, and close the socket bridge again. Subsequently, the JNI driver for the hardware service is loaded. If the service is hardware only, the driver is now registered as an OSGi service under the designated service interfaces. For co-bundles, there is already an existing service proxy which needs to be altered to redirect calls to the JNI driver and hence to the hardware service. In order to perform the juggling from the software to the hardware service in a consistent manner, all pending service method calls that are still accessing the software implementation will run to completion in software whereas all newly method calls use the hardware.

3.5 Juggle Prototype System

Our prototype system uses the Xilinx XUPV2P development board [XilinxInc.2009] containing a Virtex-II Pro FPGA with a total number of 30,816 programmable logic cells. In addition, the FPGA chip contains two PowerPC 450 hard cores running at up to 300 MHz clock. The cores can have up to 16 KB of instruction- and 16 KB of data cache and a MMU. Xilinx provides a patched Linux kernel tree which is able to run on the PowerPC cores.

The system boots off a flash device containing the system ACE file which initially
configures the PPC cores as well as programming the static parts of the FPGA required to connect the PPC cores to the peripheral hardware. The PowerPC405 program counter is set to the starting address of the Linux kernel also contained in the ACE file. During and after the boot process the kernel can use the flash card as a secondary storage device for its root file system.

Figure 3.6 shows a block diagram of the base system design used for the prototype system. A single PowerPC core is attached to a Processor Local Bus (PLB) which is part of the IBM CoreConnect Bus Architecture specification [IBM1999] and in this system serves as a communication backbone. An Ethernet connector, a serial port, and the compact flash connector for the card holding the system ACE file are attached to this bus through their controller logic cores. In addition, there is a bridge which connects to a second bus, the On-Chip Peripheral Bus (OPB). Even though this bus type is
3.6 Evaluation

The use case for evaluating Juggle is an application that requires encryption. This can, e.g., be the encryption of data on the internal storage of the device or a secure protocol which encrypts the data before transmission. Normally the encryption functionality deprecated in the recent versions of the Xilinx tools, the Virtex-II Pro internal configuration access port (ICAP) is only capable of communicating through the OPB. Later versions of Virtex chips feature ICAP devices that can be directly attached to the PLB. Our prototype contains only a single PRR, for the proof of concept, which is attached to the OPB through a socket bridge. This has the consequence that only a single hardware-accelerated service can run at any time.

The figure shows the logical structure of the base system. Physically, the PR region of the system has been placed at the right edge of the chip. The reason is that the entire memory bank is connected to the left side of the chip so that choosing the right side for the PRR keeps the number of static routes crossing the module boundaries low. As a consequence, the RP region can cover almost the full height of the device except for four rows of IOB and IOI at the top and bottom. The width of the PR region spans 8 CLBs, leading to a total size of almost 16% of the FPGA fabric (Table 3.1):

<table>
<thead>
<tr>
<th></th>
<th>Slice</th>
<th>Mult</th>
<th>Ram16</th>
<th>TBUF</th>
</tr>
</thead>
<tbody>
<tr>
<td>Entire FPGA</td>
<td>13696</td>
<td>136</td>
<td>136</td>
<td>6848</td>
</tr>
<tr>
<td>PRR</td>
<td>2240 (16.35%)</td>
<td>20 (14.7%)</td>
<td>20 (14.70%)</td>
<td>6848 (16.35%)</td>
</tr>
</tbody>
</table>

Table 3.1: Physical resources of the FPGA chip and the PRR.

The prototype runs the PowerPC core at 300 MHz and features 256 MB of external DDR SDRAM. As a Linux kernel, it uses the patched Xilinx Linux kernel based on version 2.6.35 and a Java virtual machine (three different VMs have been successfully tested). After boot, about 190 MB remain available for user-space programs such as the JVM and Juggle. Due to the constrained resources, Juggle relies on an updated version of the highly optimized Concierge [Rellermeyer2007a] OSGi technology implementation. The OSGi framework is enhanced with support for hardware-accelerated OSGi bundles through an FPGA extender. The prototype system does not feature an automatic controller for juggling software and hardware implementations of services. Instead, it registers an extension service for the Concierge shell so that the juggling can be triggered on demand by the user of the system.
**public interface** EncryptionService {
    **void** loadKey (**byte** [] key);
    **byte** [] decrypt (**byte** [] cipher);
    **byte** [] encrypt (**byte** [] plaintext);
}

**Listing 3.7:** Interface of the EncryptionService

would either be implemented in software or, if performance critical, in hardware as an ASIC. Security is unfortunately one of the areas that require a constant update of the technology used due to bugs in implementations and exploits through weaknesses in the algorithms. For instance, if the mobile device was shipped with an ASIC accelerating the encryption of data with the Data Encryption Standard [FIPS46-2] (DES), the de-facto standard until 2004, it would be obsoleted by now since with todays possibilities the DES encryption cannot be considered secure any more. If, however, the encryption is implemented as a hardware-accelerated service, the device becomes much more flexible and future-proof. First, the encryption algorithm can be exchanged at any time, e.g., with a Triple-DES encryption [FIPS46-3], even at runtime if the concrete use case allows this. Second, encryption can selectively run hardware-accelerated, e.g., when the performance of the interactive process is bounded by encryption. An example would be a user decrypting an email message. When the user switches the foreground task, e.g., to the music player, the audio decoding becomes the hardware-accelerated service and any encryption happening in the background runs through the software implementation of Triple-DES.

### 3.6.1 DES and Triple-DES as Hardware-Accelerated Services

DES and a Triple-DES encryption service have been implemented as hardware-accelerated OSGi services for Juggle. Both services are implemented in software, using either the JCE provider shipped with the VM or the BouncyCastle [BouncyCastle2000] Java library, and in hardware in the form of a partial bitstream for the PR region in the base system. Listing 3.7 shows the common service interface of both DES and Triple-DES so that one can be easily exchanged with the other.

The bitstreams for hardware-accelerated services for a given PRR have the identical size (≈145 kB in case of our prototype system) since the entire PRR is reconfigured in either case. In practice, tools for partial reconfiguration like Xilinx PlanAhead create compressed bitstreams so that there can be slight variations in the size depending on the
Table 3.2: Physical resources used by different hardware-accelerated services

<table>
<thead>
<tr>
<th></th>
<th>PRR</th>
<th>add</th>
<th>mul</th>
<th>DES</th>
<th>Triple-DES</th>
</tr>
</thead>
<tbody>
<tr>
<td>LUT</td>
<td>4480</td>
<td>90 (2.01%)</td>
<td>58 (1.29%)</td>
<td>1081 (24.13%)</td>
<td>3008 (67.14%)</td>
</tr>
<tr>
<td>Flipflop</td>
<td>4480</td>
<td>176 (3.93%)</td>
<td>144 (3.21%)</td>
<td>513 (11.45%)</td>
<td>1527 (34.08%)</td>
</tr>
<tr>
<td>Slice</td>
<td>2240</td>
<td>108 (4.82%)</td>
<td>88 (3.93%)</td>
<td>660 (29.46%)</td>
<td>1835 (81.92%)</td>
</tr>
<tr>
<td>Mult</td>
<td>20</td>
<td>0</td>
<td>1 (5.00%)</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

complexity of a design. Table 3.2 shows a detailed resource consumption of the two hardware implementations when programmed into the FPGA. As reference points we have added two simple hardware-accelerated services we used during the development of the system, an adder service and a multiplication service, which both take two Java primitive type integers as input and return the result of the arithmetic operation as a Java integer. As the table shows, the Triple-DES implementation consumes almost a factor three more chip real-estate than the DES implementation. This can be expected since the implementation contains in fact three instances of DES. Both hardware-accelerated services leave enough space so that potentially other services could run in parallel if the base system was designed to support this.

Since the reconfigurable area is always entirely overwritten, the reconfiguration time is solely a function of the target service and not of the service previously located in the PRR. Hence, the time depends on the number of elements to be reconfigured. Table 3.3 shows the exact sizes of the example bitstreams and the reconfiguration times. The static full bitstream used to boot the system is given as a reference point in terms of bitstream size but it indeed cannot be used for reconfiguration. The reconfiguration time for our examples varies between 11.2 and 24.9 milliseconds. Values reported in the literature indicate that in general the reconfiguration time of the Virtex-II Pro is between 10 and 35 milliseconds. There is an additional overhead involved in switching the socket bridge, which adds in average about three milliseconds. In total, the time to juggle a service is hence between 15 and 30 milliseconds for our examples. This indicates that an on-demand reconfiguration is feasible given the latency requirements of applications typically found on mobile devices.

The time to create the initial hardware service is in range of 100 milliseconds for our DES and Triple-DES example and includes the time to create the service proxy and the time to load the JNI driver for the hardware. Loading either of the two encryption bundles, which are implemented as co-bundles and also register a software service, takes less than a second. The startup time of Juggle, which includes the startup time of the
### Table 3.3: Size of the bitstream and reconfiguration time

<table>
<thead>
<tr>
<th>bitstream</th>
<th>size in bytes</th>
<th>reconfiguration time</th>
</tr>
</thead>
<tbody>
<tr>
<td>static full.bit</td>
<td>1448817 (100%)</td>
<td>—</td>
</tr>
<tr>
<td>add</td>
<td>123249 (8.50%)</td>
<td>11.159 msec</td>
</tr>
<tr>
<td>mul</td>
<td>128222 (8.85%)</td>
<td>24.896 msec</td>
</tr>
<tr>
<td>des</td>
<td>149087 (10.29%)</td>
<td>13.441 msec</td>
</tr>
<tr>
<td>tdes</td>
<td>149088 (10.29%)</td>
<td>13.374 msec</td>
</tr>
</tbody>
</table>

Java virtual machine, Concierge with the basic bundles, and the load time of the FPGA extender bundle, is in average 6 seconds in the prototype setup.

### 3.6.2 Acceleration through Juggle

Dynamic juggling of software and hardware services has shown to be feasible for a large class of applications. What remains to be shown is that hardware services have in fact a significant potential for speeding-up Java programs. For this purpose, we have evaluated different ways of performing TripleDES encryption on different Java virtual machines.

For the PowerPC architecture, there is no implementation of the original Sun Java HotSpot Virtual Machine. However, since the sources were released to the open source community, the IcedTea project [IcedTea2006] has implemented a portable version of the OpenJDK which is largely free of assembly code (IcedTea Zero) and has been successfully ported to the PowerPC and other architectures. The main caveat of the IcedTea Zero VM is that it is purely interpreting and does not feature just-in-time compilation. Hence, the performance of applications running on the IcedTea Zero VM is significantly lower than on a JIT-enabled virtual machine. For comparison, we have taken a version of the IcedTea VM enhanced with the Cacao [Krall1997] just-in-time compiler. Both versions were retrieved as binary packages from the Debian testing repository and are based on the same Java 6 version 1.8.2 build 18 of IcedTea. The Zero VM is version 14.0-b16, the Cacao JIT corresponds to the released version 0.99.4. The third virtual machine used is the IBM J9 VM for PowerPC. This virtual machine has JIT and is typically used in production servers of the PSeries but also runs on the PPC 405. The version used is J2RE 1.6.0 IBM J9 2.4 build pxp3260-20071123_01 and was downloaded directly from the IBM homepage.

As a first reference point, we have measured the performance of different Java TripleDES implementations on the three virtual machines running on our evaluation
prototype system. In general, cryptography for Java applications is supported through the Java Cryptography Extensions, an API for data encryption, authentication, and key management. The API used to be an extension to the Java platform but was folded into the main Java platform. All three virtual machines used in the experiments ship with a JCE provider. The providers of the two IcedTea flavors are identical, the IBM implementation is based upon a different code-base. For further comparison, we have used the open-source JCE provider BouncyCastle [BouncyCastle2000] in its latest version 1.38 for Java 6. This library can uniformly run on all three virtual machines.

The experimental setup for this and all following experiments is the encryption of a buffer filled with random bytes, using TripleDES with the same fixed key. The size of the buffer is varied in the experiments to get an impression of the overall performance characteristics of the corresponding implementations. Figure 3.8 summarizes the experimental results for the various Java implementations. As a baseline of comparison, the graph additionally contains the measurements for a C implementation using the Triple-DES implementation from Eric Young’s libdes.

The first conclusion to be drawn from the results is that the interpreting VM has a significantly lower performance than the other two VMs. Is can be expected since cryptography is computation-intensive and can hence profit to a large extent from just-in-time compilation. The IBM VM performs better with its own JCE provider whereas on Cacao BouncyCastle performs better than the built-in OpenJDK provider. For further
experiments, we will only compare against the IBM J9 JCE implementation as the best-performing Java implementations. Overall, however, even this implementation is still a factor of almost three slower than the C implementation.

The next experiment compares the performance of the two software implementations (IBM J9 JCE and C+libdes) with the performance of a hardware-accelerated Juggle service running on the different VMs. The hardware-accelerated service consists of a Java interface, a JNI driver, and the corresponding Triple-DES logic in the FPGA. The JNI driver exchanges data with Java through simple byte arrays and shuffles data to the FPGA TDES core by writing to software registers. Triple-DES is a block cypher. For the encryption of each block (of 8 bytes), the JNI driver writes the data into two 32 bit registers of the TDES core and then alters a status register to indicate that the data is ready and the requested operation is an encryption. When the core has encrypted the data, it sets the status register to a success value. The JNI driver busy-waits on the content of the status register and then reads back the encrypted result from two 32 bit registers. This design has been mainly chosen for simplicity, more sophisticated implementations might further improve the performance of the hardware. The TDES core runs with the bus clock speed, which is 100 MHz in the prototype system and therefore a factor of three lower than the CPU clock.

![Graph showing performance of Triple-DES encryption through a hardware-accelerated service](image)

**Figure 3.9:** Performance of Triple-DES encryption through a hardware-accelerated service

Figure 3.9 also shows the measured results and, as a baseline, the performance of a
Figure 3.10: Detailed performance of the JNI-based implementations and the software implementations

C implementation using the same FPGA TDES core for acceleration. Comparing to the pure software implementation, the hardware-accelerated service performs an order of magnitude better than the Java implementation and still a factor of three better than the C+libdes implementation. The difference in performance between the three different virtual machines becomes marginal even though the IcedTea Zero VM lacks JIT. The reason for this behavior is that the experiment is no longer computation-bound as far as the code executed in the virtual machine is concerned. However, neither of the three VMs can match the baseline performance when accessing the FPGA directly from C. This effect is due to the overhead of JNI and moving the data from Java into native code.

Figure 3.10 shows the measured performance in a smaller range of buffer sizes to increase the resolution. For buffer sizes between one and 8 bytes, there is no significant difference in the runtime of the encryption. This is due to the operation of the algorithm on discrete blocks of exactly 8 bytes. If the buffer size is smaller or equal to 8 bytes, the effective size of data transmitted to the FPGA is always one block of 8 bytes with the remaining bytes zeroed out and the algorithm has the same runtime. The absolute runtime of the encryption of buffers fitting into one block is initially higher for the JNI-based hardware-accelerated services (the IcedTea Zero several times higher and the curve is therefore omitted from the figure). However, also the increase relative to the buffer size for buffers larger than one block—in the figure this corresponds to the
slope of the curves—is higher than for the C+FPGA version. It can be concluded that in addition to the static overhead for switching from Java into JNI, there is also a systematic overhead that scales with the size of the buffer. The explanation of the latter effect lies in the organization of the Java heap, which can store data in any form and opaque to the application. In practice, moving data from Java into JNI involves creating a copy of the data. For instance, arrays are not guaranteed to be stored contiguously on the Java heap but in C array access corresponds to pointer arithmetic and hence requires arrays to be contiguous.

This is a known performance problem and Java code frequently interacting with native code (e.g., the operating system) therefore avoids storing the data in a Java format and instead uses the Java NIO byte buffers. These buffers are Java objects but can be backed by a direct buffer, which is a buffer that occupies a certain region of memory and stores the data in the same format identical as the underlying operating system. Therefore, these buffers are directly accessible from native code and do not require any additional intervention from the VM to be passed into JNI and manipulated by native code. Filling a direct byte buffer is slightly more expensive than writing to a Java byte array since it involves calling a method on the object and the more expensive data transfer on these typically much smaller portions of data. However, many data sources such as network sockets and the file system have a NIO interface in Java and can already provide data in the form of NIO byte buffers.

![Figure 3.11: Detailed performance when using Java NIO buffers](image-url)
Figure 3.12: Performance of Triple-DES encryption through Hardware-accelerated service using Java NIO buffers

When using a NIO byte buffer for data transfer, the picture in the micro-scale changes (Figure 3.11). Not only is the static overhead of the JNI call significantly lower in this case, also the systematic overhead is completely eliminated and the slope of the curves for hardware-accelerated Java Triple-DES encryption is now identical to the slope of the C and FPGA implementation. Figure 3.12 confirms this observation in the large scale. Hardware-accelerated encryption in Java can provide a performance equal to using the FPGA directly from C since the static overhead becomes irrelevant for realistic buffer sizes. When encrypting buffer of at least 64 bytes size, a Juggle hardware service accelerates the encryption by a factor of almost 20 compared to the best Java software service. Furthermore, when accelerating the performance-critical code through hardware implementations, a interpreting VM can reach almost the performance of a JIT-enabled VM. This is an interesting design option for highly resource-constrained systems that cannot afford the memory and storage footprint of a just-in-time compiling VM.

3.6.3 Power Consumption

Besides performance, power consumption is a major issue for mobile and battery-powered embedded devices. Therefore, we evaluated the power consumption of three different implementations of TDES by using a wattmeter introduced between the power
supply of the prototype board and the power outlet. Hence, the values measured determine the consumption in the primary circuit and correspond to the de-facto consumption observed by an operator of the device. As a benchmark, we used the TDES encryption of a buffer of 1024 bytes in 30 runs of loops of 1000 encryptions and measured the power consumption for the OpenJDK IcedTea VM with the Cacao JIT and the JCE encryption provider, for the C implementation using the libdes library, and for the same OpenJDK IcedTea/Cacao VM but using the FPGA for the encryption. Figure 3.13 shows the results for the three different implementations. The power consumption of the board in the idle state is 8.75 W, illustrated by the dashed horizontal line. The two Java implementations have a peak consumption in the first six second, which is the time that the JVM takes to start. After this startup time, the software implementation has a relatively stable power consumption of 9.1 W whereas the FPGA-based implementation uses only 8.9 W. The C implementation has a constant consumption of 9.1 W.

Integrating the power consumption over the runtime of the test run results in the total energy used for performing the encryption task. The pure Java implementation, which has a total runtime of 346 seconds (the figure does not show the entire runtime for Java/JCE), takes 3151 Joule. With the C implementation, the device consumes a total of 199.6 Joule, only slightly more than 6% of the energy for Java/JCE, mainly

**Figure 3.13:** Power consumption of the different TDES implementations
3.7 Related Work

Juggle is not the first attempt to interface between a high level language like Java and reconfigurable hardware. JBits [Guccione1999] is a set of Java libraries that can read bitstreams either generated by the toolchains or from a currently running FPGA device. It provides an API to modify a configuration bitstream and use it to reprogram the device. Unfortunately, JBits provides little abstraction over a hardware description language. Hence, it is highly platform-dependent and requires the using application to explicitly deal with low-level details such as the routing. It could, however, be an important building block for implementing relocatable hardware-accelerated services for Juggle.

Liquid Metal [Huang2008] features the Lime language which is based on Java but extended with a special and more restricted type system amenable to bit-level analysis. Lime can be compiled both into Verilog and successively into FPGA bitstreams as well as to Java byte-code. The target domain of Liquid Metal is similar to Juggle as both systems target devices with both a full-fledged CPU and an FPGA as an additional resource for dynamic acceleration. The major difference is the level on which the systems operate. Liquid Metal attempts to create a unified language for both the software and the hardware design. Juggle in turn focuses on the integration of the two worlds through composition of modules and requires both parts to be designed using their specific tools and offline. Since the runtime of an FPGA synthesis tool for non-trivial pieces of functionality is in the order of minutes and hours, it can, at least considering the current state of technology, be questioned if Liquid Metal will be able to reach its design goal and become the equivalent of a Just in Time compiler for hardware.

A different approach has been taken with JOP [Schoeberl2003], a Java optimized processor implemented on top of an FPGA. The motivation of JOP is to enable the use of high-level productivity languages like Java for programming FPGA chips. The authors point out that the programming languages usually used on top of systems on a chip like C and Assembler provide poor abstractions to the programmer. Furthermore, they claim that use of an operating system in between an FPGA chip and a managed runtime for languages like Java providing higher level of abstraction is unnecessary overhead since the main functionality of an OS can as well directly be implemented as part of
the managed runtime system. The result of this consideration is a JVM implemented as a processor on an FPGA. The original Java byte-code is translated by the processor into an address in the own microcode format of the JVM. More complex Java byte-code instructions are handled by sequences of JOP instruction. JOP represents an interesting point in the design space and a JOP VM enhanced with support for modularity can be a platform to run Juggle without the overhead of an operating system.

Ullmann et al. [Ullmann2004] present a complete approach to a module based architecture for automotive control devices. Today's automobile classes contain up to 100 control devices which becomes sooner obsolete and the product life cycle decreases from 5 to 2 years. The adaptivity of reconfigurable devices can increase the product life cycle and reduce the cost and risk for development and later maintenance. On the memory structure their partially dynamically reconfigurable structure consists of separated slots for the function module which are connected to a bus structure. A bus macro is used to ensure the physical separation between the slots and connect them the signal lines between the functional blocks and the arbitration/runtime system module. Furthermore the authors describe the communication between the function blocks and how loading and unloading is achieved.

A high-level approach to use the Xilinx FPGA reconfiguration ability is explained in the work of Williams et al. [Williams2004]. They present a modular platform for RSoC called Egret designed around the idea that complex systems can and should be designed by composition. The specification of an assembled hardware module stack is given to a software tool that constructs the appropriate FPGA configuration, as well as software infrastructure such as device drivers. An interesting twist in their approach is the usage of the built-in ICAP device. The system builds atop a Microblaze system architecture and uClinux as the operating system and presents a first ICAP driver implementation. Crucial parts of system design, e.g., how to manage reconfigurable module regions, are unfortunately not covered by their work.

## 3.8 Summary and Discussion

Juggle shows that a modular and loosely-coupled approach to integrating software and reprogrammable hardware facilitates a flexible and dynamic co-existence between software and hardware services. Applying the same management facilities to both worlds simplifies the development of such systems and at the same time gives the maintainer the opportunity to alter the setup even at runtime. OSGi is extensible enough to be used for this purpose and the extender pattern avoids large parts of the boilerplate code re-
quired for registering services. The latency of reprogramming reconfigurable areas is low enough to seamlessly switch between the software and hardware and accelerate the most critical tasks at any time. Due to the common interface that Juggle applies to both software and hardware services, the substitutability principle of modularity ensures that existing applications do not need to be modified for Juggle. Hence, their performance can gradually be improved through introducing hardware services. In practice, the degree of acceleration can be significant, as shown by the example of TripleDES, where the hardware is able to reach a speedup of 20 even with a simple and unoptimized hardware design. Generally, services taking a small amount of input data and applying a significant amount of (ideally parallelizable) computation most amenable to hardware-acceleration. In situations where the data transfer dominates the end-to-end latency, FPGAs are less likely to be applicable.

One of the lessons learned in building Juggle is the importance of the chosen interface for the performance of the system. When only considering the throughput, the interface of choice for achieving the best performance for the hardware acceleration of functionality operating on large data blocks needs to use Java NIO byte buffer. This, however, is not the most efficient interface for the Java implementation since the performance gained with NIO is based on the assumption that the endpoint receiving the data lives outside of the JVM. With the choice of NIO, the Java implementation has to pay the penalty of bringing the data back into the virtual machine.
Chapter 4

Turning OSGi into a Programming Model for Distributed Systems

4.1 Motivation

OSGi is by now a well-established system for managing dynamic modules on the Java virtual machine. In addition, the lightweight services facilitates the composition of application from loosely-coupled modules. The advantages of OSGi, however, were traditionally restricted to a single virtual machine even though the modular design appears to be a good fit for programming distributed systems.

The key insight for turning OSGi into a programming model for distributed systems is that the module boundaries instituted by centralized module management systems are generally well-suited to being repurposed as distribution boundaries. In the past, networked applications have typically distributed their functionality by interposing communication proxies at procedure calls or object method invocation, with mixed results. In particular, the issue of transparency has dogged distributed computing platforms based on these models: as Waldo et. al. [Waldo1994] point out, a remote procedure invocation has fundamentally different semantics to a local call, and consequently fundamentally different exception handling code must be written by the programmer.

In contrast, module management systems like OSGi do not assume a static environment but instead are explicitly designed to embrace dynamism. For instance, applications are required to handle unloading of modules at any time and to handle the events issued by the framework to cleanly handle services becoming unavailable. Taking advantage of this, a distribution system for OSGi can represent communication-related failures as local module and service unloading events.
The implementation of this concept is R-OSGi [Rellermeyer2007b], a conceptual extension to OSGi turning it into a programming model for distributed systems, in many cases without even changing existing application bundles. At the same time, R-OSGi can sit on top of arbitrary OSGi framework implementations and does not require customized frameworks.

R-OSGi was the first attempt to extend OSGi towards distributed systems as opposed to previous approaches that adapted OSGi to interact with middleware systems (e.g., Jini [Sun2005] or UPnP [UPnP2000]). The latter approach inherently restricted remote OSGi services to what is expressible through the middleware. In contrast, R-OSGi preserves the entire expressiveness of OSGi services by building a distribution system around the OSGi model of services.

4.2 Challenges

The effort of turning OSGi into a programming model for distributed systems results in three main conceptual challenges, all of them related to the fact that R-OSGi attempts to be consistent with the behavior that an application can expect from a local OSGi framework. As a consequence, R-OSGi does not restrict the semantics of OSGi in any regard while at the same time aims to be non-invasive with respect to both the application modules and the OSGi framework. The concrete challenges arising from this approach are:

- **Central service registry**: the expectation of a conventional OSGi bundle is that services can be looked up from the (local) service registry. In a distributed setting, however, the local service registry is unaware of all the services on remote machines.

- **Services are plain Java objects**: the result of a service binding is a direct reference to a Java object that implements all the interfaces that were requested when acquiring the service reference.

- **Services can have implicit package dependencies**: Even though there is no strict requirement in the OSGi standard that services are registered by bundles, in practice they are. As a result, a service now available for remote access might depend on imported packages or use types that this original bundle exported. On the client, a proxy service must preserve class-space consistency and cannot always make the assumption that the bundles it depends on are already available.
R-OSGi addresses these challenges by:

- Extending the local OSGi service registry with a network-enabled service registry (Section 4.3.1).
- Providing a way to generate proxy services even on mobile platforms without VM support for proxy generation (Section 4.3.2).
- Generating proxy bundles that register the proxy services and provide clients with a consistent view of the original bundle through type injections (Section 4.3.5).

In addition, it has to solve several technical problems, some of them specific to distributed OSGi or Java. For instance, the dynamic proxy generation on the client side facilitates adaptation of the proxy, e.g., if the client VM supports only a lower version of Java byte-code or if the service provider wants to push computation to the client. The structure of R-OSGi furthermore allows an easy integration of non-Java services into a distributed OSGi application, as discussed in Section 4.6.

Three different patterns of interaction between modules are researched in detail and solutions are presented to enable a consistent behavior across machine boundaries. The first pattern is direct invocation through (synchronous) remote service invocation. Services are called in the same way as methods on any other Java object are called. The second pattern is indirect (asynchronous) invocation through events. Here, a publish-subscribe pattern is applied in the form of the EventAdmin service in OSGi. The third pattern is the use of continuous data streams in services. Streams can occur as data types in service methods and need special treatment when used in the network. Section 4.4 assesses the performance of the implementation based on experiments and existing benchmarks. The following Chapter 5 then presents three applications of R-OSGi and shows how the opportunities have been taken in concrete setups.

4.3 R-OSGi

R-OSGi allows a centralized OSGi application to be transparently distributed at service boundaries by using proxies. Figure 4.1 shows a simplified example with one service provider \((I)\) and one service consumer \((J)\). To bundles on peer \(I\), the R-OSGi proxy is indistinguishable from local OSGi services such as service \(A\) and \(B\). The R-OSGi protocol on the proxy is used to make remote invocations to the original service, which is located on peer \(I\), and events from \(I\) are transparently forwarded to \(J\) and occur as if they were issued by a local bundle. Different to other systems, however, R-OSGi
requires not only proxification of services. Instead, the system needs to also provide a consistent view of the modules underlying a service.

### 4.3.1 Distributed Service Registry

OSGi is built around a centralized service registry. In order to distribute OSGi applications in a non-invasive fashion, a distributed registry implementation is required to make OSGi bundles aware of services located on remote machines. However, it is not possible to make a distributed service registry look like a local registry without changing the OSGi framework implementation.

Thus, to avoid limiting the generality of the platform, R-OSGi works with a complementary service discovery protocol and builds proxies for remote services which then register their services with the conventional OSGi service registry. Hence, conventional OSGi bundles can be used (and distributed) in R-OSGi without modification.

OSGi uses an explicit binding model whereby the client bundle invokes (as a synchronous method call) the service registry, which hands over a set of service references

---

1With the release 4.2 of the OSGi specification, service hooks have been introduced to intercept service lookups and introduce additional service references into the result set. At the time where R-OSGi was designed, no such mechanism existed.
in return. The request contains two arguments: the fully qualified class names of the interfaces of the requested service and a filter expression which can, for instance, be used to distinguish between equivalent implementations of the same service type. Filters are based on the LDAP string filter syntax (RFC 1960, [Howes1996]) and can express basic arithmetic comparison clauses, string comparison clauses, and boolean compositions thereof. If filters are used, they are matched against the registered service properties. A client in possession of a valid service reference can then attempt to establish a binding to the service, and afterwards invoke operations on it.

While the explicit binding model simplifies the handling of network and remote node failures in R-OSGi, the approach of building proxies for services introduces a potential scalability problem since in a large distributed system there might be a large number of nodes, and a large number of services. Each service proactively announcing its availability and the system generating proxies for every available service might increase network traffic, and tie up processing resources at the nodes.

R-OSGi’s distributed service registry alleviates this problem by making service discovery (and thus the proxy generation) reactive. Bundles can register services of type DiscoveryListener and set properties to convey information about the service interfaces they are interested in, optionally including a filter string. Following the whiteboard pattern, R-OSGi keeps track of all registered listeners by observing service registration events from the local service registry. It initiates remote service discovery whenever there is an entity in the system that has announced a demand for a service.

Likewise, peers announce their offers of services to the network and allow remote access to them according to a locally-determined policy. Whenever a new service is registered with the local framework with properties that indicate it should be offered remotely, R-OSGi triggers registration of this service with the remote service discovery layer.

Explicit determination of which services to offer for remote access in this way can be performed by the application, at the cost of loss of transparency (since the application must set the required properties). However, the explicit service brokering over the boundaries of the network can be completely separated from the application and added as a surrogate bundle. Such a bundle can listen for local service registrations, and selectively re-export some services remotely without requiring the application itself to be distribution-aware. At the same time, it can take measures to provision remote services to hosts where they are needed.

R-OSGi comes with a default implementation of a distributed registry using the Service Location Protocol (SLP) [Guttman1999, Veizades1997, Guttman1999a] as the underlying mechanism. Rather than using a C-based daemon implementation of SLP like
OpenSLP, we instead developed a pure Java implementation, jSLP [Rellermeyer2005]. jSLP implements all the mandatory features of the SLP protocol, plus most of the optional features, yet has a code footprint of only 55kBytes.

SLP has several compelling features for R-OSGi: its adaptivity, the inherently distributed lookup process, and a similar notion of services as in OSGi. To use SLP as a fully decentralized service registry, we exploit the adaptive behavior of the SLP protocol. In SLP, if no dedicated Directory Agent (DA) is present, the clients use multicast (as in SSDP [Goland1999]) and exchange information about services in a peer-to-peer fashion. This setup works well in spontaneous and unmanaged networks. When a DA is present, clients communicate exclusively with this central registry server. This setup is favorable when handling a high number of services or in networks where no multicast (over the entire set of participating nodes) is possible. Since SLP in a first step always tries to find a DA, and falls back to multicast discovery when a previously available DA becomes unavailable, every setup can seamlessly switch from a centralized to a distributed mode of operation and back.

Through this feature, R-OSGi implements a distributed SLP layer that can be used in a wide range of situations. In terms of naming, both OSGi and SLP identify a service by a single string. In OSGi, this is the fully qualified name of the interface under which the service has been registered. In SLP, the name is a service URL of the form service:serviceType://URL where the service type is of the form abstractType:concreteType. By describing all OSGi services by the same abstract type service:osgi and using the fully qualified name of the interface as the
concrete type, a bidirectional mapping between OSGi and SLP services is created. OSGi supports LDAPv2 filter predicates on service properties to allow more declarative and fine-granular services matching. This feature becomes particularly useful when the service registry is no longer a central but a large distributed one. With the choice of the SLP protocol that also supports LDAP filters over service attributes, R-OSGi leverages the power of expressive service predicate matching for the distributed case and is able to push selectivity to the edges of the network.

After a service is discovered, R-OSGi introduces an intermediate step before the actual service is delivered (i.e., imported into the local framework). This is important for security reasons as it allows users to, e.g., see the available remote services in a GUI before connecting to them. With such a step, R-OSGi matches the behavior of OSGi, which also uses an indirection over service references. R-OSGi also supports explicit connection to a remote peer if the application has a priori knowledge of the distribution of services in the system. In both cases, the intermediate step consists of presenting RemoteServiceReferences to the application or the surrogate bundle which encapsulate both all the facets of the OSGi side of the service and all the facets arising from the distribution. R-OSGi even takes care of keeping the service properties fresh, i.e., performs consistent updates on the local properties when the service properties of remote service change. If a bundle is interested in getting access to a discovered or already known remote service, it can import the service into the local OSGi system by calling the getService method on the R-OSGi service. This triggers the creation of a proxy module and the registration of a service proxy in the local OSGi service registry.

4.3.2 Dynamic Service Proxies

The most common form of explicit communication between modules in OSGi is through services. Once looked up and retrieved from the service registry, clients hold a direct reference to the service object and can invoke methods according to the implemented service interfaces. When using remote services across machines, proxies have to be generated and registered with the local service registry.

R-OSGi creates the client proxies for remote services on the fly. To a service client, these proxies behave as a local service and are also provided by locally-instantiated bundles. However, a proxy bundle redirects all service method calls to the original service residing on the remote machine and propagates the result of the method call back to the local client. The same pattern is applied to application-level exceptions. Since Java can use exceptions to transport information to the caller even in situations that are strictly speaking not of exceptional nature (e.g., FileNotFoundException for
an open call tells the caller that the File object is not in state open due to the absence of the underlying file) it is important to apply the same level of consistency with local behavior. Hence, exceptions thrown by the remote service are serialized and rethrown by the service proxy.

The approach of dynamically generating the proxy code at the client facilitates spontaneous interaction between services, but also reduces to a minimum the data (in the form of Java byte-code) that must be stored on the server or transferred over the network when a client binds to (or fetches) a service. Furthermore, client-side proxies can be tailored to the capabilities of the platform, e.g., to the version of Java byte-code supported by the virtual machine.

The typical information required to create a proxy for a particular set of service interfaces is determined by byte-code analysis of the original service when it is registered. When a client fetches the service interface, the service provider responds with the corresponding Java byte-code for the interface along with any serialized properties of the service.

From the interface byte-code, the client can then generate a full proxy for the service. No precompiled skeletons or stubs need to be provided by the implementor of the service, and no actual proxy code needs to be transferred. This is particularly useful in the case of servers running on resource-constrained devices, since the service provider bundle does not need to retain any code for the client proxy.

On the client side, the proxy is created through a Proxy Generator. The Proxy Generator is based on the ASM library [Bruneton2002], and it uses byte-code manipulation to create the proxies for service interfaces. First, an empty class is created that implements both the service interfaces as well as an OSGi specific interface (the BundleActivator) to provide the system with an handle to control the life-cycle of the bundle and the service proxy. The OSGi specific parts, including the registration of the service with the local framework and retrieving the R-OSGi service, are implemented by emitting generic templates. This code is primarily responsible for registering the service and getting the required information (such as the service properties) from the system. Subsequently, every method of the service interfaces is visited and the corresponding method implementation created. Each method implementation delegates the method call to the network channel provided by R-OSGi and invokes the following (reflection-style) method:

```java
Object invokeMethod(final String serviceURL,
                     final String methodSignature, final Object[] args)
                     throws RemoteOSGiException;
```

The serviceURL is known at proxy generation time and hard-coded into the proxy, since
Chapter 4. Turning OSGi into a Programming Model for Distributed Systems 67

every remote service gets its own proxy. The method signature is also a constant of each method implementation. The args array is built at runtime by aggregating the actual arguments.

Since the remote invocation follows the pattern of the Java reflection API, it has to deal with the problem of boxing of primitive types. The arguments have to be transmitted as an array of objects and since arrays in Java follow store typing, they require all entries to be of an object type. Autoboxing, as introduced with Java 5, is a mechanism of the Java compiler to automatically infer situations in which a primitive type needs to be boxed into an object type and vice versa. Unfortunately, autoboxing works only in the process of translating Java source code into byte code, whereas the proxy generation already operates on the byte-code level. Hence, the generation of the byte code for the service methods has to explicitly deal with the boxing of arguments into object types and the unboxing of return values into primitive types by selectively generating byte-code in such situations. The need for boxing and unboxing can be inferred from the types used in the method signatures in the service interface(s).

The proxy-implementation of the service interfaces is packed into a JAR file together with the service interfaces themself. The required metadata is added to the manifest to turn the JAR file into a valid bundle. Packages containing the service interfaces are exported and/or imported if they were also exported or imported by the remote bundle that has registered the original service. This allows other bundles to import the interface if it is not yet known. Otherwise, the import statement is used and the newly created bundle is linked against the existing interface to preserve consistency within the framework. R-OSGi stores the generated bundle and installs it, which leads to a registration of the proxied service. Since the service is registered under the transmitted interface names, local bundles cannot distinguish between a proxied service and a local service, thereby preserving full location transparency.

4.3.3 Method Invocation

Every method invocation corresponding to a remote service is transformed into the invokeMethod call shown before and sent through the underlying R-OSGi channel. On the other side of the channel, the first step taken is to lookup the corresponding service. R-OSGi holds references to all services that are released for remote access in a HashMap to guarantee a quick lookup. On this service object, a reflective call of the original method is performed. However, the Java reflection API requires the formal method parameters for matching and these can differ from the types of the actual arguments. This is particularly true if one of the formal parameters is an interface or an abstract class.
One option would be that for every method call, the whole type hierarchy of each of the arguments is used for matching. To avoid this overhead, the signature of the method is part of the transmitted message so that R-OSGi can use the signature to unambiguously match the original method. If the reflective method call succeeds, the result value is packed into a response message and sent back. If an exception occurs, the exception object is serialized, packed into the response message, and thrown on the other side of the channel. This makes the syntactic behavior of the remote service indistinguishable from that of local services.

4.3.4 Error Handling

Two types of errors have to be distinguished between and hence require different handling. On the one hand, there are errors (and therefrom arising exceptions) that occur independent of the fact that the service call is now crossing the boundaries of an address space and a single machine. The correct behavior compared to a local setup is, as discussed before, to rethrow the exception in the proxy. On the other hand, much of the discussion about transparent remote procedure calls or remote object calls have been centered around the kinds of errors that solely related to the network now in between caller and callee. R-OSGi takes the approach to provide location transparency but not failure transparency. This means, remote services can be accessed as if they were local services (inevitably due to the fact that the local proxy service is a valid local service). Errors, however, are explicitly passed to the application and the application has to be aware that errors can potentially occur. In the case of R-OSGi, however, most of the errors arising from unreliable networks can be mapped to dynamic lifecycle events, which OSGi applications are prepared to handle.

The consistent mapping of network errors to lifecycle operations is to remove the service proxy whenever

- the remote service has been unregistered and is no longer available.
- a message on the network could not reach the remote service (potentially after retransmissions on the protocol level).
- the host which has provided the service is no longer reachable, e.g., due to a network partitioning.

In either of the cases, R-OSGi removes the proxy module with its registered service proxies since this is consistent with the view of the client; the service is in fact no longer available to the client. If the removal was the result of an incomplete service invocation,
the proxy throws a runtime exception (which in Java does not need to be declared in method signatures and can always occur as the result of a method invocation) to indicate to the application that the service call was not successful.

In consequence, R-OSGi attempts to provide full consistency with local behavior where possible and map network errors to existing events where errors were introduced by the remote operation. This, however, does not take application-level failure semantics into account, which are in general hidden from the platform and require explicit involvement of the application. R-OSGi is a mechanism for remote service access and leaves policy decision to the applications. There is the possibility for distribution-aware applications (or surrogate bundles extending distribution-unaware applications) to push policies into the proxies by using a specific service registered by the R-OSGi bundle. This mechanism permits the creation of redundant bindings of proxies to multiple services and handling and controlling the switching between bindings for either handling errors, or implementing non-functional aspects of distribution such as load balancing. A detailed discussion of redundant bindings follows in the Section 5.2 about the Deployment Tool.

4.3.5 Type Injection

In OSGi, all code is modularized into bundles and imports of code from other bundles have to be explicitly declared in the bundle JAR manifest. Several implications arise in the context of proxy bundles which require attention to provide a view to the application that is consistent with a corresponding local setup.

The service interface might use types as formal parameters or return values of methods that do not belong to the standard Java classes and cannot be assumed to be present at the client. This can either be the case if the type is declared by a class from the original service bundle, or because the package to where the class belongs was imported by the original service bundle.

It has to be assured that the generated proxy is resolvable, i.e., it provides all the types that are used by methods of the service. R-OSGi thus has a special strategy called Type Injection to ensure type consistency for the service interface.

When the (remotely accessible) service is registered, every type occurring in the service interface is observed by a static code analysis. Algorithm 1 gives an overview how the type injection works in detail. The goal of the analysis is to determine three sets, the imports $\mathcal{I}$, the exports $\mathcal{E}$, and the type injections $\mathcal{T}$. The analysis starts with the set of service interfaces under which the service has been registered and forms the transitive closure over all referenced types. In Java, this can occur in the form of method
Algorithm 1 Determining the type injection set


Require: Output: Set of type injections into the proxy module: $\mathcal{T}$, Set of package imports of the proxy module: $\overline{I}$, Set of package exports of the proxy module: $\overline{E}$

Ensure: $\overline{I} \subseteq I$, $\overline{E} \subseteq E$, $\mathcal{T} \subseteq \text{Classes in the module}$

1: $\text{Stack} \leftarrow S$
2: $\mathcal{T} \leftarrow \emptyset$, $\overline{I} \leftarrow \emptyset$, $\overline{E} \leftarrow \emptyset$
3: $\text{Visited} \leftarrow \emptyset$
4: while $\text{Stack} \neq \emptyset$ do
5: $\text{Clazz} \leftarrow \text{pop}(\text{Stack})$
6: if $\text{Clazz} \notin \text{Visited}$ then
7: $\text{Visited} \leftarrow \text{Visited} \cup \{\text{Clazz}\}$
8: $\text{push}(\text{Stack}, \text{superclass}(\text{Clazz}))$
9: $\text{push}(\text{Stack}, \text{interfaces}(\text{Clazz}))$
10: for all $M \in \text{methods}(\text{Clazz})$ do
11: $\text{push}(\text{Stack}, \text{argumentBasetypes}(M))$
12: $\text{push}(\text{Stack}, \text{returnBasetype}(M))$
13: end for
14: for all $F \in \text{fields}(\text{Clazz})$ do
15: $\text{push}(\text{Stack}, \text{basetype}(F))$
16: end for
17: for all $C \in \text{innerClasses}(\text{Clazz})$ do
18: $\text{push}(\text{Stack}, C)$
19: end for
20: $P \leftarrow \text{package}(\text{Clazz})$
21: if $P \in I$ then
22: $\overline{I} \leftarrow \overline{I} \cup \{P\}$
23: end if
24: if $P \in E$ then
25: $\overline{E} \leftarrow \overline{E} \cup \{P\}$
26: end if
27: if $\text{Clazz} \in \text{Classes of module}$ then
28: $\mathcal{T} \leftarrow \mathcal{T} \cup \{\text{Clazz}\}$
29: end if
30: end if
31: end while
signatures or field signatures but also has to take the type hierarchy (superclass, implemented interfaces, inner classes) into account. For each type that can be discovered in this process, it is determined if the corresponding class is imported by the original bundle in which case the package automatically also becomes an import of the proxy, analogue for exports, and if the class is provided by the module under analysis, in which case it becomes part of the injections set.

Whenever a client fetches the service, the injections (i.e., the byte-code of the Java classes) are transmitted in addition to the service interface and the service properties. During proxy generation, the injections are materialized, i.e., class files are generated and stored in a JAR file that defines the proxy bundle. By construction, the injections cannot contain packages of classes that are not exported by the original bundle if the original bundle was correct.

Classes from the packages `java.*` and `org.osgi.*` are excluded from the whole process since it is assumed that they belong to the execution environment (step omitted in Algorithm 1 for readability). The result of the injection strategy is a minimal set of classes and package imports that make the service proxy self-contained and resolvable. At the same time, the injection set provides exactly the subset of the original bundle that is observable by a client through the service interfaces of the proxied service (Figure 4.3).

Beyond the described code analysis to determine the minimal set of injections, service registrations can be manually provided with classes that have to be injected into the bundle. This can be useful in particular cases, e.g., if an argument of a service method is an interface and the service provider wants to add a particular instantiable implementation of the interface.
4.3.6 Network Channels and Message Transport

The communication structure of R-OSGi is purely message-based. The design of R-OSGi is in itself modular and permits arbitrary implementation of network protocols and transports to be plugged into the system. The interface between the system and a transport provider is the ability to accept and create channels which can send and receive messages. This is in practice generic enough to encapsulate a large variety of protocols, later shown by the examples of Bluetooth (Section 4.5.1) and ZigBee (Section 4.5.3).

By default, R-OSGi uses its own native protocol. For efficiency of parsing and handling, all messages are transmitted as binary. Messages consist of a header that indicates the type of the message plus some common attributes, and a body with the parts specific to the message type. The default network channels in R-OSGi uses persistent TCP connections using the TCP keep-alive option. As long as there is traffic within the timeout period, the connection is kept open. This reduces the overhead for the TCP handshake that would otherwise precede every call to a service. At the same time, persistent TCP connections have the advantage that it can be easily detected when the host on the other side becomes unavailable. In many cases, a clean shutdown of the service can be done before a client attempts to send a message to a no longer available host so that the client encounters a regular OSGi event as opposed to a runtime exception.

When a connection through a network channel is established, the two peers exchange symmetric leases (Figure 4.4). A lease contains the names of the services that the peer offers as well as the event topics the peer is interested in. The latter is used in the context of remote events as discussed in the following Section 4.3.7. In R-OSGi, unlike in systems like Jini, a lease is more a contract between the two peers than a temporal
limitation. Whenever changes to services or to subscriptions are announced through the lease, the protocol mandates that the peer that has issued the lease sends an invalidation containing the delta so that a new lease can be constructed or the channel can be closed.

### 4.3.7 Remote Events

The second important case for explicit communication between modules is through asynchronous events. R-OSGi therefore extends the OSGi concept of events as described in the R4 specification of the `EventAdmin` service. This central service acts as an event broker and relay within a single OSGi framework instance. Posting events requires the invocation of a method on the `EventAdmin` service object, subscribing for events is done through the whiteboard pattern. A bundle registers for an event type by registering an `EventHandler` service together with the property `event.topics` and the optional property `event.filter` which is matched against the property set of occurring events. Topic strings of events follow a hierarchical structure\(^2\) and can be matched using wildcards.

In R-OSGi, event distribution is implemented as a whiteboard pattern over the now distributed service registry. Bundles initially register the `EventHandler` in the local service registry, just as in the non-distributed case. The subscription is announced to peers through the symmetric lease transmitted during the connection establishment phase. On the other side of the channel, an `EventHandler` is registered locally for the stated topics and if any matching events are outstanding, they are sent back through the channel. To publish an event, bundles post it to the local `EventAdmin` service which then sends it to all registered listeners.

In practice, a fully transparent forwarding of events has disadvantages. Most prominently, some OSGi framework implementations detect the presence of an EventAdmin and additionally post their internal events through this mechanism. These events, however, then contain properties like service references which are neither in a Java object serialization sense serializable, nor would it make sense to serialize them since they reference framework-internal entities in a local OSGi system. Therefore, a opt-in or opt-out strategy can be applied to limit the events transmitted between connected hosts.

---

\(^2\)Typically the naming schema for events adheres to the common Java reverse domain name notation also used for packages.
4.3.8 Supporting Streaming Data Services

Service invocation has the property of partitioning data into discrete units, i.e., into the arguments and return values of method calls. Events, on the other hand, implement a push-based communication pattern of discrete units of data. For some applications, however, the client wants a continuous stream of data from which it can pull on demand.

In Java, the exchange of such streaming data is typically implemented using instances of `InputStream` and `OutputStream` of the `java.io` package and its subclasses. In the case of interaction through services, the service returns an instance of a stream to the client and data is then transmitted between the service and the client through the stream. However, plain remote method invocation is not sufficient to transparently remote these services. A stream object is backed by a file, a buffer, or a network channel, i.e., a resource physically attached to the machine. When it is returned to a remote client, it has to be serialized and the backing is lost. Therefore, plain service remoting with method invocation is not sufficient to deliver streaming data produced or consumed, e.g., by many ubiquitous devices.

When R-OSGi executes a remote method call that returns an instance of `InputStream` or `OutputStream`, this instance is replaced by a handle in the result message that is sent to the calling peer. The presence of streams in method signatures can already be detected during the type injection analysis when the service is registered. On the client side, the handle is used to create a stream proxy that extends `InputStream` or `OutputStream` and this proxy is returned as result to the method invocation. Likewise, instances of `InputStream` or `OutputStream` that are passed as arguments to remote method calls are replaced with handles and represented by stream proxies in the same way. The result is a client now holding a stream proxy instance in which every call to either the single byte `read` method or the more efficient buffer-style `read(byte, int, int)` causes a request message to be sent through the network and a response message to be received. For the case of the proxified input stream, the request message contains the stream handle and the number of bytes requested and the response ships the bytes read from the original stream back to the proxy, as shown in Figure 4.5.

This mechanism allows to transparently exchange and use streams that are per se not serializable. Stream proxies look and feel like any input or output stream and thus can be used like any input or output stream, e.g., attaching a buffered stream or an object stream. Furthermore, the overhead of stream request and stream result messages is even less than the already low overhead of method invocation messages. This provides the best possible performance which is crucial as streams are often used to continu-
4.4 Experimental Evaluation

Turning OSGi into a programming model for distributed system raises the question of the implications on the performance. It has to be shown that doing so does not adversely affect applications compared to alternative technologies and programming models. The following experimental evaluation assesses the fundamental properties of the R-OSGi implementation in different scenarios.

The Javaparty/KaRMI [Haumacher2005] benchmark measures the performance of alternative RMI implementations. It calls various methods on a remote-accessible sample object using arguments of different size and complexity. We have implemented the Javaparty benchmark as an OSGi service which is transparently distributed by R-OSGi. For comparison, we have also implemented it as a service object which is distributed by RMI and as a UPnP service accessible through the Domoware UPnP implementation [Demuru2005] for OSGi. The benchmark client calls the different methods multiple times and determines the average invocation time from the accumulated runtime. Most of the arguments are instances of primitive types or primitive arrays with increasing length. We skipped the parts of the performance benchmarks that are specific to the KaRMI system and not relevant to R-OSGi.

The WSTest benchmark [Sun2004a] measures the performance of web services. It was originally used to compare web service performance in Java and in .NET. The benchmark starts a number of agents that concurrently call one of four sample methods according to a predefined mix. The arguments of the method calls are complex objects. In the variant originally used by Sun and Microsoft, only one of the methods is called

Figure 4.5: R-OSGi stream proxies
Table 4.1: Binding Time

<table>
<thead>
<tr>
<th>Service</th>
<th># Methods</th>
<th>Binding Time in µs</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>R-OSGi</td>
<td>RMI</td>
</tr>
<tr>
<td>JavaParty</td>
<td>7</td>
<td>147381</td>
</tr>
<tr>
<td>WSTest</td>
<td>4</td>
<td>97147</td>
</tr>
</tbody>
</table>

at once, concurrently by eight agents. Since UPnP is not able to use complex objects in service calls due to the lack of full type mapping, we run this benchmark only for R-OSGi and RMI.

4.4.1 Service Binding

In a first experiment, we measured the binding time. In R-OSGi, this is the time spent to establish the connection, requesting the service, receiving the interface, and building the proxy. For RMI, this is the time needed to establish the connection and to download the stub from the codebase. The results are presented in Table 4.1. As the Table shows, R-OSGi performs better than RMI even though the client has more work to do. From our observations, the download of the stub is the source of the overhead in RMI. The differences between the two benchmarks are because the binding time depends on the complexity of the service and, in this case, this complexity is related to the number of service methods. The benchmarks show that R-OSGi is more efficient in terms of binding time than RMI, an interesting result given the additional functionality that R-OSGi provides.

We have also tested how R-OSGi scales down to mobile devices by measuring the binding time for the Javaparty service on a Sharp Zaurus 5500 PDA with 802.11b wireless LAN. The PDA has a StrongArm SA-1110 CPU running at 206 MHz and 64 MiB RAM, and runs cvm [Sun2002], Sun’s implementation of the J2ME CDC Personal Profile. The measured binding time on this device is 1585 milliseconds. This is a much higher overhead but comparable with the latency of such operations on mobile devices. Furthermore, this penalty has to be paid only once for each service.

4.4.2 Service Invocation

In a second experiment we compare the cost of invoking a remote service in R-OSGi, RMI, and UPnP using the Javaparty benchmark. Since UPnP does not support complex objects as arguments in service method calls, not all test methods of the benchmarks could be implemented for UPnP.
4.4.3 Services on Consumer Devices

The benchmarks have been measured with the services running on an IBM Thinkpad R32 notebook with an 1.6 GHz Intel Pentium 4 Mobile CPU and with 512 MiB RAM. The client was a Pentium 4, 3 GHz Desktop machine with 1 GiB RAM. The network between client and service host is a 100 Mbps switched Ethernet. Both devices ran a Sun 32 bit J2SE 1.5 Java virtual machine. The R-OSGi version used in the benchmarks corresponds to release 0.64.

The results are shown in Table 4.2. R-OSGi performs slightly better than RMI in many cases, especially when the arguments are complex objects. The general conclusion, however, is that R-OSGi is competitive with RMI in all cases.

We also measured the round trip time in the test network which was 193 $\mu$s (+-7 $\mu$s). Compared with this value, the ping() method using R-OSGi has an overhead of only 1.5% whereas for RMI it is about 16%. Those tests that can be run with UPnP have an execution time two orders of magnitude larger than R-OSGi and RMI. The main reason is the high verbosity (resulting in higher network delays) and the expensive parsing of the XML involved.

A similar comparison was done using the WSTest benchmark (Table 4.3) where we measured both response time and throughput. R-OSGi has a lower response time per method and a higher throughput. We also tested the scalability of R-OSGi using this benchmark. The proposed setup of the WSTest specifications uses only eight agents. When, for instance, the echoVoid method is called by 80 concurrent agents, the response time for R-OSGi increases by only 5% whereas it increases by about 23% for RMI. This indicates that R-OSGi scales very well, even for large setups with massive distribution.

4.4.4 Services in Enterprise Setups

The consumer device setup resembles the state of the art when R-OSGi was first released as open source. Besides its original problem domain, R-OSGi has followed the trend of OSGi and increasingly gained popularity in the enterprise space. Hence, we measured the performance of R-OSGi on more contemporary hardware in a cluster setup. The machines used are dual socket Intel Xeon L5520 2.26 Ghz IBM x3550 servers with Hyperthreading enabled and 24 GiB RAM. The servers run in a cluster setup connected through a 1 Gbps network switch. As virtual machine, we use a 1.6.b18 OpenJDK 64-Bit Server VM. The R-OSGi version used corresponds to the version 1.0 release.

Figure 4.6 shows the comparison of R-OSGi and UPnP for selected methods of the Javaparty benchmark. The considerably higher computational power of the hosts and
### Table 4.2: Javaparty benchmark results

<table>
<thead>
<tr>
<th>Method invoked</th>
<th>Invocation Time in µs and STD</th>
<th>R-OSGi</th>
<th>RMI</th>
<th>UPnP</th>
</tr>
</thead>
<tbody>
<tr>
<td>void ping()</td>
<td>195.81 ±0.52</td>
<td>225.18 ±0.73</td>
<td>87938.45 ±174.04</td>
<td></td>
</tr>
<tr>
<td>int ping()</td>
<td>214.63 ±0.47</td>
<td>227.98 ±0.64</td>
<td>87335 ±27.83</td>
<td></td>
</tr>
<tr>
<td>void ping(int)</td>
<td>216.83 ±0.43</td>
<td>227.17 ±0.78</td>
<td>87844.28 ±191.74</td>
<td></td>
</tr>
<tr>
<td>void ping(int, int)</td>
<td>227.04 ±0.42</td>
<td>228.88 ±0.50</td>
<td>88558.57 ±126.76</td>
<td></td>
</tr>
<tr>
<td>void ping(null)</td>
<td>202.97 ±0.39</td>
<td>228.03 ±0.47</td>
<td>-</td>
<td></td>
</tr>
<tr>
<td>void ping(Integer)</td>
<td>218.30 ±0.41</td>
<td>324.85 ±1.16</td>
<td>-</td>
<td></td>
</tr>
<tr>
<td>void ping(byte[1])</td>
<td>246.26 ±0.55</td>
<td>273.34 ±1.31</td>
<td>88770 ±122.24</td>
<td></td>
</tr>
<tr>
<td>void ping(byte[2])</td>
<td>246.23 ±0.42</td>
<td>273.65 ±0.54</td>
<td>88822.85 ±48.61</td>
<td></td>
</tr>
<tr>
<td>void ping(byte[4])</td>
<td>246.58 ±0.51</td>
<td>274.16 ±0.55</td>
<td>88832.85 ±40.95</td>
<td></td>
</tr>
<tr>
<td>void ping(byte[8])</td>
<td>247.94 ±0.51</td>
<td>274.41 ±0.51</td>
<td>88948.57 ±86.42</td>
<td></td>
</tr>
<tr>
<td>void ping(byte[16])</td>
<td>249.46 ±0.49</td>
<td>275.37 ±0.56</td>
<td>89088.57 ±39.07</td>
<td></td>
</tr>
<tr>
<td>void ping(byte[32])</td>
<td>252.98 ±0.51</td>
<td>277.17 ±0.68</td>
<td>89122.85 ±15.77</td>
<td></td>
</tr>
<tr>
<td>void ping(byte[64])</td>
<td>257.39 ±0.47</td>
<td>284.274 ±0.457</td>
<td>89055.714 ±19.166</td>
<td></td>
</tr>
<tr>
<td>void ping(byte[128])</td>
<td>270.14 ±0.70</td>
<td>295.53 ±0.59</td>
<td>89090 ±40.70</td>
<td></td>
</tr>
<tr>
<td>void ping(byte[256])</td>
<td>278.69 ±0.63</td>
<td>317.38 ±0.47</td>
<td>89162.85 ±38.43</td>
<td></td>
</tr>
<tr>
<td>void ping(byte[512])</td>
<td>337.61 ±0.81</td>
<td>363.59 ±0.69</td>
<td>89201.42 ±104.53</td>
<td></td>
</tr>
<tr>
<td>void ping(byte[1024])</td>
<td>429.25 ±0.96</td>
<td>457.97 ±0.94</td>
<td>89467.14 ±32.38</td>
<td></td>
</tr>
<tr>
<td>void ping(byte[2048])</td>
<td>532.44 ±1.03</td>
<td>582.42 ±1.19</td>
<td>89997.5 ±24.87</td>
<td></td>
</tr>
<tr>
<td>void ping(byte[4096])</td>
<td>692.89 ±1.07</td>
<td>718.17 ±1.15</td>
<td>91098.75 ±63.13</td>
<td></td>
</tr>
<tr>
<td>void ping(byte[8192])</td>
<td>1275.49 ±7.60</td>
<td>1095.5 ±2.29</td>
<td>98631.42 ±3185.51</td>
<td></td>
</tr>
<tr>
<td>void ping(byte[16384])</td>
<td>1903.20 ±11.19</td>
<td>1872.35 ±7.36</td>
<td>97718.57 ±36.02</td>
<td></td>
</tr>
<tr>
<td>void ping(byte[32768])</td>
<td>3941.77 ±65.53</td>
<td>3932.06 ±52.93</td>
<td>157588.57 ±93.26</td>
<td></td>
</tr>
<tr>
<td>void ping(float[1])</td>
<td>251.20 ±0.59</td>
<td>275.15 ±0.75</td>
<td>-</td>
<td></td>
</tr>
<tr>
<td>void ping(float[2])</td>
<td>252.20 ±0.57</td>
<td>276.01 ±0.5</td>
<td>-</td>
<td></td>
</tr>
<tr>
<td>void ping(float[4])</td>
<td>253.92 ±0.64</td>
<td>277.67 ±0.37</td>
<td>-</td>
<td></td>
</tr>
<tr>
<td>void ping(float[8])</td>
<td>256.83 ±0.52</td>
<td>279.99 ±0.79</td>
<td>-</td>
<td></td>
</tr>
<tr>
<td>void ping(float[16])</td>
<td>262.09 ±0.48</td>
<td>287.20 ±0.60</td>
<td>-</td>
<td></td>
</tr>
<tr>
<td>void ping(float[32])</td>
<td>273.85 ±0.5</td>
<td>297.67 ±0.66</td>
<td>-</td>
<td></td>
</tr>
<tr>
<td>void ping(float[64])</td>
<td>296.17 ±0.74</td>
<td>317.40 ±0.56</td>
<td>-</td>
<td></td>
</tr>
<tr>
<td>void ping(float[128])</td>
<td>344.24 ±0.70</td>
<td>369.27 ±2.41</td>
<td>-</td>
<td></td>
</tr>
<tr>
<td>void ping(float[256])</td>
<td>439.99 ±0.92</td>
<td>470.15 ±7.57</td>
<td>-</td>
<td></td>
</tr>
<tr>
<td>void ping(float[512])</td>
<td>551.24 ±1.21</td>
<td>605.09 ±9.46</td>
<td>-</td>
<td></td>
</tr>
<tr>
<td>void ping(float[1024])</td>
<td>723.89 ±1.59</td>
<td>749.48 ±3.62</td>
<td>-</td>
<td></td>
</tr>
<tr>
<td>void ping(float[2048])</td>
<td>1224.91 ±2.27</td>
<td>1251.54 ±10.05</td>
<td>-</td>
<td></td>
</tr>
<tr>
<td>void ping(float[4096])</td>
<td>1954.01 ±11.07</td>
<td>1945.25 ±38.72</td>
<td>-</td>
<td></td>
</tr>
<tr>
<td>void ping(float[8192])</td>
<td>4105.28 ±77.57</td>
<td>3982.53 ±59.83</td>
<td>-</td>
<td></td>
</tr>
<tr>
<td>void ping(float[16384])</td>
<td>8036.28 ±132.49</td>
<td>7916.87 ±132.72</td>
<td>-</td>
<td></td>
</tr>
<tr>
<td>void ping(float[32768])</td>
<td>13460.10 ±131.23</td>
<td>13839.06 ±104.92</td>
<td>-</td>
<td></td>
</tr>
<tr>
<td>void ping(DM(1024^2))</td>
<td>918597.93 ±1306.3</td>
<td>923121.21 ±12276</td>
<td>-</td>
<td></td>
</tr>
<tr>
<td>void ping(DM(2048^2))</td>
<td>3557125 ±16284</td>
<td>3614843.75 ±23682</td>
<td>-</td>
<td></td>
</tr>
</tbody>
</table>
Chapter 4. Turning OSGi into a Programming Model for Distributed Systems 79

Table 4.3: WS Test benchmark results

<table>
<thead>
<tr>
<th>Test</th>
<th>R-OSGi Resp. time (µs)</th>
<th>R-OSGi Throughput</th>
<th>RMI Resp. time (µs)</th>
<th>RMI Throughput</th>
</tr>
</thead>
<tbody>
<tr>
<td>echoVoid</td>
<td>5799.109</td>
<td>1378.583</td>
<td>10914.879</td>
<td>732.583</td>
</tr>
<tr>
<td>echoStruct</td>
<td>11464.700</td>
<td>697.633</td>
<td>14067.500</td>
<td>568.533</td>
</tr>
<tr>
<td>echoList</td>
<td>12238.550</td>
<td>653.500</td>
<td>15390.130</td>
<td>519.767</td>
</tr>
<tr>
<td>echoSynthetic</td>
<td>2439.700</td>
<td>3275.567</td>
<td>3069.710</td>
<td>2604.667</td>
</tr>
</tbody>
</table>

Figure 4.6: Javaparty benchmark, R-OSGi versus UPnP
the factor 10 faster network lead to a better performance of UPnP. Yet, there is still an order of magnitude difference in all cases.

When comparing to the RMI implementation of Java 6 (Figure 4.7), R-OSGi is still competitive. Both protocol benefit in about the same way from the increased resources available. The detailed results are summarized in Table 4.4. Whereas the method invocations with small argument size are mainly bound by the network latency, serialization overhead and network throughput are the limiting factors for larger argument sizes. Both protocol reach a de-facto throughput of about 111 MiB/s when invoking the ping method taking a DoubleMatrix object as an argument. This is a typical maximum observed in practice, the theoretical maximum (ignoring TCP and IP headers, etc.) for a gigabit network is 119.21 MiB/s.

Especially for enterprise application, the threading model can play an important role. By default, R-OSGi runs service-side one separate thread per connection. This is equivalent to the implemented behavior of RMI, even though the RMI specification does not guarantee any particular mapping of remote objects to threads [Sun2004b]. In complex setups, however, this might not be sufficient and could potentially cause remote deadlocks. A simple example is when one method of the service waits on the monitor of an object and another service method signals all waiting threads with a notifyAll
call. This situation happens, e.g., in a producer/consumer pattern when the queue is idle. In a non-distributed case with two threads, the consumer thread can wait on the idle queue and the producer thread can wake the consumer thread up after it has placed a work item into the queue. If the queue not becomes a remote service and the server-side threading model is single-threaded (per connection), a client hosting a consumer as well as a producer thread will deadlock when the consumer thread is the first to call the remote service. The single server-side thread handling the client connection will wait infinitely on the empty queue and the request of the client-side producer thread is never handled.

This simple example illustrates the importance of the server-side threading model. Unfortunately, a service cannot foresee by how many clients it will be called and, except for a sophisticated deadlock detection based on code analysis, cannot predict how many threads it needs to avoid a deadlock scenario. From the perspective of the R-OSGi service provider, for any given number of threads, there can always be a service that deadlocks with these \( n \) threads and requires \( n + 1 \) threads to run correctly. Hence, the most practical solution that is also typically chosen by application servers is to use a thread pool for handling remote requests and expose the size of the thread pool to the administrator as a tunable parameter. Inevitably, decoupling the handling of the client connection from the service invocation through a pool of worker threads introduces additional overhead. Figure 4.8 shows a comparison of the Javaparty benchmark running with a single-thread per connection versus a version running with a thread pool with two worker threads. The overhead of the thread pool and the dispatching of requests to workers is about 11 \( \mu s \) and independent of the size of the arguments. It hence proportionally hurts the latency of the most simple method calls (9% for a ping without arguments) much more than calls to method with large arguments (0.015% for the DoubleMatrix ping) where serialization and network propagation delay are predominant. In addition, the case of the JavaParty benchmark, the invocation time of the service methods are very small, i.e., they do not contribute to the end-to-end latency. More computation-intensive services would likely yield a decrease of invocation time for the worker thread model in situations in which the service is under load.

### 4.4.5 Events

The main dimension of evaluation for the asynchronous event mechanism is the number of events that can be transmitted in a given time period. Clearly, this metric depends on the size of the event to be transmitted. In the OSGi EventAdmin specification that R-OSGi extends, events have a topic that needs to be transmitted and in addition can
### 4.4. Experimental Evaluation

#### Table 4.4: Javaparty Benchmark Results, Cluster

<table>
<thead>
<tr>
<th>Method invoked</th>
<th>Invocation Time in µs and STD</th>
<th>R-OSGi</th>
<th>RMI</th>
<th>UPnP</th>
</tr>
</thead>
<tbody>
<tr>
<td>void ping()</td>
<td></td>
<td>115.10 ±12.09</td>
<td>149.64 ±41.07</td>
<td>1445.068 ±45.97</td>
</tr>
<tr>
<td>int ping()</td>
<td></td>
<td>119.42 ±0.40</td>
<td>118.67 ±0.64</td>
<td>1446.482 ±43.25</td>
</tr>
<tr>
<td>void ping(int)</td>
<td></td>
<td>119.00 ±0.23</td>
<td>118.91 ±1.04</td>
<td>1447.172 ±35.14</td>
</tr>
<tr>
<td>void ping(int, int)</td>
<td></td>
<td>119.06 ±0.66</td>
<td>115.44 ±0.22</td>
<td>1471.171 ±21.52</td>
</tr>
<tr>
<td>void ping(null)</td>
<td></td>
<td>119.97 ±0.16</td>
<td>116.61 ±0.38</td>
<td>-</td>
</tr>
<tr>
<td>void ping(Integer)</td>
<td></td>
<td>123.14 ±0.25</td>
<td>140.94 ±0.79</td>
<td>-</td>
</tr>
<tr>
<td>void ping(byte[1])</td>
<td></td>
<td>140.96 ±11.25</td>
<td>130.11 ±0.20</td>
<td>1456.422 ±22.00</td>
</tr>
<tr>
<td>void ping(byte[2])</td>
<td></td>
<td>135.95 ±0.41</td>
<td>129.85 ±0.32</td>
<td>1462.911 ±22.25</td>
</tr>
<tr>
<td>void ping(byte[4])</td>
<td></td>
<td>135.20 ±0.21</td>
<td>130.32 ±0.53</td>
<td>1453.091 ±12.96</td>
</tr>
<tr>
<td>void ping(byte[8])</td>
<td></td>
<td>136.08 ±0.17</td>
<td>129.81 ±0.40</td>
<td>1452.560 ±13.10</td>
</tr>
<tr>
<td>void ping(byte[16])</td>
<td></td>
<td>136.32 ±0.17</td>
<td>136.81 ±1.43</td>
<td>1450.385 ±9.15</td>
</tr>
<tr>
<td>void ping(byte[32])</td>
<td></td>
<td>136.40 ±0.12</td>
<td>135.88 ±3.27</td>
<td>1466.538 ±27.42</td>
</tr>
<tr>
<td>void ping(byte[64])</td>
<td></td>
<td>137.08 ±0.16</td>
<td>130.85 ±0.18</td>
<td>1450.210 ±13.98</td>
</tr>
<tr>
<td>void ping(byte[128])</td>
<td></td>
<td>132.63 ±0.33</td>
<td>132.17 ±0.39</td>
<td>1463.363 ±23.99</td>
</tr>
<tr>
<td>void ping(byte[256])</td>
<td></td>
<td>135.36 ±0.12</td>
<td>135.29 ±0.15</td>
<td>1462.656 ±17.49</td>
</tr>
<tr>
<td>void ping(byte[512])</td>
<td></td>
<td>137.61 ±0.19</td>
<td>143.43 ±3.28</td>
<td>1488.731 ±30.20</td>
</tr>
<tr>
<td>void ping(byte[1024])</td>
<td></td>
<td>155.39 ±0.18</td>
<td>162.98 ±0.23</td>
<td>1518.262 ±29.54</td>
</tr>
<tr>
<td>void ping(byte[2048])</td>
<td></td>
<td>171.03 ±0.14</td>
<td>182.76 ±0.21</td>
<td>1564.433 ±38.48</td>
</tr>
<tr>
<td>void ping(byte[4096])</td>
<td></td>
<td>202.16 ±0.25</td>
<td>207.34 ±5.37</td>
<td>1799.876 ±55.77</td>
</tr>
<tr>
<td>void ping(byte[8192])</td>
<td></td>
<td>241.07 ±1.48</td>
<td>236.36 ±1.73</td>
<td>2114.380 ±35.95</td>
</tr>
<tr>
<td>void ping(byte[16384])</td>
<td></td>
<td>330.81 ±1.03</td>
<td>328.56 ±6.80</td>
<td>2801.818 ±52.44</td>
</tr>
<tr>
<td>void ping(byte[32768])</td>
<td></td>
<td>473.30 ±0.97</td>
<td>484.90 ±0.82</td>
<td>4346.722 ±85.67</td>
</tr>
<tr>
<td>void ping(float[1])</td>
<td></td>
<td>132.86 ±0.52</td>
<td>140.85 ±0.13</td>
<td>-</td>
</tr>
<tr>
<td>void ping(float[2])</td>
<td></td>
<td>132.01 ±0.87</td>
<td>140.46 ±0.27</td>
<td>-</td>
</tr>
<tr>
<td>void ping(float[4])</td>
<td></td>
<td>124.54 ±1.30</td>
<td>132.71 ±0.38</td>
<td>-</td>
</tr>
<tr>
<td>void ping(float[8])</td>
<td></td>
<td>123.26 ±0.16</td>
<td>132.81 ±0.24</td>
<td>-</td>
</tr>
<tr>
<td>void ping(float[16])</td>
<td></td>
<td>123.95 ±0.14</td>
<td>134.01 ±0.21</td>
<td>-</td>
</tr>
<tr>
<td>void ping(float[32])</td>
<td></td>
<td>125.86 ±0.20</td>
<td>136.07 ±0.45</td>
<td>-</td>
</tr>
<tr>
<td>void ping(float[64])</td>
<td></td>
<td>128.81 ±0.18</td>
<td>146.73 ±0.17</td>
<td>-</td>
</tr>
<tr>
<td>void ping(float[128])</td>
<td></td>
<td>139.05 ±2.91</td>
<td>146.33 ±2.75</td>
<td>-</td>
</tr>
<tr>
<td>void ping(float[256])</td>
<td></td>
<td>160.93 ±1.65</td>
<td>162.93 ±0.45</td>
<td>-</td>
</tr>
<tr>
<td>void ping(float[512])</td>
<td></td>
<td>185.50 ±0.22</td>
<td>181.55 ±0.17</td>
<td>-</td>
</tr>
<tr>
<td>void ping(float[1024])</td>
<td></td>
<td>223.74 ±2.47</td>
<td>218.80 ±0.28</td>
<td>-</td>
</tr>
<tr>
<td>void ping(float[2048])</td>
<td></td>
<td>275.85 ±0.35</td>
<td>295.30 ±1.35</td>
<td>-</td>
</tr>
<tr>
<td>void ping(float[4096])</td>
<td></td>
<td>370.89 ±1.10</td>
<td>391.91 ±1.06</td>
<td>-</td>
</tr>
<tr>
<td>void ping(float[8192])</td>
<td></td>
<td>518.56 ±1.10</td>
<td>531.06 ±1.38</td>
<td>-</td>
</tr>
<tr>
<td>void ping(float[16384])</td>
<td></td>
<td>798.94 ±0.48</td>
<td>798.69 ±8.42</td>
<td>-</td>
</tr>
<tr>
<td>void ping(float[32768])</td>
<td></td>
<td>1344.33 ±9.68</td>
<td>1352.58 ±1.22</td>
<td>-</td>
</tr>
<tr>
<td>void ping(DM(1024,1024))</td>
<td></td>
<td>71875.00 ±13063</td>
<td>72000.00 ±300.00</td>
<td>-</td>
</tr>
<tr>
<td>void ping(DM(2048,2048))</td>
<td></td>
<td>286000.00 ±16284</td>
<td>285900.00 ±300.00</td>
<td>-</td>
</tr>
</tbody>
</table>
contain arbitrary key-value pairs as properties. The experimental setup used for evaluation of the event mechanism consists of a host serving as an event source that transmits events in a tight loop and a client that measures the received events per time interval. The size of the events is varied by introducing a single property having a value in the form of a byte array. The byte array gives a good estimation for arbitrary other properties of the same aggregate size as long as the time spent for serialization remains significantly smaller than the network propagation delay. Figure 4.9 shows the received events per second as a function of the size of this byte array (the payload). Overall, R-OSGi is able to handle high volumes of events, in this configuration up to 37,000 events per second. The first point to be observed is the difference between events without payload and events with a small payload. The explanation for this behavior is that no properties corresponds to a null value in the event object and this can be much more efficiently transmitted than a Hashtable (as the most prominent implementation of a Dictionary). Hence, the difference between zero payload and a payload of 1 byte resembles the increase in message size caused by the property dictionary. After this point, the event throughput remains relatively stable until a payload of 1024 bytes has been reached. At this point, the message size exceeds the MTU of the network (here...
1500 bytes) and fragmentation splits the message into multiple packets. From this point on, doubling the payload yields in approximately half the event throughput.

### 4.4.6 Streaming

The third pattern of interaction between modules is the use of continuous data streams. In the experiment, one host provides a service with a single method that returns an `InputStream`. A client calls this method through an R-OSGi remote service invocation and gets a stream proxy bound to the original stream of the service. In the service implementation, the stream is fed by reading from a large (1 GiB) data file filled with random content.

The performance of the streaming highly depends on the amount of data pulled from the stream on the client side. Figure 4.10 shows the measured throughput in MiB/s as a function of the client buffer size, which is the buffer handed to the `read` method of the `InputStream`. R-OSGi’s stream proxy implementation sends the size of the client buffer with the request message so that as much data is transmitted as the client can handle. When this buffer is small and the payload per message is consequently small, the overhead of the messaging becomes dominant and the reached throughput is small.
4.5 Alternative Transports Protocols for Mobile and Ubiquitous Devices

Many aspects of the OSGi and R-OSGi model are perfectly matching the requirements of applications on mobile and ubiquitous devices. First, the devices involved in these networks form a heterogeneous set. Different hardware platforms and operating systems are available on the market and in use. The abstractions provided by the Java VM dramatically simplify the development of software for these devices. Furthermore, ubiquitous networks often involve a large quantity of devices that have to be managed by the user. The module lifecycle management of OSGi allows to consistently update software modules among all devices.
4.5. Alternative Transports Protocols for Mobile and Ubiquitous Devices

However, these devices are highly resource-constrained and thus often unable to run a Java virtual machine. They also often lack a TCP/IP interface and use other, less expensive communication protocols (e.g., Bluetooth [BluetoothSIG1998], ZigBee [ZigBeeAlliance2002]). Since services are generally used to abstract away from concrete communication and hide implementation details behind the service interface, this principle can be used to provide a variety of transports for remote service invocation and other communication patterns. Due to its own modular and extensible design, R-OSGi is able to seamlessly interact through alternative transport protocols typically used with mobile and ubiquitous devices.

4.5.1 Bluetooth Transport

To support R-OSGi services over Bluetooth radios, we need to separate the service identifier from the underlying transport. Hence, instead of SLP service URLs, we use opaque URIs. The URI of a service can use any schema and thereby identify which transport implementation is required. By doing this, the address of a service now also contains information on the transport protocol it supports. For instance, the URI of a service accessible through the TCP/IP can be: \texttt{r.osgi://some.host\#21}. The network channel through which the service can be accessed is identified by the URI’s schema, host, and port components (if differing from the default port). The fragment part of the URI describes the local identifier that the service has on the other node. The same service through Bluetooth transport would be, e.g., \texttt{btspp://0010DCE96CB8:1\#21}.

Bluetooth is a network solution covering the lower five layers of the OSI model [Zimmermann1980]. In our implementation we use RFCOMM, which provides a reliable end-to-end connection, as transport protocol. We modified the channel model of R-OSGi so that every implementation of a transport protocol that can accept incoming connections can natively create and register a channel endpoint for them. Thus, messages received via Bluetooth do not have to be bridged over TCP but can be processed directly by the corresponding channel endpoint.

To evaluate the performance of the Bluetooth implementation, we tested the system again with the JavaParty benchmark. Figure 4.11 shows the results of the benchmark on Bluetooth, compared to the baseline (first column), which is an ICMP ping over Bluetooth PAN. The invocation of the void method with no arguments takes only slightly more time than the ICMP ping. This shows that R-OSGi adds little overhead. Furthermore, when the size of the arguments increases, the invocation times scales relative to the size of the arguments.
4.5.2 Bluetooth Service Discovery

R-OSGi supports service discovery in a modular fashion through provider plugins. Alternative discovery protocols can be used since the service discovery operates behind a common discovery handler interface that is not restricted to a particular implementation, e.g., SLP. To match the Bluetooth transport we implemented a corresponding discovery handler, which offers Bluetooth service discovery via Bluetooth’s own Service Discovery Protocol (SDP) [BluetoothSIG1998].

A major challenge of the OSGi service discovery with SDP is the granularity mismatch. In the Bluetooth service model, R-OSGi itself is the Bluetooth service and the OSGi services it offers are conceptually sub-entities of the R-OSGi service. However, SDP has no notion of service hierarchy. Furthermore, SDP is very limited in the way how services can be described, which leaves a semantic gap between Bluetooth services and OSGi services. A distinct feature of the OSGi service model is that requests can be filtered to reduce the result set to the subset in which the client is interested. When running service discovery over the network, the ability to push selectivity to the edges of the network yields in a reduction of network consumption. SDP supports queries but only on UUIDs (Universally Unique Identifiers) [OASIS2004] whereas OSGi allows semantically rich and complex LDAP filter strings [Howes1996]. In addition, SDP data types

---

Figure 4.11: Javaparty benchmark over Bluetooth 1.2 transport
do not suffice to express the properties of OSGi services. This can lead to problems when selecting services as the limited expressibility in SDP results in many more services answering a request than is necessary. Unfortunately, this hurts the performance on a network technology like Bluetooth with a limited bandwidth over-proportionally.

To provide OSGi-like service discovery over Bluetooth, the discovery implementation keeps track of services that are marked to be remotely accessible through SDP. The most common form of service filter involves the service interfaces. In order to support searches similar to those in OSGi, the UUID of every service interface is adapted so that filters over service interfaces can be expressed as SDP service searches over the corresponding UUID. In the implementation, the UUID is a hash over the interface name. For each service, the serialized properties are stored as sequences of bytes in the service record. These properties are retrieved lazily and only if a filter expression poses constraints over these particular properties. A modification time-stamp indicates the version of the properties so that clients can cache them for future filters and easily check if the cached version is still valid. A hash value over all service blocks is used to detect if new services were added or some were removed since the last search. Through this structure, OSGi services can be fully described directly in the Bluetooth service discovery mechanisms while at the same time minimizing the amount of data that has to be transferred per request and particularly optimizing the protocol for the common case (filter on the service interface names).
4.5.3 802.15.4 Transport

The IEEE 802.15.4 standard [IEEE2006] defines PHY and MAC layers for short range, low power radios. 802.15.4 is the basis for the lower layers of ZigBee, which represents a full network solution including upper layers not defined by the 802.15.4 standard.

In contrast to TCP/IP and Bluetooth, 802.15.4 only defines the lowest two layers of the OSI model and thus does not provide a reliable transport on its own. While ZigBee would provide the missing layers and reliable transport, this solution provides a lightweight solution that can even run on, e.g., TinyOS powered sensor network devices. The R-OSGi transport has been implemented atop a simple transport layer developed in our research group. This layer is similar to TCP by using acknowledgments, timeouts, and retransmissions to provide reliable end-to-end communication on top of 802.15.4. The implementation of the transport layer represents a proof of concept only and is not optimized in any way. Reliable transport over packet oriented lower layers is a research topic on its own and we will eventually exchange our implementation with a more sophisticated algorithm tailored to the characteristics of 802.15.4 radios.

Atop of this transport we implemented an 802.15.4 network channel for R-OSGi, similar to the implementation for Bluetooth. Since the message format used is compatible to TinyOS’ active messages, the URI schema for this transport is tos. An example of a URI for this transport is tos://100#5 with tos being the schema, 100 an example for a node address, and 5 being the service identifier on the peer. The transport behaves like Bluetooth or TCP transports. Once a connection of the underlying layer is detected, a logical R-OSGi channel to this peer is established over the connection and message exchange can start.

4.6 Non-Java Services

One obvious limitation of the R-OSGi implementation is that it requires at least a Java CDC VM and an OSGi framework to run on the participating nodes. For small devices, we had to address this issue and come up with solutions for devices with limited computation power and resources. This also includes embedded systems on micro-controller architectures with no Java VM and limited operating system support.

When two R-OSGi peers establish a connection they exchange a list of the services they provide through the symmetric leases. Each entry in this list comprises the names of the interfaces that the service implements and its properties. When a caller invokes a remote service, the instance of R-OSGi on the callee side sends the full interfaces including all methods to the caller. On the caller, a proxy that implements the interfaces
4.6. Non-Java Services

of the remote service is built on the fly and registered with the local OSGi framework like other local services. Conceptually, the integration of non-java services exploits the fact that the service is created and registered as a local proxy service on the caller side to hide non-OSGi services behind an OSGi-like interface. The bridging between the caller – which must be a full OSGi framework – and the callee is done at the proxy.

A common obstacle for integrating devices programmed in different languages is the transfer of data. Since OSGi services are inherently Java-centric, R-OSGi uses the Java type system as a pervasive data exchange format and thereby the Java serialization protocol as a wire format. This mandates all participating devices to understand the Java serialization protocol. However, the requirements for a service are much more restricted. The communication between a service and a client is fully described by the service interface. Not only does the interface define the verbs used—in the form of the service methods—it also defines the nouns—in the form of the formal parameter and return type. Hence, a service providing host only has to understand how to serialize the types occurring in the service interfaces, which is an entity over which the service has full control and that is created as an integral part in the design of a service.

In practical terms, this means that a low-performance host can decide to only offer services using primitive types and not using exceptions and thereby omitting most of the serialization protocol. A more powerful node can define and use more expressive types and implement a custom subset of the serialization protocol required to handle exactly these types. In principle, the code required to serialize the types occurring in a service interface could be automatically generated from the interface.

The first platform to prove the concept and show the consequences of the R-OSGi design is an embedded Linux devices running services written in C or C++, thereby removing the dependency on Java. The second implementation works on top of TinyOS [Hill2000] running on Tmote Sky [Moteiv2005] nodes featuring 10kB of RAM and 48kB of code storage, thus removing the dependency on a full fledged operating system.

4.6.1 Services in C for Embedded Devices

Supporting services written in languages other than Java is mainly a problem of adapting the data types. The usual way to invoke native code from Java code is through the Java Native Interface (JNI) [Sun2004], an interface that exposes an API that can be used to create and access objects and to invoke methods.

The core of the R-OSGi C/C++ runtime is a stand alone library that implements the JNI interface but without actually being a Java VM (Figure 4.13). The JNI runtime library does not have the overhead of interpreting any Java byte-code and operating on
a virtual stack machine. Instead, it implements only the `JNIEnv` which is normally the handle to the JNI interface of a Java VM. The runtime library maintains internal data structures for classes, fields, methods, and object instances. Methods are implemented by C functions operating on the data structures. Common classes like `java.lang.String` or `java.lang.Integer` are already implemented and part of the library. Service implementations can register new models of Java classes through an extension mechanism, if they require other classes.

With the help of the runtime, JNI code can run without a Java VM. The R-OSGi protocol is implemented in the R-OSGi daemon, which accepts incoming TCP/IP connections and handles the requests. It is possible to implement R-OSGi daemons for different transports as well. JNI service implementations are registered with the R-OSGi daemon through a configuration file, together with the location of a Java interface implementation of the service. This interface is not used for the execution of the JNI service but only as a static resource exchanged in the R-OSGi protocol. Furthermore, the daemon also implements the Java serialization mechanism. If the class of an incoming object is known to the JNI runtime (either by default or provided by the service), the R-OSGi daemon can create a corresponding instance in the JNI runtime which can then be used by the service implementation in the same way a corresponding Java class inside a VM would be used. Hence, arguments for service calls can be deserialized (and return values serialized again) to preserve the full expressiveness of Java/OSGi but without the overhead of a Java VM.

To quantify the performance of our solution, we again use the JavaParty benchmarks.
The first series of measurements were taken on two Linux notebooks using Java 5. Figure 4.14 shows that the native implementation of R-OSGi is slightly faster in some cases but no significant performance gains can be observed. This can be expected since the limiting factor on devices with plenty of resources is the network bandwidth and not the processing. Furthermore, the HotSpot VM benefits from just-in-time compilation.

The picture changes when the same experiment is repeated on a Linksys NSLU2, an embedded Linux device with a 133 MHz Intel XScale IPX420 CPU (Figure 4.15). This time, a JamVM [Lougher2003] is used as a reference. The native R-OSGi implementation performs significantly better than the interpreting VM in all cases. Even the call with 32KiB of arguments completes in less than 20ms.

### 4.6.2 Sensor Nodes as Services

The implementation of an R-OSGi runtime for services on TinyOS [Hill2000] is based on the 802.15.4 transport presented in the preceding section and, like all TinyOS applications, implemented in NesC [Gay2003]. NesC adds the concept of components, interfaces, and wiring of components to the C programming language. The Tmote sky nodes implement the TelosB [Crossbow2004] platform of TinyOS and all hardware compo-
Figure 4.15: Javaparty benchmark on the Slug

Components are supported in the current TinyOS release 2.0.2.

Conceptually, this implementation differs largely from the implementation for embedded Linux, despite the fact that NesC is just an extension of C and the programming languages are thus very similar. Dynamic loading of shared libraries is not available on TinyOS and resources are very limited. Additionally, the capabilities and therefore the services provided by a sensor node are usually static. Therefore, the approach of emulating a full JNI environment is not feasible for sensor nodes and TinyOS. Instead, the services provided by the sensor node are statically composed at build time and also include the appropriate interfaces and data types in advance.

The R-OSGi application for TinyOS uses the 802.15.4 transport to listen for incoming connections. When a connection is established, the application composes the lease message containing the interfaces and properties of all services available by collecting the data from the installed services. Services implement a (NesC) Service interface which provides commands for getting the serialized (Java) service interfaces, getting the serialized, built-in properties of the service, and executing a method on the service. Furthermore, it defines an event that is signaled upon completion of the method invocation. This way it is easily possible to have multiple, different services installed in one node as they can be managed uniformly by the R-OSGi application. A configuration file that
is used at compile time defines the services that are available on the node.

When a remote R-OSGi client invokes a method, the R-OSGi application dispatches the call to the correct service by invoking its `invoke` command with the signature of the method and the serialized method arguments as arguments. The service then dispatches the invocation internally according to the signature. For example, a temperature service might offer `getRaw()` as well as `getCelsius()` and `getFahrenheit()`. The invocation is implemented as split-phase operation. Once the invocation has completed, `invokeDone` is signaled and the return value passed as argument. The serialized return value is then sent to the calling peer as response to the invocation of the method. To deserialize the arguments and serialize the return value, services use a `Serializer` component custom tailored to the types involved in the services.

## 4.7 Related Work

Models and frameworks for building distributed systems have a long history. The conventional approach is to make remote invocations identical to local procedure or method calls, as exemplified by Remote Procedure Calls (RPC) [Birrell1984], Java Remote Method Invocation (RMI), the Common Object Request Broker Architecture (CORBA) [OMG1995], or the Distributed Component Object Model (DCOM) [Brown1998]. While providing a form of distribution transparency at the level of invocations, the application must nevertheless be manually factored into distributed components, and the large-scale structure of the application usually reflects this factoring. The same is generally true for analogous operating system-based approaches, such as Amoeba [Mullender1990] or SOS [Shapiro1989].

Alternatively, centralized applications written in a component framework can be automatically factored into distributed components. Coign [Hunt1999] partitions COM-based Windows applications into two parts that can be distributed in a client/server configuration. Coign instruments the code through binary rewriting, analyzes the dependencies between COM components and calculates a graph-cutting according to a cost metric for introducing network communication between the subgraphs. Similarly, JOrchestra [Tilevich2002] automatically partitions a program by rewriting byte-code to replace local methods with remote invocations, and object references with proxy references. In these approaches, the distribution is orthogonal to the original design, and occurs along object boundaries which were typically not designed with distribution in mind, giving rise to the kind of transparency and performance problems described in [Waldo1994].
Recent centralized module management systems, e.g., MJ [Corwin2003] and OSGi, in contrast to typical component frameworks, impose boundaries between modules which are explicit at the level of program code. This is done to better deal with dynamically loading, updating, and unloading of modules at runtime.

However, the existing efforts to add distribution support to OSGi have either followed the OSGi specifications in providing protocol adapters to existing Jini [Waldo1999, Sun2005] and Universal Plug and Play (UPnP) [UPnP2000] infrastructures, or (as with the Newton Project [Paremus2006]) introduce an additional component model for distribution independent of OSGi's module boundaries and based on an existing infrastructure like Jini. Both approaches are what might be termed “invasive”: they require the application to be explicitly structured (or restructured) around the distribution model provided by Jini or UPnP, and hence the application must be factored in such a way as to conform to one of these component models. At the time where R-OSGi was developed, there was no way to remote OSGi instances without losing the generality of the OSGi model, or, equivalently, to allow an OSGi application to be easily distributed along OSGi module boundaries.

This has recently changed with the release of the OSGi Enterprise Specifications [OSGiAlliance2010], which has standardized the RemoteServiceAdmin service. This service encapsulates the mechanism for importing remote services into the scope of a local OSGi framework and exporting local OSGi services through distribution providers. In its concept, the RemoteServiceAdmin of the Enterprise Specifications is similar to R-OSGi. In fact, R-OSGi has influenced the standard and there is a beta version of R-OSGi which implements this interface and the concepts behind it with only few technical adaptations to the original R-OSGi code. The scope of the specifications is broader than the one of R-OSGi in the sense that it is designed to work with arbitrary existing distribution providers like RMI or Web Services. However, it thereby possibly limits what kind of services can be exported to what is expressible by a concrete distribution provider. At the same time, its scope is narrower in the sense that it only deals with remote services but not with the consequences for modularity. There is no such mechanism like type injection. Instead, provisioning all required dependencies is up to the maintainer of a distributed setup. Complementing the RemoteServiceAdmin which encapsulates the mechanism, there is a TopologyManager dealing with the policy decisions, i.e., what to import from where, etc. Since non-invasiveness was not a requirement, the Enterprise Specifications can rely on the newly created Service Hooks to become aware of clients requesting services so that implementations can mix information about remote services with local matches on demand whereas R-OSGi still requires an explicit request for creating service proxies.
4.8 Summary and Discussion

Transparently distributing programs designed for a single address space context has been a problematic concept. Waldo et. al. [Waldo1994] provide a comprehensive summary of the main problems: networked systems are fundamentally different in behavior to centralized ones, and the semantics of an invocation are also fundamentally different. Consequently, the argument goes, it is unlikely that a centralized program will perform with acceptable performance, let alone correctly, when factored into distributed components. The basic problems here are communication latency and non-determinism, and unreliability (either due to message loss or partial failure of the nodes or network). These arguments are powerful and persuasive.

R-OSGi sidesteps these issues by intelligently exploiting the way that OSGi programs are already written – the assumption of unknown performance characteristics of cross-bundle calls. Furthermore, rather than masking distributed failures, R-OSGi exposes these events to application bundles, but in a form that the bundle is already designed to handle: the disappearance of service bundles through module unloading.

R-OSGi conceptually maps failures arising from the distribution of components to local service events. From the OSGi model, developers are used to guarding the code against the case that parts of the system are not available. Usually, this is done by listening to service events or using the OSGi ServiceTracker. In a purely local configuration, services can become unavailable when some entity in the system, in particular a user of the system, decides to stop or to uninstall the bundle that has provided the service. By mapping network malfunctions to these events that are already handled by the applications, we introduce no failure patterns that are not already possible in purely centralized situations.

For instance, if a service providing peer fails, we detect the breakdown of the network channel and the failure to reconnect. Having observed this, R-OSGi immediately uninstalls the proxy bundle. Even if the network operates without failures, the original service can throw exceptions. R-OSGi serializes these exceptions and rethrows them in the proxy bundle to mime the exact behavior of the original service.

OSGi Services give no guarantee about execution time regardless of whether they are local or remote. Side effects of services such as threaded design or database accesses (e.g., a persistence service), can lead to an execution time that appears to be non-deterministic from the client point of view. A user might even decide to replace a fast implementation of a service by an extended but overall slower implementation and the client has to live with this situation. Furthermore, services are often event-driven and since events in OSGi are typically dispatched asynchronously, no assumptions about
timing can be made. This is a considerable difference to plain objects, that are most of-
ten expected to execute methods within a very short time. A further difference between
R-OSGi in comparison to systems like CORBA is that the granularity of distributed
entities is much larger. In OSGi, services encapsulate whole functional units and the
dependencies between services are typically restricted to semantical dependencies at
the application level. Objects in contrast tend to have a larger number and often nested
interconnections that make bad effects of the network more severe.

The premise of the R-OSGi line of work was to show that turning OSGi into a
programming model is possible and results in more flexible, easier to program systems.
The focus of this chapter was on demonstrating the feasibility of the approach, both in
terms of the challenges involved and in terms of the implications on performance. The
following chapter demonstrates its utility based on other systems that have been build
on top of R-OSGi.
4.8. Summary and Discussion
Chapter 5

Applications of Distributed OSGi

5.1 Opportunities

R-OSGi is the mechanism to turn OSGi into a programming model for distributed systems by bridging the gap between a single Java virtual machine and a set of interconnected virtual machines. Modularization as exercised in OSGi applied to distributed systems provides ample opportunity, most importantly:

- **Separating design from deployment:** The compositional approach of deployment modularity can in the same way be applied to distributed systems. Rather than designing an application for a concrete deployment (assignment of functional units to hosts) the application is designed as a set of loosely coupled modules which can dynamically be deployed to hosts and the deployment can even change at runtime.

- **Separating application logic from communication logic:** Services and service interfaces can be used to abstract from concrete communication protocols. The code for generating and sending messages over the network is not entangled with the application logic and appropriate implementations for a concrete communication setup can be chosen at runtime.

- **Embracing dynamism:** Distributed systems consist of many hardware components which can all individually fail, exhibit misbehavior, or can simply be turned off. Hence, distributed systems are by nature dynamic systems. With the OSGi model of localized behavior and a mandate to react to changes in the environment the effects arising from this kind of dynamism can be made tractable for the application.
This chapter presents three systems built on top of R-OSGi that take advantage of these opportunities to enable advanced functionalities in distributed systems. The first application is a tool for providing drag-and-drop distribution of software modules without any need to change the code of the modules. R-OSGi provides the communication between modules that were previously designed to run on a single virtual machine. The second application provides a practical solution to problem of updates in distributed systems. The idea of versioning is already attached to bundles and packages in OSGi. Since R-OSGi preserves a consistent modularity across machine boundaries through proxy modules, type injections, and dependency management, it is possible to also keep the versions consistent. Thereby, R-OSGi can provide a practical solution to the update problem in distributed systems (Section 5.3). The last application, AlfredO, introduces a more coarse-granular but semantically richer distribution model in the form of detachable tiers which is implemented atop R-OSGi.

5.2 Drag-and-drop Distribution with the Monitoring and Deployment Tool

The R-OSGi Monitoring and Deployment Tool (RDT) turns the Eclipse IDE into a powerful tool for developing, deploying, and monitoring distributed applications.

Firstly, the RDT can analyze OSGi applications inside Eclipse and generate graphical representation based on Eclipse’s Graphical Editing Framework (GEF) [Eclipse2003]. Users of the tool can turn this initial centralized deployment into a distributed deployment by dragging and dropping bundles from one machine to another. The resulting deployment description is transformed into tasks for R-OSGi.

Secondly, RDT exposes R-OSGi’s transparent support for load-balancing and fault-tolerance. Such facilities can be added to existing applications with a few clicks, and the resulting system immediately deployed from Eclipse.

Thirdly, RDT can be used to visualize the structure and status of a running distributed application in real time, including such issues as node or network failures and concurrency issues. As well as its use in a software management context, this facility can be used to explore the impacts of different failure models and decide on appropriate fault tolerance strategies.

Finally, RDT can capture all network messages and make them available to the Eclipse user. These traces can be used by developers to for profiling, testing, debugging, and benchmarking distributed applications throughout the entire development cycle.

RDT greatly illustrates the principle of distributed OSGi of separating design from
Figure 5.1: Example setup

deployment. The design of a distributed application is conventional OSGi modules and design considerations are solely the separation of an application into functional units (modules) and interfaces between them (services). The deployment is created by the tool and in a very simple way: by dragging and dropping modules in a graphical representation of the system. Communication is added by R-OSGi along those services that now become the boundaries of distribution.

5.2.1 Implementation

The RDT itself consists of two different parts: The actual Plugin which extends the Eclipse IDE and the Deployment Agent in charge of distributed deployment. Figure 5.1 shows an example consisting of the development machine running the Eclipse instance and three nodes running each an OSGi framework with R-OSGi and the Agent. The graphical representation of the deployment is transformed into sets of declarative tasks for each nodes.

The purpose of the Plugin is to add support to Eclipse for creating deployment graphs from an existing OSGi application, allowing the user to modify the graph, and to visualize the resulting application once it is deployed. The Deployment Agent is the primary communication channel between the Eclipse instance and the OSGi frameworks running on the nodes. It mainly features remote management commands and processes the tasks disseminated by the Plugin during the deploying step. Once the application
Drag-and-drop Distribution with the Monitoring and Deployment Tool

5.2.2 Code Analysis and Graph Generation

The manifest of OSGi bundles explicitly declares package imports and exports but does not give any information about the services provided and consumed by a bundle. Therefore, the service dependency structure can be normally determined only from the running bundles of an application. The Deployment Tool uses code analysis to reason about the services used by bundles. Since the source code of the application bundles is usually available when developed in Eclipse, the tool operates on the Abstract Syntax Tree (AST) maintained by Eclipse for every Java source code resource.

As an initial step, the user specifies the main (root) bundle of the application through a wizard. This is the anchor point for the system from which the dependency analysis starts. An AST node visitor traverses the syntax trees of all source files belonging to the bundle project to identify which services are registered (through `context.registerService`) and consumed (`context.getServiceReference` and `context.getService`). For binary resources, the same could be achieved through byte-code analysis (e.g., using the ASM [Bruneton2002] library, also internally used by R-OSGi). The code analysis then recurses over all bundle dependencies of the root bundle and their dependencies.
The result of the analysis is the complete set of bundles making up the application, together with information about the registered and consumed services for each bundle. In the next step, the tool creates an initial deployment graph depicting the structure of the application if it was launched on a single OSGi framework.

### 5.2.3 Creating a Distributed Deployment

This initial graph can be manipulated by the user to turn the centralized application into a distributed one. New nodes running OSGi frameworks can be added to the graph. The bundles of the application can be dragged and dropped from one node to another, thereby changing the structure of the deployment. Dependencies between the bundles on the level of services are illustrated by connection arrows. The amount of connections between frameworks gives the developer already a coarse estimation on how much network activity a specific distribution setup might involve. Figure 5.2 shows the Deployment Editor with an opened deployment of an application on three different nodes.

For each service which is accessed by remote frameworks, the R-OSGi system transparently creates a service proxy on the consumer peers which calls the original remote service. Dependencies on the level of package imports and exports are automatically resolved by R-OSGi. Events raised through the OSGi EventAdmin service are transparently forwarded to those nodes that have a corresponding EventHandler registered.

The Deployment Tool does not only facilitate the creation of distributed applications out of software modules, it also offers to add advanced features such replicating modules for load balancing, or failover redundancy. The user can create redundant copies of bundles through the visual interface by dragging a bundle to a different node with the control key pressed. By default, these copies are unbound. With different GEF connection tools, the user can (graphically) bind the services of this copy to consuming bundles and thereby specify for which purpose the copy should be used. With the tool, the implications of distributed failures can be tested and their impact on the running application evaluated. For education purposes, different failure scenarios can be simulated by disabling nodes of the deployment, or by creating new channel implementations which simulate network failures.

R-OSGi supports binding service proxies to a single primary service and adding redundant remote services. The default behavior of R-OSGi is to remove the client-side proxy when a service becomes unreachable or a message cannot be transmitted from the client to the service. If the failover policy is defined for these additional bindings, R-OSGi will instead switch to the next service whenever the invocation of the primary service fails due to unavailability or network failures. Two different load balancing
strategies are selectable. The **ONE** strategy selects the best available service on the first invocation (the one with the least service proxies bound to it). The **ANY** strategy is intended to be used with stateless, idempotent services. With this strategy, the best available service is reselected with every new invocation.

### 5.2.4 Integration into Eclipse

In a traditional OSGi application, the start order of bundles does not depend on package imports and exports but is affected by the services that bundles exchange. If bundle $B$ consumes a service provided by bundle $A$, $A$ has to be started before $B$ in the general case so that $B$ will find the service when it starts. Some bundles are designed to handle missing services (e.g., leaving out certain features or delaying the startup until the service is present) but this behavior is not mandatory and depends on the degree of dynamism that the developer of the bundle has taken into account.

In a distributed setting, these constraints cross the boundaries of a single VM. The naive approach to start the local bundles, register every local service for remote access and building a proxy for every remote service on each peer (thereby making the deployment a fully connected graph) does not work. If a bundle requires a service provided by a remote machine, the remote service bundle has to be started prior to the start of the local bundle. In general, it has to be assured that not only the total order of bundle installation is preserved among the peers but also that the remote access of services is established in the right order.

The Deployment Tool addresses this challenge by maintaining a global task queue which is processed in sequential order. This resembles the behavior of a local framework where the starting of bundles is also sequential. The total order of bundle installation is not relevant; therefore it is performed in parallel on all frameworks and prior to the application startup phase. For each bundle, the service dependencies are determined and a `BUNDLE_START_TASK` is enqueued in such a way that the dependency structure is properly reflected. If a bundle has any dependencies to services of a remote bundle, the system inserts a `TASK_GROUP` instead that consists of three phases:

1. Importing all remote services that are consumed by the bundle
2. Starting the bundle
3. Exporting all services that are consumed by remote bundles

Through this structure, the resulting startup regime is guaranteed to be identical to that of a centralized OSGi framework. Since the installation is done in parallel, only little
communication overhead is incurred for coordinating the startup order. To minimize the overhead of dependency analysis, the Deployment Editor keeps an up-to-date task queue which is initialized with the tasks required for a centralized application (only tasks to install and start each bundle) and modifies the task queue in response to changes on the structure of the deployment.

When the user triggers the deployment of a application, the bundles are generated from the referenced Eclipse projects. The current prototype implementation of the tool interacts with the Concierge [Rellermeyer2007] Eclipse Plugin to create highly portable OSGi R3 bundles which also run on very resource-constrained devices.

Each framework involved in a deployment is expected to run a Deployment Agent bundle offering a Deployment Service. The services from all frameworks are transparently imported through R-OSGi to the Equinox OSGi framework on which Eclipse and the Deployment Tool run on. As a result, the tool can use the services of the remote frameworks (through R-OSGi service proxies) as if they were local services making the deployment and all further interaction easier. The Deployment Service offers methods to control the framework remotely, e.g., to install and start bundles or to establish a connection to another remote framework in order to import a service required for the application. Since the bundles involved in the deployment remain unmodified, the registration of the services for remote access cannot happen by the bundles themselves (since they were not designed to support distribution through R-OSGi). For this case, R-OSGi allows surrogate registration by a third bundle. In the case of the R-OSGi Deployment Tool, the surrogate registrations are performed by the Distribution Services.

5.2.5 Monitoring the Running Application

As soon as the application is running in the network, the Deployment Tool opens the Monitoring View for this specific application (Figure 5.3). The view shows the actual state of the OSGi framework on each involved node, i.e., the bundles and services. Once again, the Agents on the frameworks are contacted to get this information. Changes in the state of bundles and services are propagated to the Plugin via EventAdmin events which are transparently forwarded by R-OSGi.

Furthermore, the system determines which pairs of frameworks communicate through channels. A channel is established if there are remote accesses to services from one framework to the other. R-OSGi offers an extensible model for channels through NetworkChannelFactories. The tool takes advantage of this and registers its own channel type for the protocol “r-osgi+managed”. These channels provide feedback about activity and exchanged messages. This information is as well propagated to the Plugin through
events and visualized in the Monitoring View. The developer can not only supervise the network activities of the deployment through visual feedback but also inspect the messages in the style of tools like Ethereal/Wireshark [Wireshark1998].

5.3 Distributed Module Updates

*Software evolution* [Lehman1997] describes the phenomenon of change in software systems over time. The term replaced *software maintenance* frequently used in previous literature; thereby emphasizing that the effort put into software systems does not primarily stem from a natural decay of the software over time but in response to a changing environment. Lientz and Swanson [Lientz1980] categorized the causes of software to evolve into four main categories:

- Correcting post-delivery deviation from an intended correct behavior.
- Adapting a delivered software product to reflect a changed environment.
• Improving the performance after delivery.

• Preventing latent faults in software before they become effective faults.

One way in which software evolution manifests itself is in discrete units of *updates*. An update transforms a software system from its current version to a new version by partially or fully replacing code. These changes can either replace existing functionality without changing the API (non-breaking changes) or cause changes visible to external code (breaking changes). When changes are applied to a running system and typically without interrupting the operation of the system, they are called *online updates*.

One distinct advantage of modular system as opposed to monolithic designs is the impact of applying changes through updates. Whereas in a monolithic, tightly intertwined architecture very little can be predicted about the behavior of the system after an update, modularity helps to reason about updates locally since the interaction between modules is formally defined and can be introspected. For instance, OSGi allows online updates of running bundles, even though it is not always guaranteed that running services do not get interrupted. The detailed behavior of OSGi is discussed in the following section.

Online updates of distributed system are highly challenging for a number of reasons. First of all, updates affecting the interfaces of distribution can cause clients to break and in many systems the number of clients is unbound and unknown. Second, applying such an update consistently requires also updating the clients at the same time, which involves communication and requires coordination. Third, multiple hosts can use their own local versioning. A certain version of the communication interface can mean different things to different clients since in practice in many systems the communication interface is copied to every machine and is no longer managed actively by the system. R-OSGi as a runtime for modular distributed systems avoids some of these problems and provides effective ways for applying updates automatically and consistently across multiple machines.

5.3.1 Updates in OSGi

In the OSGi standard great emphasis has been put on providing ways of consistently updating applications running atop an OSGi framework. The framework internally keeps track of which bundles are exporting packages and which bundles are consuming these exported packages through imports. Delegations between exporting and importing classloaders are established when resolving the bundles and tracked as *wires*. Once
wired, a delegation remains immutable. The rationale behind the OSGi update strategy is to keep bundles in a functional and consistent state. Therefore, actions which indirectly affect running bundles by changing shared code are deferred until a framework user explicitly triggers a package refresh. The two actions which potentially lead to such changes are the uninstallation of bundles and bundle updates, as they both remove (old) packages.

The uninstallation of a bundle is only considered safe if no exported package is in use by other bundles at the point in time when the action occurs. This is trivially true for bundles not exporting any package. If the action is not safe the classloader has to be marked for removal but the packages will remain accessible to existing consumers until a package refresh for all or only the affected packages is triggered. All packages not in use are removed from the system by removing the corresponding package information from the internal package database of the framework. Subsequently, bundles arriving after the uninstallation of a bundle no longer be wired to an uninstalled bundle any more. Bundle updates follow the same pattern. Packages which are not exported or are not used by downstream consumers are immediately updated. For packages in use, the classloader hosting the original version of the bundle has to remain accessible for exactly these packages. Newly appearing consumers of packages which were previously not in use will be wired to the classloader serving the new code and hence they will see the new version. Figure 5.4 shows such an update where packages 1 and 4 are in use by other bundles and therefore preserved after the update.

Regardless of the safeness of the update, an UPDATED event is signaled to those bundles which have opted in for being informed about state changes by registering a BUNDLELISTENER with the framework.

Packages can be explicitly refreshed through the PACKAGEADMIN service, which is a service described in the core specifications [OSGiAlliance2009]. Even though this service is optional, most existing OSGi framework implementations provide it by default. A refresh can be triggered by any bundle installed on the framework. Examples of such a bundle are a console or a GUI which facilitate user intervention.

With each refresh, an initial set of bundles that need to be refreshed is passed as an argument to the package admin. This can be either a fixed array of bundle objects, or

---

1 Except when rewired through a package refresh, which, as discussed later, requires explicit intervention.

2 In Release 3 of the specifications at most one version of a package can exist at any time. Release 4 allows multiple versions of packages to coexist in disjoint classloader spaces, i.e., if there is no bundle in the system that directly or indirectly has access to both versions.

3 When running with permissions enabled, only bundles with appropriate permissions can trigger a refresh.
null, which is defined to be all bundles in the system. From this initial set of packages, the package admin service then determines the exported packages which have been marked for removal or which have a deferred version update. For this (smaller or equal) set of packages for which a refresh actually has an effect, the package admin recursively builds the transitive closure of all bundles that are dependent due to an incoming wire; i.e., if a bundle imports a package from a bundle that is subject to a refresh, it gets refreshed as well and thus all its exports are also marked to be refreshed. The result is a set of packages (and thereby a set of exporting bundles) that are affected by the refresh action.

To effectively release all references to old packages, the refresh process itself first stops all affected bundles in reverse start level and installation order. Exceptions occurring during this process of stopping the bundles are signaled through framework events to subscribed listeners but otherwise ignored. After the last bundle has left the active state, stale packages (as results of uninstallations) are removed by disposing of the classloader, and deferred package updates are applied by swapping the classloader reference to the classloader providing the updated version of a bundle. Subsequently, all bundles which have not been uninstalled are restarted corresponding to their start level and original installation order. The result is a system which now entirely operates on the refreshed packages. Any reference to uninstalled packages or previous versions are dropped. However, bundles that previously were active may no longer be resolved because required package dependencies could have been uninstalled.
5.3.2 Distributed Updates

Many of the advantages of modularity with regard to software updates also apply to a distributed module system like R-OSGi. For instance, OSGi has a built-in support for dealing with versioning. R-OSGi extends this to distributed systems. On the one hand, this happens through the type injections since the classes made available through the exports of the proxy module preserve their versions. On the other hand, R-OSGi actively manages dependencies of remote services and is therefore able to track updates.

Still, the implementation of support for distributed updates atop OSGi frameworks is inherently more complex than the support for local updates and the reasons are threefold. First, local updates are handled by the framework which has full knowledge about package versions, wires, etc. The distributed update is handled through the distribution software which itself is just a bundle and has no direct access to the internal state of the framework. Second, local updates happen on the granularity of entire bundles. The distribution software, however, primarily operates on the granularity of services. Therefore, it generates stripped down proxy bundles that only provide the service interface and necessary type injections. As a result, updates to the service bundle cannot be directly applied to the proxy bundles. The distribution software has to handle this case specifically. Furthermore, as discussed in section 5.3.3, the update can potentially be applied by the local framework in two steps, the immediate and the deferred part. The system has to apply both parts at the right time to the client nodes. Third, consistency is harder to achieve when multiple peers are involved. Atomicity is, as we will show, not possible without additional knowledge about the update.

The proxy bundle partly inherits the package import behavior as part of the type injection mechanism. Out of the original package imports, the proxy must import exactly those packages that are required to resolve the classes that are injected into the proxy bundle. Therefore, the package import set of the proxy is a subset of the import set of the original bundle. The proxy exports exactly those packages that were exported by the original bundle and that have at least one member in the form of a class injected into the proxy bundle.

As a consequence, the proxy bundle generated on the client side can have dependencies that have to be resolved in order to have a working service. Different to other distribution software, R-OSGi does not assume that all bundle dependencies of a service are present on each client node. If a package import is not resolvable on a client, R-OSGi creates a clone of the bundle that resolved the package import of the original bundle to mimic the behavior of a single local framework (Figure 5.5).

However, this approach poses the challenge of consistently handling updates occur-
ring on the original service bundle. The OSGi framework itself is not capable of dealing with distribution, hence, it does not provide any support for distributed bundle updates. R-OSGi in turn has the knowledge about where clones of bundles have been created and can actively manage these clones. Furthermore, interaction in R-OSGi is inherently fix-term and restricted to the time period in which client and service provider have connectivity. Once this is not the case any more, no state is persisted on the client so that manageability is always given.

5.3.3 Detecting and Preparing the Update in Practice

Since the distribution system sits on top of the framework as a regular application, it can only detect bundle updates through the events they generate. Unfortunately, the events signaled by the framework do not give sufficient information about the exact update procedure. After a successful update of a bundle, a `BundleEvent` of type `UPDATED` is issued. As a payload, it contains the information on which bundle the update happened but not what the precise effect of the update is. The whole picture has to be constructed by correlating such events with feedback information gathered through interaction with the Package Admin. The package admin provides an interface for retrieving information about exported packages of a bundle and their versions. The prerequisite for detecting an update, however, is that the distribution middleware has sufficient information about the previous state of the bundle to determine the set of packages that have been changed. However, this assumes that updates follow a strict

![Figure 5.5: Update problem for managed proxy dependencies](image)
versioning discipline, i.e., the package version is actually incremented in the meta-data whenever a class of the package has changed. The other possibility would be to track the state of individual classes by calculating hashes over the entire byte-code or certain parts of the class as done for serial version UIDs in the Java Serialization Protocol. Since updates which do not change the package versions can have side effects (e.g., breaking dependent bundles) even in the case of a local OSGi framework, R-OSGi was not implemented to support per-class tracking of updates.

This version might or might not be persisted by the framework in its private storage but there is no reliable and framework implementation-independent way of accessing this data. With the information about changed packages, however, the system can reconstruct an update bundle. The corresponding byte-code can be retrieved from the classloader. For the state of the bundle after the package refresh, this is easier because all packages and classes are then provided by the same classloader. For reconstructing the state right after the update event, there are potentially two classloaders that refer to the byte-code of the classes. One is the old classloader which serves the deferred packages and one is the new classloader which serves the updates version. Therefore, knowing one class that has changed and one class that does not have changed is sufficient to reconstruct the state of the bundle in the deferred case.

If the update affects dependency bundles, this generated update bundle can be directly applied. For the service bundle, the generated update bundle contains more classes than the proxy bundle. However, the type injection property is an invariant that is expected to hold after the update. Hence, the same code analysis algorithm can be used that is already part of R-OSGi for determining the type injection set. Applied to the new state of the bundle after update, the code analysis generates the corresponding update bundle.

5.3.4 Applying the Update Consistently

In order to apply the updates atomically, the distribution software has to take on the role of the package admin on the corresponding client peers. It has to execute the algorithm described in 5.3.1 but introduce a distributed consensus between the stop and start phases that involves a possible rollback. Thereby, the semantics of a local OSGi update is consistently mirrored to a set of client peers.

However, even with this effort, applying the update atomically in a global sense is not possible. The distribution software can only react to events and the update event is generated after the update has already been successfully applied locally. It therefore can only observe an update but not influence (e.g., abort or delay) it any more. There
are two ways of fixing this behavior and turning the entire distributed update into a single atomic transaction. One is to alter and intercept the operation of the framework. If the distribution software can influence the outcome of the initial update, it can achieve a global atomic update semantic. However, non-invasiveness against the standardized framework implementations was a clear design decision for R-OSGi. The other approach is to apply the updates through an external distribution-aware interface and not directly through the local framework. For instance, the Deployment and Monitoring Tool (RDT, Section 5.2) would provide such a mechanism. The same infrastructure can be used to apply updates to bundles to all affected peers atomically because it enables the distribution software to act before the update is applied on the local framework.

5.4 AlfredO

The mobile phone is quickly transforming itself from a mobile telecommunication device into a universal information manager not only able to support communication among people, but also the processing and manipulation of an increasingly diverse set of interactions. The trend of a phone as a point of convergence for the user’s activities, in some respects, has already begun. South Korea Telecom has introduced mobile payment technology and added RFID readers to phones to allow people to get information about shopping products [SouthKoreaTelecom2007]. Nokia has integrated GPS receivers to enable sports activity tracking, car navigation, and multimedia city guides [Nokia2007a, Nokia2006]. Motorola is researching how to allow its nomadic devices to interact with a car’s components: if the car airbags deploy, the phone makes an emergency call; if the driver is maneuvering on a busy road, an incoming phone call is postponed; and if an urgent calendar entry is approaching, it can pop up on the car’s display [Motorola2006]. Applications of this type are usually based on ad-hoc implementations and customized to specific scenarios.

In the past, distributed systems have supported interactions among embedded devices by either statically pre-configuring the execution environment or by dynamically migrating code, data, and service state from one device to another. However, the lack of flexibility of the former approach and the increased security risks of the latter have hampered their actual adoption in mobile environments. To overcome these problems and make a more general approach feasible for resource-constrained mobile phones, we propose AlfredO, a lightweight middleware architecture that enables users to flexibly interact with other electronic devices while providing ease of maintenance and security for their personal devices.
The design of AlfredO stems from two key insights. The first insight is that most interactions with electronic devices such as appliances, touchscreen, vending machines, etc., are usually short-term and ad-hoc. Therefore, the classic approach of pre-installing device drivers for each target device is not practicable. Instead, the AlfredO architecture proposes to adopt a software distribution model based on the concept of software as a service. Each target device represents its capabilities as modular service items that can be accessed on-the-fly by any mobile phone. The second insight is that the evolution of client-server computing from mainframes hooked to dumb user terminals to two-tier architectures (i.e., the classical client-server architecture) and to three-tier architectures (e.g., Web applications) has shown how partitioning server functionality yields better overall performance, flexibility, and adaptability. Therefore, we model each service item as a decomposable multi-tier architecture consisting of a presentation-tier, a logic tier, and a data tier. These tiers can be distributed to the interacting mobile phone thus configuring multi-tier architectures between the mobile phone and the target device.

The approach aims at turning nearly-ubiquitous mobile phones into universal interfaces to the surrounding electronic world. Mobile phones nowadays have sufficient computation power to participate in sophisticated applications. However, they have by design inherent characteristics which distinguish them from typical general-purpose mobile computing devices, such as laptop computers. Phones have a different form factor, different display sizes and screen resolutions, and different input devices. Treating mobile phones like laptop computers overstrains their capabilities and provides unfeasible solutions. On the other hand, considering mobile phones as downsized versions of conventional computers neglects the benefits and unique capabilities they offer, such as built-in cameras and various sensor devices. Our goal is to look at the phone platform in its own right and leverage as much as possible its unique characteristics.

AlfredO incorporates three main mechanisms: (1) a service-based software distribution model for the support of an unbounded number of service interactions between phones and other electronic devices, (2) a multi-tier service architecture to flexibly configure the service interaction, and (3) a device-independent presentation model to achieve device independence and provide interface customizability. In its implementation, AlfredO uses OSGi to manage its modules and R-OSGi for substituting service calls with communication across machine boundaries and for managing the modules on both sides. The challenge of AlfredO is to translate its high-level application model (detachable tiers, declarative user interfaces) with its own meta-data and constraints to the OSGi bundles, services, and dependencies.
5.4.1 Service-based Software Distribution Model

When a phone needs to interact with an electronic device available in the surrounding environment it needs to obtain the required software. A simple approach would be to pre-install the necessary software on the phone and require a third party to authenticate it. Yet, this approach would result in poor flexibility as mobile phones will more likely need to interact with devices casually encountered in the environment. Furthermore, each time the original software is updated, the update needs to be propagated to all phones where the software was previously installed. As the number and type of electronic devices increase, explicitly distributing, installing, updating software on each phone would become an unmanageable task.

Another possible approach is to dynamically transfer the software from the electronic device (or from the Internet) to the phone at the beginning of the transaction. Unfortunately, this approach would expose the phone to several security risks since in the common case the interaction occurs with unknown devices. Furthermore, downloading, installing, and configuring all necessary software is a time-consuming task that very often requires the user involvement and that consumes lots of communication and computational resources.

The AlfredO solution to software distribution is based on the concept of *Software as a Service (SaaS)*, which has been traditionally applied to Internet services. According to this logic, the new business model for most Internet’s commodity software is not selling software, but building services enabled by that software. We believe SaaS can bring interesting benefits also to mobile phones, especially due to the impossibility for such resource-constrained devices to possess all software necessary for every possible interaction.

We adopt a service-based software distribution model where software available on electronic devices is made available to mobile phones in the form of flexible service items. Specifically, we package the functions provided by each electronic device as modular services that can be invoked, decomposed, and distributed using the service-oriented architectural approach of R-OSGi. In R-OSGi, services encapsulate whole functional units and dependencies between services are typically restricted to semantical dependencies at the application level. In the simplest case, a phone acquires on-the-fly the interface of a service of interest, generates a local proxy for it, and discards it once the interaction is completed. In this way, phones are released from the duty of downloading, installing, and maintaining the software necessary to interact with all surrounding devices and the number of possible interactions can therefore grow unbounded. Furthermore, by letting phones acquire interfaces to arbitrary services high flexibility is
provided and a phone’s functionalities are not limited anymore to what their software platform and middleware layers are pre-configured for.

Another advantage that this service-based distribution model brings to mobile phones is its concept of software as a process. Instead of software products that need to be engineered to exactly follow the given specifications, this model allows software to undergo frequent changes thus flexibly integrating a user’s new requirements, technological advances, and emerging data models as soon as they become available. Hence, software on electronic devices can be changed and upgraded without compromising their interactions with the external world.

5.4.2 Detachable Tiers

We envision most interactions between mobile phones (clients), and other electronic devices (target devices) will occur in an ad-hoc manner. A mobile phone may contact a target service directly if its address is known (e.g., the contact address is provided at the bottom of the touchscreen) or upon service discovery. The mechanism used is the R-OSGi service discovery interface which can use arbitrary protocol implementations. Alternatively, the target device itself may periodically broadcast invitations to nearby devices. AlfredO makes the information about new devices available to the user and the user can decide whether to connect to a discovered device. Once the connection is established, the two devices exchange symmetric leases that contain the name of the services that each device offers. Thereby, the user can choose which service to invoke.

As Figure 5.6 shows, in the AlfredO model services are built using a multi-tier software architecture, e.g., consisting of a presentation tier (i.e., the user interface), a logic tier (i.e., computational processes), and a data tier (i.e., data storage). Tiers can be distributed according to different distribution logics and the boundaries of distribution can be adjusted dynamically. Typically, at the beginning of an interaction, the phone and the target device agree on the distribution configuration. This decision may depend on the phone’s capabilities as well as its current execution context. For example, if a phone has low free memory, only the presentation tier is shipped to the phone, whereas if the communication link is unstable also the logic tier is shipped, thus reducing the communication overhead.

In the current implementation, the data tier always resides on the target device, while the presentation tier always resides on the client. By default the service logic tier is not transferred to the mobile phone, but we support also the case in which parts of the service logic are transferred to the mobile phone.

Initially, the target device provides the mobile phone with two elements: the inter-
The default behavior is to generate a local proxy for the service interface and host only the presentation tier on the mobile phone. The client device runs the UI locally and triggers computation on the remote target device by interacting with the local proxy. As all computation and data management occur on the target device, this configuration minimizes the load on the resource-constrained phone. The mobile phone either self-generates a suitable UI based on the abstract description of the UI (see the example with the smart phone in Figure 5.6) or directly receives the UI from the target device (see the example with the communicator in Figure 5.6). We envision this will be the case for most interactions as they are likely to occur in unknown and untrusted environments.
Indeed, a main advantage of this configuration is security. On the server side, the target device has full control on the implementation of its functions thus limiting attacks from malicious clients. On the client side, the device can decide which capabilities to expose to the target device in order to support the interaction. Furthermore, if only a stateless description of the UI is shipped to the mobile phone the configuration provides the security benefits of a sandbox model.

AlfredO also permits configuring more complex two-tier architectures, where the client not only acquires the presentation tier but also parts of the service logic (see the example with the tablet in Figure 5.6). The client can request additional services that appear in the list of service dependencies provided by the descriptor and run them locally. For each requested service, the client receives the associated descriptor (listing the service dependencies of the new service) and its service interface. In trusted environments, this approach can be effective in reducing the communication overhead and improving the application’s responsiveness.

The descriptor provides a declarative description of the system comparable to other declarative approaches like XForms [W3C2007], but it allows for more flexibility. Indeed, our approach is not restricted to typical interfaces with input validation and content submission. Instead, it supports all the interaction patterns of the R-OSGi system, such as asynchronous communication through events, high-volume data exchange through transparent stream proxies, and synchronous service invocations between services. Due to the service-oriented approach that abstracts away from the concrete implementation of the communication, local services and remote services are fully interchangeable and the deployment of an AlfredO application can be altered at runtime. The information contained in the descriptor can be used by the engine to do an impact analysis of different possible assignments of the application tiers to the two devices involved.

The example in Figure 5.7 shows how a mobile phone can configure a customized client application capable of interacting with the remote target device. In a typical interaction some services will run on the mobile phone’s platform (e.g., KeyboardDevice, PointingDevice, etc.) and others on the target device’s platform (e.g., ApplicationService). The client device receives a descriptor of the target service and generates the application’s View and Controller.

Instead of defining layouts that typically break on different screen resolutions and ratios, the UI is specified using abstract controls and relationships. The Render running on the mobile phone decides how to turn this abstract UI into an implementation (the application’s View) that is tailored to the phone’s hardware capabilities.

The AlfredOEngine generates the application’s Controller based on the service requirements specified in the descriptor. The Controller defines how events generated
through the UI (View) can affect the state of the application consisting of application data as well as configuration parameters and proxy settings. For example, at some point of the interaction, in order to improve the application’s responsiveness the client can decide to acquire additional services currently running on remote devices. Likewise, the Controller also defines how events generated by the target device can modify the application’s state. The Controller, for instance, may periodically poll a certain service method provided by the remote device and react to its changes by invoking another service method or by changing the implementation of a control command of the UI.

**5.4.3 Declarative User Interfaces**

In our approach, we consider mobile phones as general-purpose platforms for interactions with various electronic devices and applications but without disregarding the specific characteristics of each device. Electronic devices provide a wide range of different input and output hardware capabilities. In many cases, these are customized to the functions each device is designed for. Clearly, a phone cannot offer every conceivable hardware capability, but capabilities of one device can be mapped to those of another one. For example, the mouse of a desktop computer is equivalent to the joystick of a phone or the knob of a coffee machine.

The service descriptor provides a device-independent specification of the UI. Ideally, an application developer should describe the input and output needs of his applications through this description, devices should provide specifications of their input and output
capabilities, and users should specify their preferences [Myers2000]. The system can then self-implement a suitable interaction technique that takes all these requirements into account.

In AlfredO, the service logic remains agnostic to the specific hardware drivers available on each device. In other words, the logic tier builds on an abstract UI. Input and output capabilities that are used by a specific UI are modeled as OSGi services and accordingly their abstract definition is given by their corresponding service interfaces. All OSGi service interfaces are then organized in a hierarchy. For example, the NotebookKeyboard service implements the KeyboardDevice service interface which is used for entering characters as well as the PointingDevice service interface which is used for moving the mouse pointer through the cursor keys.

Depending on the capabilities offered by the interacting phone, the abstract description of the UI can be rendered differently, i.e, each phone generates the UI in a different manner. A device platform without a mouse or trackpoint can only build a GUI implementing the KeyboardDevice interface and without the PointingDevice service. Or a phone may have the choice to use a trackpoint or an accelerometer to implement the PointingDevice interface. Likewise, on a phone a KeyboardDevice interface may be implemented using the small keyboard of the phone or a handwriting detection that operates with a stylus. In principle, multiple devices can be federated to implement the abstract specifications of the given UI. Furthermore, the UI can be partly on the local phone, partly on the target device, and partly on other external devices. For example, in Figure 5.6, the phone may decide to use a notebook’s screen with larger resolution; in this case, the ScreenDevice service would be implemented remotely by the notebook platform and invoked on the phone through a local proxy.

The implementation of the UI can use different rendering engines that are provided by the client platform. Currently, the default rendering engine produces a Java AWT [Zukowski1997] application where the abstract user interface is rendered with AWT panels. Another rendering engine supported by AlfredO is based on the SWT toolkit [EclipseFoundation2004]. This is especially useful for devices for which an implementation of the Embedded Rich Client Platform (eRCP) [EclipseFoundation2006] exists. As eRCP runs on top of OSGi, it requires only a small set of additional bundles to turn an eRCP device into an AlfredO client. For phone platforms that do not support any graphical toolkit, it is possible to use a web browser that is fed by a servlet [Sun1994] renderer. This produces HTML enriched with AJAX [Garrett2005]. In this case, the web browser can serve as a graphical environment to interact with the headless AlfredO platform.
5.4.4 Use Case

To demonstrate how AlfredO allows a phone to quickly transforming itself in a universal remote controller we have built a MouseController. This is a simple but very powerful service that allows a mobile phone to control the movement of the mouse on a notebook’s screen. Figure 5.8 shows how a browser application running on a notebook can be controlled using the communicator’s cursor keys. In the figure, the user is minimizing the window opened on the notebook’s screen.

The user interface is declared as part of the AlfredO descriptor and rendered in different ways on different phone clients, depending on the capabilities of each particular device. For example, the description of the user interface retrieved by the phone specifies that input commands utilize the PointingDevice service interface. On a Nokia 9300i phone, this interface is implemented with the cursor keys of the keyboard. On an iPhone, the same interface is implemented using the integrated accelerometer, thus allowing the user to move the mouse pointer on the notebook’s screen by moving the phone itself.

Figure 5.8: MouseController running on a Nokia 9300i phone

On the phone’s screen a small snapshot of the notebook’s screen is displayed. Since
the interactions causing the mouse to move are typically occurring at a high update rate, there is often not enough network bandwidth left to send the large updates of the snapshot back to the phone. Therefore, the application uses asynchronous events between the service and the phone and sends updates whenever there is enough bandwidth.

Listing 5.9 shows the part of the AlfredO descriptor that declares the detachable tiers of the application. The first tier (presentation tier) is implicitly given by the declarative user interface (now shown in Listing 5.9).

The next tier encapsulates parts of the application logic. It depends on a Pointing-Device to be available, which abstracts from a concrete input device provided by the platform. Optionally, the tier can also make use of the keyboard as an input device. The tier provides a continuous stream of images that is picked up by the UI to update the image box. Additionally, there are platform constrains declares, which help the engine to decide whether or not the tier should be moved to the client device. The connectsTo declaration tells the system through which service this tier is connected to other tiers.

The third tier is fixed, i.e., it cannot be moved to the client device. The reason in this case is that it is tied to the display of the notebook and moving it to the client would change the semantics of the application. By providing the RemoteCursorService that the first tier connects to, the coupling of the two tiers is fully defined. The coupling between the first two tiers is made through the ImgStream service.

This simple application mainly stresses the communication facilities of R-OSGi since it needs to provide a sufficient feedback to the user in order to allow for a smooth control of the remote mouse cursor. More complex applications taking full advantage of the flexible deployment have been presented in [Rellermeyer2008].
Listing 5.9: Tier declarations in the AlfredO descriptor for the MouseController application
5.5 Related Work

5.5.1 Dynamic Partitioning of Applications

Several approaches have proposed the partitioning of existing applications in order to create distributed applications. The Coign [Hunt1999] project uses instrumentation of COM components through binary rewriting, analyzes the dependencies between the components and calculates a graph-cutting based on a cost model for introducing network communication between the subgraphs. Similarly, JOrchestra [Tilevich2002] partitions a Java program by rewriting byte-code to replace local methods with remote invocations, and object references with proxy references. However, JOrchestra requires in-depth knowledge of the source program in order to do the fine-granular distribution of the mobile classes through a distribution plan. The high number of classes in modern Java applications can make this procedure time-consuming and error-prone. In the Eclipse-R-OSGi Deployment Tool, we exploit existing modularization and use the visualization capabilities of Eclipse to allow the developer to create deployments in a more intuitive way. Furthermore, partitioning of existing applications is an iterative process that has to be revised from time to time in order to adapt to the changing environment or to optimize for performance. Hence, we support monitoring of the deployments and advanced features like load-balancing.

5.5.2 Monitoring and Visualization of Distributed Systems

Visualization of distributed systems for testing and education purposes has been proposed by projects like Parade [Naps1999], PVaniM [Carothers1997], or ConcurrentMentor [Carr2003]. However, these projects describe standalone tools which operate on existing deployments and do not help to actually develop and deploy distributed application. Since the R-OSGi Deployment Tool is based on Eclipse, visualization can be easily applied to any of the intermediate steps during the design phase. The OverView [Desell2004] project has proposed a comparable extension to visualize distributed applications developed in Eclipse. However, they achieve the high level abstractions through explicit modeling in their own Entity Specification Language. In contrast, the R-OSGi Deployment Tool uses existing abstractions of the application based on OSGi bundles and services and does not require any further modeling efforts. Thereby, it provides a more rapid and intuitive way of designing and visualizing distributed applications.
5.5.3 Updates in Distributed Systems

Several authors have discussed the challenges of runtime updates in different domains in a number of research projects, e.g., [Lee1983, Gupta1996, Hicks2005, Stoyle2007, Baumann2005]. The Ginseng system [Neamtiu2006] allows to update software without stopping and restarting it. It generates dynamic patches that can be applied to a running application with the help of the runtime system. We do not require updates to fully running applications but instead leverage the standardized OSGi model to only stop and restart those parts of the application consistently that are actually affected by the update.

The author of [CechPrevitali2007] identifies software evolution and thus updates to software at runtime as an important property of distributed applications in particular. She proposes to consider updates as cross-cutting concerns and therefore express them as aspects which are woven into the application at runtime. We also see handling of updates as a concern of the middleware layer and we agree that proper means and support for software evolution are a necessity for distributed systems. In contrast to the approach presented, the authors of [Ajmani2006] research systems for which it cannot be assumed that updates can happen for all nodes at once. Instead, the authors discuss a system that gradually updates nodes.

Research has also been conducted on using different versions of code at the same time and letting old and new code interact [Sewell2001]. Coexisting versions of code can also arise in our approach when old packages are still in use by other bundles. However, the typical focus of applications written in OSGi is to eventually remove all old uses of old versions of packages and only execute one version of code throughout the system.

5.5.4 Spontaneous Interaction with Devices

Research on distributed systems and ubiquitous computing has variously focused on the problem of how users can dynamically interact with devices embedded in the surrounding environment. Proposed solutions can be roughly grouped into two categories: those that assume an a priori configuration of the interacting devices and those that configure the devices on-the-fly by downloading the necessary software from the Internet or by migrating it from a nearby device.

For example, systems like Personal Server [Want2002] provide the user with a virtual personal computer environment. Data and code necessary to interact with external input/output interfaces are pre-stored and pre-installed on the mobile devices. As these approaches require a pre-configured infrastructure they can suit only static environments.
The second class of systems allows for increased flexibility and can therefore suit dynamic environments. Technologies based on mobile code [Fuggetta1998] have been considered in several domains, but they are usually disregarded because of their security and trust concerns. These security problems are alleviated by systems that rely on a third party (e.g., Internet) for authentication purposes. CoolTown [Kindberg2000] assume a web presence that connects all embedded devices. Each device advertises its presence and offered services through URLs. SDIPP [Ravi2005] augments the Bluetooth service discovery protocol with web access. A user can download the required service interface directly from the nearby device using Bluetooth or from service directories implemented as web services.

Although these approaches can provide some flexibility, AlfredO achieves even higher flexibility by organizing the services into decomposable tiers that can be distributed to configure one-tier or two-tier architectures among the interacting devices. Security is also improved by transferring to the mobile phone only a description of presentation tier, thus allowing the device to self-implement its UI. Furthermore, AlfredO does not rely on Internet connectivity and targets the resource constraints of mobile phones: it is lightweight and highly efficient.

Web services have also been considered in this context. Microservers [Hartwig2002] embed web servers in Bluetooth devices and use WAP over Bluetooth for communication. Specifically tailored to mobile phones, Mobile Web Server [Nokia2008], also known as Raccoon, provides a mobile phone with a global URL and with HTTP access thus enabling a mobile phone to host a universally accessible website. Even though web services are not employed in the current implementation, they could be utilized as well. We opted for R-OSGi because it provides a lightweight implementation optimized to minimize the resource consumption on phones.

We borrow the notion of abstract user interfaces that can be rendered in different ways on different devices from other research projects [Ponnekanti2001, LaPlant2004], especially in the field of human-computer interaction [Nichols2002]. However, these projects mostly focus on how to generate the user interface and typically rely on centralized infrastructures. Instead, our focus is on the system and infrastructure issues.

### 5.6 Summary and Discussion

Three distinct advantages of modularity when applied to distributed systems have been mentioned. All three come to effect in the applications shown.

A tool like the RDT would not be possible without the notion of modules that can
be deployed to different machines and without the effect that the design is separated from the deployment. RDT makes the compositional approach of deployment modularity visible and understandable. At the same time, the monitoring facilities visualize the communication happening behind the scenes to implement the interaction between remote services. This helps to understand the concrete impact of substituting local service invocation with remote service invocation in different deployments. The separation of application logic from communication logic through the abstraction of services is the fundamental principle that enables a flexible deployment. At the same time, the dynamism arising from changes in the environment is handled by allowing for redundant bindings between services and policies to use the redundant links, e.g., for failover. Again, the prerequisite for doing so is that the application logic encapsulated in the modules is not tainted by code explicitly handling the communication.

The solution to the update problem takes advantage of the fact that modules provide natural boundaries within an application and the impact of changes remain local, unless they cross one of the declared dependencies. Therefore, parts not affected by updates can remain untouched and run without interruption. The contribution of R-OSGi is to preserve the invariants of modularity across the different machines involved, which finally makes distributed updates feasible and tractable.

Finally, AlfredO has built a higher form of abstraction and granularity atop the basic modularity. In practice, the tiers can be implemented as single modules but in most cases are more likely constitute compounds of modules in order to not violate the cohesions of the modules. The flexibility of deployment that a distributed module system provides is deliberately restricted to a subset of deployments that are designed to support the application well. This restriction is expressed through the high-level dependencies and the constraints in the AlfredO descriptor. At the same time, the descriptor adds application-level information semantics otherwise not available to the runtime system.

The RDT and AlfredO show two different paths towards implementing fluid systems atop R-OSGi. Whereas the focus of RDT is more on static and controlled re-deployment through a system administrator, AlfredO exploits additional knowledge about the modular decomposition in the form of the descriptors to autonomously change the deployment of modules at runtime. However, AlfredO understands mainly a single axis as degree of freedom: the axis between the mobile client device and the stationary server system. Furthermore, none of the two approaches actively deals with the state of modules. It is assumed that moving a module from one host to another does not affect the running system. The following chapter presents a system which does not make any of these assumptions. Instead, all possible degrees of freedom between a mesh of modules are potentially exploited and state is actively managed by the runtime system.
5.6. Summary and Discussion
Modern hardware platforms, either as multi-core computers or as in cloud computing services, offer many opportunities for horizontal (scale-out in the cloud) and vertical scalability (scale-up, in multi-core). There are, however, also plenty of challenges attached to it. One example is the programming of scalable and reliable applications for cloud infrastructures. As a matter of fact, cloud fabrics under the hood are distributed systems. The large scale of typical deployments hence exacerbate the traditional challenges of distributed computing such as achieving robustness in the presence of node and network congestion or failures [Waldo1994]. The pool of available resources and the implicit topology created by the individual nodes can, however, also change at any time as a result of explicit resource management, e.g., the addition or removal of nodes by the operator of the application. Applications for the cloud are in turn expected to seamlessly adapt to the elasticity of the cloud fabric. In practice, this means that they have to continuously adapt to the changing set of resources by migrating software components and workloads to the available machines, which is often beyond what programming models for traditional distributed systems are able to achieve.

Alternative programming models have hence been proposed to raise the level of abstraction and thereby making the application logic less tied to specific topologies and deployments and more scalable and elastic. For instance, model-driven engineering has been discussed (e.g., [Bruneliere2010]) as an alternative. Platforms such as MapReduce [Dean2004], Hadoop [ApacheHadoop], or Dryad [Isard2007] provide high-level programming models for a specific class of applications like batch processing. An example of a novel general-purpose programming model for the cloud is Orleans [Bykov2010]. It runs atop the CLR and is based on the principle of software componen-
tization (grains). Communication is hidden behind an actor-like concurrency model. It requires, however, all software to be written against this new programming model and language extensions to C#.

In contrast, the approach presented in this chapter builds upon OSGi as an existing and well-established application model, and presents a novel runtime platform, Cirrostratus, to seamlessly turn OSGi applications into cloud applications. As shown previously in Chapter 4, loosely-coupled interaction between modules can be turned into remote communication on demand, which is a prerequisite for the location-transparency required to implement elasticity. For instance, R-OSGi already supports redundant bindings to multiple services and switching from one to the other, as illustrated by the RDT (Section 5.2). In order to be successfully applied to the cloud, however, additional challenges need to be solved in order to simplify the development and deployment of cloud applications:

**Location Transparency** In cloud computing fabrics the notion of a physical location and of a network address are weaker than in traditional distributed systems. At the same time, changes to the topology happen more frequently. We show that it is possible to virtualize the application modules and the bindings between them, provide full location transparency, and to make the runtime system appear to the application as a single system image. While this significantly simplifies the development of new applications by removing the burden from the programmer to deal with the dynamism in the application code, it also makes it easy to deploy and run modular legacy applications to cloud platforms.

**Orthogonal Replication** Cloud computing operates at a large scale with multiple tenants sharing machine and network resources. It is therefore particularly important to make applications resilient to changes of resource availability or even outages. The typical solution to address this challenge is over-provisioning and redundancy. We provide a mechanism to add stateful replication to OSGi services without requiring the developer to change the code. By using an abstract interpretation technique, the runtime is able to infer the locations of state changes from the code and weave the bytecode in such a way that it runtime state changes are replicated across multiple replicas and according to a customizable replication policy.

**Fine-granular Monitoring and Management** Large cloud application often serve or support critical business activities. It is therefore particularly important to effectively monitor and measure the vitality and the performance of the system. The virtualization in cloud fabrics prohibits performance monitoring on the operating system or hardware level. We provide a fine-granular alternative on the application level by providing the operator of a cloud application with the mechanisms for tracking performance and live-
liness on the granularity of single modules and services. The collected information can be used to change the physical deployment of the application or changing the resource assignment, e.g., by migrating services to other machines or adding additional replicas of services and controlling the load balancing on a per-service level.

As a runtime system for elastic applications, we have developed Cirrostratus, a platform that provides full location transparency by providing a global registry based on a DHT in which all modules are made available to all machines in the resource pool. This gives Cirrostratus the ability to dynamically deploy modules and reconfigure the application at run time. Cirrostratus sits on top of a set of OSGi runtime systems and itself implements the interface of a single OSGi runtime spanning the complete pool of machines. Since it provides a single system image to the modules, unmodified OSGi applications can run atop Cirrostratus and be made fluid and elastic in a transparent manner.
6.1 The Cirrostratus Cloud Runtime

The Cirrostratus runtime operates atop multiple hosts running Java virtual machines with arbitrary OSGi framework implementations (implementing at least revision 4.0 of the specifications). It can expand the resource pool by adding new machines and contract it by releasing machines, as cloud platforms like Amazon EC2 support it. Figure 6.1 shows an overview of the system architecture. The Cirrostratus software stacks on each node communicate through an overlay network spanned by all the hosts currently participating in the Cirrostratus deployment. Even though Cirrostratus is a distributed system, it exposes the interface of a single, coherent OSGi framework to the application, thereby providing a single system image. Cirrostratus itself does not implement the entire functionality of an OSGi framework but leverages the respective local host-frameworks where possible. It virtualizes the host-frameworks by intercepting those calls to the runtime system which require a global view of the entire (distributed) system or resulting in actions that affect the global state of the system. For instance, it provides a global distributed service registry that makes the services of all nodes available. Those parts of the calls are handled within Cirrostratus to achieve a consistent view among the nodes. The remaining functionality like resolving the bundles and persistently storing them is provided by the respective local host-framework.

An OSGi application constitutes itself from a set of modules that would normally be deployed to a single OSGi framework running on a single machine. On Cirrostratus, bundles installed on one machine are as well accessible to all other machines belonging
to the same Cirrostratus instance. The same is true for services. By structuring an application into modules, the runtime gains the freedom to distribute the modules based on the current shape of the underlying fabric and the current load on the system broken down to the load on individual modules. What the application sees is a virtual deployment of modules (upper layer of Figure 6.2). Every module appears exactly once, every distinct service instance registered by a module appears exactly once. By adapting the modules, the runtime can replicate modules and rebind services on demand. It has the full control over the physical deployment (lower layer of Figure 6.2) in which modules can appear arbitrarily often and services can either be accessed remotely or through a local replica.

In the example in the figure, the bottleneck of the application is the functionality provided by a service of Module C, called by Module A. The data exchanged in the calls is relatively small while resulting in a computation-intensive execution. The interaction between Module A and Module B happens less frequently but more data is exchanged for each call. Such a scenario would likely result in the physical deployment shown in the figure where multiple physical replicas of Module C exist whereas Module A and Module B are co-located on the same machine.

The only constraint for physical deployments is that each element (module, service) of the virtual deployment has to be provided at least once. In practice, there are also other good reasons for holding more than one instance, e.g., for fault-tolerance. Since the application remains agnostic against the physical deployment, the runtime can at any time redeploy modules and services to optimize the overall resource utilization or to react to changes in the work load. Full location transparency is achieved from an application’s point of view because the services are delocalized, meaning that they are observable by a client but their precise physical location is not of significance and may be subject to changes. Choosing the best physical deployment for a given set of machines and a virtual deployment is the task of a controller, which can take application-specific requirements into account. The following sections discuss what is involved in order to map virtual to physical deployments and deal with the state replication, the prerequisite for location transparency.

6.2 Virtualizing an OSGi Framework

The only interface between application bundles and the OSGi runtime is the Bundle Context. The context is given to the bundle through the Bundle Activator which is the entry point of an active bundle requiring interaction with the framework and called
when the bundle is started or stopped. The bundle context is, e.g., used to control the life
cycle of other bundles, register listeners for change events, or providing and consuming
services. In fact, all the distinct functionality of an OSGi bundle is rooted in the use
of the bundle context, the remaining code is standard Java. In order to run a bundle on
the Cirrostratus runtime rather than on the host framework the system passes a virtual
bundle context to the bundle. This forged bundle context provides the system with an
indirection to intercept calls to the runtime system and either handle the operation on its
own or at least partly delegating the operation to the local host-framework.

6.2.1 Local and Global Scope

Bundles in OSGi are singleton entities. A distinct bundle version can only be installed
once on an OSGi runtime. Multiple versions of a bundle can co-exist since revision 4 of
the OSGi specifications but only in disjoint classloader spaces. No bundle in the system
can import more than one version of an exported package at any time. This is a bare
necessity in Java since instances of the same class provided by multiple bundles and
hence loaded through different classloaders are runtime-incompatible.

The goal of Cirrostratus is to provide a platform for running applications in large
deployments like the cloud. Many of such applications require multiple clients to access
the application. The clients can be viewed as multiple instances of the same module (or
sets of thereof), and typically of the same version. Since clients and backend services,
however, are part the same Cirrostratus instance, the existence of multiple clients would
violate the singleton principle.

As a remedy, Cirrostratus introduces the concept of scopes which form a visibility
boundary inside an OSGi deployment. By default, bundles are installed and operate
in the global scope which results in full visibility of exported packages and registered
services to all other nodes in the system. In addition, each node provides its own local
scope. Bundles installed in the local scope can see all resources of the global scope
and the local scope but their own resources remain hidden to other nodes. In the same
spirit, the bundle can bind to all services in the global and local scope but its own
services are only accessible to other bundles installed in the local scope of this node. By
construction, the uniqueness criteria for modules remains valid within each scope but
still all instances of bundles in the local scope can access the same bundles in the global
scope.

Figure 6.3 illustrates the effect of the different scopes. A shell bundle 1 installed in
the local scope of node 1 would return B1, B2, and B3 as a result of a bundles command
since it sees its own local B1 and the bundles installed in the global scope. Different
instances of B1 can coexist in the local scopes. The shell bundle \( \triangleright \) would return the same list of bundles but pointing to its local bundle B1. The bundle B1 can bind to services provided by B2 and B3 but since it is a local bundle, its own registered services are only visible to bundles in the same local scope. Finally, the shell bundle \( \bowtie \) only returns B2 and B3 since the global scope is agnostic to any local scope.

6.2.2 Bundle Start and Meta-Resolving

The first step to start a bundle on Cirrostratus is to get it resolved on the host framework so that the activator class can be loaded. In order to do so, Cirrostratus has to resolve all dependencies of the bundle. This potentially requires resources physically located on other nodes so that prior to the resolving process on the local host framework, the runtime has to determine and locate the required resources on other nodes and provision them to the local node. This process is the meta-resolving.

Each bundle contains a list of imported packages required to make the bundle resolvable. The runtime maintains a distributed bundle registry which indexes the bundles according to their exported packages. The meta-resolving process can hence easily find out the locations where potential candidates for package imports are installed. Choosing the right virtual bundle follows the same resolving rule that a conventional OSGi framework applies. In general, a framework attempts to use the highest version permitted within the version constraints attached to the import. The absence of a version constraint corresponds to the interval \([0.0.0, \infty)\). At the same time, however, the framework has to assert that no bundle in the system is wired either directly or indirectly through its transitive dependencies (classloader space) to more than one version of a package. In essence, a bundle can only be part of one classloader space at the same time. For resolving, this means that the choice of a certain package provided by a specific bundle would attach the bundle that is resolved to the classloader space of the resolver and might preclude the choice of other candidates for further dependencies. This is exactly
the case when there is at least one package exported in both classloader spaces.

Conventional OSGi frameworks can afford to check the integrity of classloader spaces and the validity of candidates on the fly during the resolution process by traversing the existing wires of the bundles. In Cirrostratus, in contrast, this information is distributed across the entire deployment. In order to accelerate the meta-resolving process, Cirrostratus keeps the information about exported packages on the granularity of entire classloader spaces so that the integrity checks can be distributed to the nodes holding the meta-data and the resolving process can be parallelized.

The result of the meta-resolving process is a set of virtual bundles which resolve all the package dependencies of the new bundle. Picking a specific physical bundle instance corresponding to a virtual bundle is a policy decision and permits the runtime to optimize, e.g., for network bandwidth consumption, latency, or node utilization since the bundle bytes have to be requested from the other node and transmitted through the network in order to install a local copy on the host framework. In the end of this process, the new bundle is installed, becomes resolved, and the activator class can be loaded. Cirrostratus then calls the start method of the activator but passing its own virtualized bundle context to the bundle. In the virtual context it can intercept all calls to the framework where required and, e.g., after doing operations involving the virtualization layer to delegate the call to its local host-framework. Since Cirrostratus is itself an ordinary bundle running on each local host framework, it holds a bundle context of the local host which it can use on behalf of the virtualized bundle. As a consequence of the virtualization, the application bundles never actually becomes active (the state where it is started) on the host framework but instead virtually active on Cirrostratus. The Cirrostratus runtime and not the local host-framework is in charge of handling the life cycle of the application bundles since this amounts to state with global effects.

### 6.2.3 Achieving Location Transparency

The application bundle has access to all other bundles (in the global scope) as visible in the virtual deployment. Through its bundle context, it can request a bundle object for each of them which facilitates, e.g., manipulating the life cycle state of the bundle or updating it with a new version, even for bundles which happen to be physically located on a different node. In order to support this global view, Cirrostratus maintains a shared bundle and package registry. The registry is based on a distributed hash table (DHT) implementing the Chord [Stoica2001] algorithm with functional extensions.

The base line for the behavior of Cirrostratus is a conventional OSGi framework running on a single machine. The difference in Cirrostratus is that the provider and the
consumer of a bundle are now potentially located on two different nodes. For reasons of consistency, it has to be avoided that a bundle becomes unavailable due to any third, not directly involved node becoming unavailable. Therefore, Cirrostratus does not store the bundles or package information in the DHT but uses it as an inverted index pointing to the locations where the actual objects reside. In order to turn a Chord-like DHT into an inverted index, the map functionality of the hash map has to be extended to multimap semantics, which means that the DHT in Cirrostratus is able to store multiple values per key. In the case of bundles, the DHT maps virtual bundles to physical bundles. For packages, the DHT contains the meta-data about the classloader space, which consists a set of virtual bundles. Each classloader space is indexed by all packages that it exports through its member bundles.

Bundles are indexed in the DHT for lookup by their bundle ID, which is a globally unique numeric identifier, and their symbolic name + version. These are the two properties of bundles for which applications can do a lookup through the OSGi API. For each virtual bundle, there can be multiple physical bundles. The DHT implementation is enhanced to provide this indirection and also support different policies for mapping virtual to physical bundles. Since policies typically require knowledge about the application, Cirrostratus exposes interfaces to the application to implement their own policies. The default policy effective when the application does not provide its own is to pick a random physical bundle.

Services are the main (implicit) communication channels among bundles in Cirrostratus and on-demand binding and re-binding services in a location-transparent manner is the *conditio sine qua non* for providing elasticity. Again, the DHT acts as an inverted index mapping the virtual services to the physical locations. Each virtual service is indexed by its global service ID and all service interface names that the service is registered under. A particularity of OSGi services is that they can have properties attached and these properties. Requesting client bundles can express RFC 1960 LDAP filters [Howes1996] against them to find matching services only. The flexibility of the DHT implementation in Cirrostratus handles these queries exactly like policies and evaluates them on the node containing the meta-data for a service to push the selectivity of the filter into the network.

### 6.2.4 Providing and Consuming Services

Registering a service involves creating an index entry for the service in the DHT. The service properties have to be stored on the responsible DHT as well so that the filters can be evaluated on the DHT nodes. Properties can be altered by the service after
registration. As a consequence, not only the deregistration of a service but also property updates have to be propagated and the index updated.

Consuming a service on a node in the first hand triggers the import through an R-OSGi proxy if the service does not happen to be local. As in R-OSGi, the service proxy is provided by a proxy bundle which also provides the type injections, which are all classes of the original bundle that are reachable from the service. This allows for global consistency in the versioning and type system because it ensures that a client and a service use the same versions of the same types within a communication relationship.

Subsequent requests by clients for this service will then be served through the proxy service and all calls to the service redirected to the remote service through remote calls. Using R-OSGi for remote service access is the base case since it does not affect the physical deployment. The proxy bundle—in abstract terms—only provides a type- and version-safe communication channel to a remote service. In this case, the cost of using the remote service is paid for every service invocation due to the network latency. Arguments have to be serialized and transmitted through the network. After either successful or failed invocation, the result (either the return value or the exception) have to be again serialized and sent back to the caller. Under certain conditions, this cost can be higher than having a local replica. The cost of using a replica stems from the communication required to synchronize changes to the shared state. If this shared state is small enough or changes happen infrequently (not with every invocation of the service), the replica potentially performs better than the remote service.

In order to give the system the freedom to seamlessly switch from one mode to the other, the generated proxies in Cirrostratus are more dynamic than in R-OSGi. They can be on demand replaced by replicas without interrupting the consumer. Providing a replica of a service changes the physical deployment because it leads to a duplication of the bundle providing the service. Furthermore, other bundles can establish a remote connection to a replica in the same way as they do to the original service. All replicas in Cirrostratus are read and writable, all are equal and consequently all appear as first-class citizens in the service registry. Choosing a specific physical service for a service request is once again a policy decision and can be influenced by the application.

6.3 Managing the State of Modules

In abstract terms, a service consists of methods exposed through a service interface and each of these methods takes a set of input parameter, does some local computation that possibly involves reading from and writing to local memory, and finally returns a
result. The module hence describes the behavior of a functional unit through its services. However, the code is written for the case that the service runs on a single machine. The approach of understanding modules as a design principle for scalable systems is to take these modules, infer the behavior of the module through symbolic execution, and adapt the behavior for the shared runtime.

A key concept for the adaptation is the notion of state. In an object-oriented language like Java, state is created by the set of objects on the heap and their fields, which can contain either primitive values or references to other objects. For an OSGi service, the situation is even more structured. The starting point is always a single object, the service object. Hence, the state of an OSGi service can be described as the value of the fields of the service objects and the value of all the fields of the directly or indirectly referenced objects.

OSGi provides the tightly coupled way of sharing code through dependencies and the loosely coupled through services. The model of Cirrostratus is that only services are containing state. If code is shared through an import, it is assumed that no state is leaked out of the module through this dependency.

### 6.3.1 Symbolic Execution

Symbolic Execution is a technique traditionally used to, e.g., verify the correctness of a program [King1976] or to generate test cases [Khurshid2003]. Instead of running the program on a distinct set of actual input arguments, the code is executed with symbols as inputs. As a result of symbolic execution, it can be inferred if and how a certain instruction depends on the inputs or other symbolic values.

Unlike in previous applications, we are using symbolic execution to infer application semantics that help to identify places in the code that have to be adapted in order to replicate a module. These places include read and write field accesses for fields that amount to the state as defined before, and method invocations. In Java, like in most object-oriented languages, a field can be aliased, e.g., through a local variable or as argument of the current method. The strength of symbolic execution is that for each instruction, the potential data dependencies can be determined and aliases can be resolved.

The symbolic execution starts with instantiating a symbolic call stack for the first entry point to the module. In Cirrostratus, this means it starts with the first method of the first object registered as a service. The call stack consists of a symbolic this reference and a typed argument symbol for each formal parameter followed by a set of empty symbols for the local variables. Each instruction of the method is then interpreted and the effects are reflected on the symbolic call and execution stacks.
Three different kinds of symbols can occur on the stack as results of an instruction. Argument symbols indicate that the value has been passed to the current method as an argument. Field symbols indicate that the value is the field of an object. Method call symbols are placeholders for the result of a nested method call. In the absence of conditional control flow, all other values on the stack can be considered to be constant from the perspective of the service. Conditional control flow potentially render the meaning of a value ambiguous because there can be more than one assignment to the value. Like in static single assignment form [Cytron1991], the symbolic execution places $\phi$-Symbols on the stack to indicate this. If a $\phi$-Symbol occurs later in the instrumentation and the cases covered are semantically different (e.g., a constant and a field symbol), a runtime check has to be inserted into the code.

The symbolic execution takes, in contrast to the regular execution, $O(n)$ time where \( n \) is the number of instructions interpreted. The reason is that loops and other control flow, which lead to jump instructions on the byte code level, do not have to be followed. In practice, the number \( n \) is smaller than the number of instructions contained in a module because only the instructions reachable through services will be interpreted. The result of a symbolic execution is a method descriptor that indicated the locations where state changes have been detected. When a nested method call is detected that has not been symbolically executed before, the interpreter recurses into this method. Otherwise, there is a cached symbolic method descriptor that can be linked to the method call symbol generated.

Java does not have pointers in the language but uses virtual method calls by default. This means that the method to be called is determined by the runtime type of the object (dynamic dispatch) and not statically at compile time. For non-virtual calls, the access patterns of the method referred to through a nested call can be broken down to field reads and writes and folded into the descriptor of the caller. For virtual calls, however, it cannot be statically determined for the general case which method will be called at runtime. This only works if the the callee is constructed within the module and the reference is immutable. Otherwise, the analysis can capture the behavior within the scope of the calling method and then locally in the scope of all potential candidates for callee. The full picture has to be constructed at runtime by a facility that tracks the entry into the nested methods and signals the effective reads and writes.

Candidate methods within the same module can be instrumented to encompass this facility. Methods belonging to objects external to the module cannot be instrumented because the code is not under control of the runtime. They can belong to a module that is already loaded, or to the JVM class path. However, external method calls can have side effects, in which case the execution has to be replayed on all nodes to achieve
consistency. The symbolic execution will therefore be performed on external method calls to exclude those methods for which it can be inferred that they do not have side effects. If there are write accesses to fields in the call tree of an external method, then a side effect is apparent. But there are two cases in which no information about side effects can be inferred at all. First, calls against interfaces or abstract methods (purely virtual calls), and second calls into native (JNI) code. The symbolic execution conservatively has to assume side effects in these cases and mark the calls for later instrumentation.

Other important analytics involve the location of thread creation synchronization and the detection of interaction patterns between threads and shared state. For instance, worker threads can be detected by their blocking behavior on shared variables like sockets or queues. By doing so, the runtime can later adapt the inherent parallelism of a module to, e.g., scale vertically (adding more threads) or horizontally (adding more machines).

### 6.3.2 Adapting the Code

Once the symbolic execution of a module has completed, all locations that affect the state of a module are known. In a subsequent step, the runtime applies an instrumentation strategy to adapt the code in such a way that the state can be kept consistent among replicas of a service. Indeed, every possible way of replicating the state can be used with Cirrostratus provided that the required adaptation can be expressed as transformations of the analyzed code. In particular, externalizing the state into a database is a valid solution. Implementations of state replication technology typically require highly customized code adaptation strategies and the right state replication technology is again application-dependent. To illustrate the process of code adaptation, we focus for the remainder of this section on the replication strategy that we use by default in Cirrostratus. This strategy keeps the fields of the objects intact and uses them as caches for the global state which is kept consistent through an entry consistency protocol [Adve1996] that operates asynchronously.

**Service Entry Points**

In order to track the state changes in the code, the service has to be endowed to interact with the corresponding state replication facility in Cirrostratus. The default strategy turns service method calls into transactions. Each service is equipped with its own replica manager since by inserting a private field into the service class into which the authoritative replica manager instance is injected at runtime. Furthermore, each service method is prefixed with a `BOT` call that signals the replica manager the beginning of a
new transaction. This internally causes the replica manager to allocate a new transaction context, which is responsible for tracking state changes. In order to allow for concurrent transactions to happen simultaneously, the replica manager stores the transaction contexts as thread locals so that they do not interfere. Every regular exit point of the service method is preceded by a EOT call which causes the replica manager to propagate the state changes to other replicas and free the resources associated with the transaction.

Field accesses

According to the definition of state in Cirrostratus, every object which is part of the state of a service is directly or indirectly referenced by the service object. Hence, Cirrostratus can assign hierarchical field identifier to each field and denote the path to the field starting with the service. For instance, if the service has a field name of a certain type NameInfo and the state changed is the firstname field of this NameInfo object, then the identifier for the operation is this.name.firstname. Through this identifier, other replicas can find the path to the modified object by iteratively applying a reflective getField operation for each level of the path.

Each field access detected through the symbolic analysis causes a code adaptation. Since the default strategy uses entry-consistency, all reads of fields are cumulatively transmitted to the replica manager as part of the BOT call so that the replica manager can construct the barrier. Write operations are signaled where they happen in the code so that the execution stack can be used to easily determine the value written to the field. For instance, the code sequence that writes the value 100 to a field

```
ALOAD 0
BIPUSH 100
PUTFIELD MyObj.field : I
```

is adapted as follows:

```
ALOAD 0
BIPUSH 100
  DUP
  ISTORE 2
PUTFIELD MyObj.field : I
  ALOAD 1
  LDC "this.field"
  ILOAD 2
  INVOKEVIRTUAL ch/../TransactionContext \
       .addFieldUpdate : (Ljava/lang/String;I)V
```
The code adaptation introduces a temporary local variable to store the written value. Among multiple writes, the same local variable slot can be reused so that the number of local variables increases only by one, or two if `double` or `floats` are involved.

**Method calls**

Calls to methods that are part of the same module and are also adapted for state replication involves setting the context in which the nested method operates. This means that when the call takes place, the replica manager has to know which arguments within the scope of the callee relate to fields. Even though Java exclusively uses call by value semantics, it can pass reference types by value so that nested methods can perform state changes if the referenced objects are mutable. For instance, effect of the following program fragment

```java
private Sum sum;

public void foo() {
    bar(sum);
}

private void bar(Sum s) {
    s.add(10);
}
```

with regard to state changes of the sum field can only be detected correctly if the replica manager understands the binding between the field `sum` and the argument `s` that happens during dispatching to the method `bar`. Hence, the code has to be adapted to shared this information with the replication facility. Prior to a call, a new context is created that contains all bindings to fields if such exist for the particular method call. All state changes within the called method can then be related to the context and the replica manager can decide if in this particular context a certain call changes the global state or only a local value. After the call, the context is popped off the context stack again. This mirrors the behavior of the JVM when creating a new frame prior to a method call and disposing off the frame afterwards.

**External method calls**

Methods external to the module cannot be adapted and calls to external methods containing side effects thus have to be replayed on all replicas. In order to do so, the replica
manager has to capture the operand stack for a method call. This is exactly the part of the execution stack that is consumed by the method call. The precise format depends on the formal parameter of the method. Whereas all reference values and most primitive typed values take one word on the stack, the primitive types long and double consume two words. Since the type of the formal parameter can be easily inferred by the symbolic analysis, it can generate a custom stack unwinding sequence that stores the arguments in an object array (and duplicates it) so that they can be passed to the replica manager. Consequently, the object array can be winded to the stack again, restoring the stack to the same state as before and allowing the method call to be performed. Primitive values have to be boxed into reference types during this process and unboxed afterwards since autoboxing is a feature of the Javac compiler but not part of the instruction set of the VM.

The advantage of the described procedure is that it can be implemented very efficiently and in-place. In concrete terms, the code sequence

```
DUP_X1
SWAP
BIPUSH index
SWAP
AASTORE
```

(ignoring primitive types) has the following properties: i) it only touches the topmost element of the arguments on the stack in each iteration, meaning that each element is touched only once, which is optimal. ii) when started with an empty object array as top of the stack (the future arguments underneath), and applied repetitively for each argument, it preserves the invariant that after each iteration, the stack consists of the object array reference followed by the remaining arguments. iii) the code sequence increases the maximum size of the stack by at most two (three when considering possible boxing). The stack winding code sequence is

```
DUP
BIPUSH index
AALOAD
SWAP
```

and has the reciprocal properties. The only minor disadvantage of the approach is that neither of the two code sequences has a correspondence in the Java language and thus cause decompilers (e.g., sometimes used in runtime debugging tools) to stumble across them. Most of them, however, simply print the code sequence in byte code instructions
and gracefully proceed with the rest of the code. Code verifiers, in turn, are not affected since they operate on the byte code level and the sequence is legal byte code.

An important distinction has to be made between state and values. Either can appear as arguments of method calls. A value would be an object that is not referenced by any field and usually has a life time limited to the current execution context. Consequently, it has to be passed by value, meaning that it is serialized and shipped to all nodes replaying the method invocation. State, in turn, has to be passed by reference to preserve the global identity. For instance, if the content of a field (hence state) is passed to the add method of a LinkedList, it is expected that the object returned by a subsequent remove call still passes the == equality check against the field, even if the remove call happens on a different node. If the argument was passed by value, each node replaying the add method call would deserialize a new copy of the field value which would not provide reference-equality any more. If by reference semantics are used, the each node passes its own content of the field to the add method and regardless of which node does the remove, it compares against its own field again.

### Locking and Thread Coordination

Special attention has to be dedicated to the concurrency mechanisms of the JVM instruction set. The code is expected to express the behavior of a program running on a single machine. It is likely to rely on the globality of monitors and thread synchronization primitives. When the same code is now running in a distributed setting, both facilities unmodified would now only act locally, which is likely to break the code. Hence, the code has to be adapted to use Cirrostratus’ distributed locking and thread coordination system.

In the Java language, critical sections are written as synchronized blocks. Methods declared synchronized are equivalent to a large synchronized block containing the entire method body and synchronizing on the containing instance. On the level of byte code, this translates to a much more irregular sequence of a MONITORENTRY instruction accompanied by at least two MONITOREXIT instructions, one for the regular exit and one for the case that catches any exception thrown by the embraced code to avoid the thread escaping the critical section without releasing the monitor. If the code within the synchronized block contains return statements, each of them is preceded by a MONITOREXIT as well. In order to achieve a consistent behavior, the monitor enter and each monitor exit instruction has to be adapted to call the distributed locking facility. The same has to happen with calls to Object.wait (with and without timeout) and Object.notify or Object.notifyAll. The distributed locking system has
6.4 Managing elastic modules

Location transparency and state replication facilitate the re-deployment of modules and services on demand, without requiring the programmer to change the modules. The Cirrostratus runtime provides not only the platform to re-deploy but also the information required for deciding when to do so. The final decision, however, is not the responsibility of Cirrostratus but of an external Controller. The reasons for this design decision are apparent. Binding services to a particular instance, migrating functionality between machines, or favoring replicas over remote calls are all policy decisions that cannot be made without knowing the application and its requirements. Cirrostratus can infer semantics of the code but not of the application. Most importantly, policy decisions do typically not only involve technical aspects but also, e.g., economical aspects or other non-functional requirements such as service level agreements.

On the actor side, Cirrostratus provides a management interface for actions like replicating or migrating a service from one machine to another. If no threads are involved, a migration is simply a temporal replication that after successful completion drops the original instance of the service. Threads require special treatment and more adaptation to provide checkpoints at which the stack can be captured. Other actions involve the explicit rebinding of a client to another service instance or touch points to the underlying platform to start new machines or to shutdown machines no longer required. With these actions, the deployment becomes completely fluid. The controller can gradually transform every deployment into every possible form that fulfills the contract implicitly given by the requested virtual deployment.

On the sensor side, the throughput and the latency of the communication channels that underly the remote services are continuously monitored. The same happens for the traffic generated by the state replication. When requested, Cirrostratus can even monitor the potential traffic generated through either a remote service or a replica. It then generates the message that would be used in the proxy or in the state replication protocol and averages the message size over time. It can also report the size of a module or the traffic generated by a service replication process. Together with the knowledge about the current network link conditions, a controller can thus decide whether or not
the current deployment is optimal and how to improve it.

A valid replacement of a controller is a human being that reads the performance monitors and takes decisions. Depending on the application and the requirements, an autonomic controller can provide the same efficiency. The design and implementation of particular autonomic controllers or even their fitness for particular applications is beyond the scope of this work. For smaller applications and simple requirements that can be expressed by single values, however, we have successfully used the constraint logic engine of Rhizoma\cite{Yin2009} to control the deployment.

6.5 Evaluation of the Scalability

In order to run applications on an elastic set of nodes, the runtime itself needs to scale to not adversely affect the application. In Cirrostratus, the majority of operations run locally. The potential scalability bottlenecks are the distributed registries, which are accessed for every new bundle installed and for every new service binding. Whereas the number of entries for bundles is fixed to two entries per virtual bundle (one for the bundle symbolic name + version and one for the bundle ID), classloader spaces and virtual services have a variable number of entries per object. For services, this scales with the number of service interfaces under which a virtual service is registered. Classloader spaces depend on the number of packages exported. Hence, it is a reasonable assumption that each node participating in a Cirrostratus deployment has to maintain registry entries in the range of dozens to hundreds of entries, in extreme cases (few nodes, many bundles and services) up to thousands of entries.

In a first scalability experiment, two nodes form a Cirrostratus deployment and each node has a variable number of entries (in this case entries corresponding to virtual services) in the registry. One node does a lookup for a key that maps to a service on the other node and the resulting lookup latency is measured. Figure 6.4 shows the result of this experiment using between 100 and 1,000,000 entries per node. Each data point was measured after a warmup period of 2000 lookups and for a loop of 150 lookups. The values shown are the averages over 10 to 15 iterations, the error bars correspond to the standard deviation. The impact on the lookup latency is small, only about 10% when going from 100 to 100,000 service entries. It is hence unlikely that the number of entries becomes a scalability bottleneck in practice.

What is more likely to limit the scalability is the influence of the size of the deployment and the communication latency on the access to the distributed registries. This factor, however, highly depends on the class of deployment and the network character-
6.5. Evaluation of the Scalability

Figure 6.4: Scalability in terms of registry entries

6.5.1 Cirrostratus on a Cluster

The following experiment uses a variable number of nodes which are part of the same cluster. The machines are dual-socket quadcore Intel Xeon L5520 machines running at 2.26GHz and equipped with 24GiB RAM. Oracle HotSpot Server VM build 1.6.0_02-b05 is used as a JVM. All machines are interconnected through a switched gigabit Ethernet with an average ping round trip time of 77\(\mu\)s. This time, the number of entries in the registry is fixed to 100 per node and the metric is the access time relative to the number of nodes in the deployment.

Figure 6.5 summarizes the results of the experiment for one to eighteen cluster nodes. The minimum lookup time is the trivial case in which the lookup happens on the node itself. Therefore, the graph shows a line for the minimum over all lookup times excluding the local case. The almost linear shape of the curve assesses the sanity of the network conditions over the time of the experiment. The second dotted line represents the maximum lookup times. The average is over all lookups, including the local case since this is what an application, agnostic about the distribution of keys in the system,
would encounter. Overall, the results indicate that the Cirrostratus registry scales close
to logarithmically with the number of nodes. This is due to the underlying algorithm of
the Chord DHT with its log-scale finger table. Despite the increased functionality of the
Cirrostratus registry, the good scalability of Chord is maintained.

The irregularity between 4 nodes and 5 nodes is due to the distribution of the keys.
The Chord algorithm only scales perfectly logarithmically when the keys are reasonably
equally distributed. Hosts typically calculate their keys as secure hashes over their IP
address or host names to avoid collisions with a high probability. As Figure 6.6 illus-
trates, the keys of node 4 and node 5 are very close in the key space. Whereas this does
not effect the experiment as such (entries are equally distributed to the nodes), it does
affect the effectiveness of the finger table. In this case, node 5 does not contribute to any
lookup latency except to its own since the distance between node 4 and node 5 happens
to be below the resolution of the finger table.

The graphs in Figure 6.7 shows the full lookup latency matrix for 6, 12, and 18
nodes. The white diagonal is the local case where the node requests one of its own
entries. Latencies left of the diagonal tend to be lower than on the right, which is due to
the clockwise message propagation in the ring. Looking up an entry on the predecessor
node potentially takes a full round on the circle. With a growing number of nodes,
the finger table becomes effective and the graph shows regions where the latency drops when going from one node to its successor node.

In summary, the distributed registry is the crucial part of the system since all internally shared data in the platform is stored in it. The experiments confirm that the latencies experienced on a cluster are low enough to not adversely affect the application and scale well enough to support a larger number of nodes.

### 6.5.2 Cirrostratus on PlanetLab

The cluster environment can be taken as an indication how well Cirrostratus behaves and scales within a single data center. In practice, however, applications on the cloud often require to be operated on several different data centers, e.g., in order to reduce the risk of a blackout. Therefore, we have ran the same experiment on PlanetLab, a globally distributed network of machines.

Figure 6.8 summarizes the result of the experiment running on 2 to 50 PlanetLab nodes. The resources in this network are shared with a high number of other users running experiments at the same time so that the absolute numbers can be considered a worst case scenario for a cloud deployment. The 50 nodes deployment corresponds to a
Figure 6.7: Registry lookup latencies (Cluster)
6.5. Evaluation of the Scalability

Figure 6.8: Scalability on PlanetLab

deployment to 36 physical sites, depicted in Figure 6.9.

Depending on the network conditions, an average lookup time for an entry in the distributed registry is around 400 milliseconds. The worst case lookup time is slightly below 800 milliseconds. Clearly, the much higher latency needs to be taken into account when creating globally distributed deployments. Applications which want to probe, e.g., the service registry frequently and within the critical path might not be good candidates for a large-scale distribution.

In terms of scalability, Cirrostratus has no problem scaling to the higher number of nodes. Figure 6.10 illustrates the latency matrix for 12, 20, and 50 PlanetLab nodes. Unlike on the cluster, the latency matrices form much more irregular patterns. The quality of service in terms of lookup time suffers from two factors. One is the inhomogeneity of the network links in terms of latency and the machine resources, which results in some nodes being always slower to access than others. The other is the volatility of these properties over time due to network congestion and scarcity of memory and CPU cycles on the shared nodes resulting in variations of the access time and a higher average standard deviation as in the previous experiment.

The key distribution (Figure 6.11 for 50 nodes) in this scenario is well enough to assume an equal distribution of entries in an application. However, the current im-
implementation of Cirrostratus assigns keys to the nodes by applying a secure hashing function to the host name, a strategy known from other implementations of DHTs (e.g., [Rowstron2001]). This leads to a good distribution but not necessarily to an optimal latency in the resulting DHT. Nodes physically located on the same site are likely to get keys assigned with are not adjacent in the keyspace. Methodologies like Proximity Neighbor Selection (PNS, [Castro2002]) can provide remedy by correlating keys to the proximity in the network and result in lower average latencies.

On the other hand, most cloud applications are unlikely to be distributed across a high number of data centers. For practical deployments, it is of higher importance to introduce a hierarchical distribution of values into the registry to account for the different granularities of distribution (same rack, same data center, different data center).
6.5. Evaluation of the Scalability

(a) 12 Nodes

(b) 20 Nodes

(c) 50 Nodes

**Figure 6.10:** Registry lookup latencies (PlanetLab)

**Figure 6.11:** Key distribution
6.6 Performance Evaluation

Cirrostratus is able to turn a modular application into an elastic application. In many cases, elasticity is an option for future expansion and not necessarily an immediate need. Therefore, it is important to assess the cost for opting into this flexibility.

The first attempt to do so is a micro benchmark. Figure 6.12 show the runtime overhead of having the state replication support instrumented into methods. `nop()` is a void method call that does nothing. In this case, the runtime overhead is zero. `readField()` is a method reading a field of the same class. The instrumentation places the entry barrier for applying pending updates to the field, which results in a small overhead. As the `readField10()` case shows, this overhead scales with the number of fields read (ten in this case). `writeField` and `writeField10` show the same behavior for field writes. In this case, each write has to be instrumented individually and the overhead is larger, yet within acceptable margins. The four following cases cover nested method calls. Again, the `nop` case does not involve any overhead. In the other cases, a context for the nested call has to be pushed and the arguments have to be captured. Passing a variable argument which is instrumented as a reference is not more expensive than passing a constant which is passed by value. The last method takes ten arguments, which scales well enough compared to the case with just one argument.

The experiment shows that for very small methods with just a few operations all manipulating the shared state, there is a significant cost of having the flexibility to replicate shared state on demand which can reach in some situations a factor of up 2.5. Even though the cost is high for single operations, this is in fact the worst case. In the following experiment we show that for realistic code with a mix of local operations and global state, the instrumentation overhead is small enough and that the ability to replicate services actually renders a benefit for realistic work loads.

6.6.1 Dacapo Xalan Benchmark

The DaCapo benchmark suite [Blackburn2006] is a collection of open-source Java benchmarks. Different to other existing benchmarks, DaCapo focuses on real applications using real software. For our purposes, the Xalan benchmark is best suited because it decouples the client generating the load and the processing threads through a queue. Xalan itself is an XSLT processor for performing XML transformations in Java (a version for C also exists) and is frequently used in web applications and content management systems. The server end of the queue used in the benchmark is driven by a number of worker threads dequeuing the work items, which are locations of XML files.
to process. The workers then do some XML transformations on the files using Xalan and save back the results. Client and worker threads all run in the same Java VM. Due to this structure, the transition into a Cirrostratus application is straightforward. Client and server simply become separate modules and the queue is registered by the server as an OSGi service which the client looks up to submit the items. Thereby, client and server can now run on different machines. Since the queue has become a service, it can be replicated to a set of machines and Cirrostratus manages the state replication. Thereby, multiple machines can transparently share the work load generated by the client.

In a first experiment, we compare the performance of our OSGi version of the benchmark on a conventional OSGi framework (Equinox [EclipseEquinox]) with the performance on Cirrostratus running only on a single node, again atop Equinox. The experiment ran on an 1.86 GHz Intel Pentium M system with 2 GiB main memory and a 32 bit JDK 1.5.015 Sun Hotspot VM. Figure 6.13 shows that the runtime of the benchmark on Equinox and on Cirrostratus with instrumentation explicitly disabled is very close to identical. It can be concluded that Cirrostratus itself does not impose any overhead on a running application. The third column shows that the runtime of the adapted benchmark is slightly higher but the overhead for such a more realistic application is not a factor of more than two but as low as about 4%. Hence, the runtime cost of having the services

![Figure 6.12: Microbenchmarks](image-url)
instrumented can be justified with the benefit of achieving increased flexibility through elasticity.

### 6.6.2 The Effectiveness of Elasticity

The next question to answer is how the Xalan benchmark profits from elasticity. The following experiments were performed on a cluster of dual socket dual core AMD Opteron 275 machines running at 2.2 GHz and having 4 GiB main memory. The JVM used was a 64 bit JDK 1.6.0_0-b11 Sun Hotspot VM. Figure 6.14 shows the change in runtime of the benchmark in a vertical scaling scenario for different numbers of worker threads. By default, the benchmark uses 8 worker threads. The results show that the runtime is optimal for four threads, which means each core runs one thread. This indicates that the benchmark is mainly CPU-bound. Adding more threads then increases the runtime significantly due to scheduling effects and contention on the shared data structures. The increase in performance between two, three, and four worker threads, however, is not linear. It can thus be concluded that on a larger machine with more cores available, the lock contention would limit the vertical scalability.

Figure 6.15 shows the runtime for the benchmark running with different numbers of
Figure 6.14: DaCapo Xalan benchmark with different thread counts

Figure 6.15: Replicated DaCapo Xalan benchmark running with variable numbers of replicas
replicas on dedicated machines, hence in a horizontal scaling scenario. As the values show, the DaCapo Xalan Benchmark cannot immediately profit from replication. Despite the lower total computation power, the unreplicated case beats all replicated cases and the performance even degrades quickly after more than four replicas. The reason for this behavior is that the data set processed, which are the XML files, have a small size. In average, the file size is only 13.25 kiB. This has the effect that the actual computation time per file is very low and the overhead of the distributed locking is becoming dominant. An additional indication for this finding is that three threads provide better performance than four, which means that the benchmark is becoming IO bound when running replicated. In order to optimize the execution time of the Xalan Benchmark workload, the Cirrostratus runtime should hence choose the non-replicated 4 threads operation.

With a larger data set, generated by duplicating the content of the XML files, the picture changes (Figure 6.16). In this run, the data set was increased by a factor of nine through concatenating copies of the inner nodes only so that the XML schema of each file was not violated. The values show that under these conditions replication pays off and can significantly decrease the total processing time. When using the larger data set in a vertical scale-up, the benchmark still exhibits the same behavior as the smaller data.

Figure 6.16: Same setup, larger data set
It takes 40.2 ms (±0.57 ms) to complete the benchmark using the larger data set with four threads, 43.3 ms (±0.51 ms) with five threads, and 44.5 ms (±0.71 ms) with six threads.

The conclusion of the Xalan Benchmark is that the sweet spot for parallelizing an application depends on a number of factors and is not easily predictable. This even more underlines the benefit of making applications elastic so that the runtime can change the deployment adaptively. However, besides the number of replicas, their placement with regard to service clients is also important. The following use case illustrates this with a large and non-trivial application.

### 6.7 Use Case: Stendhal

Stendhal is a multiplayer online game based on the Arianne engine [ArianneProject2005]. The game and the engine are written in Java and normally used in a client-server setting. Figure 6.17 shows a screen-shot the running Java client. In order to assess the feasibility of the Cirrostratus runtime for realistic code bases and larger distributed systems, we turned the game into a modular structure and bundled the different components.

The server is turned into a big module with different services encapsulating distinct functionality of the system, e.g., maintaining the information about players or handling the loading of bitmaps resources. The server module encapsulates about 83,000 source lines of code (SLOC). The client becomes another module, amounting to about 40,000 source lines of Java code. All explicit communication between the client and the server are eliminated and replaced by service invocations. Furthermore, the original game engine had the state externalized into a relational database connected through JDBC. We replaced the database tables by sets of objects with fields and thereby turned the server into an in-memory system.

In order to run experiments, we implemented a simulated player in the client that causes the character to perform random walks through the game world. The character chooses a direction to move with the probability of 1/5 for each of the four directions or stops for a round with the probability of 1/5, which models a think time. When the character has chosen a direction, it moves into this direction until hitting an obstacle or a 1 of 80 chance for stopping is hit. In these two cases, it will start over with choosing a direction. This implementation provides a smooth movement that is (at least visually) similar to a human player.

The first experiment involves one machine (node A) running a server instance and one machine (node B) running a client. Both machines are part of the cluster used for
Figure 6.17: Stendhal

Figure 6.18: Stendhal elastic redeployment
the previous benchmark. The network in this cluster had an average ping round trip time of 92 $\mu$s. The client communicates with the server through a service which is, when the client starts up, in the first place implemented as a remote service invocation. After stabilizing, this causes a traffic of 81.428 Bytes/s from the client on node A to the server on node B and 32.890 Bytes/s back from B to A. The average latency for service invocations is 753 $\mu$s. A simple controller makes the decision to create a replica of the server on node A. After stabilizing again, the traffic from A to B drops to zero because there are no writes happening on the original service instance. The local replica is now used by the client, which causes in average only 64.440 Bytes/s in synchronization traffic to node A. Using a lazy replication protocol, this traffic could be completely avoided. The latency for service invocations has dropped to 510 $\mu$s.

The second experiment is a more complex scenario involving a server machine and two client machines (node 1 and 2). While the clients are in the same cluster and have an average RTT of 130 $\mu$s between them, the server node resides in a different cluster with an average RTT of 533 $\mu$s to the clients. Figure 6.18 summarizes the impact of several stages in the optimization of the deployment. The upper graph shows the bandwidth consumption in Bytes per second for the server and the two client nodes, both through the remote service and the replication protocol. The lower graph shows the latency observed by the two clients in milliseconds. The experiment begins at time T0 where the server and the client on node 1 run. At time T1, the second client starts up, connects to the service through a remote service, and causes the overall bandwidth consumption of the system to increase. The controller reacts at time T2 by replicating the server to node 1. This causes the server bandwidth consumption to drop as well as the bandwidth consumption of the replication protocol to rise but with a lower aggregate consumption. The latency drops for node 1 which now has a local replica and increases for node 2 since the server now also has to process updates from the replica on node 1. At time T3, the controller creates a second replica of the server on node 2, which now puts all the load on the replication protocol but is able to decreased the aggregate bandwidth consumption and latency. If the policy had allowed the controller to even drop the original server, it would have turned the client server setup into a full peer to peer system. A more sophisticated controller and additional clients in different places would then cause the system to go back to servers again due to the increasing overhead of the state replication for larger numbers of replicas. The absolute numbers measured are not of great significant since the highly optimized R-OSGi protocol for the remote service invocation competes against the widely unoptimized state replication protocol of the Cirrostratus prototype. However, the experiment illustrates the effectiveness of elastic redeployment and how simple transformations of the setup can optimize the resource
consumption of the system.

6.8 Related Work

Providing a single system image for Java applications was initially the domain of the distributed Java virtual machines (dJVMs). Prominent representatives include the cJVM [Aridor2000] system and Jessica2 [Zhu2002]. Both operate on the granularity of threads and are thereby not well designed for elasticity. Furthermore, dJVMs are full replacements for JVMs and cannot catch up with the high performance of today's industrial strength JVM implementations providing just-in-time compilation. Another approach for a customized cluster-enabled JVM is Java/DMS [Yu1997] which externalizes shared objects into a second heap under the governance of a distributed shared memory system. Clustering approaches like JavaSplit [Factor2003] sitting atop the JVM could partly mitigate the performance problem of dJVMs by, like Cirrostratus, adapting the byte code of the application rather than requiring custom and less optimized JVM.

Historically, there are two different strategies for implementing the idea of objects which are shared between different nodes. The first strategy is object caching and involves that replicas of objects are read-only and invalidated whenever somebody writes to the primary object. It was implemented, for instance, in systems like Javanaise [Hagimont1998] designed for applications in wide-area networks. One of the pioneers of distributed shared objects based on an update mechanism is the ORCA [Bal1990] parallel programming language, initially built on top of the Amoeba distributed operating system [Mullender1990]. The model of ORCA guarantees sequential consistency on shared data, which is represented as abstract data types (ADTs). Similar update mechanism approaches also exist for Java. For instance, the Mocha [Topol1998] system listens to lock and unlock that local threads try to acquire on replicated objects and turns them into distributed locks (ReplicaLocks). Changes performed on the locked object are propagated to all replicas depending on the chosen consistency protocol. Cirrostratus provides a simple default replication protocol but facilitates the application to provide and configure alternative implementations. It thereby enables arbitrary consistency, tunable per service, and can inter-operate with any of the approaches described before.

Most approaches for distributed shared objects, involve synchronous operations on the replicated objects in some flavor. In contrast, the Active Objects pattern completely decouples the invocation of a method on a shared object from the actual execution [Lavender1995]. When an invocation occurs, a request is asynchronously stored in a
queue. Hence, the method call of an active object is non-blocking. As a placeholder for the method result, a *Future Objects* is returned which delays the computation of the method invocation until the result is effectively needed. Each active object has its own thread running a scheduler which determines which request is to be executed next. This pattern is, for instance, implemented in the *ProActive* [Baduel2006] middleware. In its asynchrony, active objects are similar to the default state replication strategy of Cirrostratus that processes updates outside the critical path. Instead of requiring future objects, Cirrostratus uses the entry barrier to block if unprocessed updates are pending.

*Open Terracotta* [Pepperdine2006] is the freely available version of the Terracotta Clustering Server. The software is able to transparently replicate Java objects and thereby clustering Java applications. As an input, the Terracotta server requires a configuration file that precisely describes which objects have to be replicated and which replication technique should be used. The clustering server can either rely on existing critical sections or use named locks on methods. In this, Terracotta differs greatly from Cirrostratus which can largely infer this information from the code. The Terracotta server is the central unit in the system which stores the object data of shared objects and coordinates the threads running on the application nodes. Accesses to fields of shared objects are instrumented at load time and redirected to the Terracotta transparency libraries which reside in the JVM of each node. In contrast, Cirrostratus is a peer to peer system without a single point of failure. Additionally, Terracotta replaces classes of the Java class path with its own implementations. This can, under some conditions, break applications. For instance, Terracotta is known to currently be incompatible with certain `java.util.concurrent` classes. Cirrostratus, in turn, does not modify the Java class path and only instruments the application, meaning that it is forward compatible to upcoming releases of the JVM and their class paths.

Adaptation of code is often associated with *Aspect-Oriented Programming* and languages extensions like AspectJ [Kiczales2001] or systems like PROSE [Nicoara2008]. AOP systems operate on imperative constructs like join points and point cuts requiring knowledge about the structure of a program or even the source code. The code adaptation in Cirrostratus, in contrast, are driven by symbolic execution and match on declarative patterns independent of concrete code.

### 6.9 Summary and Discussion

The Cirrostratus system is a versatile platform for transparently turning modular OSGi applications into elastic applications. The system addresses the three key components
required to add elasticity to Java applications. It provides the module as a unit of parallelization through dynamically re-deploying and re-binding the services used as abstract communication channels between modules. It provides mechanisms to transfer and synchronize the state of services in response to the now distributed state and does so by inferring the required information from the byte code. And, it provides a management infrastructure facilitating the application or an external controller to adapt the deployment on demand to supervise and meet the non-functional requirements. The runtime is carefully designed to be infrastructure and not hard code policy into the system. With the Xalan benchmark and the ability to run a large Java application atop the system, we have shown that the system is actually able to add elasticity to non-trivial Java applications.

What has so far been done in terms of composition of modules during the compilation process or the installation process on a single machine now also applies to deployments on a variable set of machines. Again, the loosely-coupled points of interaction (in OSGi through services) become the boundaries of potential distribution. As the substitution principle mandates, concrete physical instances can at any time be exchanged but this time the spacial distribution of modules can be arbitrarily altered without losing internal state. The reason why the state can be captured through symbolic execution is the encapsulation of modules. Internal state cannot arbitrarily leak through the system but only through well-defined interfaces. In fact, the main restricting assumption made in Cirrostratus compared to a general OSGi application is that the internal state is perfectly encapsulated and hidden behind service interfaces and not arbitrarily shared through package exports. Thereby, the access patterns leading to changes in the internal state are finite and all rooted in the methods of the services exposed. If there was no such encapsulation, the symbolic execution could never capture all code paths leading to state changes.
6.9. Summary and Discussion
Chapter 7

Conclusions

7.1 Modular Systems

The fundamental challenge motivating this thesis is the increasing dynamism of environments in which software systems are expected to run in. Examples for such environments include modern and future hardware with the ability to enable and disable resources on demand as well applications of cloud computing operating on variable sets of machines. Such a degree of flexibility and adaptability is not well supported by today's software systems. This thesis contributes towards taking a holistic approach on software design by applying the idea of modularity not only on a logical but also on a physical level with respect to future deployments. Three main systems have been presented in this thesis and each is able to make use of different properties of software modularity to add a certain behavior to modularized applications.

With Juggle, we have shown how modularity helps to manage heterogeneity and the ability to reprogram hardware resources and makes them usable to the application. Through the substitution principle and strict encapsulation, it is possible to treat modules written for completely different environments equally as long as they adhere to the same interface. Juggle applies this to conventional Java OSGi services and services backed by a reprogrammable FPGA device. The system is able to apply the same management facilities to both worlds and seamlessly switch from one implementation to the other. Thereby, it facilitates on-demand acceleration of parts of the application without changing either the structure of the application or any line of code. The existing boundaries created through the modularization and the existing points of loosely-coupled interaction through service interfaces are enough for a runtime system like Juggle to substitute functionality through hardware accelerators. The effectiveness of the acceleration is illustrated by the example of the Triple-DES encryption algorithm, which experiences
a 20-fold improvement when the Java software implementation is substituted with an FPGA-based implementation.

R-OSGi shows how modularity helps to separate the design of an application from an actual deployment. The flexibility of the design gained by decomposing the system into modules and taking care that the modules can be composed and re-used in other applications is preserved to the deployment phase and the runtime of the application by not only applying logical but also physical modularity. In the concrete example of R-OSGi, this flexibility is achieved by turning conventional modular OSGi applications transparently into distributed systems through the introduction of remote services and proxy modules. Developers can focus on designing the functional units as modules and the interfaces as services. The network communication is introduced by the runtime where necessary so that the same, unmodified modular application can be distributed in different deployments and even redeployed on demand. As an additional effect, network failures can be mapped to module unload events which OSGi applications are already prepared to handle gracefully. As a result, modularity becomes a lightweight programming model for distributed systems. The concrete boundaries of distribution can be shaped and adapted at runtime since the modules already provide plenty of opportunities through loosely-coupled interaction. Making late choices about where to introduce communication and in which form does not result in any substantial runtime-overhead, as the experience with R-OSGi shows. Using R-OSGi and placing a remote invocation using its own protocol between a Java party service and client is competitive with using RMI, which requires the developer to adapt the code and explicitly deal with the remote communication at design time. One of the challenges is to deal with the tightly-coupled dependencies as well, which potentially limit the flexibility of the deployment. R-OSGi does so by providing the type injections into the proxy modules and actively managing the dependencies of remote services. Both takes advantage of the declaratively self-containedness of software modules.

Cirrostratus is the next logical step into the direction of embracing the dynamism with fluid software modules. It implements a runtime system capable of adding elasticity in an orthogonal manner and thereby shows the potential of modular software to deal with the challenges in cloud computing. Modules define a virtual deployment which describes the levels of interaction between functional units. The runtime, however, has the freedom to choose any physical deployment that complies with the requirements expressed through the virtual deployment. In particular, the runtime can replicate modules and services on demand and in an elastic manner. Regions of shared state among replicated modules are detected through a symbolic code analysis at load time by the runtime. With this knowledge, the runtime system instruments the modules to intro-
duce the interaction with replication protocols and to insert performance monitors for controlling. Using replication to exploit inherent parallelism works in practice, as the example of the Stendhal game shows.

The three systems researched occupy different points in the design space and illustrate different advantages of software modularity. Neither of the systems makes the claim to be the only possible or most effective solution to the problem that it tackles. In particular, some of the challenges that had to be dealt with in order to implement the systems could be avoided by choosing a particular and more restricted application model than plain OSGi modules. For instance, the problem of state in Cirrostratus can be solved by mandating that all state has to be externalized into a database system. OSGi was chosen as the foundation for the systems and as a base for the research on software modules due to its ability to actively manage the deployment of an application in an easy and intuitive way even from within the system. The generality of OSGi, which is the entire expressiveness of the Java language, rendered it a good object of study. This, however, does not restrict any of the concepts presented in this thesis to OSGi or even Java.

The lessons learned from implementing and researching modularity are that a well-designed modular decomposition together with an intelligent runtime system is a very promising approach for dealing with increasingly dynamic environments in which no longer strong assumptions can be made at design or compile time. In particular, the systems presented in this thesis clearly indicate that very different concerns like hardware acceleration and large-scale distribution can be incorporated into a systems design in a uniform way by simply introducing structure and degrees of freedom through loose coupling.

Unfortunately, in practice a minority of existing applications turn out to be properly modularized. We took the effort and attempted to modularize different existing large code bases as, e.g., the Apache Derby database engine [BenCharrada2008], with mixed results. It turned out that besides the expected points of high coupling for reasons of performance optimization, certain programming practices lead to an unpredictable degree of accidental coupling which hampers the effort of modularization. In the case of Derby, this is, for instance, the use of a single huge data structure on which all initial, intermediate, and result data is attached during the processing of requests and which has a transitive hull that contains almost the entire database engine. This clearly is a poor separation of concern and fixing such a problem essentially comes close to a complete redesign of the application. Therefore, to a certain extent the approach of using software modules as a systems design principle is more applicable to implementing new system than to refactoring legacy software.
7.2 Directions for Future Work

A possible direction of future work is to further generalize the approach of using modules as a system design principle in practice and choose platforms other than Java and OSGi. For instance, in an accepted project proposal for the Microsoft Innovation Cluster for Embedded Software (ICES) we outline how a module runtime system could become a viable application platform for embedded systems running an enhanced .NET micro framework. The challenges connected to this research goal are to make modules a first-class citizen of the host language and to introduce the necessary support for modularity into the language runtime.

Another possible research focus is the exploration of modularity for increased parallelism and concurrency. An uncompromised modular design exhibits great potential for building highly concurrent and parallel systems to deal with the challenges of both clusters of machines and multi-core machines that are increasingly becoming distributed system on their own. Especially the parallelism resulting from multiple users using the same software modules can be exploited through intelligent and self-adaptive runtime systems. With the emergence of massively multi-core chips like the Intel SCC that deliberately avoid the overhead of cache-coherence, the challenges of software architecture for such platforms become similar to the challenges of designing software for clusters of machines. The difference is the high-bandwidth interconnect that permits a higher degree of coordination between nodes than typically practical in distributed systems. The concept of modularity in combination with intelligent runtime systems like Cirrostratus can be a key technology to successfully exploit such emerging hardware.

Finally, an interesting research question is whether it is possible to design languages in a way that is more friendly to a modular application design. OSGi, for instance, uses Java, which is an object-oriented programming languages. As already indicated earlier, single objects are unlikely to be feasible boundaries for modularization for a flexible systems design. On the other hand, it features logical modularity in the form of packages but without any form of dependency management and without any connection to physical modularity. OSGi arguably fixes this by mapping collections of packages to bundles and declaring their runtime-dependencies. The question, however, remains if a language which treats modules as a first-level citizen and applies more rigid enforcement of the principles of modularity could potentially provide much of the advantages outlined in this thesis with less effort by applying cross-layer optimization between the language and the module management system.
Bibliography


