The Software Fabric for the Internet of Things

Jan S. Rellermeyer[1], Michael Duller[1], Ken Gilmer[2], Damianos Maragkos[1], Dimitrios Papageorgiou[1], and Gustavo Alonso[1]

[1] ETH Zurich, Department of Computer Science,
8092 Zurich, Switzerland,
{rellermeyer, michael.duller, alonso}@inf.ethz.ch
{dmaragko, dpapageo}@student.ethz.ch

[2] Bug Labs Inc.
New York, NY 10010
ken@buglabs.net

Abstract. One of the most important challenges that need to be solved before the "Internet of Things" becomes a reality is the lack of a scalable model to develop and deploy applications atop such a heterogeneous collection of ubiquitous devices. In practice, families of hardware devices or of software platforms have intrinsic characteristics that make it very cumbersome to write applications where arbitrary devices and platforms interact. In this paper we explore constructing the software fabric for the "Internet of Things" as an extension of the ideas already in use for modular software development. In particular, we suggest to generalize the OSGi model to turn the "Internet of Things" into a collection of loosely coupled software modules interacting through service interfaces. Since OSGi is Java-based, in the paper we describe how to use OSGi concepts in other contexts and how to turn non-Java capable devices and platforms into OSGi-like services. In doing this, the resulting software fabric looks and feels like well known development environments and hides the problems related to distribution and heterogeneity behind the better understood concept of modular software design.

1 Introduction

The notion of an "Internet of Things" refers to the possibility of endowing every day objects with the ability to identify themselves, communicate with other objects, and possibly compute. This immediately raises the possibility to exchange information with such objects and to federate them to build complex composite systems where the objects directly interact with each other. These composite systems exhibit very interesting properties and can have a wide range of applications. Yet, they are complex to develop, build, and maintain in spite of the purported spontaneity of the interactions between the objects. A crucial reason for these difficulties is the lack of a common software fabric underlying the "Internet of Things", i.e., the lack of a model of how the software in these different objects can be combined to build larger, composite systems.

The problem in itself is neither new nor restricted to pervasive computing. For instance, a great deal of effort has been invested in the last decades to develop concepts for modular software design. There, the problem is how to build a coherent application out of a possibly large collection of unrelated software modules. Conceptually, it is intriguing to consider the problem of building the software fabric for the “Internet of Things” as another version of the problem of modular software design. Both problems are closely related, in spite of addressing different contexts, and being able to establish a connection between them will mean that we can bring all the experiences and tools available for one to bear on the other.

This is the premise that we pursue in this paper. We start from OSGi [1], a standard specification for the design and management of Java software modules. Despite its recent success for server-side application, OSGi was originally developed for managing software modules in embedded systems. Taking advantage of recent developments that have extended OSGi to distributed environments (R-OSGi [2]), we show how to turn small and ubiquitous devices into OSGi-like services.

Many aspects of the OSGi and R-OSGi model are perfectly matching the requirements of applications on the “Internet of Things”. 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.

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 [3], ZigBee [4]). To accommodate these characteristics, in the paper we describe extensions to R-OSGi that:

- make communications to and from services independent of the transport protocol (in the paper we show this with Bluetooth and 802.15.4 [5]),
- implement an OSGi-like interface that does not require standard Java or no Java at all (in the paper we demonstrate this for CLDC [6], embedded Linux, and TinyOS [7]), and
- support arbitrary data streams to and from services.

Through these contributions, the resulting fabric for the “Internet of Things” can be treated and manipulated as an OSGi framework that is also applicable to small devices, alternative transport protocols, and non-Java applications. This means that we can treat devices as composable services with a standard interface, services can be discovered at runtime and changed dynamically, data exchanges can use the Java types even for non-Java applications, and failures can be masked behind service withdrawal events. Application developers see a collection of software modules with a homogeneous interface rather than a collection of devices. In the paper we give an example of an extensible hardware platform – commercially available – where each hardware module is treated as an OSGi module, thereby proving the feasibility of the premise that we are pursuing in this paper.

The paper is organized as follows. In Section 2, we review the main ideas behind OSGi and motivate our approach. In Section 3, we discuss how to make OSGi frameworks independent of the transport protocol. In Section 4, we describe how to provide OSGi-like services on platforms either running reduced versions of the JVM, programmed in languages others than Java, or lacking a full-fledged operating system. In Section 5, we present an extension to R-OSGi to support data streams. In Section 6, we describe how all these extensions make the OSGi model general enough to become the software fabric for the "Internet of Things".

2 Background on OSGi and Motivation

2.1 The OSGi Model

OSGi [1] is an open standard for developing and managing modular Java applications. Efforts around OSGi are driven by the OSGi Alliance, with contributions from software manufacturers like IBM, BEA, or ORACLE, as well as device vendors like NOKIA, SIEMENS, or BOSCH. OSGi is already in use in a wide range of systems (including the development environment ECLIPSE, but also in embedded systems such as those in the BMW 3 series car).

In OSGi, software modules are called *bundles*. Bundles can export packages and import shared packages of other bundles. The runtime of OSGi (called *framework*) handles the dependencies between the bundles and allows the active control of every bundle's lifecycle. Each bundle is loaded through a separate Java classloader and shared code is handled by a delegation model. It is possible to extend an application by installing new bundles at runtime as well as updating existing bundles or unloading them. To avoid link-level dependencies between code of bundles, which would limit the flexibility of a modular architecture, OSGi provides a *service* abstraction. A service is an ordinary Java object which is published to the framework's *service registry* under the names of the interfaces it implements. Additionally, a service can be described in more detail by attaching properties to the service's registration. These properties can also be dynamic, i.e., change at runtime. Other bundles can make use of services by requesting the implementation from the framework. Since service clients only have to know the interface but not the actual implementation, this leads to a loose coupling of the bundles. Use of services relies on common, published interfaces but the implementation of each service remains a black box. It is, thus, possible to exchange the implementation of a service at runtime. It is also possible to have several alternative implementations of a service which can then be matched at a finer level through the use of properties.

Existing OSGi frameworks include commercial implementations such as IBM's SMF [8] as well as open source projects such as ECLIPSE EQUINOX [9], APACHE FELIX [10], CONCIERGE [11], or KNOPFLERFISH [12].

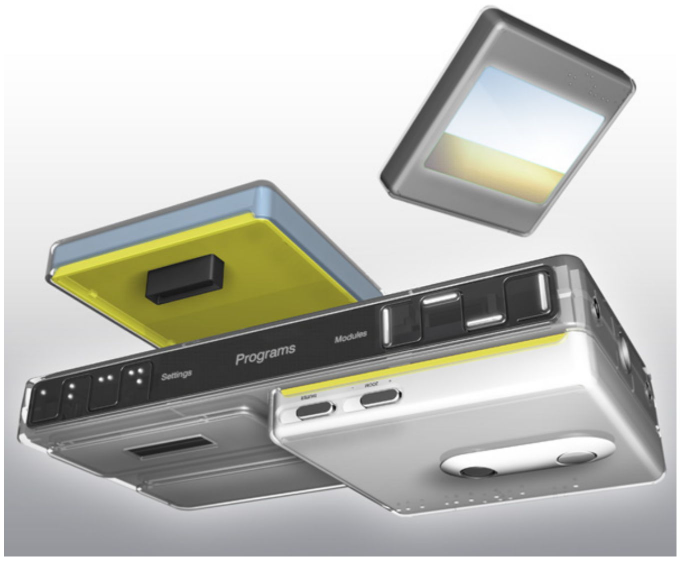

Fig. 1. The Bug with several modules

2.2 Example: The BUG Platform

One can question the wisdom of trying to generalize OSGi beyond module management in a centralized setting. However, this has been done already and quite successfully. An example of a commercial device which internally uses OSGi to utilize modular hardware is the Bug (Figure 1), developed by the New York based company Bug Labs Inc. The device consists of a base unit (computer) and hardware modules that allow the system to be extended. The base unit is a handheld ARM-based GNU/Linux computer that runs the Concierge OSGi framework on a CDC VM.

A primary feature of Bug is that it's expandable. A dedicated 4-port bus known as the *Bug Module Interface* (BMI), allows external devices (*hardware modules*) to interface with the base unit. The BMI interface is hot pluggable both in hardware and software, meaning that modules can be added to and removed from the base unit without restarting the computer or manually managing applications that depend on specific modules. The BMI interface is comprised of eight standard interfaces such as USB, SPI, and GPIO which are all available through each connector. Each hardware module exposes via OSGi a set of services that local or remote applications can utilize.

At the software level, BMI behaves like any other hot-pluggable bus, such as PCI and SB. Hardware modules can be added to and removed from the Bug base unit without restarting the device. Based on what modules are connected, OSGi services are available that can be utilized by local and remote OSGi-based applications. Some examples of hardware modules are LCD screen, barcode scanner, weather station, and motion detector.

In a way, the Bug device can be considered as a physical actualization of the OSGi service model: each hardware module contains devices that, when connected to a base unit, register themselves within the OSGi runtime as a service. When a user attaches a hardware module to the base unit, one or more hardware service bundles are started in the OSGi runtime, in addition to any application bundles that depend on those services. Similarly, when modules are removed, the same bundles are removed from the system. Since OSGi was designed to support dynamic services, applications written for Bug work seamlessly with the highly variable nature of the hardware.

The Bug implements the idea of a service-oriented platform for a collection of hardware devices plugged to each other. The vision we want to articulate in this paper is one in which OSGi plays a similar role but in the context of the "Internet of Things" rather than for hardware modules. In order to do this, however, OSGi needs to be extended in several ways. First, it has to be able to support distributed applications which the current specification does not [2]. Second, a framework is needed capable of running on small devices, something that only a few of the existing frameworks can do but at a considerable performance penalty [11]. These two points have been covered by recent developments (see the remainder of this section). From the next section on, we tackle the rest of the necessary extensions to make the OSGi model suitable for the "Internet of Things": making the distribution protocol independent, removing the dependency on a regular Java Virtual Machine, removing the dependency on Java altogether, and extending it to support continuous media streams.

2.3 R-OSGi

OSGi can only be effectively used in centralized setups where all bundles and services reside within the same Java Virtual Machine. Not only the dependency handling requires all modules to reside on the same machine, the service registry is a centralized construct that is not designed for distribution.

R-OSGi [2] is an open source project designed to address these limitation and provide a distributed version of the OSGi model. In R-OSGi, service information is registered with a network service discovery protocol so that a node can locate services residing on other nodes. The original implementation of R-OSGi was targeting TCP/IP networks and used RFC 2608, the Service Location Protocol (SLP) [13, 14]. The notion of services in OSGi and SLP are very close to each other and SLP can run in both managed networks where a *Directory Agent* acts as a central service broker, and in unmanaged networks by using multicast discovery. The connection between two applications through an R-OSGi service is called a *network channel*. To access remote services, R-OSGi generates a dynamic proxy for the service on the caller's side, which makes the service look like a local service. The proxy itself is implemented as an ordinary OSGi bundle. To deal with dependencies, R-OSGi determines the minimal set of classes which are required to make the service interface resolvable. Those classes from this set that do not belong to the VM classpath are additionally transmitted to the client and injected into the proxy bundle. Method calls from the proxy to the remote service are executed by exchanging appropriate messages between the R-OSGi instances running in the proxy's framework and running in the remote service's framework. Arguments to the remote method are serialized at the caller's site and sent with the call request. At the callee's site the arguments are deserialized again and the method is invoked with the deserialized arguments. If the method returns a return value, the value is serialized and sent to the caller's site together with the acknowledgment of the successful method invocation. Finally, the return value is deserialized at the caller's site and returned to the proxy.

In addition to method calls to the remote services, R-OSGi also transparently supports event-based interaction between services as specified in OSGi's Event Admin service specification [1].

R-OSGi can be effectively used to build applications from a federation of distributed services. However, it works only with TCP/IP and requires a (at least J2ME CDC compliant) Java VM with an OSGi framework implementation. Finally, R-OSGi has no support for data streams such as those produced by, for instance, RFID readers. Thus, it is not suitable for the applications we have in mind as part of the "Internet of Things". In what follows we show how this general model we have just described (starting from R-OSGi) can be extended so as to build the underlying software fabric for the "Internet of Things".

3 Extending R-OSGi to other Transport Protocols

The current Internet is based on TCP/IP both for wired and wireless communication (802.11 wireless LAN). In contrast, the "Internet of Things" often involves communication over low-power, short-range networks such as Bluetooth [3] or ZigBee [4]. R-OSGi, however, is limited to TCP/IP and not suited well for the context of the "Internet of Things". Thus we have modified R-OSGi, which is available as open source, to make it independent of the underlying transport protocol. We do this by describing the necessary extensions to support Bluetooth and 802.15.4 protocols.

There are two design properties that prevent R-OSGi to be effectively used with transports others than TCP/IP. First, the SLP service discovery is an integral part of R-OSGi and pervades all the layers. For instance, remote services are identified by their SLP service URL. The SLP protocol, however, is tightly bound to IP networks. Second, the network channel model [2] of R-OSGi allows to plug in alternative transports for outgoing communication but the acceptor for incoming connections only supports TCP/IP sockets. In the model of R-OSGi, it is assumed that alternative transports would bridge to TCP over local loop, like the HTTP transport provided by R-OSGi. This approach works for desktop computers with powerful and versatile operating systems but is unfeasible for ubiquitous devices communicating over Bluetooth or ZigBee.

3.1 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

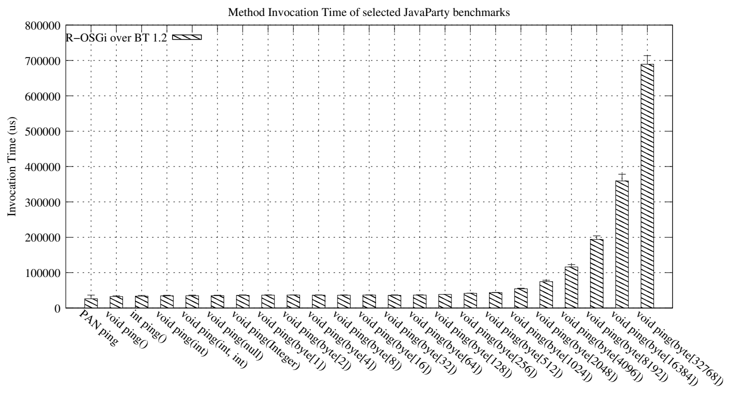

Fig. 2. Javaparty Benchmark over Bluetooth 1.2 Transport

URLs, we use opaque URIs. The URI of a service can use any schema and thereby identify which transport implementation it requires. 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 TCP/IP can be: *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., *btspp://0010DCE96CB8:1#21*.

Bluetooth [3] is a network solution covering the lower five layers of the OSI model [15]. 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 with the JavaParty benchmark. The JavaParty benchmark was originally developed by the University of Karlsruhe as part of the JavaParty [16] project to test the performance of an alternative RMI implementation. In the OSGi version of the benchmark, a test client calls different methods of a remote service with arguments of increasing size. Figure 2 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.

3.2 Bluetooth Service Discovery

R-OSGi supports service discovery through SLP but only over an IP-based network transport. We have changed service discovery to make it an orthogonal concern. SLP discovery is still supported in the form of a separate and optional service that gives hints to the application when certain services are found. Alternative discovery protocols can now be used as well since the service discovery operates behind a common discovery handler interface that is not restricted to 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) [3].

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 subentities 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 gap between Bluetooth services and OSGi services. SDP supports queries only on UUIDs (Universally Unique Identifiers) [17] whereas OSGi allows semantically rich and complex LDAP filter strings [18]. In addition, SDP data types do not suffice to express the properties of OSGi services. This can lead to problems when selecting services as the limited expressibility in SDP will result in many more services answering a request than is necessary.

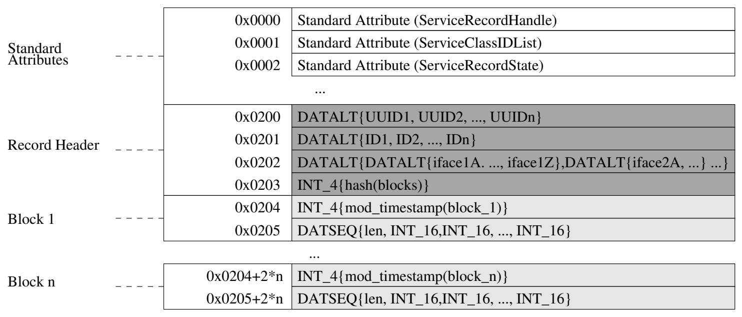
Standard Attributes
0x0000 Standard Attribute (ServiceRecordHandle)
0x0001 Standard Attribute (ServiceClassIDList)
0x0002 Standard Attribute (ServiceRecordState)
...
Record Header
0x0200 DATALT{UUID1, UUID2, ..., UUIDn}
0x0201 DATALT{ID1, ID2, ..., IDn}
0x0202 DATALT{DATALT{iface1A. ..., iface1Z},DATALT{iface2A, ...} ...}
0x0203 INT_4{hash(blocks)}
Block 1
0x0204 INT_4{mod_timestamp(block_1)}
0x0205 DATSEQ{len, INT_16,INT_16, ..., INT_16}
...
Block n
0x0204+2*n INT_4{mod_timestamp(block_n)}
0x0205+2*n DATSEQ{len, INT_16,INT_16, ..., INT_16}

Fig. 3. SDP Service Record for R-OSGi Services

To provide OSGi-like service discovery over Bluetooth, we keep 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 our 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.

3.3 802.15.4 Transport

The IEEE 802.15.4 standard [5] defines *PHY* and *MAC* layers for short range, low power radios. 802.15.4 is the basis for the lower layers of ZigBee [4], 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, we wanted a more lightweight solution that can even run on, e.g., TinyOS powered devices. Hence, we have implemented a simple transport layer similar to TCP that uses acknowledgments, timeouts, and retransmissions to provide reliable end-to-end communication on top of 802.15.4. Our implementation of the transport layer represents a proof of concept only as we did not optimize it 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 OSGi-like Services on non-standard Java

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 microcontroller 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. 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 of the remote service is built on the fly and registered with the local OSGi framework like other local services. We exploit 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.

The first platform we adapt to running OSGi is the Java CLDC [6]. It lacks certain properties that are required for running OSGi, for instance, user-defined classloading. The second platform we adapt are embedded Linux devices running services written in C or C++, thereby removing the dependency on Java. Finally, the third implementation works on top of TinyOS running on Tmote Sky [19] nodes featuring 10kB of RAM and 48kB of code storage, thus removing the dependency on a full fledged operating system.

4.1 CLDC

CLDC is the smallest version of the Java Virtual Machine. It ships with almost every mobile phone and is also used on embedded devices with very little resources (minimum requirements are $160kB$ non-volatile memory as storage and $32kB$ volatile RAM). The VM for CLDC is hence called the *Kilobyte Virtual Machine* (KVM) [6]. On top of the VM, different *profiles* can run. The profile defines which subset of the standard Java classes is available, which CLDC-specific classes are provided, and which packages can be optionally included. For instance, the MIDP [20] profile is the most widespread on mobile phones and allows to run applications packages as MIDlets.

The reason why OSGi does not run on CLDC devices is mainly the sandbox model in the underlying KVM. For instance, in the MIDP profile, MIDlets cannot access code of other MIDlets and no new classes can be loaded that are not part of the MIDlet itself. For such devices, we locally run an adapter that understands the R-OSGi protocol and communicates with the local services.

The adapter is implemented as a single MIDlet that contains a stripped-down implementation of the R-OSGi protocol (since it only needs to deal with incoming calls) and the services that the device offers. Since the KVM does not support reflection, the dispatching of remote service invocation calls to the service methods has to be done explicitly. A further limitation of the KVM is the missing Java serialization. This is a severe limitation because not only the arguments of service methods might involve complex Java objects, also the R-OSGi protocol itself has some messages that ship reference types, for instance, the properties of the service. The solution to the problem is to generate custom serialization and deserialization methods for every used complex type ahead of time when the MIDlet is assembled. The generation procedure uses bytecode analysis to inspect the classes and generates a serialization for every complex type used in a service interface. Since the arguments of service methods are known at the time where the MIDlet is generated, arbitrary types can be serialized. Also the customized method calls are generated at build-time so that the service does not have to be changed in order to run on CLDC. Clearly, the statically assembled MIDlet does not support all features that an OSGi service running on a framework supports.

Although it is possible to enable and disable services through a management service, the update of a single service requires an update of the whole MIDlet.

4.2 Embedded Linux

Supporting services written in languages other than Java is mainly a problem of adapting the data types. OSGi, that is Java, allows to use very expressive classes which cannot be easily transformed to other languages. The usual way to invoke native code from Java code is through the Java Native Interface (JNI) [21], an interface that exposes an API that can be used to create and access objects and to invoke methods.

In the R-OSGi C/C++ implementation, we developed a stand alone library (JNI Runtime) that implements the JNI interface but without actually being a Java VM (Figure 4). The JNI runtime library does not have the overhead of interpreting any Java bytecode 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*. The daemon 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. In this file, the developer also has to specify 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

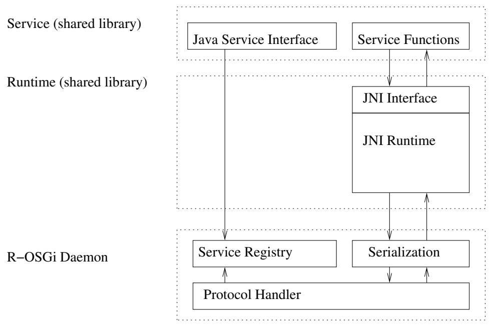

Fig. 4. R-OSGi for Native Code Services

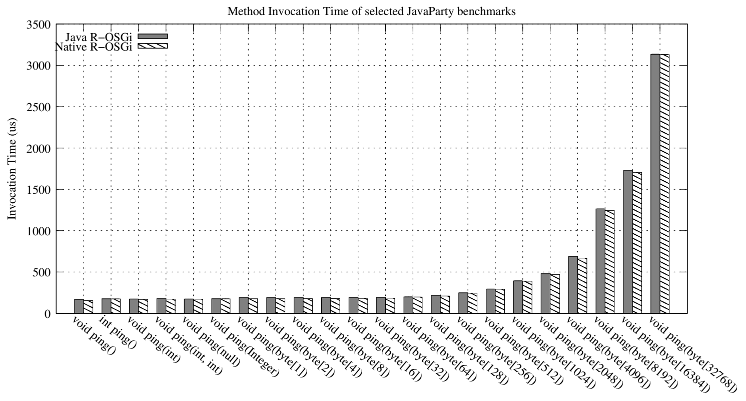

Fig. 5. Javaparty Benchmark on Linux

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 5 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 huge 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 6). This time, a JamVM [22] is used as a reference. The native R-OSGi implementation performs significantly better than the interpreting VM in all cases. Even the call with $32kB$ of arguments completes in less than $20ms$.

4.3 TinyOS

Our implementation of R-OSGi services for TinyOS [7] is based on the 802.15.4 transport presented in the preceding section and, like all TinyOS applications, implemented in NesC [23]. NesC adds the concept of components, interfaces, and wiring of components to the C programming language. A TinyOS application consists of several components that are wired together. TinyOS provides a collection of ready to use components, mainly hardware drivers. The Tmote sky nodes implement the TelosB [24] platform of TinyOS and all hardware components that this platform is composed of are supported in the current TinyOS release 2.0.2. Therefore and because of previous experience with Tmote sky nodes and TinyOS, we chose to implement R-OSGi services for sensor nodes using TinyOS.

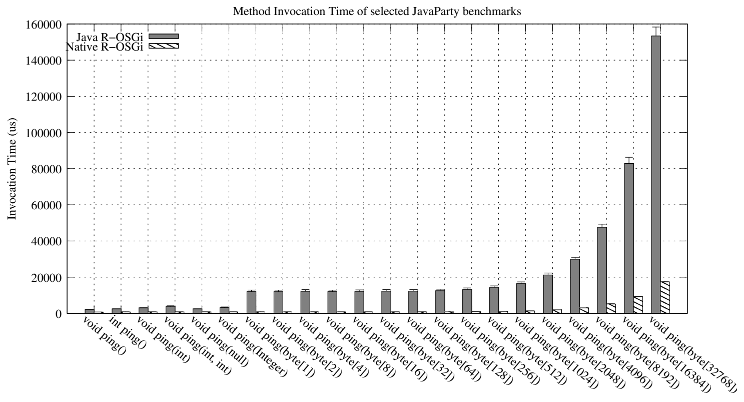

Fig. 6. Javaparty Benchmark on the Slug

Conceptually, this implementation is more similar to the implementation for CLDC than 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. We rather compose the services provided by the sensor node at build time, like the MIDlet for CLDC, 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 our (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 will be 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 peer 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 our *Serializer* component.

5 Supporting Streaming Data Services

R-OSGi was designed for remote method invocation which corresponds to a conventional request-response interaction. However, many small and ubiquitous devices only deliver or consume data streams, e.g., a sensor node that periodically reports the temperature. Also services offered by more powerful devices can require the exchange of streaming data, e.g., a CCTV camera.

Typically, exchange of streaming data in Java is 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. 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 by many ubiquitous devices. We present an extension to R-OGSi that transparently supports streams. We also discuss quality of service issues related to streaming data.

5.1 Transparent Streams on Java

To allow for transparent remote access to streams we make R-OSGi stream aware by intercepting the exchange of stream objects and introducing a separate mechanism for exchanging data remotely over streams.

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. There the handle is used to create a 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.

Calls to the stream proxy will be converted to R-OSGi stream request messages which will be parsed and result in respective calls to the actual stream. The data returned (in case of input streams) is sent back as R-OSGi stream result messages and returned to the read call to the stream proxy.

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 continuously transmit significant amounts of data compared to method invocation which occurs sporadically and usually with a small argument size.

5.2 Streams on non-OSGi Devices

Similarly to the implementation of OSGi-like services in devices without OSGi, we provide support for data streaming in such devices through the adapters.

CLDC already supports *InputStream* and *OutputStream* types. The R-OSGi adapter for CLDC simply replaces instances of streams with handles in the same way R-OSGi does. Incoming stream request messages result in calls to the actual stream and the data returned is sent back as stream result messages. The same applies for embedded Linux platforms with the only difference that the stream, like every other object, is implemented in JNI instead of Java.

For TinyOS, the *Service* interface is extended with two additional commands and two additional events for reading from and writing to streams. When a TinyOS service returns a serialized stream handle as result, it also passes a local handle to the R-OSGi adapter. This handle will be passed back to the service's commands for reading and writing when a stream request message arrives. The service then dispatches the request internally. Once data has become available (in case of input streams) or been processed (in case of output streams), the corresponding events are signaled and the result is passed as argument. The result is then sent to the service caller as response.

5.3 Quality of Service

The possibility to stream large amounts of data requires careful considerations of the possible impact on the whole system. R-OSGi maintains one communications channel to every remote host it is connected to. Through this channel all communication takes place in the form of messages. The data exchanged by streams is transmitted in stream request and stream result messages which are exchanged over the same communication channel over which all other messages like method invocation or lease renewal are exchanged.

To ensure correct and smooth operation of R-OSGi, messages of other types are prioritized over stream request and stream result messages to avoid clogging of the communication channel by a burst of stream messages. On the other hand, R-OSGi-aware applications can choose to prioritize traffic of specific streams over other streams or even over the other message types. This is useful for real-time streams like, e.g., audio or video streams, which are in general more time-critical than method invocations.

6 Discussion

OSGi simplifies constructing applications out of independent software modules. Moreover, it provides facilities for dynamically managing the insertion and withdrawal of software modules, keeping track of the dependencies between modules, enforcing dependencies, and interaction through loosely-coupled service interfaces. With the extensions to R-OSGi that we have just presented, we have a software fabric for the "Internet of Things" that gives us exactly the same features but for distributed applications residing in heterogeneous devices.

Having this support implies, for instance, that spontaneous interactions between two appliances become service invocations within a distributed OSGi framework. Switching one of the appliances off appears as a module withdrawal and the framework – the software fabric – handles the cleanup of all dependencies and informing users of this service. It is also possible to dynamically replace the implementation of a service, or even the device itself, without the caller being aware of the swap.

Service discovery can be made more precise and efficient by using the filters provided by OSGi. That way, the amount of information exchanged during a spontaneous interaction can be minimized, an important factor when communication is through a low bandwidth radio link. The same mechanism is helpful to prevent that, when several devices are present, all of them have to reply to indiscriminate broadcast looking for services.

In the software fabric we just described, all devices independently of language, platform, or size present a uniform frontend which makes developing applications far easier than when programmers have to cope with a wide range of technologies and paradigms. It is also important to note that this common frontend does not reduce the exchanges to the minimum common functionality of all devices involved. Finally, note that this common frontend and standardized interfaces apply not only to software but, as the example with the Bug illustrates, also to hardware.

We are not aware of any other solution of the many that have been proposed that is capable of bridging heterogeneity as well as the software fabric we propose. In particular, with the very low overhead that it introduces.

7 Related Work

The idea of constructing distributed systems out of cooperating services has been pioneered by the Jini [25, 26] project. In Jini, every Java object can be a service. The information about the services around is maintained by a central *Lookup Server* which has to be initially discovered by each peer in order to communicate. The later communication between services is point-to-point and does not require the lookup server any more. Jini does not pose restrictions on the protocol that this device to device communication follows. However, since Sun's standard implementation of Jini uses RMI for the communication between devices and the lookup server, most applications use RMI also for the device interactions. Although spontaneity was one of the main goals of Jini, the requirement of a lookup server restricts the applications of Jini to networks with some management. In contrast, the approach presented in this paper does not require a central component and also works in opportunistic networks with no management, such as ad-hoc networks. This is crucial for applications around the "Internet of Things", where interaction is often driven by the momentum of opportunity. Furthermore, Jini is tailored to TCP/IP networks and cannot be easily run over other transports.

Other projects that are discussing how to contruct applications from cooperating services and provide specifications for design and interaction of these are ODP [27], CORBA [28], and DCE [29].

For consumer devices, UPnP [30] is a standard for discovering and interacting with equipment. UPnP is able to discover new devices through the multicast protocol SSDP [31] and even includes an addressing schema derived from Zeroconf [32, 33] for networks where no infrastructure is available which assigns IP addresses. Devices and services are described by XML descriptors which can be used to select devices and services based on fine-granular properties. The communication between devices uses SOAP-based XML-RPC. Since SOAP can basically run over any transport, UPnP is technically not restricted to TCP/IP networks. However, the high verbosity of the XML involved requires networks with sufficient bandwidth and limits the feasibility for low-power communication. Furthermore, UPnP limits the arguments of calls to a small set of specified types. General type mappers for complex objects as known from web services are not supported.

Recently, the idea of *Service Oriented Device Architecture* (SODA [34]) has gained momentum. SODA tries to incorporate principles of SOA into the world of devices to facilitate their integration into enterprise systems. The work presented in this paper can be seen as an implementation of this idea. Instead of an enterprise bus based on web services, OSGi and R-OSGi are used for the communication between services on devices. Since the R-OSGi protocol is far more lightweight than web services, it appears to be more amenable to implement the "Internet of Things" in resource constrained devices.

8 Conclusions

In this paper we have outlined a software fabric for the "Internet of Things" that is based on the ideas borrowed from the field of modular software design. The next steps we intend to pursue include developing applications on this fabric to better test its properties, continue extending the fabric to encompass a wider range of devices and protocols, developing tools that will simplify the development of applications that federate collections of services provided by ubiquitous devices, and explore bridging of heterogeneous network architectures as well as multi-hop scenarios in general.

References

1. OSGi Alliance: OSGi Service Platform - Release 4. (2005)
2. Rellermeyer, J.S., Alonso, G., Roscoe, T.: R-OSGi: Distributed Applications Through Software Modularization. In: Proceedings of the ACM/IFIP/USENIX 8th International Middleware Conference. (2007)
3. Bluetooth Special Interest Group: Bluetooth. `http://www.bluetooth.com` (1998)
4. ZigBee Alliance: Zigbee. `http://www.zigbee.org` (2002)
5. IEEE: 802.15.4-2006 Part 15.4: Wireless MAC and PHY Specifications for Low Rate Wireless Personal Area Networks (WPANs). (2006)

6. Sun Microsystems, Inc.: J2ME Building Blocks for Mobile Devices: White Paper on KVM and the Connected, Limited Device Configuration (CLDC) (2000)
7. Hill, J., Szewczyk, R., Woo, A., Hollar, S., Culler, D.E., Pister, K.S.J.: System Architecture Directions for Networked Sensors. In: Architectural Support for Programming Languages and Operating Systems. (2000)
8. IBM: IBM Service Management Framework. `http://www.ibm.com/software/wireless/smf/` (2004)
9. Eclipse Foundation: Eclipse Equinox. `http://www.eclipse.org/equinox/` (2003)
10. Apache Foundation: Apache Felix. `http://felix.apache.org` (2007)
11. Rellermeyer, J.S., Alonso, G.: Concierge: A Service Platform for Resource-Constrained Devices. In: Proceedings of the 2007 ACM EuroSys Conference. (2007)
12. Makewave AB.: Knopflerfish OSGi. `http://www.knopflerfish.org` (2004)
13. Guttman, E., Perkins, C., Veizades, J.: RFC 2608: Service Location Protocol v2. IETF. (1999)
14. Guttman, E.: Service Location Protocol: Automatic Discovery of IP Network Services. IEEE Internet Computing **3**(4) (1999)
15. Zimmermann, H.: OSI Reference Model-The ISO Model of Architecture for Open Systems Interconnection. IEEE Transactions on Communications **28**(4) (1980)
16. Haumacher, B., Moschny, T., Philippsen, M.: The Javaparty Project. `http://www.ipd.uka.de/JavaParty` (2005)
17. Leach, P., Mealling, M., Salz, R.: A Universally Unique IDentifier (UUID) URN Namespace. RFC 4122 (2005)
18. Howes, T.: A String Representation of LDAP Search Filters. RFC 1960 (1996)
19. Moteiv: Tmote sky. `http://www.moteiv.com/` (2005)
20. Sun Microsystems: Java Mobile Information Device Profile. (1994)
21. Sun Microsystems: Java Native Interface. `http://java.sun.com/j2se/1.5.0/docs/guide/jni/index.html` (2004)
22. Lougher, R.: JamVM. `http://jamvm.sourceforge.net` (2003)
23. Gay, D., Levis, P., von Behren, R., Welsh, M., Brewer, E., Culler, D.: The nesC language: A holistic approach to networked embedded systems (2003)
24. Crossbow Technology: The TelosB Platform. `http://www.xbow.com/Products/productdetails.aspx?sid=252` (2004)
25. Sun Microsystems: Jini Specifications v2.1. (2005)
26. Waldo, J.: The Jini architecture for network-centric computing. Communications of the ACM **42**(7) (1999)
27. Linington, P.F.: Introduction to the open distributed processing basic reference model. In: Open Distributed Processing. (1991)
28. Object Management Group, Inc.: Common Object Request Broker Architecture: Core Specification (2004)
29. CORPORATE Open Software Foundation: Introduction to OSF DCE (rev. 1.0). Prentice-Hall, Inc., Upper Saddle River, NJ, USA (1992)
30. UPnP Forum: Universal Plug and Play Device Architecture. (2000)
31. Goland, Y.Y., Cai, T., Leach, P., Gu, Y., Albright, S.: Simple Service Discovery Protocol (Expired Internet Draft). IETF. (1999)
32. Cheshire, S., Aboba, B., Guttman, E.: RFC 3927: Dynamic Configuration of IPv4 Link-Local Addresses. IETF. (2005)
33. Guttman, E.: Autoconfiguration for IP Networking: Enabling Local Communication. Internet Computing, IEEE **5**(3) (2001)
34. de Deugd, S., Carroll, R., Kelly, K., Millett, B., Ricker, J.: Soda: Service oriented device architecture. IEEE Pervasive Computing **5**(3) (2006)