Implementation of D-Bus support for Jolie

May 22nd 2013

Spring semester 2013

IT University of Copenhagen

Authors: Niels Martin Søholm Jensen, nmar@itu.dk

and Jan Aagaard Meier, jmei@itu.dk

Supervisor: Marco Carbone, maca@itu.dk

Total pages (including front page and appendices): 71 pages

Number of characters (excluding appendices): 91.090 characters (~38 standard pages)

Preface

This report was produced as part of a bachelor's project running from January through May 2013 at the IT University of Copenhagen under supervision of Marco Carbone.
Many thanks goes to Fabrizio Montesi, for his great help in making us understand Jolie, and for contributing to feedback on this report.

Content

Introduction

Service-oriented computing is a software design methodology in which the system is composed of individual entities, called services. These services communicate using standardized protocols. Jolie is a service-oriented programming language, that abstracts away the implementation details of these protocols.

Service-oriented computing is used both in network as well as local service communication, but the currently supported protocols of Jolie are either designed for communication over a network or for communication between Jolie services in the same runtime. The network protocols incur unnecessary overhead when used for communication between local services.

D-Bus is a protocol and communication medium for local interprocess communication that supports both request-response and publish-subscribe messaging patterns and allows services to publish a description of their interfaces. Numerous projects, such as Skype, and the GNOME and KDE desktop environments, expose interfaces through D-Bus.

Problem

We propose to implement D-Bus support for Jolie, enabling services written in Jolie to communicate with each other and with any D-Bus compatible application. This should be done in an effective and correct way that is true to Jolie as well as D-Bus.

Background

In the following we outline the three overall entities which form the foundation of our project. Supporting service-oriented architectures is the key motivation of Jolie. Jolie is the language we are seeking to extend and an understanding of the most basic language constructs is needed to grasp exactly how our project extends the capabilities of that language. Finally, we elaborate on the origin and core features of D-Bus in order to clarify the need for a D-Bus Jolie extension.

Service-oriented architectures

Service-oriented architectures (SOA) involve software design where the product is represented as a composition of multiple individual services. Each service has a clearly defined and self-contained concern (Valipour et. al. 2009 p. 36). The service composition as a whole provides its functionality by communication between its services using standardized communication protocols. This makes interoperability a natural feature of the architecture and it promotes modular and extensible systems (ibid. p. 37). However, using standardized protocols introduces the task of implementing and understanding such protocols or at least a protocol-specific binding library for the programming language of choice. A service composition may even use multiple protocols internally or to communicate with external systems.

Jolie

Jolie (http://www.jolie-lang.org) is a service-oriented programming language, which abstracts away the implementation details of protocols. In Jolie, services are separated into behavior and deployment. The behavior describes the implementation of the service, in the form of statements and expressions forming the execution path of the application. The deployment describes the specific details of how the service communicates, among other things which protocol(s) to use (Montesi et. al. 2012, pp. 3-6). This separation makes it possible to develop a service without taking a specific communication protocol into account, as well as changing which protocol to be used without changing the actual interface and implementation of the service.

We will not elaborate much on the syntax details of behaviors because, in essence, they are not relevant to a communication extension - just like a thorough understanding of a communication extension is not needed to develop advanced behaviors. Think of them as implementations just like any other object-oriented language, such as Java.

Communication ports

In Jolie, the information that describes how to establish communication links between services in the SOA is called *Deployment*. Deployment details are syntactically described as communication ports using the keywords `inputPort` and `outputPort` which precedes a name and a block containing parameters about the deployment. Below, we have listed two examples of communication ports.

A service that uses TCP/IP sockets to *listen* on port 8000 might use a communication port like Figure 1:

```
inputPort MyService {
    Location: "socket://service.location.com:8000"
    Protocol: sodep
    Interfaces: TheServiceInterface
}
```

Figure 1 – An inputPort definition in Jolie

A service that wishes to *invoke* the service above might use a communication port like Figure 2:

```
outputPort TargetService {
    Location: "socket://service.location.com:8000"
    Protocol: sodep
    Interfaces: TheServiceInterface
}
```

Figure 2 – An outputPort definition in Jolie

In this example we look at two matching services written in Jolie, however this does not have to be the case. A Jolie program may communicate with any service in any language as long as they implement a supported protocol/location. The right output port or input port just has to be defined.

Location

A location is a syntax construct of Jolie that declares the communication medium which a communication port is to use when sending and receiving messages. Some examples of the currently supported media are: TCP/IP sockets, Java remote method invocation (RMI) and UNIX domain sockets. Communication media are implemented by Jolie contributors as self-contained Jolie extensions with a common abstract design. The runtime will know which extension to use via the prefix of the location string given in the port specification.

A location declaration for a D-Bus service might look like Figure 3:

```
Location: "dbus:/com.location.service:/service"
```

Figure 3 – A location declaration in Jolie

Protocol

A protocol is a syntax construct of Jolie that declares which rules and formats should be used to transmit messages on a communication port. Some examples of the currently supported protocols are: Hypertext Transfer Protocol (HTTP), Simple Object Access Protocol (SOAP) and JavaScript Object Notation Remote Procedure Call (JSON-RPC). Similar to a location extension it is possible to implement self-contained protocol extensions for Jolie. The protocol keyword is in some cases optional as some location extensions do not require a protocol extension to be declared.

A protocol declaration for HTTP looks like Figure 4:

```
Protocol: http
```

Figure 4 – A protocol declaration in Jolie

Interfaces

Interfaces describe the operations to be supported by a communication port. It provides type definition for arguments. It may contain both `OneWay` and `RequestReponse` operations, the distinction being whether the operation is expected to provide a return value or not:

```
interface OkularInterface {
        OneWay:
                goToPage ( long ),
                openDocument ( string )
        RequestResponse:
                currentPage ( void ) ( long ),
                currentDocument ( void ) ( string )
}
```

Figure 5 – Interface description of the PDF reader Okular in Jolie

Interfaces do not contain any protocol or location specific syntax and thus a single interface can be used across all protocols supported by Jolie.

Type system

The Jolie type system consists of simple data types, arrays and tree structures. An operation in Jolie can take only a single argument, so in order to pass several values one will have to define a tree structure holding the values. Each node of the tree structure can either be a simple type, an array, or a further nested data structure. Arrays in Jolie are allocated dynamically, and can hold any simple type or nested data structure. The root node of a data tree cannot be an array, but any child nodes are dynamically allocated arrays.

Concurrency

Jolie has several syntax constructs to declare concurrent execution. The behavior associated with an input port may declare how incoming requests are handled via the `execution` keyword. It describes whether the service should terminate after a single session, do concurrent sessions allowing requests to be handled simultaneously or do sequential sessions where requests are handled one at a time:

```
execution { single | concurrent | sequential }
```

Figure 6 – Declaring execution mode for input ports in Jolie

Statements in the behavior can be run in sequence, parallel or a combination. A semicolon is used to separate statements to be run in a sequence. A vertical bar is used to separate statements to be run in parallel:

```
{A ; B} | {C ; D}
```

Figure 7 – Executing Jolie statements in sequence and parallel

Statements may be local computation or requests to other services through output ports.

The D-Bus protocol

D-Bus is a messaging protocol for interprocess communication (IPC). In the beginning it was built around the goal of replacing Bonobo (based on CORBA) and DCOP which were the previous IPC systems of the popular desktop environments GNOME and KDE respectively (Love 2005). These predecessor systems were critiqued for being too closely tied to their associated desktop environments as well as being complex and performance heavy (Burton 2004). D-Bus was designed to be simple and efficient, and in comparison with the previous alternatives it delivers low-latency and low-overhead (Burton 2004). It was developed by members from the GNOME and KDE communities under the freedesktop.org initiative which to this day has formed a shared location for numerous open source projects focused on sharing technology across desktop environments (Rayhawk, H. et al. 2013).

D-Bus is available in all major Linux desktop distributions (Vervloesem 2010). A port for windows also exists, but the following only describes the UNIX reference implementation. In addition to the daemon (`dbus-daemon`) and the C binding (`libdbus`) the UNIX version also includes a tool called `qdbus`, which can be used to query D-Bus objects through the command line.

Communication is done through a *message bus* daemon that accepts connections from multiple processes, and forwards messages between them. Once a process is connected to the bus it can query it to discover what other processes are present on the bus, and what their interfaces are.

D-Bus exposes two busses, the system bus, and the session bus. The system bus is global, and might be used for sending signals such as “new hardware added” etc. The session bus is specific to each logged in user, and allows the user’s applications to communicate with each other.

D-Bus supports both request-response, where messages are forwarded from one application to another and publish-subscribe where application can subscribe to certain *signals* emitted by other applications. A request-response operation could be asking a PDF reader for its current page and getting a response. A publish-subscribe operation could be to subscribe to receiving pageChanged signals published by a PDF reader.

When applications connect to the bus they must first authenticate with the daemon and invoke the method `Hello`. Invoking this method assigns the application a unique connection name, a bus name. The application cannot begin communicating with other applications before it has been assigned a unique connection name. Henceforth we will refer to this as the “initialization of the connection” or “the initialization overhead”. The daemon is itself an application on the message bus, and the initialization call is invoked like a regular D-Bus method call. Once a connection has been initialized, it can request additional so-called *well known* bus names from the bus. The well known names take the same form as Java package names, containing sequences of character and numbers, separated by dots. Applications that wish to expose methods on the bus will often request a well-known name, in order to have a human-readable name that is the same each time the program is running. As an example, the PDF reader okular will request the name `org.kde.okular-` followed by a unique identifier, allowing multiple instances of okular to exist on the bus at the same time.

An application usually has a single connection to the bus. This connection can request several well known names, each of which can expose different interfaces. Method calls are not invoked directly on connections, but on *objects*, exposed by the connection. A connection can expose multiple objects, and each object can implement several interfaces. Objects can be thought of as analogous to an object in Java, in that they expose methods for remote connections to call. The diagram below shows how three applications might be connected to D-Bus:

- Okular, a PDF reader, has the unique name `:2.55`, and has additionally requested the well-known name `org.kde.okular-57`. It exposes two objects on the connection.
- A Jolie server application has the unique name `:1.33`, and has requested two additional well-known names. The application exposes a single object on the connection.

- A Jolie client application has the unique name `:1.42`. Since the client application only sends requests to others and does not expose any objects, it has not requested any well-known names.

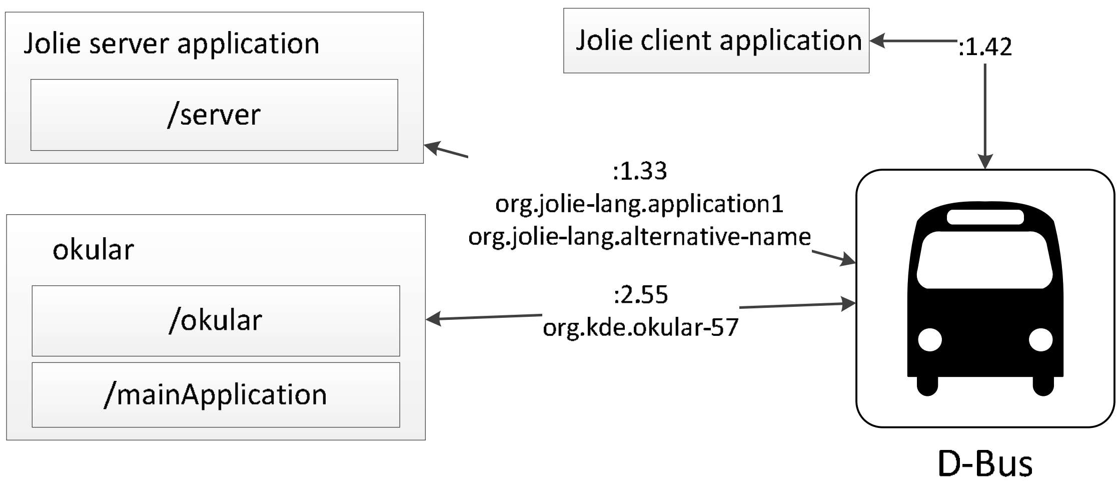


Figure 8 – Three applications communicating via D-Bus

Since D-Bus is intended for communication between processes located on the same machine it uses a local communication medium - namely UNIX domain sockets. An application using D-Bus passes messages to the D-Bus daemon, which takes care of passing the message on to the right recipient, which can either be specified by its unique name or any well known name it may have obtained.

Messages that are sent through the bus must follow the marshalling format specified by D-Bus, known as the *wire format*. The wire format is a binary serialization format (Pennington et al. 2013, Introduction) consisting of a message header and a message body. A binary format is chosen because it adds less overhead than for example XML (Pennington et. al. 2013, Introduction), which uses a lot of extra characters to keep track of the 'formatting' of the data and has to convert all data to and from strings when marshalling and demarshalling. The message header keeps track of the following:

- Recipient and sender of the message
- Name of the method to invoke (and optionally the interface the method was defined in)
- Length of the body

- Signature of the method
- Message serial (used by the sender to identify the response corresponding to a request.)

This header is followed by the body of the message, the method arguments or response values. Whereas in XML each argument to a method is part of an element identifying the name of the argument, arguments in the D-Bus wire format are not named - they are simply written to the body of the message in the order that the recipient expects them, and the order specified in the signature. Thus, the signature is needed when de-marshalling the data.

Introspection

Objects on D-Bus may expose their interface to callers by implementing the `org.freedesktop.DBus.Introspectable` interface, which contains an `Introspect` method. A response to this method must follow the Introspection Data Format (Pennington et. al. 2013, Introspection Data Format), which as an XML format for specifying the objects exposed by the connection, their interfaces, and their methods and arguments.

Implementing Introspection is not required, but a lot of services implement it. Introspection data is a form of “standardized documentation,” in that it allows the author of a service to tell other developers exactly what arguments are expected and what is returned. Most of the language binding libraries available for D-Bus can turn the introspection data into an interface declaration in the given language (Rayhawk, H. et al. 2013).

Problem analysis

With the basics of SOA, Jolie and D-Bus covered, we can start to think about challenges involved in putting those three together. The first question of how a Jolie location extension should communicate with D-Bus is addressed in the section below. After that discussion, we dive deeper into the technicalities, with discussions of how Jolie interfaces should be mapped to D-Bus, and how our extension should handle concurrency. Finally we discuss the possibilities of implementing signals, a feature which is not currently supported in Jolie.

Interacting with D-Bus

There are several options for creating the communication link between Jolie and D-Bus. One solution would be to communicate directly with D-Bus by simply creating messages conforming to the wire format and writing those the correct socket, but this is a very error-prone solution, and does not leverage the object-oriented nature of Java. D-Bus adds the abstractions of connections, objects and methods which makes calling methods seem more similar to Java, but if you had to construct the messages and serialize them yourself, much of this abstraction would not be put to use. Therefore we have decided to use the dbus-java binding, which does all the tedious work of connecting to and authenticating with the bus, wrapping messages to the wire format and writing them to the socket. It is possible to interact with the Java binding in two ways - through the high-level and through the low-level API. Both of these are described below, followed by a comparison of their advantages and disadvantages.

The D-Bus Java binding

High-level binding

The high-level binding is closest to calling regular methods in Java, in that it works with objects that you can call methods directly on. Say that we know of a connection that:

● has the well known name:	`org.jolie-lang.Test`
● exposes the object:	`/Test`
● implements the interface:	`org.freedesktop.DBus.Introspectable`.

Because the `Introspectable` interface is part of D-Bus, it is already defined in the Java binding and thus we can simply call the `getRemoteObject` method with that class, and invoke the `Introspect` method.

```
DBusConnection conn = DBusConnection.getConnection(DBusConnection.SESSION);

Introspectable intro = conn.getRemoteObject("org.jolie-lang.Test", "/Test",
Introspectable.class);
String data = intro.Introspect();
```

Figure 9 – Introspecting a D-Bus object through the high-level binding

If we wanted to invoke a remote object that implements another interface, that interface would first have to be defined in Java. This can be done by hand, or by using the `CreateInterface` command line tool included with the binding, provided that the target object is introspectable.

Exporting an object so it can be called on D-Bus is also done in an object-oriented fashion with the high-level binding. To expose a connection on:

● the well known name:	`org.jolie-lang.Hello`
● that exposes an object:	`HelloObject`
● implementing the interface:	`org.jolie.HelloInterface`
● with a single method:	`Hello`
● which takes and returns a:	`String`

the following code is required:

```
package org.jolie

public interface HelloInterface extends DBusInterface {
    public String Hello(String name);
}

public class Hello implements HelloInterface {
    public String Hello (String name ) {
        return "Hello " + name;
    }
11 }
12
13 DBusConnection conn = DBusConnection.getConnection(DBusConnection.SESSION);
14 conn.requestBusName("org.jolie-lang.Hello");
15
16 conn.exportObject("/HelloObject", new Hello());
```

Figure 10 – Exporting an object on D-Bus with the high-level binding

In the above example the interface `HelloInterface` is defined in the package `org.jolie`. The combination of the package name and the name of the interface gives the name of the interface on D-Bus. A connection to the bus is obtained at line 13, and the program then requests the bus name `org.jolie-lang.Hello`. A class is defined at lines 7 to 11, which implements the `HelloInterface`, and an instance of that class is exported on the object path `/HelloObject` at line 16.

The high-level binding automatically takes care of adding a proper `Introspect` method when exporting an object, so all objects exported by the high-level binding implement the `org.freedesktop.DBus.Introspectable` interface.

Low-level binding

The low-level binding does not provide abstractions in the form of classes and interfaces representing D-Bus objects and interfaces. Operations on the low-level are much closer to what is actually happening when a message is sent - you create a message and write it to a stream. The advantage compared to creating and writing messages directly to the socket is that messages - method calls, method returns, errors and signals - are represented as separate classes, providing some form of type safety, and that the `Message` class takes care of creating the right headers and converting the method body from an array of Java objects to the binary wire format. The example from the high-level binding (Figure 10) above of getting the introspection data from an object looks like this when using the low-level binding:

```
01  BusAddress address = new
02  BusAddress(System.getenv("DBUS_SESSION_BUS_ADDRESS"));
03  Transport transport = new Transport(address);
04
05  MethodCall call = new MethodCall("org.jolie-lang.Test", "/",
    "org.freedesktop.DBus.Introspectable", "Introspect", (byte) 0, "");
06  transport.mout.writeMessage(call);
07
08  MethodReturn ret = (MethodReturn) transport.min.readMessage();
09  String data = (String) ret.getParameters()[0];
```

Figure 11 - Introspecting a D-Bus object through the low-level binding

The code above is not much longer than for the high-level binding, but there are still some differences to note. One is that the example contains two type casts:

- At line 7, we typecast when the response to the Introspect call is read from the transport. This is needed because `readMessage` will read all types of messages sent to its transport, also method calls and signals, so therefore we have to cast this specific message to a `MethodReturn`. In practice we would also need to call `readMessage` in a loop and check the type and id of every message, because several other messages might arrive before the response that we are waiting for. In the high-level binding a method call is blocking, meaning we did not have to worry about matching up the right message.

- At line 8, we typecast when we collect the return value from the message. This is needed because `MethodReturn` is a type representing every type of return message possible. In the high-level binding the invoked object was known to implement the `Introspectable` interface and therefore its return type was also known. In the low-level binding the type must be obtained by knowing the return type of the method (in this case we know that `Introspect` returns a single argument, a `String`) or by looking at the signature of the method.

High-level vs. low-level binding

Since we have already dismissed the option of interacting directly with the bus, we are left with two options - the high-level or the low-level version of the dbus-java binding. The choice is basically between full type-safety on the one hand; and full control of the messages that are sent and their contents on the other. With the high-level binding we can get a Java object representing each remote object and get automatic introspection support when objects are exported to the bus, whereas with the low-level binding we have full control of what messages are created, their signatures and arguments, as well as when the messages are read and sent.

When we first looked at the bindings, we started out by investigating the high-level binding, simply because it is the best documented of the two. It quickly became clear however, that the way that the high-level binding handles invoking and exposing methods is very different from how Jolie expects a location extension to work.

When a remote method is invoked in Jolie the interpreter passes a representation of that message (its name and arguments) to the extension, which is expected to send the message, and then return. At some later point Jolie will then invoke another method on the extension, telling the extension to start looking for a response to the message. Since Jolie supports multiple communication methods and allows multiple code paths to be executed in parallel, it makes sense for Jolie itself to handle the scheduling of message reception, rather than handing over control to individual extensions. The high-level binding however dispatches the method call, and blocks until it receives a reply, as in the example above of calling `Introspect`.

All method calls invoked using the high-level binding, both synchronous and asynchronous, must be invoked on a Java object, representing the interface of the remote object. Our D-Bus location extension will have access to a Java representation of the Jolie interface, but the high-level binding needs an actual Java interface in order to get a remote object. In theory we could generate a Java interface .class file from the Jolie interface and then load that into the classloader when the extension is initialized, but having to generate extra Java code in order to be able to use the binding would complicate things unnecessarily.

With the limitations of the high-level binding we started to look at the low-level. The low-level binding has a lower level of abstraction, working with messages that are written to a stream.

This level of abstraction actually fits well with how Jolie handles messages, as described above. When Jolie asks our extension to send a message, we can convert it to a `MethodCall`, and send that message to the bus. After the message is sent, the low-level binding does not care more about the message, there is no way to wait for a reply to the message. When Jolie at some later point tells our extension that it should receive a reply for the given message, it will simply start looking in the input stream for a message matching the one that was sent previously. All messages sent to a connection are put into the same input stream, so using the low-level binding adds some complexity in having to match up received messages with sent ones, but it means that Jolie can receive and process method calls and replies when it wants to.

After having looked at the two bindings, the biggest difference lies in when the code associated with a method call or a method return is executed. When an object is exported onto D-Bus through the high-level binding, that object must implement the actual code which is to be executed upon a method invocation over D-Bus. When a method exposed through the low-level binding is invoked, a `MethodCall` which represents the call and its arguments can be read from the input stream, but the low-level binding does not care when the code associated with that call is executed. This behavior is very much in line with how Jolie works with location extensions - Jolie only needs the extension to look for messages and alert it when a message has been received. At some later point, Jolie will then ask the extension to handle the message, and convert its arguments into something that a Jolie program can understand.

The combination of the facts that the high-level binding adds unnecessary abstractions in the form of remote objects which must implement a Java interface, and that the low-level binding works very similar to how Jolie expects location extensions to work means that we have chosen the low-level binding for this project.

Marshalling and type systems

Marshalling is the process of converting data structures in Jolie to the equivalent data structures in D-Bus, and the other way around. Both Jolie and D-Bus have extensive type systems, supporting multiple types, such as arrays and maps. However, there are also some differences between the two type systems that have to be mapped to each other. This section will concentrate around two parts of marshalling. The first section concentrates on how to map the structure of methods and method signatures between D-Bus and Jolie, and the other section will talk about converting from one type system to the other, and obtaining the right D-Bus signature for a set of Jolie types.

Methods and method signatures

To talk about the difference in method signatures between D-Bus and Jolie, we use a simple Jolie type and interface written in Jolie as our vantage point. The interface contains a single exponentiate method which takes two integer values, the base and the exponent. It returns the exponentiation in the form of an integer:

```
type ExponentiationRequest:void {
    .base: int
    .exponent: int
}

interface Math {
    RequestResponse:
        exponentiate ( ExponentiationRequest ) ( int )
}
```

Figure 12 – Jolie interface and type declaration for the exponentiate method

Methods in Jolie only take a single argument, so in order to accept both an exponent and a base, we must define a type to hold the two in a tree structure. The first intuition when translating this Jolie interface to a D-Bus interface might be to simply accept that a Jolie service will always expose an interface with zero or one argument for each method. This would mean that the D-Bus introspection data for the exponentiate method would look like this:

```xml
<method name="exponentiate">
    <arg name="ExponentiationRequest" type="a{si}" direction="in"/>
    <arg type="i" direction="out"/>
</method>
```

Figure 13 – D-Bus introspection data for the exponentiate operation

The data tree in Jolie is the equivalent of a map - it maps strings to values. In this case it maps the keys base and `exponent` to two different integers. This is expressed in D-Bus introspection as an array of map entries, each entry mapping a string to an integer. In Java terms the method takes a `Map<String, Integer>`.

Converting from Jolie to D-Bus we have lost a great deal of information. D-Bus does not allow arg nodes in introspection data to contain subnodes, so there is no way to show the caller what the keys of the maps should be. One could argue that it would be natural to assume that a method for exponentiation calls its arguments base and exponent, but the arguments might as well have been called index and power, and there is no way to tell the caller that. This means that the caller would have to refer to external documentation to know the names of the arguments. A key intention of SOA is to make services self-contained entities that can easily be called by other services, even though they are written in other programming languages. The interface above however is very closely tied to Jolie. In Jolie it is easy to instantiate a new tree structure when calling the method, but other languages would first have to instantiate a new `Map<String, Integer>`, and add the key/value pairs.

An approach that would preserve more of the information from the Jolie interface would be to take advantage of the fact that methods in D-Bus can take several arguments, and that each argument can be named. If each of the subnodes of the `ExponentiationRequest` type were regarded as a separate method argument, the introspection data would look like this:

```xml
<method name="exponentiate">
    <arg name="exponent" type="i" direction="in"/>
    <arg name="base" type="i" direction="in"/>
    <arg type="i" direction="out"/>
</method>
```

Figure 14 – Alternative D-Bus introspection data for the exponentiate operation

Now suddenly the caller knows, by only looking at the D-Bus introspection data, which arguments this method takes, and the signature of the method is more akin to a D-Bus interface and less a Jolie interface. Even though Jolie and D-Bus have two different approaches to how to specify their method signatures, they still achieve the same goal - they tell the invoker about the type and the name of the arguments that are expected. Therefore we feel that the second approach is superior, since it manages to convey the same amount of information in D-Bus as is found in the Jolie program. It only slightly changes how the type system of Jolie is interpreted - the outermost level of a type tree is the method arguments, while all its subtrees are regular maps.

With the mapping of method signatures from Jolie to D-Bus sorted out, one might think that mapping the other way around, calling methods in D-Bus from Jolie would be just as easy. That is not always the case however, since the D-Bus type system is not as restrictive as that of Jolie. One problem is that the name attribute of method arguments is optional in D-Bus. This means that if someone else implemented the exponentiate method from before, it might very well look like this:

```xml
<method name="exponentiate">
    <arg type="i" direction="in"/>
    <arg type="i" direction="in"/>
    <arg type="i" direction="out"/>
</method>
```

Figure 15 - D-Bus introspection data for the exponentiate operation without argument names

This is not uncommon in D-Bus, the below example is taken from the introspection data of the official D-Bus daemon (`org.freedesktop.DBus /`):

```xml
<method name="StartServiceByName">
    <arg direction="in" type="s"/>
    <arg direction="in" type="u"/>
    <arg direction="out" type="u"/>
</method>
```

Figure 16 – Introspection data from the official D-Bus daemon

For the above method, `StartServiceByName`, we could try to inspect the type of the arguments, but with the less elaborative interface there is no way to know which branch of a

Jolie tree structure maps to what argument since both have the same type. Furthermore, it is not even required for services on D-Bus to implement the `Introspectable` interface, which means that we might also be faced with no information about the method at all, not even about the types of or the number of arguments.

Tree-nodes in Jolie must have a name, so even though they do not have a name in D-Bus we must come up with a naming convention to use in Jolie for methods in which the arguments do not have names. One solution would be to make it convention that the order that arguments are defined in the Jolie type should be the same as they are expected in D-Bus, such that the type below on the left would mean `exponentiate(base, exponent)`, while the one on the right would mean `exponentiate(exponent, base)`.

```
type ExponentiationRequest:void {
    .base: int
    .exponent: int
}
```

```
type ExponentiationRequest:void {
    .exponent: int
    .base: int
}
```

Figure 17 – Two Jolie type declarations for ExponentiationRequest

However, it is not possible for Jolie extensions to know the original order of the arguments in the definition of the type, only the names of the arguments, sorted alphabetically. Relying on the order of the arguments would also make the extension more error prone. Someone might decide to change the arguments into alphabetical order, suddenly completely changing the interface of the program in D-Bus without any noticeable difference in Jolie because argument order has no significance in Jolie.

Since the nodes must have a name, and we cannot rely on their order, we must make up a naming convention that allows the developer to easily show the argument order in Jolie if the arguments aren't named in the D-Bus introspection data, or if the introspection data is not present. We have decided to make the convention that nodes must be named arg*n*, where *n* is the index of the argument in the method call. If we know (from some other form of documentation) that the exponentiate method has the form `exponentiate(exponent, base)`, the Jolie type would look like on the type on the right:

```
<method name="exponentiate">
    <arg type="i" direction="in"/>
    <arg type="i" direction="in"/>
    <arg type="i" direction="out"/>
</method>
```

```
type ExponentiationRequest:void {
    .arg0: int // exponent
    .arg1: int // base
}
```

Figure 18 – D-Bus and Jolie declarations for the exponentiate operation, without argument names

Having to annotate each node with a comment in order to know what it actually contains is not ideal in Jolie terms, but we think it is the best compromise for mapping a D-Bus type that does not name its arguments to a type in Jolie.

Introspection

As we have seen above, the interface in Jolie becomes much more readable, if it can be modeled from proper D-Bus introspection data. Since we rely heavily on introspection data from other programs, it makes sense to also do an effort to provide proper introspection data when exposing a service in Jolie. Luckily the mapping from Jolie to D-Bus is relatively easy, because as previously mentioned; the Jolie type system is more restrictive than that of D-Bus. Subnodes must always have a name in Jolie which means that we can always specify the names of the arguments in the introspection data. How actual data types are mapped between Jolie and D-Bus is described below.

Mapping the type systems

The type systems of both D-Bus and Jolie are split into basic types and container types. We discuss the two separately below.

The basic types in D-Bus and Jolie have a pretty clear one-to-one mapping. One exception is that D-Bus supports both signed and unsigned integer types, whereas Jolie only supports signed integers. This is not a problem when exposing a Jolie interface on D-Bus, where we will simply expose an interface that only expects and returns signed integer types. The problem arises when we want to map a method in D-Bus that takes an unsigned integer to an operation in Jolie. As an example, we can use the goToPage method of the PDF reader Okular, which expects an unsigned 32 bit integer (type code “u”):

```
<method name="goToPage">
    <arg name="page" type="u" direction="in"/>
</method>
```

Figure 19 – Introspection data for the Okular goToPage operation

There is no way to specify an unsigned integer type in Jolie, and calling the method with a signature of a signed integer or a signed long will not work - both give the error message 'No such method'. Since we cannot show in Jolie that the page argument must be unsigned, we must obtain this information some other way. Luckily, Okular is introspectable, which means we can query it to get the correct signature. Whenever a remote object on D-Bus is introspectable, querying it to get its signature should be preferred over generating the signatures ourselves, because it eliminates the need to do reflection on the passed arguments and because the type will always be as precise or more precise (we will know whether the method expects a signed or an unsigned integer, which we cannot know simply by looking at the values passed from Jolie).

If the remote object that we are invoking a method on is not introspectable however, there is no way to know that signature in advance and thus we will need to generate it from the passed arguments.

D-Bus and Jolie support container types - types that hold other types. The table below shows the container types in the two languages, and their special features. The third column lists the compromises that we have taken to map the two types to each other.

D-Bus	**Jolie**	**Compromises**
Dictionary A dictionary in D-Bus is a mapping between a simple type and any type	**Tree** A tree structure with strings as keys and any type as value	In D-Bus the key might be of any simple type meaning it can be integers, strings, booleans etc. In Jolie the key must be a string, which means that it will not be possible to use a number type as a key in Jolie.
Array	**Array**	*None*

Struct A struct is a structure containing other types. Unlike structs in languages like C, the members of a D-Bus struct do not have any name, instead they are identified by their position in the struct.	***None*** Jolie does not have any structure resembling a struct	Jolie does not support a collection of types without any names. We might have made it convention that the members of a struct in Jolie were called `member0`, `member1` etc. in Jolie, much in line with what we have decided to do for methods that do not give names to their arguments. The difference between the two is however that method arguments will often have names, and therefore we only had to come up with a solution to support the rare cases where arguments are not named. If we were to do the same for structs we would change the meaning of the struct type in D-Bus by always adding member names. Furthermore, there would be no natural way to distinguish between structs and trees in Jolie if the members of both have names. For these reasons we have decided not to support structs in this implementation
Variant A variant can have any type, it simply tells that the type is specified later, in the body of the message, right before the	**(undefined)** Used in Jolie to specify that the subnode of a tree can have any type **Multi-typed arrays**	When defining a type in Jolie, one might simply say that it is undefined, meaning that the type can hold anything. We can map this behavior with Variant, which is the D-Bus equivalent of Java's `Object`.

actual value.	In Jolie it is possible to write `a[0] = 0; a[1] = "a"`, creating an array with multiple types. One cannot map this structure with a Jolie type, other than saying that the type is undefined	Variant can also be used for arrays that contain several different types, similar to an `Object[]` in Java.

Table 1 – Mapping of D-Bus and Jolie container types

Mapping interface names

It is possible, but not required, to specify the interface of a method when invoking it through D-Bus. We have decided to omit the interface name, since there is no way for a Jolie program to specify the proper name for a D-Bus interface. This is because interface names in D-Bus must contain at least one dot (Pennington et. al. 2013, Interface names), which is not allowed in Jolie. Interface names are only used when resolving ambiguities, i.e. when an object exposes several methods with the exact same signature. Since having multiple methods with the same signature on the same object is not allowed in most languages (including Java and Jolie), we believe that this ambiguity will only arise very seldom.

Concurrency and connection persistence

Due to Jolie's syntactically integrated support for concurrency and its highly distributed nature there can be no doubt that attention has to be paid to the thread-safety of a location extension. On top of that, concurrency support offers better utilization of modern hardware for some applications. In this chapter we discuss how we might gain performance through connection sharing and/or reuse. Please refer to the benchmark chapter for concrete estimates on the significance of these performance gains.

The D-Bus Java binding does not offer a low-level implementation that is thread-safe. As described in the background it is just an input stream and an output stream. The binding documentation (Johnson 2011 p. 19) simply states, without further elaboration, that reading and writing is not guaranteed to be thread-safe.

The initialization overhead of registering a new connection on the D-Bus daemon means that every time a channel is opened, an initial request will have to be made before any actual requests can be made. Without reuse or sharing of channels this will add an overhead to every single request, as a Jolie request would, in essence, always be two D-Bus requests instead of one.

Jolie supports abstract reuse of communication channels as well as parallel usage of a single channel, provided that the thread-safety of the channel is stated to be true. In this way, reuse can be leveraged without much effort with regards to Jolie and the Java binding. However, parallel usage will require implementation of a thread-safe wrapping as part of our D-Bus Jolie location extension.

Solutions

We have considered a number of ways to support concurrency and channel persistence. Even though Jolie offers support for parallel channel usage it does not just happen automatically and without any drawbacks. The channel has to be made thread-safe and this should be done in an effective way. On top of that, there is still an overall interface and execution flow which has to be followed.

Mutual exclusion

A simple way of ensuring thread-safety is to control access to the shared resource by using the Java `synchronized` keyword. This will ensure mutual exclusion with regards to the exact code block that interacts with the D-Bus transport.

However in the case of output ports it is not just that simple. Request and response handling is handled by two different method calls, potentially in two different threads. Jolie expects the send operation to terminate as soon as the message has been sent. A receive request for the response will be made at some point and the channel should be able to pair it with the original request. This complicates parallel usage of a single communication channel.

One way of solving this parallel request/response pairing issue is to use a thread-safe input buffer such that receiver threads keep an eye on the input buffer while attempting to receive their respective responses. This seems harmless at first but as the number of concurrent requests increases the amount of CPU time spent on context shifting increases and ultimately the solution is less scalable.

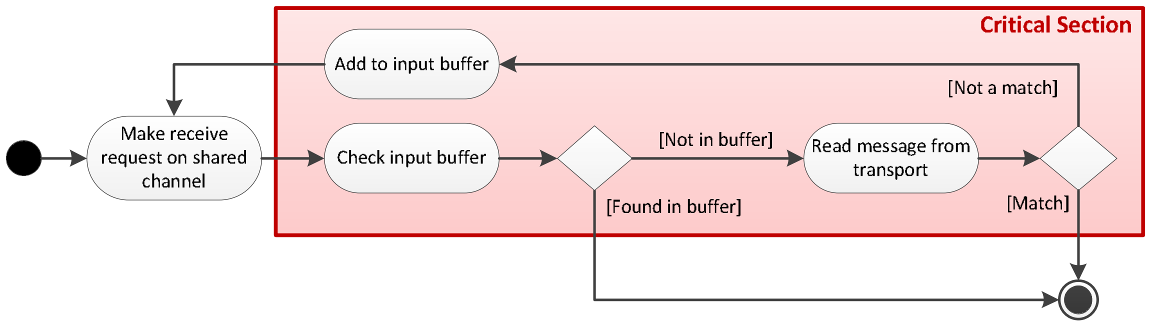


Figure 20 – Receiving messages through mutual exclusion

Deferred callbacks

An alternative that may provide more scalability is to maintain a callback map of functions which provide the execution paths to be resumed when the responses arrive.

This solution would remove the need for mutual exclusion as request threads would never attempt to interact with the non-thread-safe transport and thus context shifting would be kept to a minimum. Instead a single listener thread would continue to listen when at least one request is in the callback map. All the request threads would terminate and no receiver threads would be created until a matching response had been received by the listener thread.

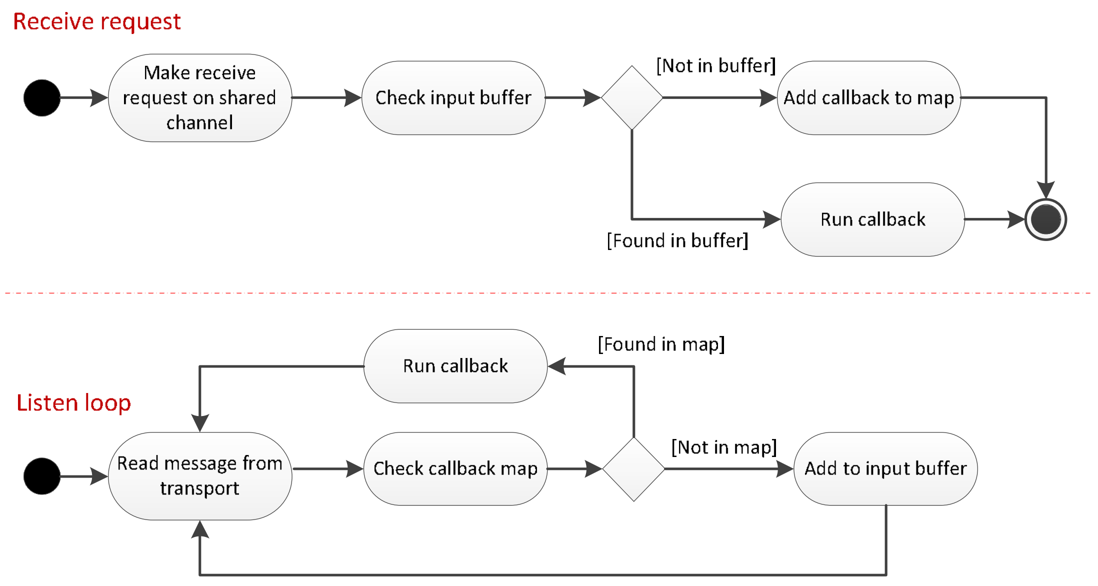


Figure 21 – Receiving messages through deferred callbacks

A drawback of this solution is that it is not well supported by Java since method-pointers do not exist. It is entirely possible to do it through a dedicated callback interface or by simply using the `Runnable` interface but either way it is not elegant. On top of that it does not align well with how Jolie expects a channel to work. Jolie's communication core will create a thread and call receive and thus it is not a responsibility of the channel to handle this.

No thread-safety

Provided that the initialization overhead mentioned above is sufficiently insignificant, a solution without thread-safety may be competitive. The gain would depend on the level of concurrency in the Jolie application being executed and the size of its messages.

It is a more simple solution which has the bonus of being easier to reason about and guarantee correctness for. It simply involves creating a channel for every request. This eliminates the need to pair requests and responses as the channel will only receive a single response - the one associated with its single request.

Threads will never be blocked by mutual exclusion, but idling channels waiting for their response may pile up causing an extra memory usage. And we mustn't forget that it retains any initialization overhead we could have saved by sharing the channels.

Channel persistence

Connection persistence is a way to save some of the initialization costs, even if an application does not involve high concurrency. This is accomplished by saving and reusing channels for multiple requests.

With channel persistence a Jolie application execution involving multiple sequential requests and no concurrent requests will only contain a single channel initialization. This may offer a significant performance benefit if the application produces many tiny sequential requests in quick succession.

Signals

In addition to request-response operations, D-Bus also supports publish-subscribe operations, called signals. This is a way for one application to send a message to multiple recipients, who have subscribed to this certain type of messages.

Sending and listening for signals is fully supported by the dbus-java binding but there are some complications as to how to specify in a Jolie program that you wish to send signals. This is because all other location extensions in Jolie communicate in a request-response scheme - either you are sending a request and (optionally) expecting a response; or you are expecting a request and (optionally) sending a response.
With signals we would be introducing a new scheme - publishers can send out signals, and subscribers can subscribe some code to be executed each time a publisher sends the signal in question.

A simple way of implementing signals could be to simply extend the interface in Jolie with another construct to represent signals, much similar to how signals are defined as part of the interface in D-Bus:

```
interface I {
    RequestResponse:
        ...
    OneWay:
        ...
    PublishSubscribe:
        ...
}
```

Figure 22 – a Jolie interface describing a PublishSubscribe (signal) operation

While this may be a good syntax for Jolie programs running D-Bus, it would create some complications for programs using other location extensions. One of the key features of Jolie is to abstract away the implementation details of various communication protocols, such that one might simply change the location string of the port from using one location extension to another. Other extensions would still work if they chose to simply ignore the `PublishSubscribe` part of the interface, but it would not be very clear to the Jolie developer, especially if he weren't familiar with D-Bus

Since publish-subscribe is a D-Bus specific feature, it would make more sense to have the user explicitly specify that they want to use a D-Bus features, instead of having a part of the interface that is only used by some location extensions. Using signals would then require the user to import some specific D-Bus interface into his program, much in the style of how console is used:

```
include "dbus.iol"

main {
    args.connectionName = ...
    args.objectPath = ...
    args.signalName = ...
    args.value = ...

    emitSignal@DBus(args)
}
```

Figure 23 – suggested Jolie code to emit a signal

The above shows an example of what the Jolie syntax for emitting a signal through D-Bus might look like.

The other way around, registering interest in a certain signal is not so straightforward, however, since it requires some way to specify a piece of code that should be executed when the specific signal is emitted. It is possible to define several procedures in Jolie using the `define` keyword, but pointers to procedures or executable code is not supported so we still need a way for the user to point to the code that should be executed when the signal is received. What we want to be able to do is something along the lines of:

```
01  include "dbus.iol"
02
03  define somesignalHandler {
04      // Code to handle signal here
05  }
06
07  main {
08      args.connectionName = ...
09      args.objectPath = ...
10      args.signalName = "somesignal"
11      args.handler = somesignalHandler
12
13      registerHandler@DBus(args)
14  }
```

Figure 24 – suggested Jolie code to attach a signal handler

In the code above, we would like some way to refer to the procedure in `somesignalHandler` at line 3, but that is not possible currently in Jolie. Furthermore, procedures in Jolie are not invoked with any arguments, and do not possess a local variable scope, meaning there would be no way to pass the arguments of the signal to the procedure.

Adding signals to the Jolie implementation of D-Bus would be very beneficial since it is one of the large selling points behind D-Bus, but it is not possible within the scope of this project. No matter how signal were implemented, they would require some form of extension of the core Jolie classes / syntax. D-Bus support for request-response and one-way operations only requires creating a new location extension, because changing of protocols is well supported in

Jolie, and Jolie has an architecture that makes it natural to create a new extension. Signals are not part of what a location extension normally supports however, so implementing them would extend the scope of the project from creating an extension, to making fundamental changes to the Jolie language. Therefore we have decided not to implement D-Bus signals in this project. For further pointers on how signals might be implemented, refer to the chapter on further work.

Implementation

This chapter covers the details behind our D-Bus location extension. After a description of the overall implementation details, we delve into some of the issues described in the problem analysis, describing how the solutions were implemented.

Overview

Our D-Bus extension for Jolie (framed in red in Figure 25) is written entirely in java, with tests and examples written partly in Java and partly in Jolie. The entire message flow involves 4 different contributors and 3 different languages:

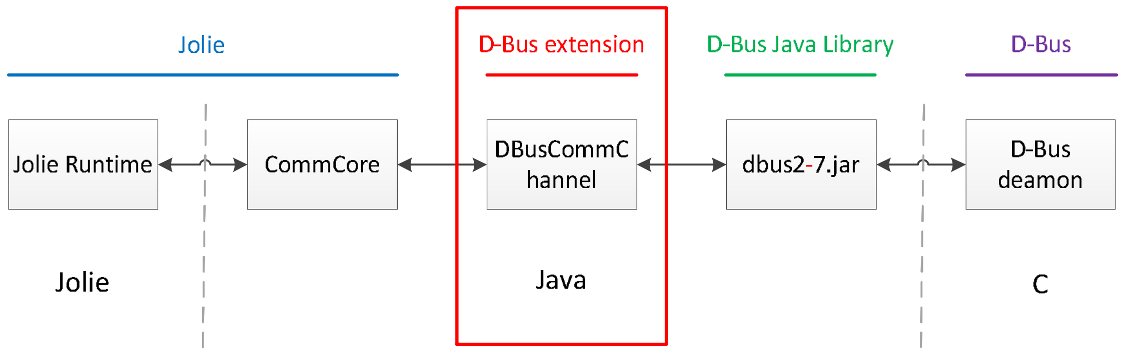


Figure 25 – Message flow from Jolie to D-Bus

Jolie Runtime refers to Jolie code being run by the Jolie interpreter and ultimately by the Java Virtual Machine.

CommCore is the internal module of the Jolie language implementation which handles all ingoing and outgoing messages. It produces Java objects of Jolie's own `CommMessage` class and hands them to the specified Jolie location extension.

DBusCommChannel is the primary class of our extension. It is the connection between `CommCore` and the D-Bus Java Library. It enables `CommCore` to send and receive messages in an abstract way that is identical to that of all other Jolie location extensions. See the following sections for more details about this and rest of our D-Bus extension.

dbus2-7.jar is the library that provides an API for Java to interact with the D-Bus daemon. We only use the low-level part of the library. We use some of the marshalling utilities provided by the library. See the background chapter for more details.

D-Bus daemon is a background service written in C and is included with many popular Linux distributions. It enables discovery of running D-Bus services and routing of messages between them. See the background chapter for more details.

Output

The full execution flow of an outgoing message is illustrated on the sequence diagram below. The red dashed lines indicate a separation in time and runtime. In other words, the individual parts are separated by an unknown amount of time and are likely to run in separate threads. We elaborate on each step using a numerical list corresponding to right-hand side annotations on the diagram in Figure 26:

1 A `CommChannelFactory` is created for each output port during initialization of a Jolie program. The concrete type of the factory is determined by the location string of the port.

2 When the interpreter encounters a statement that utilizes a given output port it will call the associated factory which returns a `CommChannel` instance.

3 A `CommMessage` representing the request is passed to the `send` method of this channel. The entire execution flow up until this point utilizes abstract types of Jolie and is thus completely identical no matter the communication medium.

4 `DBusCommChannel` marshals and sends the message through our `DBusMarshalling` API and sends it to the bus. It is asynchronous in the sense that the `CommChannel` won't block and wait for a response. It will however, do marshalling in the same thread and return when the message has been sent on its way.

5 The reference to the channel is maintained and a separate receiver thread will call the `recvResponseFor` method on `DBusCommChannel` which listens for a response and blocks till a matching response is received.

6 Upon retrieval the message will be marshalled to Jolie's `CommMessage` type and returned to `CommCore`.

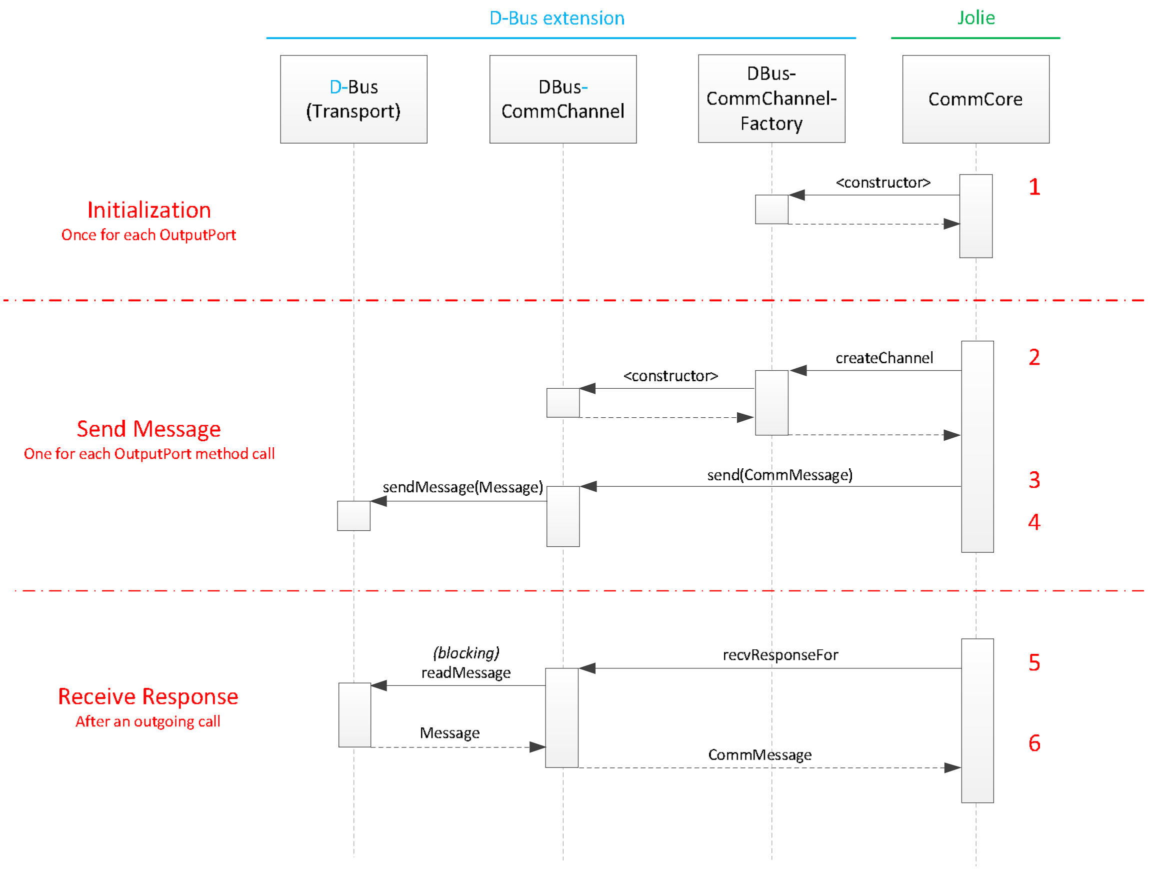


Figure 26 – Output flow

Input

Upon initialization of a Jolie program there will be created a location type specific `CommListener` for each input port declared in that program. Listeners implement `Runnable` and are in essence separate threads. A listener has a reference to Jolie's internal `CommCore` class and will schedule incoming messages for receival and execution. The listen loop and execution flow spawned by it is illustrated below using a sequence diagram (Figure 27) and an elaborative list.

1 `DBusCommListener` has a single `DBusCommChannel` which it uses in an infinite listen loop by invoking the blocking `listen` method of the channel continuously.

2 Every time the loop encounters an incoming message it will schedule it in `CommCore` and continue to listen. When a message is scheduled in `CommCore` a reference to the associated `DBusCommChannel` that heard the message will be passed along.

3 In a separate thread `CommCore` will call the receive method on `DBusCommChannel` which will marshal the D-Bus message and return it as a `CommMessage`.

4 The `CommMessage` is passed to Jolie's runtime which executes the relevant Jolie code.

5 Eventually the runtime returns and if the message is not one-way it will provide a return value in the form of a `CommMessage` which is marshalled and sent using `DBusCommChannel`.

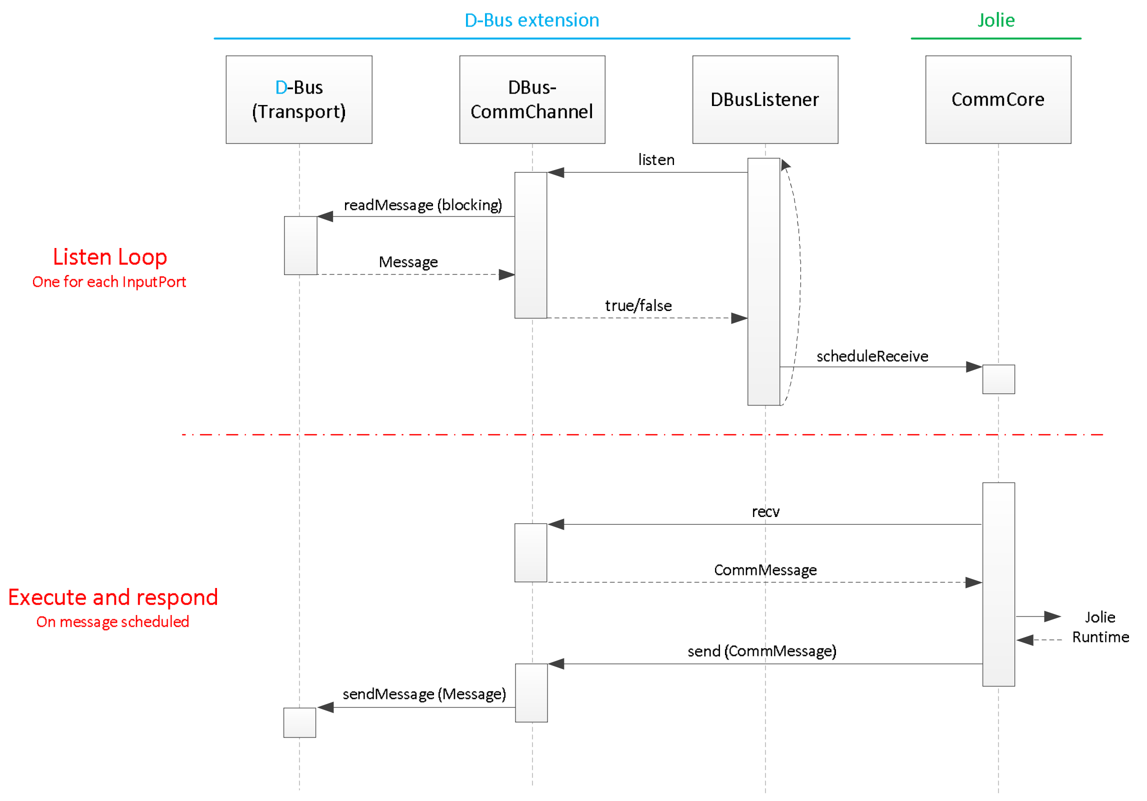


Figure 27 – Input flow

Marshalling

Marshalling of data happens in two places: when converting a Jolie `CommMessage` to D-Bus, and when converting D-Bus `MethodCalls` and `MethodReturns` to a `CommMessage`.

Marshalling of outgoing messages

When a message is sent from Jolie, two things must happen in the marshalling implementation: the Jolie `CommMessage` must be converted to an array of objects that can be passed as arguments to a D-Bus `Message`, and a signature matching the arguments must be obtained.

Introspection data can be useful for both conversion of arguments and obtaining signatures. Whether we have introspection data depends on the message type and target remote object:

- If the message is a method return, the introspection data has already been generated for the server's own Jolie interface during initialization of the channel.

- If the message is a method call and the remote object is introspectable, we have already obtained the signature via an introspect call during initialization of the channel.

For more details on how introspection data is obtained see the introspection section below.

How a signature is obtained depends on whether we have introspection data. If we have introspection data we use the signature from there, which has two advantages - first of all it is slightly faster, because it avoids looking through the arguments passed to the method and doing reflection to find their signature[1]. Secondly, we also obtain a more accurate signature, because not all of the types available on D-Bus are expressible in Jolie, as explained in the problem analysis section on marshalling. If the remote object is not introspectable, we generate a signature by collecting the types of the values, while we are converting them from Jolie to D-Bus.

The method used to convert outgoing messages is called `valueToDBus`, and has two overloads. One takes a value and a `StringBuilder` which can hold the signature string, and is used when no introspection data is present. The other takes a value and a string array containing the names of the arguments that the method expects, and should be used when we have introspection data. Both methods return an array of objects which is the arguments to the D-Bus message. The flow of the two methods are very alike, therefore they are depicted in a single diagram below (Figure 28). A bigger version can be seen in Appendix B:

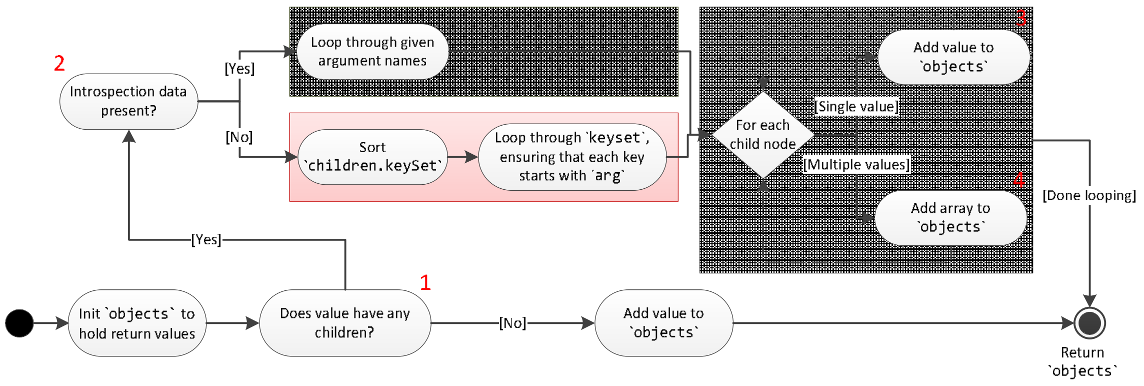


Figure 28 – Marshalling flow

[1] For measurements on the speed of marshalling with and without reflection, refer to the validation chapter on benchmarks

Both methods start out by initializing a list to hold the return values, and then checking whether the passed value is a container type or simple type, i.e. whether it has any children (1). If no; we just take that simple value and return it as the only value of the return value list. If the value is a container type, we know that the method expects several arguments, as discussed in the problem analysis on marshalling. When this is the case, we need to know in which order the subnodes of the tree structure should be marshalled, in order to match the expected argument order of the method. How we do this depends on whether we have introspection data (2). If we have introspection data (green) for the method, we loop through the given method names and extract the subnodes in that order. If we do not have any introspection data (red), we fall back to the assumption that all arguments must be named `arg0`, `arg1` etc.

For each child node (grey) we check whether the child contains only a single value, in which case we add that directly to the return values (3), or if it contains multiple values, in which case we add those as an array (4). Since arrays in Jolie are dynamically allocated, child nodes will always contain an `ArrayList` even though they only contain a single value.
When no introspection data is found, we also collect the types of each method argument while we are collecting the values from the Jolie `CommMessage`. We then use a class called `Marshalling` from the dbus-java binding to convert this list of types into a signature string.

Marshalling of incoming messages

When marshalling an incoming message, we use the same method for instances with and without introspection data. This is because the information we need, the signature string, will always be present when we receive a remote method call or a method return, as per the D-Bus specification. We use the `Marshalling` class from the dbus-java binding to convert the signature string to an array of Java types. If the signature is only a single type, and not a list we put the value of that type directly into the root of a Jolie value and return that. Otherwise we go through each of the types, and add each of their values as a child to a Jolie value, using the passed `argNames` if they are present, or using `arg0`, `arg1` etc. if they are not. The reason we do not add the argument directly to the root of the Jolie value if it is a list is, that the root of a Jolie value can only contain a single value, not an array. Therefore we need to create a subnode for this special case, even though it results in a tree with only a single subnode.

Introspection

Introspection data is created for each input port in Jolie. The introspection string is created and cached as soon as the port is initialized, so it is ready when a caller requests it. The introspection string is created by the `DBusIntrospector` class, which is responsible both for parsing introspection strings from remote objects and creating the introspection string for Jolie programs. The introspector gets passed the Jolie interface, and creates an XML introspection string based on that. The introspection data must contain an `interface` node with a name, but it is not possible to get the name that was declared for the interface of a port in Jolie. This is because a port in Jolie might implement several interfaces, which are merged together. From the point of the Jolie developer it will look like in Figure 29.

```
interface i1 {
    RequestResponse
        operation1 (int) (int)
}

interface i2 {
    OneWay
        operation2 (int)
}

inputPort p {
    Interfaces: i1, i2
}
```

Figure 29 – inputPort in Jolie declaring multiple interfaces

But from the perspective of our D-Bus extension the interfaces have been merged into one, and once that happens there is no longer any way to find a meaningful name for the combined interface. Since the interface must have a name, we have decided to use the bus name declared in the location string instead, since it resembles the required format for interface names almost completely, the only difference being that interface names must not contain hyphens. Using the bus name as the name of an interface is also a common practice in D-Bus, for example the D-Bus daemon exposes the interface `org.freedesktop.DBus` under the same bus name.

Using the bus name means that we do not have to verify the well-formedness of the name; it has already been verified by the call to `RequestName` on the bus - if the name was not well-formed or already taken our extension would have thrown an exception during initialization. Location strings can be set during runtime, so if the Jolie developer knows that there is a risk of naming conflict the Jolie application can be developed to fail more gracefully.

After having created the interface node, we create a `method` node for each operation in the interface by looping through the request-response operations and the one-way operations of the interface and creating a method entry for each. The only difference between the two is that request-response creates argument entries for both `in` and `out`, whereas one-way only creates `in` arguments, and that one-way operations are annotated with the `org.freedesktop.DBus.NoReply` annotation.

When generating argument entries for each node, we need to determine the D-Bus signature of that argument. The interface data contains the Jolie types of the interface, so we map those to Java types and use the `Marshalling` class to create a signature string for each argument. The mapping is done recursively, checking each type in the following order: if the maximum cardinality of the type is greater than 1, we know it to be an array; if not we check if it is a tree structure, by checking whether it has any subtypes. Finally if it is neither, we know it to be a simple type.

While we are creating the introspection string, we also collect the names of the input and output arguments to each method, so they can be used to map a Jolie value to and from D-Bus messages with the right argument order.

Concurrency & connection persistence

The D-Bus extension is implemented with concurrency and channel persistence in mind. Specifically the `DBusCommChannel` has been implemented to be both thread-safe and reusable.

Output

Concurrency is supported by output channels through mutual exclusion. The implementation is true to the proposal mentioned in the problem analysis.

Requests are sent through the `sendImpl` method which does not modify any shared resources before the final statement which invokes `writeMessage` on the D-Bus `Transport`. This has been wrapped in a synchronized block and thus multiple threads using the channel will have to take turns obtaining the lock for this.

The response part is more complicated. As mentioned in the problem analysis, concurrent pairing of requests and responses is not trivial. The input stream can't be read in parallel and you can't peek into the stream - once a message has been read it has also left the stream. Due to this we have implemented a `listenFor` method which is a slightly more complicated mutual exclusion setup. It involves maintaining a message buffer and taking turns reading messages from the input stream, saving message in the buffer if they are not the one which the thread is looking for and otherwise returning once the correct message has been found. The execution flow is true to Figure 20 from the problem analysis.

Input

Concurrent calls on channels used by input ports are handled almost entirely by the D-Bus daemon and Jolie's communication core. The daemon ensures that concurrent messages are written sequentially to the input stream. Our `DBusCommListener` implementation maintains a single listen loop which reads messages and schedules them for execution in Jolie's communication core. It is up to the core to decide when an operation is executed and at some point it will return a response through the `DBusCommChannel` maintained by `DBusCommListener`. The send implementation uses mutual exclusion to ensure thread-safe writes, as mentioned above.

Connection persistence

Most of the implementation to support connection persistence is already embedded in the communication core of Jolie. To enable it we have to tell `CommCore` that the channel should not be closed, but released. This is done by invoking the `setToBeClosed` mutator of the parent class `CommChannel`. We do this in the `DBusCommChannel` constructor. Furthermore, we have to provide a release implementation which will be invoked after each request has finished instead of the close implementation. The release procedure is pretty simple, it adds the channel to a `persistentChannels` map of `CommCore`, but first makes sure to obtain a lock for the channel. Once this has been done the release implementation terminates and the channel will remain in the map till another thread requests such a channel or the application terminates in which case all channels will have their close implementation called. The close procedure involves making a disconnect call to the D-Bus daemon.

Validation

The following section takes several approaches to validating the correctness of our implementation. The initial validation happens through an application from the Jolie repository named Vision, which previously relied on the qdbus command line tool but can now use our D-Bus extension. After the initial validation we dive deeper into the code, and specifically test two aspects of our implementation, which we highlighted during the problem analysis, namely marshalling and concurrency. Finally we benchmark the performance of our D-Bus extension against the Jolie extensions for SOAP over TCP sockets.

Vision

The Jolie repository contains examples of various Jolie programs. One of these is an application called “Vision”, which might best be described as a distributed PDF reader. Vision allows a group of computers to view the same document at once, and synchronize what page they are viewing.
Before the program is started a PDF reader instance must already be running on the computer. The implementation from the repository supports several different readers, but we have chosen only to add D-Bus support to the reader named Okular, as a proof of concept. When a reader instance is running, Vision can be started. Depending on the parameters passed to the program, it will either act as a server or a client. Any computer can act as a server, but only one may do so at a time. Whenever the server opens a document or changes the current page, it tells all connected clients to do the same.

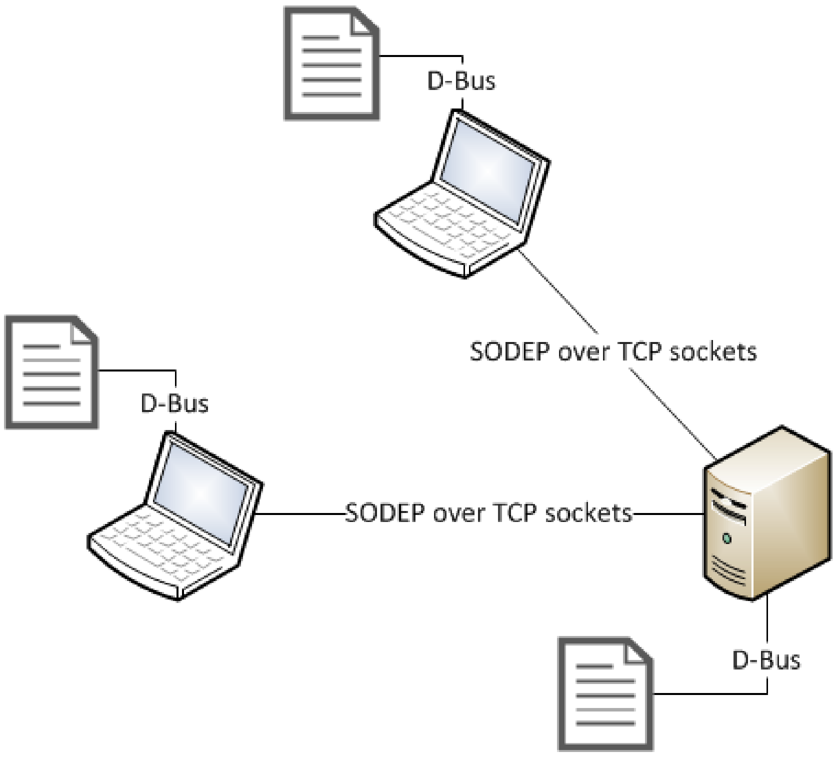


Figure 30 – setup of the Vision application.
Note: The computer case in the diagram only signifies that the computer has a special role, not that it runs any type of dedicated hardware.

The Vision example already uses D-Bus to communicate with Okular, but it does so via the `qdbus` command line tool. This means that when the Jolie client wants to instruct Okular to change page it creates a string like this: "`qdbus org.kde.okular-42 /okular goToPage 5`" and executes that string using the Jolie Exec interface. This is similar to executing the command directly on the command line, and adds an overhead in parsing the received response from a string to a Jolie value.

With our D-Bus extension we can now specify the possible operations on Okular as an output port in Jolie, and invoke operations on that, instead of using the Exec service. Being able to use Okular as an actual service in Jolie, instead of proxying the calls through the command line has changed code such as in Figure 31.

```
// BEFORE
cmdStr = "qdbus org.kde.okular-42 ";
exec@Exec( cmdStr + "/okular goToPage " + request.pageNumber )()

// AFTER
goToPage@OkularInstance(request.pageNumber)
```

Figure 31 – calling Okular goToPage, with and without our D-Bus location extension

Before, the code was executed via qdbus, and the result was discarded. This meant, that if Okular was to change their interface between versions, such that the goToPage method no longer existed, the program wouldn't know, because we are simply discarding the reply from qdbus. With our D-Bus extension on the other hand, an error would be returned from Okular, saying that the particular method was not found, and that error would be propagated up to the Jolie program. In the same fashion the poller class, which periodically checks the current page of the document of the server, now calls the `currentPage` operation directly on Okular. In the interface of Okular we can specify that the operation returns an integer, whereas when using qdbus and Exec, the exec operation would return a string, which would then have to be parsed to an integer.

Using the D-Bus extension has allowed us to rewrite the Vision example in a more type-safe way. The Vision application has helped us verify that our implementation of D-Bus for Jolie works in a real world application.

Currently the application still uses a polling loop to check whether the page of the server has changed. The program could benefit from signals, which would allow it to subscribe to a pageChanged signal once, and then not worry about checking the page any more. For more on signals, refer to the further work chapter.

Marshalling and introspection

Marshalling covers converting data in Jolie to the D-Bus wire format, and converting data in the D-Bus wire format to Jolie.

The most complex part of our marshalling implementation is in relation to our decision about how to map method signatures between Jolie and D-Bus, especially with regards to converting the upper level of tree structures to represent several arguments. Due to this complexity we have focused a lot of testing effort on complex types and Jolie to Java conversion. Finally, we have tests for simple types and variant. All four aspects of marshalling are described by their own paragraph below.

Simple types

When passing simple types there is no need to wrap the argument in a type tree, since there is only a single argument. This test case calls three methods on a server to obtain a predetermined value, integer, boolean and string respectively. The test asserts that the argument is found directly in the response (i.e. not in a type tree), that the correct values are returned, and that the introspection data is correct.

Complex types

In this test we have created a complex type tree in Jolie, to make sure that all the values are getting correctly marshalled and de-marshalled. The tree contains a string, an integer, an array of integers, a map from string to boolean, and an array of maps from string to long. The test client creates a predetermined instance of this structure and sends it to the server, which asserts that all values have been correctly transferred, and sends it back to the client, which also checks that all values are the same as those that was sent. The client also asserts that the introspection data from the server correctly matches the Jolie interface.

Variant

This test case uses exactly the same data as the test for complex types, but the interface definition contains a lot of nodes which have the Jolie data type `undefined`. In D-Bus this is mapped to the data type called ‘Variant’, which means that each value in the body will have its own signature. This test checks that Jolie can properly unwrap these variants. The assertions happen in the same way as in the complex types test.

Jolie to Java

This test creates a D-Bus service in Java, using the high-level dbus-java binding, and a client in Jolie. The Jolie program then calls an operation on the Java service, which takes multiple parameters. This test case makes sure that a Jolie type tree is actually converted into multiple arguments when sent over D-Bus, such that Jolie programs may call methods on D-Bus services which expect multiple arguments.

Concurrency

As concluded in our problem analysis, the most difficult part of supporting channel sharing is to do concurrent request-response pairing. On top of this, there is the problem of the non-thread-safe Java binding which has to be accessed using mutual exclusion.
In order to validate the correctness of our solutions to these problems we have constructed two different repeatable concurrency test suites.

Jolie Client/Server scenarios

This test suite involves running a number of tiny Jolie programs designed specifically for the purpose of testing concurrency. Every test involves a client and a server. The same interface is used throughout all the tests:

```
interface TwiceInterface {
    RequestResponse:
        twice ( int ) ( int )
}
```

Figure 32 – Jolie twice interface

It has a single request-response operation called `twice` which is supposed to return twice the value of an integer argument.

Each test involve one of three different servers that implement the twice interface and are either set up to execute in concurrent, sequential or single mode (see background on Jolie for elaboration on this language feature).

Each test involves one of two different clients that makes 50 requests. This is done either in parallel or sequentially. Each request has a different integer argument between 0 and 49. The responses are stored in an array which is eventually printed.

The correctness of request-response pairing is asserted by examining the values outputted by the client. Below is a table of the 6 different tests:

no.	**client**	**server**
0	50 parallel requests	concurrent mode
1	50 sequential requests	concurrent mode
2	50 parallel requests	sequential mode
3	50 sequential requests	sequential mode
4	50 parallel requests	single mode
5	50 sequential requests	single mode

Table 2

Note: The final 2 tests expect failure because the server terminates after the first request. The client should fail with "NoReply" or "ServiceUnknown", depending on if their requests run in parallel or sequence.

CommChannel invocation with tactically placed thread sleeps

The above test suite only tests our extension within the Jolie runtime and thus within the boundaries of the current implementation of Jolie's `CommCore`. To truly make sure that we have implemented a thread-safe communication channel we have created a test suite that puts `DBusCommChannel` under harder strain than it would likely be in the current implementation of Jolie.

The tests are entirely written in Java and involve instantiating a single `DBusCommChannel` and sharing it across a high number of threads. Each thread attempts to send a unique request through the shared channel as well as receiving the corresponding response. To ensure that the threads truly challenges the request-response pairing mechanism, by making request that are not sequential, the threads are assigned random delays before and after sending their request. This should also increase the chance that multiple threads will attempt to enter critical sections at the same time.

Since the tests are using the `TwiceInterface` mentioned above and have unique requests it is possible to assert the correctness of request-response pairing by comparing responses with requests. However, if the `DBusCommChannel` is the slightest bit non-thread-safe the tests are likely to deadlock or crash before they even get so far.

The test suite contains two distinct tests, one for input ports and one for output ports. Each of them involve 2.000 threads and requests with random delays between 20ms and 40ms. A successful test-run like this does not guarantee thread-safety but with such a high number of threads it does contribute to a much higher confidence in the thread-safety of `DBusCommChannel`.

Benchmarks

These benchmarks employ execution-time measurements. They primarily compare the performance of our D-Bus location extension with that of sending SOAP messages through TCP sockets. All benchmarks that are used for direct comparison between each other have been run on the same machine, immediately after each other, to ensure that the measurements are as comparable as possible. The full data sets can be found in the source code repository, and on the attached DVD (refer to Appendix C).

Technique

Execution-time measurement is not an entirely obvious activity in Java. A common utility to use for time measurements is `System.currentTimeMillis()` which has the limitation of 1 millisecond accuracy at best (Holmes 2006). We have chosen to use the relative-time clock represented by `System.nanoTime()` which, depending on the system, offers a much greater resolution (usually microseconds accuracy) (ibid.). It is especially useful for elapsed time comparison of benchmarks run on the same system.

Code warmup is needed due to the compilation nature of most JVMs. They will usually run the byte code many times before doing the Just-In-Time (JIT) compilation to machine code (Boyer 2008, Code warmup). This is to be able to collect profiling information in order to discover the frequently executed blocks (“hot code”) that need to be compiled. Code warmup involves running all involved code blocks multiple times to reach a steady-state execution profile and first then starting the actual benchmark.

Measurement spread needs to be accounted for. Code warmup addresses the potential variance caused by the JVM profiling and the associated compilation mentioned above, but there are still likely to be other sources of variation. For instance, we have to consider things such as automatic resource reclamation, process scheduling and memory allocation. There exists no surefire way to eliminate all of this on a modern system but it can be remedied by taking a high number of measurements and employing basic statistics. The vital factor to consider is the measurement spread and for this we calculate the standard deviation. If we take this into account we can decide with some confidence whether two benchmarks are significantly different, despite the high amount of variance. We have decided to draw probability density functions to visualize this spread further for some of the benchmarks. A common rule of thumb is that the means should, respectively, be 3 standard deviations apart for one to be able to conclude a significant difference (Boyer 2008, Part 2 Statistics) and thus we have plotted this in our diagrams as well.

In-source measuring is achieved by utilizing the log calls that are already present in our source code. The `java.util.logging` framework offers the ability to attach different handlers during runtime. Before a benchmark we attach a memory handler which stores accurate logging timestamps in memory during the benchmark, which can be done with minimal performance impact. The timestamps are formatted and written to a file after the benchmark has concluded.

Marshalling performance

One of the big selling points for D-Bus is the performance of its serialization format, the wire format, which is a binary format. The binary format is not humanly-readable, and is only usable in combination with its signature, which tells how to demarshal it. At the opposite end of the spectrum of serialization formats we find SOAP (Simple Object Access Protocol)(W3C 2007), which is a serialization format that uses the text-based, humanly-readable XML (W3C 2008) format for transmitting data. Contrary to the D-Bus wire format, SOAP contains all the information needed to deserialize it in the data itself, no signature or other external data is needed.

Even though D-Bus and SOAP are two very different serialization protocols, we still feel that there is value in comparing them to each other, because they still fulfill the same purpose, which is interprocess communication that is completely programming language agnostic. SOAP is also agnostic to the communication medium, whereas the D-Bus wire format is closely tied to D-Bus as a communication medium.

The SOAP serializer is found in the `SoapProtocol` class. Since SOAP can be used over many communication media, it extends the `CommProtocol` class, and thus must conform to a specific pattern when marshalling: when marshalling a message, it takes an `OutputStream`, a Jolie message and an `InputStream`. It is then responsible both for serializing the message, and writing it to the `OutputStream`. Equally, when demarshalling a message, it is passed an `InputStream` and an `OutputStream`, and is responsible for reading the data from this stream. The D-Bus serializer, found in the class `DBusMarshalling`, on the other hand takes a Jolie message, and returns it in the wire format, optionally creating a signature for the message in the process. Since the wire format is closely tied to the D-Bus protocol there is no need for the D-Bus serializer to extend the `CommProtocol` class, and thus it can define its own interface, and avoid having to write to a stream, it can just return the marshalled data. Since the `SoapProtocol` is doing more work when marshalling (both marshalling and writing to a stream), our first concern before comparing the two was to make sure, that writing to and reading from streams did not have any significant impact on the time it takes to marshal a SOAP message. To measure this, we added logging output to the `send` and `recv` methods before and after the streams are accessed to measure how much time is spent on writing to the streams, and how much time is spent doing actual Jolie to SOAP conversion. The time spent writing a message to the stream in the send method turned out to be on average 2,4 % of the total time spent, and the time spent reading and parsing the message (done in one operation) in the receive method turned out to be 0,8 % of the total time spent. With such low numbers we feel that it is still feasible to compare D-Bus to SOAP, even though the SOAP protocol does a little more work when marshalling. The full measurements for time spent marshalling vs. time spent accessing streams can be found in Appendix A.

Setup

The test setup for both tests includes creating a Jolie value, with 200 child nodes, which are all integers. Even though a value with 200 children is not very common in Jolie, this number was chosen in order for the operations to take sufficiently long time to get some reliable measurements. For D-Bus the `valueToDBus` method is called once before any measurements are made, in order to create the introspection string that will be used for demarshalling in the benchmarks. For SOAP a `SoapProtocol` is instantiated, and used to create the XML string that will be used for demarshalling.

Benchmarking

D-Bus

The D-Bus benchmark measures three individual function calls in each run: marshalling the value with known argument names and known signature, marshalling the value without argument names and with an unknown signature (meaning the signature is generated each time), and demarshalling with known argument names and signatures. The methods are all static, so there is no initialization of the `DBusMarshalling` class.

SOAP

The SOAP benchmark measures two functions in each run: `send` and `recv`. As already mentioned the send method writes the result of its marshalling to a stream instead of returning it. `Send` is passed a stream that writes to the null device, `/dev/null`. `Recv` is passed an in-memory byte array stream from which it reads the SOAP XML string. New streams are instantiated with each run of the test, and closed and flushed after each measurement.

Results

The results for marshalling and demarshalling are seen below. The density function of the normal distribution has been plotted for each measurement.
The full data set can be seen in Appendix C 1.3. For both marshalling and demarshalling it can be seen, that D-Bus is both significantly faster, and has a significantly smaller spread. Even when taking into account the standard deviation, depicted by the dotted lines, there is a difference between D-Bus and SOAP of 0,24 ms for marshalling (Figure 33) and 0,97 ms for demarshalling (Figure 34) respectively. Thus we have confirmed that marshalling to the D-Bus wire format is indeed significantly faster than to SOAP.

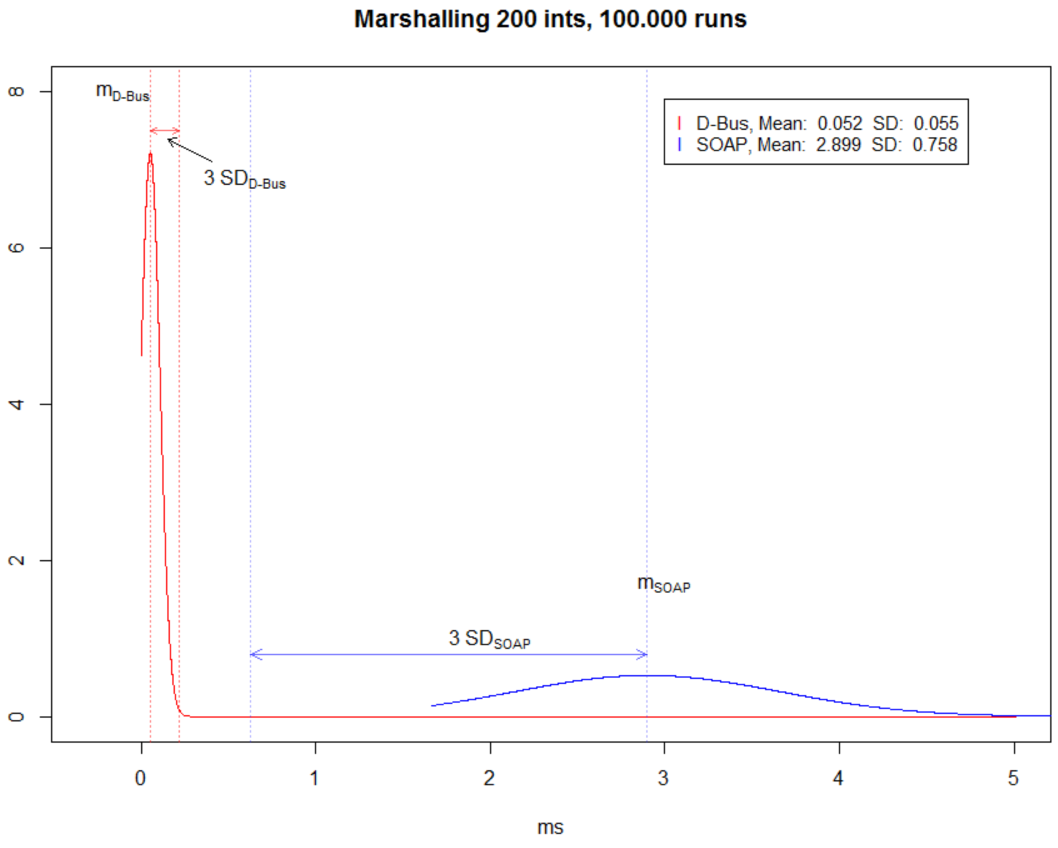


Figure 33 – Probability density distribution of the time to marshaling 200 integers in D-Bus and SOAP

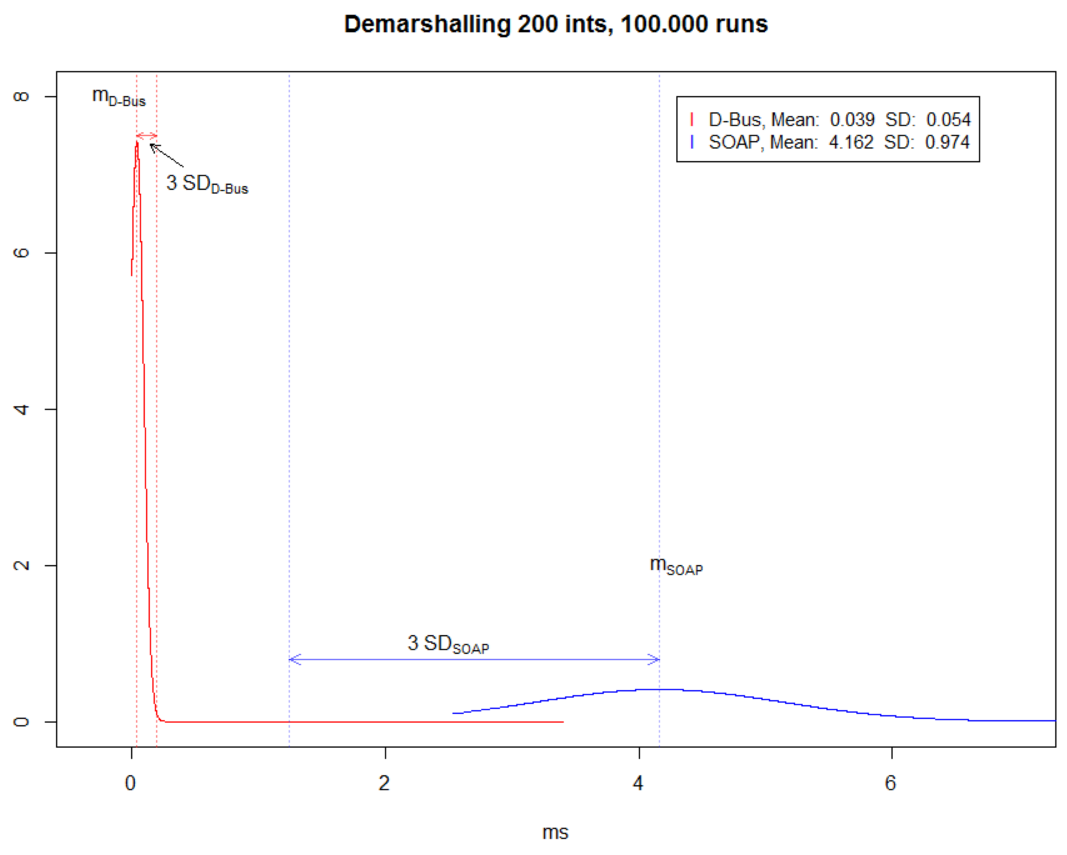


Figure 34 – Probability density distribution of the time to demarshaling 200 integers in D-Bus and SOAP

In addition to comparing SOAP and D-Bus, we also compared two D-Bus marshalling techniques, namely marshalling with and without known signatures. When the signature is not known, the `DBusMarshalling` class uses reflection to obtain the types of each arguments, which are then used to create the signature string. Even though there is a difference of almost 1ms between the means of the two methods, we cannot say anything conclusive about the difference because of the standard deviation. The plot in Figure 35 shows how the two curves clearly overlap, and how the peak (the mean value) of marshalling with a known signature is fully contained within 3 times the standard deviation of marshalling with unknown signatures.

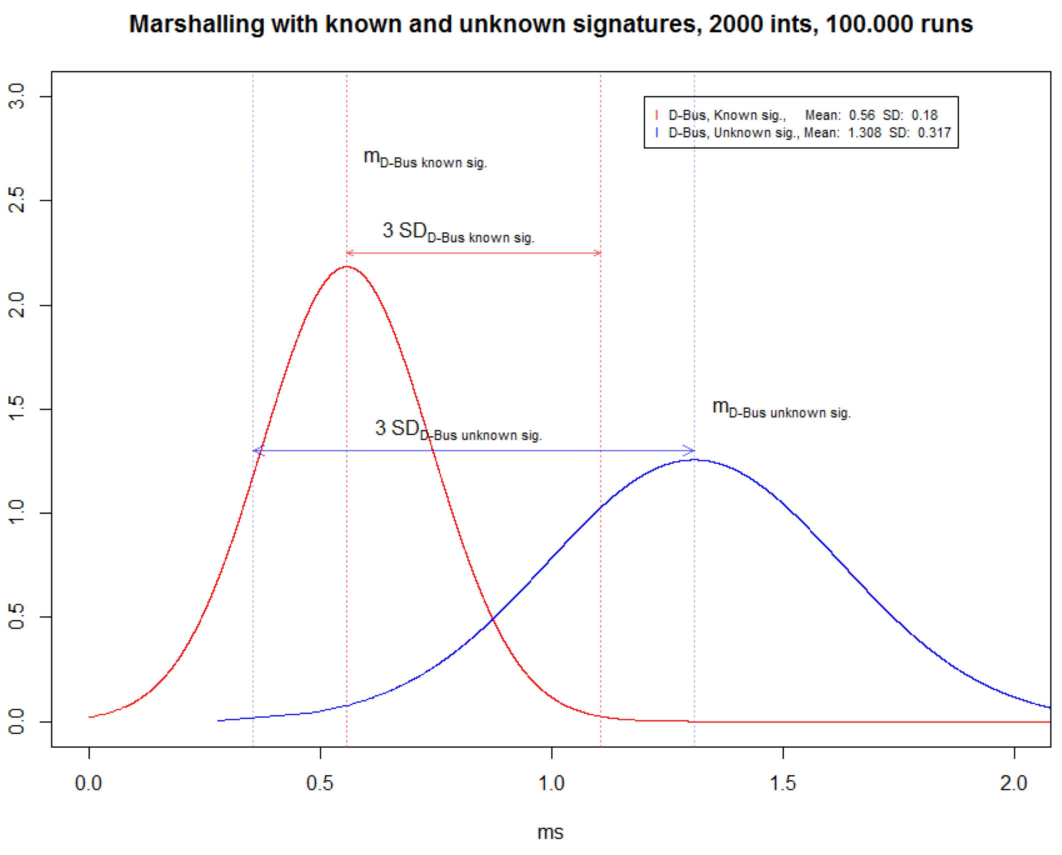


Figure 35

Execution time distribution of request-response in a Jolie runtime

The proportions of each activity in a request/response execution is very different for the two protocols. Their respective communication media are UNIX domain socket and TCP socket. The most relevant conclusions are the ratio between time spent on marshalling contra time spent on writing to the medium and vice versa.

Marshal represents the time from send has been called till the message has been marshalled
Write represents the time spent on writing to the socket
Trip+Read represents the time from just after write was completed and till the response has been read from the socket. Trip and read could not be separated for the DBus binding without modifying and recompiling the source code of the binding with measurement calls which we consider a bit out of scope. Besides, the ratio between trip and read is not that relevant.

Demarshal represents time from the response have been received from the socket till it is returned to the Jolie runtime.

DBus has a ratio where socket interaction dominates (Figure 36). For marshal and write it is approximately **1: 12**.

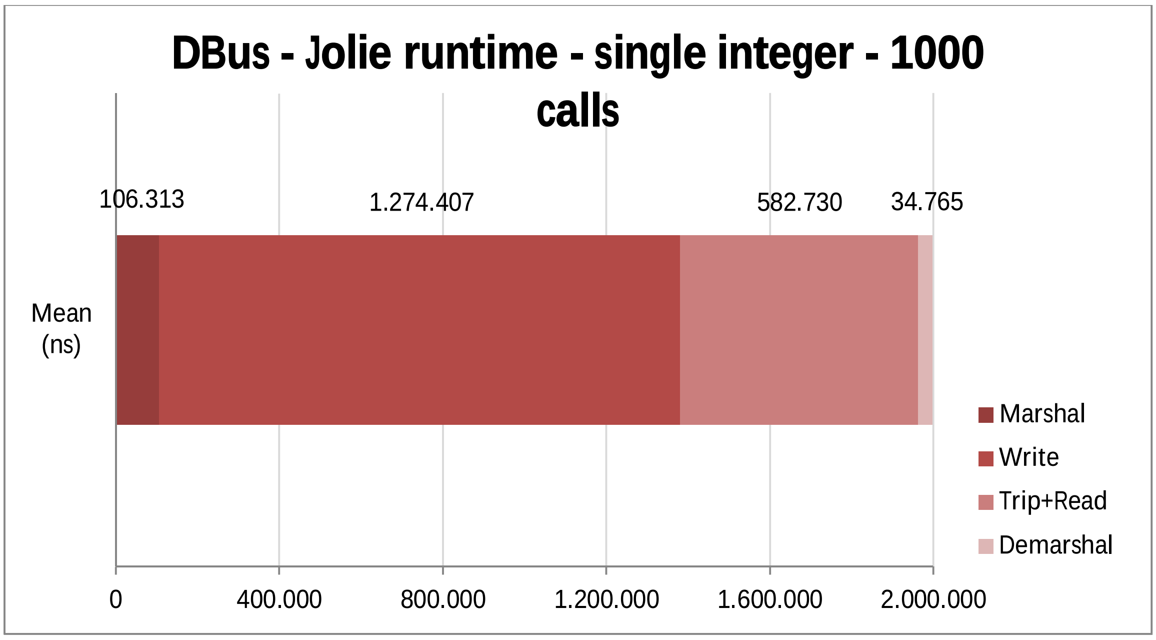


Figure 36 – Sending messages via D-Bus, measured in the Jolie runtime

SOAP has a ratio where marshalling dominates (Figure 37). For marshal and write it is approximately **1 : 0.6**.

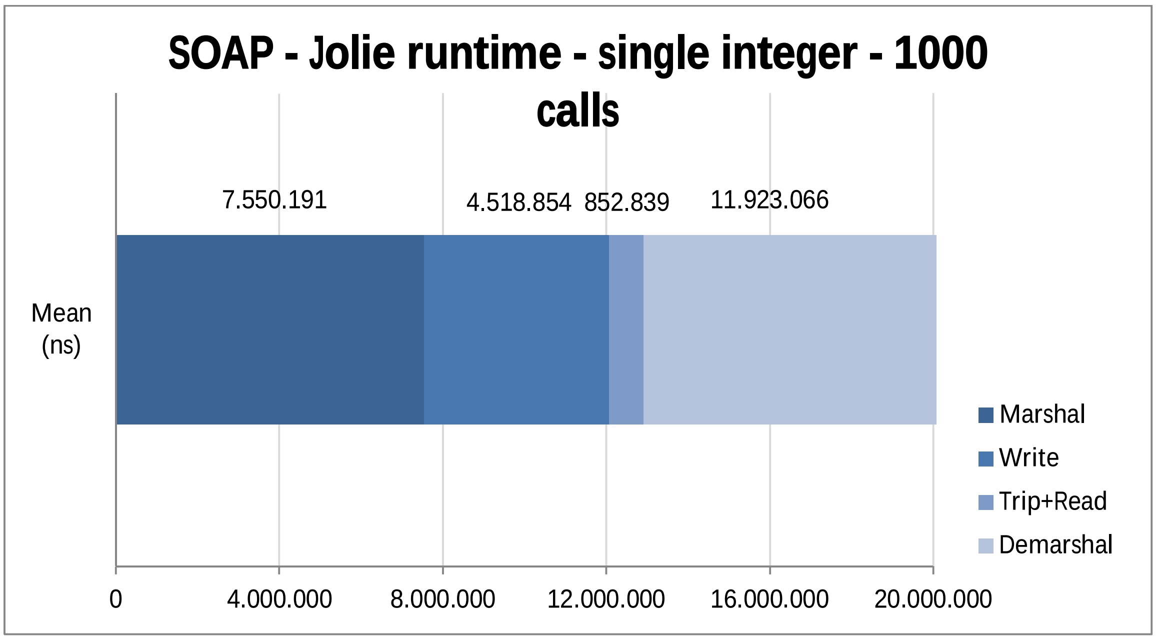


Figure 37– Sending messages via SOAP, measured in the Jolie runtime

Initialization cost

As discussed in conjunction with concurrency there is a significant initialization cost associated with creating D-Bus communication channels. It would be tempting to conclude that this is a disadvantage of D-Bus that can't be found in a SOAP/TCP setup; however this is not the case. TCP requires an expensive 3-way handshake (see Radhakrishnan et. al. 2011, figure 1 for measurements of the impact of the handshake) to be completed before a connection can be established. As seen below the initialization cost of a TCP socket is just as costly as D-Bus, if not slightly worse.

A comparison would indicate that it takes about twice as long to initialize a D-Bus channel as it takes to make a simple request. This suggests an overhead of 200 % on every request for a D-Bus implementation without channel sharing and reuse.

D-Bus channel initialization activities can be grouped into two separate activities: "Initialization" and "Introspect". Both of which involve a D-Bus request-response call:

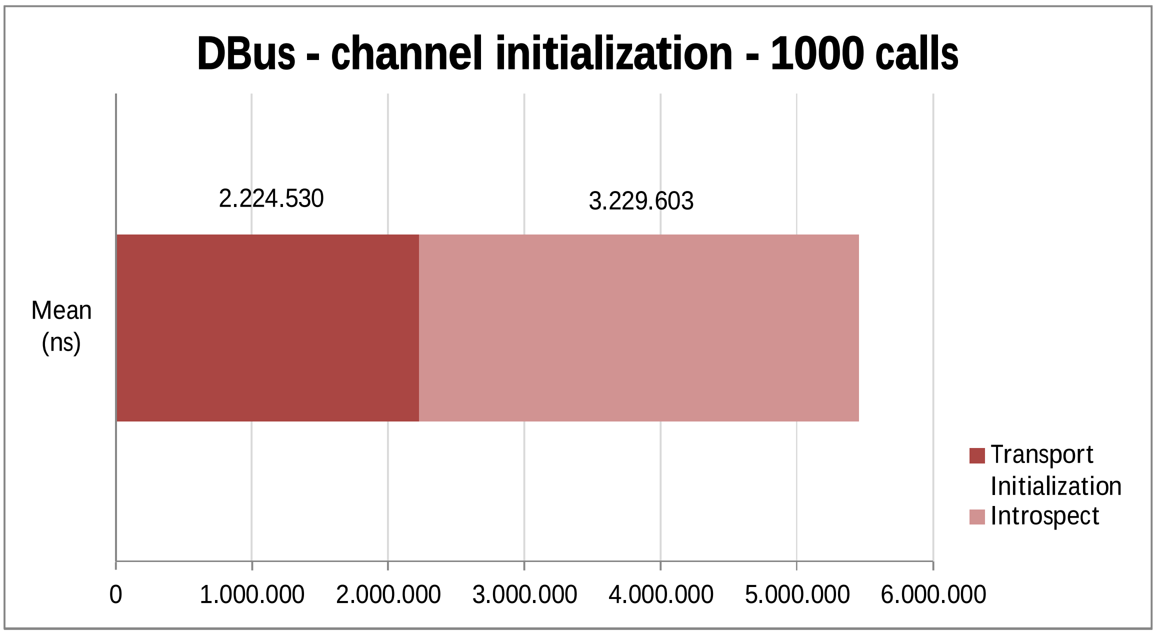


Figure 38 – Initialization of a DBusCommChannel

TCP channel initialization is dominated by socket initialization:

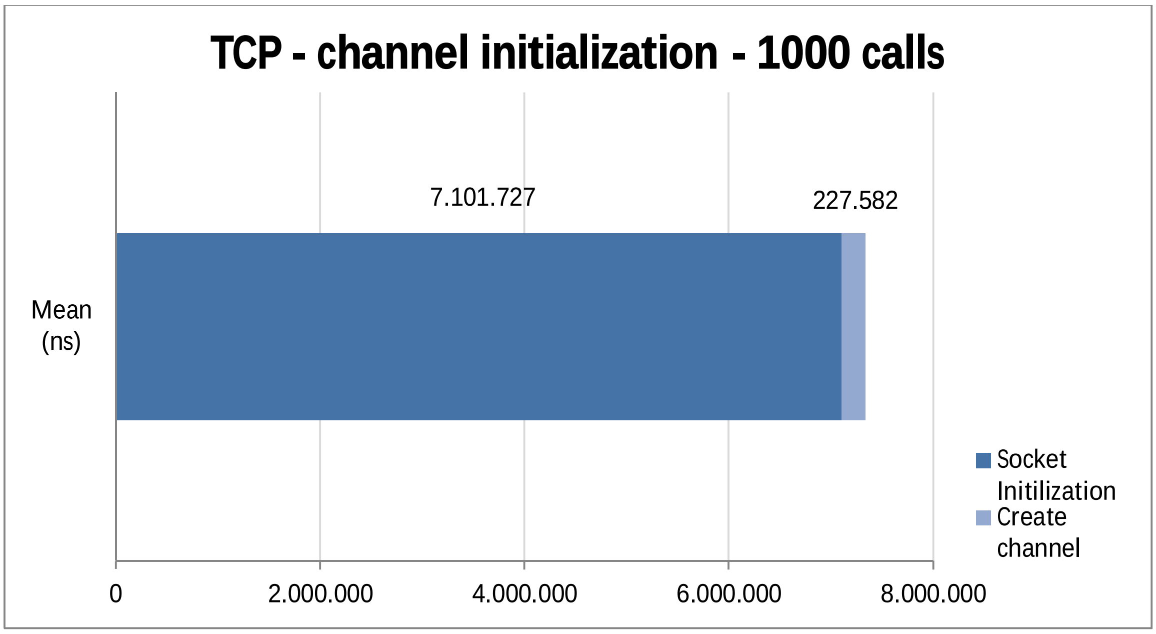


Figure 39– Initialization of a SocketCommChannel

Conclusion

We have implemented a prototype implementation of D-Bus support for the Jolie language. The implementation enables Jolie applications to communicate with any local D-Bus compatible applications. Highly competitive performance of the implementation is ensured through concurrency support as well as direct invocation of an effective low-level Java library. Our confidence in the correctness of the implementation is strong due to extensive testing of marshalling as well as full Jolie runtimes with and without concurrency. Decisions have been made with regards to how the Jolie type system is matched up with the one of D-Bus, and we feel that we have reached a compromise which offers a development experience true to the norms of both Jolie and D-Bus.

Further work

Our current implementation of D-Bus in Jolie fully supports request-response operations and one-way operations, which means that D-Bus can be used on equal terms with other location extensions in Jolie. There are however still some extensions that could be made to the D-Bus support in Jolie, two of which are described below. Signals is the hardest of the two, but in our mind also the most satisfying, since it would mean full support for the D-Bus protocol, and an improvement to the vision example (as explained in the validation section). A `dbus2jolie` command line tool would be easier to implement, since there is already a very similar tool in Jolie for WSDL (Web Service Description Language) conversion. Such a tool would greatly assist Jolie developers, especially if they are not familiar with querying D-Bus objects via `qdbus`, or with the D-Bus introspection format. It would provide a “full-circle” of D-Bus to Jolie, meaning the developer would not need to know any details about D-Bus in order to use it in a Jolie application.

Signals

As hinted at in the problem analysis on signals, the primary work to implement signals would not be the actual work of implementing the Java code to send and receive signals, but rather the work in deciding what the Jolie syntax for publish / subscribe operations should look like. The actual Java code for sending and receiving messages is trivial, given that one can reuse a lot of the code from the request-response implementation. Sending a signal requires creating a new instance of the class `DBusSignal`, and writing it to the transport. The signal takes the same parameters as a `MethodCall`, and the arguments of the signal can be converted from Jolie to D-Bus using our `DBusMarshalling` class, exactly like when invoking a request-response operation. Listening for signals requires calling the `addMatch` method on the bus, after which the signals will be routed to the connection where they can be read from the transport in exactly the same way as method calls and returns.

Signals are a big part of D-Bus, and implementing them in Jolie would be a huge step forward for Jolie programs utilizing D-Bus. The Jolie vision example would become more responsive without a CPU time cost, since it would no longer need to check periodically for page changes, it would simply wait for the signal.
We feel that Jolie as a language would benefit greatly from D-Bus signals, even though it does require some substantial changes to the language.

Full D-Bus to Jolie introspection

Full D-Bus to Jolie introspection covers being able to convert the introspection data from a D-Bus service into a Jolie interface. At the present, the programmer will have to write a Jolie interface matching that of a D-Bus service himself. In order to do this he may query the object for introspection data using a tool like `qdbus`, and then write a corresponding interface. The biggest problem with this technique is the argument names - if the introspection data contains argument names, these must be used and if it does not the arguments must be named `arg0`, `arg1` etc. - a distinction that might be hard to remember for the programmer. Furthermore, the introspection data describes the types in the style of the D-Bus type system, with letters representing the different types, e.g. i for integer, u for unsigned integer etc. When writing the Jolie interface the programmer would then have to look up what each type code stands for in order to write the right type in Jolie.

The solution to this issue would be to create a command line tool to query a service over D-Bus, parse the introspection XML data, and convert that into a Jolie interface file. The tool would most likely use the low-level D-Bus binding directly, since the introspection only requires two calls - one to say Hello to the bus, and one the query the object in question.

Jolie already has a similar tool called `wsdl2jolie` (Web Service Description Language to Jolie) which obtains the interface of a web service in the XML-based format WSDL, and converts it into a Jolie interface, which is basically the same that we would be doing - obtain the interface, parse the XML and output a Jolie interface. A lot of the code and structure from this program could probably be reused to create a `dbus2jolie` command line tool.

References

Boyer, B., 2008. *Robust Java Benchmarking*, Available at: https://www.ibm.com/developerworks/java/library/j-benchmark1 [Accessed May 10, 2013].

Burton, R., 2004. Connect desktop apps using D-BUS. *IBM developerWorks*. Available at: http://www.ibm.com/developerworks/linux/library/l-dbus [Accessed May 19, 2013].

Holmes, D., 2006. *Inside the Hotspot VM: Clocks, Timers and Scheduling Events*, Available at: https://blogs.oracle.com/dholmes/entry/inside_the_hotspot_vm_clocks [Accessed May 10, 2013].

Johnson, M., 2011. *D-Bus programming in Java 1.5*, Available at: http://dbus.freedesktop.org/doc/dbus-java/dbus-java.pdf [Accessed March 5, 2013]..

Love, R., 2005. Get on the D-BUS. *Linux Journal*, (130). Available at: http://www.linuxjournal.com/article/7744 [Accessed May 19, 2013].

Montesi, F., Guidi, C. & Zavattaro, G., 2012. Service-oriented programming with Jolie. In *Handbook of Web Services*. Available at: http://itu.dk/~fabr/papers/soc_jolie/jolie_wshandbook2012.pdf [Accessed May 19, 2013].

Pennington, H. et al., 2013. *D-Bus Specification*, Available at: http://dbus.freedesktop.org/doc/dbus-specification.html [Accessed April 2, 2013].

Radhakrishnan, S. et al., 2011. TCP fast open. In *Proceedings of the Seventh Conference on emerging Networking Experiments and Technologies*. p. 21. Available at: http://dl.acm.org/citation.cfm?id=2079317 [Accessed May 19, 2013].

Rayhawk, J. et al., 2013. freedesktop.org Wiki, 2013. Available at: http://www.freedesktop.org/wiki/ [Accessed May 18, 2013].

Valipour, M.H. et al., 2009. A brief survey of software architecture concepts and service oriented architecture. In *2nd IEEE International Conference on Computer Science and Information Technology, 2009. ICCSIT 2009*. 2nd IEEE International Conference on Computer Science and Information Technology, 2009. ICCSIT 2009. pp. 34–38.

Vervloesem, K., 2010. Control Your Linux Desktop with D-Bus. *Linux Journal*, (199). Available at: http://www.linuxjournal.com/article/10455 [Accessed May 19, 2013].

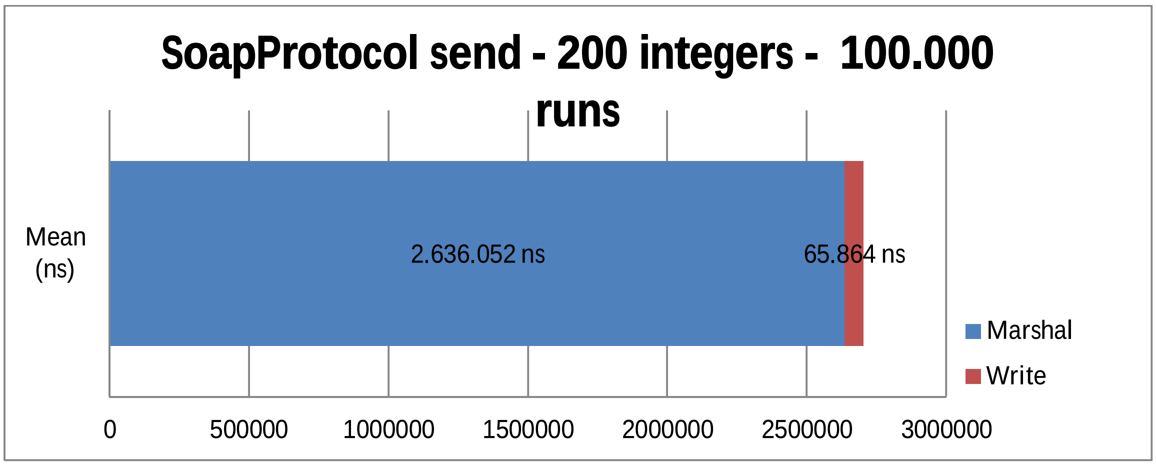
World Wide Web Consortium (W3C), 2007. SOAP Version 1.2 Part 0: Primer (Second Edition). Available at: http://www.w3.org/TR/soap12-part0/ [Accessed May 12, 2013].

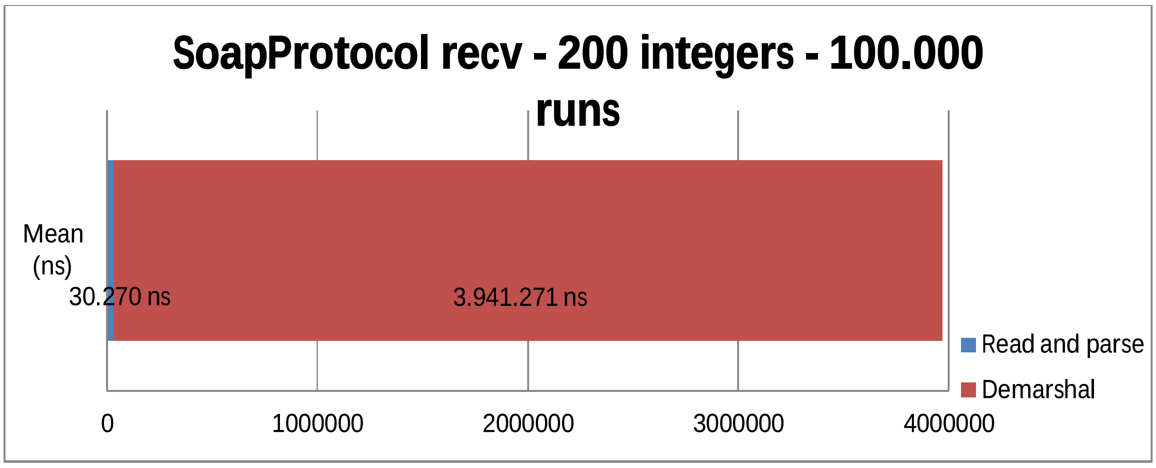
World Wide Web Consortium (W3C), 2008. Extensible Markup Language (XML) 1.0 (Fifth Edition). Available at: http://www.w3.org/TR/2008/REC-xml-20081126/ [Accessed May 18, 2013].

Appendix A – SOAP Protocol internals

SOAP is a protocol specification for serialization and deserialization of data. In Jolie SOAP is implemented in the `SoapProtocol` class, which is a subclass of `CommProtocol`. This means that the `send` method which is used to convert a Jolie value to SOAP is also responsible for writing the serialized data to a stream. We want to compare only the impact of the actual serialization to SOAP when comparing to D-Bus, so we want to make sure that writing to and reading from streams does not have any impact on performance.

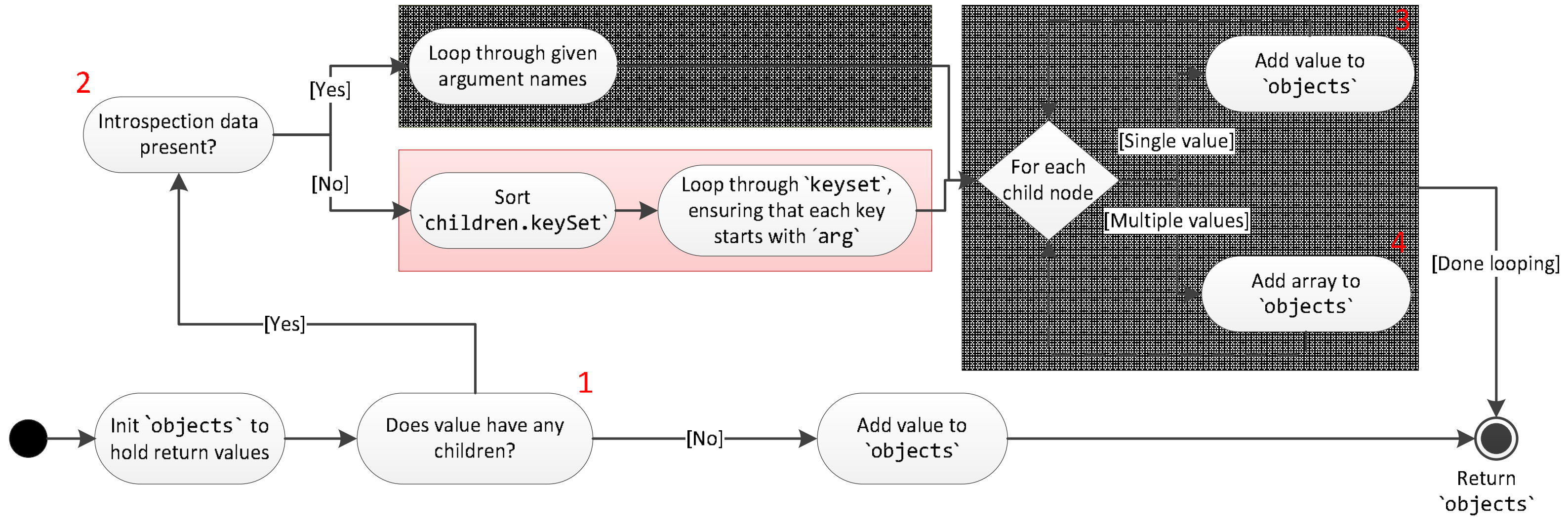
To do this benchmark we added three logs of the System.nanoTime() to the `send` method, and three to the `recv` method. In `send` the benchmarks are done at the start of the method, before data is written to the stream, and after it has been written and flushed to the stream. In `recv` the benchmarks are taken at the start of the method, after the message has been read and parsed from the stream, and before the value is returned. When the `SoapProtocol` is used in a real Jolie program, there will of course be a considerable overhead from writing and reading streams, but we tried to remove this impact as much as possible in our tests by writing to the null device (/dev/null), and reading from an in-memory byte stream. With our measurements, we have calculated the time spent marshalling and the time spent writing for `send` and the time spent reading+parsing and the time spent demarshalling for `recv`. The means of these are plotted below. For `send` the time spent writing represents 2,4 % of the total time, and for `recv` that number is 0,8 %. With such low percentages we feel that it is still completely possible to compare the `send` and `recv` operations of `SoapProtocol` with the `valueToDBus` and `ToJolieValue` operations of `DBusMarshalling`. The full data set can be found in Appendix C 1.3.


SoapProtocol send - 200 integers - 100.000 runs
Mean (ns)
2.636.052 ns
65.864 ns
Marshal
Write
0
500000
1000000
1500000
2000000
2500000
3000000


SoapProtocol recv - 200 integers - 100.000 runs
Mean (ns)
30.270 ns
3.941.271 ns
Read and parse
Demarshal
0
1000000
2000000
3000000
4000000

Appendix B – Larger diagrams

Figure 28 – marshalling flow



Appendix C – DVD Content

This appendix is a list of the content found on the attached DVD, and in the source code repository (https://github.com/nielssj/JN-DBusJolie). It is used to make it easier to refer to files on the DVD within the report.

1. benchmark results
   1.1. execution distribution - Benchmark of the time spent sending and receiving messages
   1.2. initialization cost - Benchmarks of the cost of initializing Jolie `CommChannels` for D-Bus and TCP Socket
   1.3. marshalling performance - Benchmarks of marshalling performance of D-Bus and SOAP, in addition to internal benchmarks of the `SoapProtocolCommChannel`.
2. jolie-src
3. tests
4. diagrams
5. report (this)