Report Number: WUCSE-2006-37

2006-01-01

Sliver: A BPEL Workflow Process Execution Engine for Mobile Devices

Gregory Hackmann, Mart Haitjema, Christopher Gill, and Catalin-Gruia Roman

The Business Process Execution Language (BPEL) has become the dominant means for expressing traditional business processes as workflows. The widespread deployment of mobile devices like PDAs and mobile phones has created a vast computational and communication resource for these workflows to exploit. However, BPEL so far has been deployed only on relatively heavyweight server platforms such as Apache Tomcat, leaving the potential created by these lower-end devices untapped. This paper presents Sliver, a BPEL workflow process execution engine that supports a wide variety of devices ranging from mobile phones to desktop PCs. We discuss the design decisions that allow... **Read complete abstract on page 2.**

Sliver: A BPEL Workflow Process Execution Engine for Mobile Devices

Gregory Hackmann, Mart Haitjema, Christopher Gill, and Catalin-Gruia Roman

Complete Abstract:

The Business Process Execution Language (BPEL) has become the dominant means for expressing traditional business processes as workflows. The widespread deployment of mobile devices like PDAs and mobile phones has created a vast computational and communication resource for these workflows to exploit. However, BPEL so far has been deployed only on relatively heavyweight server platforms such as Apache Tomcat, leaving the potential created by these lower-end devices untapped. This paper presents Sliver, a BPEL workflow process execution engine that supports a wide variety of devices ranging from mobile phones to desktop PCs. We discuss the design decisions that allow Sliver to operate within the limited resources of a mobile phone or PDA. We also evaluate the performance of a prototype implementation of Sliver.

2006-37

Sliver: A BPEL Workflow Process Execution Engine for Mobile Devices

Authors: Gregory Hackmann, Mart Haitjema, Christopher Gill, Gruia-Catalin Roman

Corresponding Author: ghackmann@wustl.edu

Web Page: http://mobilab.wustl.edu/projects/sliver/

Abstract: The Business Process Execution Language (BPEL) has become the dominant means for expressing traditional business processes as workﬂows. The widespread deployment of mobile devices like PDAs and mobile phones has created a vast computational and communication resource for these workﬂows to exploit. However, BPEL so far has been deployed only on relatively heavyweight server platforms such as Apache Tomcat, leaving the potential created by these lower-end devices untapped. This paper presents Sliver, a BPEL workﬂow process execution engine that supports a wide variety of devices ranging from mobile phones to desktop PCs. We discuss the design decisions that allow Sliver to operate within the limited resources of a mobile phone or PDA. We also evaluate the performance of a prototype implementation of Sliver.

Type of Report: Other

Sliver: A BPEL Workflow Process Execution Engine for Mobile Devices

Gregory Hackmann, Mart Haitjema, Christopher Gill, and Gruia-Catalin Roman

Dept. of Computer Science and Engineering, Washington University in St. Louis

Abstract. The Business Process Execution Language (BPEL) has become the dominant means for expressing traditional business processes as workflows. The widespread deployment of mobile devices like PDAs and mobile phones has created a vast computational and communication resource for these workflows to exploit. However, BPEL so far has been deployed only on relatively heavyweight server platforms such as Apache Tomcat, leaving the potential created by these lower-end devices untapped. This paper presents Sliver, a BPEL workflow process execution engine that supports a wide variety of devices ranging from mobile phones to desktop PCs. We discuss the design decisions that allow Sliver to operate within the limited resources of a mobile phone or PDA. We also evaluate the performance of a prototype implementation of Sliver.

1 Introduction

In today's world, there is an ever-growing need for collaboration among teams of people on complex tasks. *Groupware* is a special class of systems that support and facilitate these collaborative activities. The *workflow* model offers a powerful representation of groupware activities. This model is defined informally as "the operational aspect of a work procedure: how tasks are structured, who performs them, what their relative order is, how they are synchronized, how information flows to support the tasks and how tasks are tracked" [1]. In other words, workflow systems coordinate and monitor the performance of tasks by multiple active agents (people or software services) towards the realization of a common goal.

Many traditional business processes — such as loan approval, insurance claim processing, and expense authorization — can be modeled naturally as workflows. This has motivated the development of Web standards, such as the Business Process Execution Language [2] (BPEL), which describe these processes using a common language. Each task in a BPEL process is represented as a service that is invoked using the Simple Object Access Protocol [3] (SOAP). A centralized BPEL server composes these services into complex processes by performing an ordered series of invocations according to the user's specifications. Because BPEL builds on top of standards like XML and SOAP that are already widely deployed, it has been accepted readily in business settings.

The ubiquity of inexpensive mobile and embedded computing devices, like PDAs and mobile phones, offers a new and expanding platform for the deployment and execution of collaborative applications. In 2004, over 267 million Java-capable mobile phones were deployed worldwide, and Sun estimates that up to 1.5 billion will be deployed by the end of 2007 [4]. Though each device is individually far less powerful than a standalone server, their aggregate computation and communication potential is remarkable, and has yet to be fully realized. Examples of applications that could incorporate such devices advantageously include:

- Wireless sensors are attached to inventory items in a warehouse and form a mesh network. An embedded device like a Stargate [5] exposes the sensors' readings to the local 802.11b network in the form of a service. The warehouse's security team carries mobile phones which make decisions based on these readings (e.g., notify the the fire department if smoke is detected).
- A complex construction job requires workers to carry out certain tasks in a specific order. The job is modeled as a workflow, which is divided into a series of distributed processes [6]. Each process is distributed to the corresponding worker's PDA; as each process executes, it tells the worker the next task that must be completed.
- A bank goes through a series of checks and assessments during its loan application process, and may require the manager's approval for some loans. While the manager is away on a business trip, her mobile phone is equipped with a simple service which asks her to approve or disapprove new applications. This service is invoked as part of a loan application workflow.

Mobile applications like the ones described above can benefit greatly from Web standards like BPEL and SOAP. By defining a common language for inter-service interactions and data flow, these standards encourage the composition of simple services into powerful distributed applications.

Unfortunately, these applications pose several important challenges that the current state-of-the-art in SOAP and BPEL systems cannot meet. First, typical mobile devices feature severely constrained hardware requiring a very lightweight software infrastructure. Second, in the absence of a stable Internet connection, it may be impossible, impractical, or too expensive for mobile devices to contact centralized servers. Finally, wireless network links among mobile devices may be disrupted frequently and unpredictably. These challenges necessitate a lightweight, decentralized Web service middleware system which can perform on-the-fly replanning, reallocation, and/or reconfiguration in the face of network failure. Addressing these issues is a significant software engineering challenge.

In this paper, we describe Sliver, our first milestone in this long-term effort. Sliver supports the execution of SOAP services and BPEL processes on mobile devices like mobile phones and PDAs. Because Sliver builds on existing Web standards, it can be used in conjunction with a wide array of existing development tools. We emphasize that Sliver is not intended to *replace* existing SOAP and BPEL middleware: rather, it extends the Web services paradigm to new

devices which did not previously support it. In Section 2, we discuss the fundamental characteristics of mobile devices that compel a new kind of middleware. Section 3 provides an overview of the engineering challeges we faced in designing and implementing Sliver. The resulting prototype implementation is evaluated in Section 4. Finally, we discuss work on related middleware systems in Section 5, and give concluding remarks in Section 6.

2 Problem Statement

Today, developers can choose from a wide variety of support platforms for SOAP services and BPEL processes. SOAP services are typically developed using a Web service middleware platform, like Apache Axis [7]. BPEL execution engines, such as ActiveBPEL [8], WebSphere Process Server [9], and Oracle BPEL Process Manager [10], host the BPEL processes that invoke these services. These SOAP services and BPEL execution engines are then deployed on top of Java application servers, such as Apache Tomcat [11] and JBoss Application Server [12]. These general-purpose application server platforms provide developers, users, and system administrators with a wide variety of common services, such as event logging, execution monitoring, and load balancing.

Unfortunately, there are several practical issues that prevent existing SOAP and BPEL middleware packages from being deployed on mobile devices. The first issue is the combined footprint of the middleware and its support layers. For example, Apache Tomcat 5.5.17 (including the requisite Java Standard Edition 1.4.2 runtime) consumes 78 MB of disk space and 20 MB of RAM even before a single application has been deployed. Installing the ActiveBPEL engine increases this footprint to 92 MB of disk space and 22 MB of RAM; deploying BPEL processes on ActiveBPEL will consume even more resources. These requirements are reasonable for desktop computers and servers, where gigabytes of hard disk space and hundreds of megabytes of RAM are standard. However, only a handful of the highest-end mobile phones and PDAs have enough storage space and RAM to support these systems. Mobile devices also typically have far less powerful CPUs than standard PCs; even if it were possible to deploy an application of this scale on these devices, its runtime performance would be a concern.

The second issue is that these middleware frameworks and their support layers are often designed with Java 2 Standard Edition (J2SE) in mind. Runtimes which support this standard are widely available for many desktop and server platforms, but are not generally available for mobile devices. Instead, mobile device manufacturers include a more limited Java runtime, such as one based on the Mobile Information Device Profile (MIDP) standard, on the device's firmware. Such runtimes support only a fraction of the features provided by a full J2SE runtime. For example, the MIDP 2.0 standard, which is supported by virtually all modern mobile phones, has only 145 classes in its public API; in comparison, there are 2,723 classes in the J2SE 1.4.2 public API. Among other features, MIDP 2.0 does not offer most of J2SE's abstract data structures; its support for runtime reflection is minimal; and it features a unified API for file

and network I/O that is incompatible with J2SE's I/O APIs. Existing SOAP and BPEL middleware systems depend directly on these missing and altered APIs for their core functionality, and hence could not operate on most currently available mobile devices even if the hardware was sufficiently powerful.

Finally, existing BPEL systems typically use HTTP for all communication between hosts. However, this protocol is not a reasonable choice for many mobile devices. Because of network restrictions, many mobile devices (such as most mobile phones) cannot accept incoming TCP/IP sockets, and hence cannot serve HTTP requests. Incoming requests are often restricted to less-conventional transports, such as SMS messages, which current systems do not support.

Thus, if a SOAP or BPEL execution engine is to be deployed on mobile devices, it must embody three major traits: (1) it must have a suitably small storage and memory footprint, including all the libraries on which it depends; (2) it must depend only on the Java APIs that are available on all devices; and (3) it must support a wide variety of communication media and protocols flexibly. In the next section, we discuss how these traits influenced our design and implementation of Sliver.

3 Design and Implementation

Sliver's architecture features several characteristics motivated by the traits identified in Section 2. First, Sliver provides a clean separation between communication and processing. Communication components can be interchanged without affecting the processing components. Second, Sliver only depends on two lightweight external libraries, which themselves were designed with mobile devices in mind. This minimizes Sliver's footprint and ensures that Sliver can be deployed on a broad range of devices. Third, Sliver's SOAP components do not depend on its BPEL components. Developers can deploy SOAP services on mobile devices without needing Sliver's full BPEL engine, further reducing Sliver's footprint.

The resulting architecture is shown in Figure 3. At the lowest level of the architecture, the transport layer wraps various network media and protocols with a consistent interface. The transport layer exchanges message objects in the form of serialized XML strings. These strings are converted to and from Java objects by the XML and SOAP parser layers.

The SOAP server layer wraps user-provided Java services with a Web service interface. When deserialized messages arrive from the XML and SOAP layers, the SOAP server directs them to the corresponding service. The service's response is serialized by the XML and SOAP layers, and is then sent over the network by the transport layer.

Many of these layers are re-used by Sliver's BPEL server. The XML parser layer, in conjunction with the BPEL parser layer, converts user-provided BPEL documents into concrete executable processes. The BPEL server layer hosts the processes that these layers produce. Like the SOAP server, the BPEL server

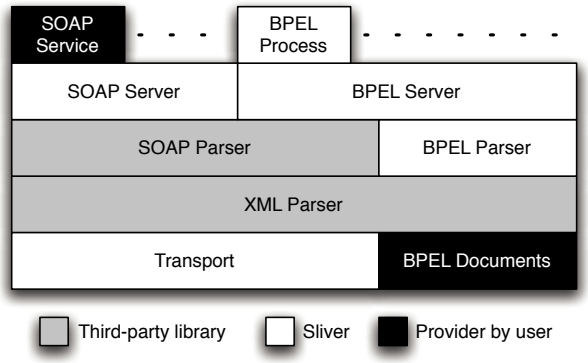

Fig. 1. The architecture of the Sliver execution engine

consumes the requests that arrive from the transport layer and routes them to the appropriate process.

In the remainder of this section, we discuss the design of these layers in further detail. We focus specifically on the design decisions that allow Sliver to meet the three traits identified in Section 2: small footprint, narrow API dependencies, and flexible communication. We also provide a brief analysis of the trade-offs made by this design.

3.1 Transport Layer

The transport layer is responsible for the transmission of messages produced by the upper layers. As we discussed in Section 2, BPEL and SOAP engines typically transmit messages using HTTP. However, HTTP is overly complex for some purposes, and devices which serve HTTP requests must have a publicly-routable IP address. Thus, it is not always reasonable to use HTTP on mobile devices. In fact, each device may support a unique set of communication protcols and media. For example, though MIDP's I/O framework defines 12 basic protocols, vendors are only required to support two (HTTP and HTTPS) [13]. Worse, MIDP devices are not required to serve *incoming* HTTP and HTTPS requests: they must only be able to create *outgoing* requests. Moreover, Sliver targets a wide range of Java runtimes and API specifications, ranging from PCs (J2SE) to mobile phones (MIDP)[1]. Because these runtimes have incompatible communication APIs, no single communication codebase can support both kinds of devices.

Sliver addresses these issues by using a transport layer with pluggable communication protocol support. When "tranditional" protocols like HTTP are not feasible, Sliver can use whatever protocols are available, like SMTP or SMS. This

[1] Sliver also supports many devices, like PDAs, which have capabilities in-between mobile phones and PCs. Some of these devices use a MIDP runtime; others feature a runtime which resembles an older version of J2SE.

design is inspired by the pluggable protocol layers used by TAO [14] and Apache Axis 2.0 [15]. Mobile devices support a wide array of communication mediums and protocols. So, Sliver's transport layer is specifically designed to facilitate the support of new protocols. If a developer wishes to support a new protocol, he must implement only two public interfaces with a total of eleven public methods. Currently, Sliver supports raw TCP/IP sockets on J2SE and MIDP devices, as well as HTTP on J2SE devices.

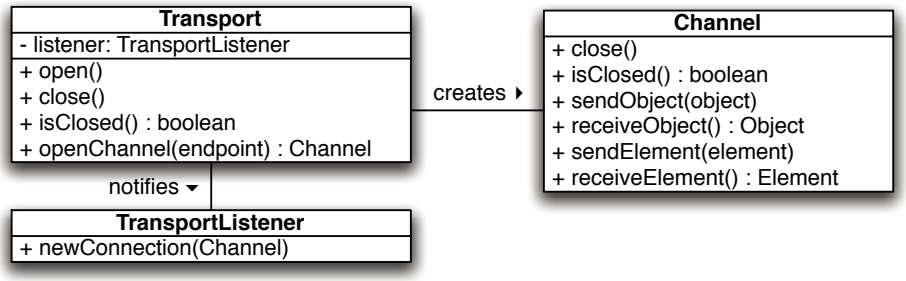

Fig. 2. `Channel` and `Transport` class diagrams

The `Channel` interface, shown in Figure 2, wraps an underlying communication medium/protocol with methods for sending and receiving messages. As we discuss in Section 3.2, Sliver represents these messages in two different forms. Sliver provides simple helper methods which feed both forms of messages into the XML and SOAP parser layers, converting these messages to and from SOAP-encoded strings. Implementing the `send` and `receive` methods thus only involves invoking the appropriate helper method, and then transmitting the result over the network. The `Channel` interface also mandates a method for closing the underlying communication protocol, which often simply calls the protocol's own close method(s).

The `Transport` interface, also shown in Figure 2, acts as a factory for creating `Channels`; each `Channel` type has a corresponding `Transport` type. The `openChannel` method creates a new connection to a remote host, as described by the provided endpoint object. The type and contents of this endpoint object are specific to each type of `Transport`; e.g., an HTTP provider may use URLs encoded as strings to describe its endpoints. Users may provide the `Transport` with a `TransportListener` object, which is notified of incoming connections. Like the `Channel` interface, the `Transport` interface provides a means for opening and closing the entire transport, which is again generally trivial to implement.

3.2 XML/SOAP Parser

Because Sliver leverages standards like SOAP and BPEL, all messages exchanged over the transport layer are encoded in XML form. Sliver uses the third-party

kXML [16] package to parse XML documents. kXML is designed with mobile devices in mind: it has a small footprint (11 KB of storage space on MIDP devices) and can operate on most available Java runtimes.

kXML implements the XML Pull Parser [17] (XPP) API, which breaks the XML into a stream of tokens that represent each tag. kXML also includes the kDOM package, which collects these tokens into nested `Element` objects. These objects describe each tag's name, attributes, children, etc.; `Element`s are nested in a way that mirrors the original XML document's structure. kXML also supports the serialization of tokens and `Element` objects back into XML strings.

As we discussed at the beginning of Section 3, Sliver encodes its messages according to the SOAP protocol. Sliver uses the third-party kSOAP [18] library to encode and decode messages. kSOAP sits on top of kXML's XPP API and converts Java objects to and from SOAP-encoded XML documents. Like kXML, kSOAP's small footprint (under 47 KB when combined with kXML) and careful use of the Java API make it ideal for deployment on mobile devices.

In the interest of brevity, we do not discuss here how SOAP encapsulates data in XML form; the interested reader may consult [3] for more information. However, it is worth noting that, like many other object-oriented languages, SOAP constructs objects out of primitive types (integers, strings, etc.) and other well-known objects. Each type of object has an associated name and namespace. SOAP namespaces are used to differentiate between different types with the same base name, and are roughly analagous to Java packages or C++ namespaces.

kSOAP maps name/namespace combinations to Java classes. It automatically recognizes a handful of types included with Java (`Integer`, `String`, etc.). It also provides a `SoapObject` class, which service providers can extend to create their own custom message types. Figure 3 shows an example of a custom `addResponse` type.

```
public class AddResponse extends SoapObject {
    public AddResponse() { this(0); }
    public AddResponse(Integer value) {
        super("http://example.com/SomeNamespace", "addResponse");
        addProperty("value", value);
    }
    public Integer getValue() {
        return (Integer)getProperty("value");
    }
}
```

Fig. 3. Code for a custom SOAP type

To minimize kXML's code size and runtime overhead, it performs very little validation on the documents that it parses. Specifically, kXML ensures that the document is *structurally* sound (e.g., that all tags are closed), but not that it is

semantically sound (e.g., that tags appear in a specific order). As we discuss in Section 3.4, this is a reasonable trade-off: for our purposes, semantic validation is often unneeded at runtime, and can be performed manually where required.

3.3 SOAP Server

The SOAP protocol provides a standard for remote procedure calls (RPC) in addition to message encoding. Sliver's `SOAPServer` class implements the RPC portion of the SOAP protocol. Again, in the interest of brevity, we do not discuss in detail how these RPC calls are encoded. We note that requests and responses are encapsulated as SOAP objects, and that these objects contain the call's parameters/return values as nested children. Like any other SOAP object, these requests have an associated type name and namespace, which are used to route requests to the appropriate service.

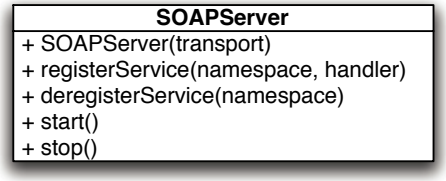

Fig. 4. `SOAPServer` class diagram

The `SOAPServer` hosts Web services on top of the transport, XML, and SOAP layers. It has five methods relevant to the user, as shown in Figure 4. The constructor, `start` method, and `stop` method are self-explanatory. The `registerService` method allows the user to deploy SOAP services, and the `deregisterService` method subsequently disables these services.

As noted above, requests are handled according to their name and namespace. In the interest of simplicity, Sliver uses a simple policy to map these request types onto Java method calls. Each namespace is associated with one Java class, and each request type within the namespace is associated with one method in that class. The user specifies the namespace-to-class mappings using the `registerService` method. On J2SE devices, Sliver's `MethodCallHandler` class automatically directs requests to the method that shares the same name and parameter types. Figure 5 shows an example of a complete SOAP service provider, which provides the `add` SOAP service within the namespace `http://example.com/SomeNamespace`.

Unfortunately, the MIDP API lacks the features necessary to direct requests to the correct methods automatically. On these devices, each service handler must implement an `invoke` method in addition to its services. When a request arrives, Sliver will pass the service's name and parameters to the correspond-

ing class's `invoke` method; the `invoke` method must direct the request to the appropriate method. An example of this procedure is provided in Figure 6.

```
public class SampleServiceJ2SE {
    public static AddResponse add(int in1, int in2) {
        return new AddResponse(in1 + in2);
    }
}
public class SOAPServerExample {
    public static void main(String [] args) throws Exception {
        Serialization.registerClass("http://example.com/SomeNamespace",
            AddResponse.class);
        Transport transport = new SocketTransport(9000);
        SOAPServer server = new SOAPServer(transport);
        server.registerService("http://example.com/SomeNamespace",
            new MethodCallHandler(SampleServiceJ2SE.class));
        server.start();
    }
}
```

Fig. 5. Code for a simple SOAP service on J2SE

```
public class SampleServiceMIDP implements SOAPServiceProvider {
    public static AddResponse add(Integer in1, Integer in2) {
        return new AddResponse(in1 + in2);
    }
    public Object invoke(String methodName, Object [] params) throws Exception {
        if(methodName.equals("add")) {
            Integer in1 = (Integer)params[0], in2 = (Integer)params[1];
            return add(in1, in2);
        }
        throw new Exception("No service named " + methodName + " found");
    }
}
```

Fig. 6. Code for a simple SOAP service handler on MIDP

3.4 BPEL Parser

BPEL workflows are treated as highly-specialized SOAP services: they are invoked using exactly the same protocols as other services. Unlike standard SOAP services — which are generally stand-alone entities implemented in any of a

wide variety of languages — BPEL processes use a standarized XML schema to describe interactions among other SOAP services. Sliver uses the XPP API described in Section 3.2 to parse these process descriptions.

Sliver represents each tag in the BPEL specification with a corresponding Java class; e.g., the <assign> tag is handled by the `Assign` class. Each class has a constructor which takes in an `XmlPullParser` object provided by the kXML library. The `XmlPullParser` maintains an internal "cursor" which points to the start of the next unparsed tag. Each constructor is responsible for parsing everything between the corresponding start and end tags, including recursively constructing the classes for any child tags. Figure 7 shows a simplified version of the `Assign` class's constructor.

```
Assign(parser, scopeData) {
    parser.require(START_TAG, "assign")
    // Require <assign> opening tag
    parser.nextTag()
    // Move cursor to start of first child

    copies ← ∅
    while parser.getName() = "copy"
        copies.add(new Copy(parser, scopeData))
    // As long as there are more child <copy> tags, parse them

    if copies = ∅
        throw "<assign> must have at least one <copy>"
    // Validate that there's at least one child

    parser.require(END_TAG, "assign")
    // Require </assign> closing tag
    parser.nextTag()
    // Move cursor to start of next tag
}
```

Fig. 7. Pseudo-code for `Assign`'s constructor

This policy has three major advantages. First, parsing a BPEL document is very straightforward. An entire document can be parsed simply by creating an `XmlPullParser` from the document's contents and invoking the appropriate constructors. Second, tags are represented as self-contained entities which can be added to Sliver incrementally. Because BPEL is a complex and evolving standard, Sliver does not yet fully support the entire BPEL language. However, as is shown by our evaluation in Section 4, Sliver can already parse and execute many useful BPEL processes. Third, each class's constructor can make local decisions which verify the BPEL document's validity. For example, the `Assign` class's constructor throws an exception if the <assign> tag does not have any

nested <copy> tags, as required by the BPEL specification. These local decisions eliminate much of the need for a heavyweight, fully-validating XML parser.

BPEL tags fall into one of two basic categories. *Activity* tags represent an actual action that can be executed — receiving a request from a client, manipulating variables, invoking SOAP services, etc. Other tags create or modify a *scope*. Scopes contain state information that is not executed directly, such as variable declarations, links (explicit dependencies among activities), and partner links (communication links to clients or SOAP services). Conceptually, BPEL scopes are much like scopes in traditional programming languages: constructs declared in one scope are only exposed to activities nested within that scope, and scopes can be nested within other scopes.

The `ScopeData` class encapsulates all the information known about each scope at parse time. As the document is parsed, a reference to the current `ScopeData` is passed to each class's constructor. Because these constructors recursively pass their own `ScopeData` to their children, all objects in the same scope will share the same `ScopeData` information. When a BPEL process declares a new nested scope (e.g., using the <scope> tag), the corresponding class copies the current `ScopeData` and operates on this copy instead. This way, changes to the new `ScopeData` are not propagated to objects outside of the new scope.

Note that the `ScopeData` only contains information that is known statically at parse time. For example, `ScopeData` contains information about the names and types of variables used by the process, but not their values. Sliver maintains runtime information in a seperate `InstanceData` class. Each time the process is executed, a new `InstanceData` object is created; it tracks variable values, link status information, and partner link bindings.

Sliver also includes an abstract `Activity` class which supports runtime execution of activity tags. All classes which represent activity tags extend the `Activity` class. `Activity` provides two important methods: `execute` and `executeImpl`. `execute` waits for the activity to be ready to execute (i.e., to ensure that all of its predecessors have completed successfully) and then invokes the `executeImpl` method. `executeImpl` is overridden by each activity class so that it actually carries out the specific action; e.g., `Invoke.executeImpl` performs a SOAP call. Each activity is responsible recursively for invoking its children's `execute` method when necessary. As activities execute, they update the provided `InstanceData` object as needed (e.g., to change a variable's value). Hence, the entire process is executed by simply creating an empty `InstanceData` and passing it to the first activity's `execute` method.

3.5 BPEL Server

The `BPELServer` class hosts the BPEL processes that the BPEL parser generates. `BPELServer` extends `SOAPServer` to invoke these processes in place of SOAP services. It also adds several additional methods to its public API, shown in Figure 8.

The `addProcess` method creates a BPEL process in the specified namespace; it reads the processs BPEL specification from the provided InputStream. This

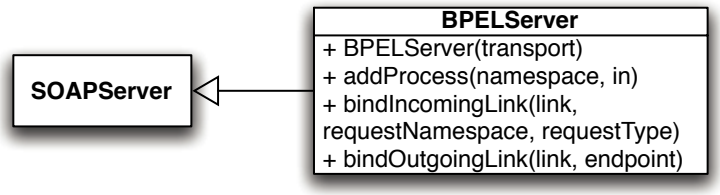

Fig. 8. `BPELServer` class diagram

method invokes the BPEL parser described above, which encapsulates the entire workflow in a `Process` object.

In the interest of being transport-agnostic, the BPEL specification does not define how to assign actual communication endpoints to each partner link. Thus, `BPELServer` offers the `bindIncomingLink` and `bindOutgoingLink` methods. The former method maps the names of incoming partner links to the kinds of messages that they accept as input. The latter method maps the names of outgoing partner links to concrete endpoints. Figure 9 shows how these methods are used to implement a complete BPEL process.

```
public class BPELServerExample {
    public static void main(String [] args) throws Exception {
        String namespace = "http://example.com/ExampleWorkflow";
        Transport transport = new SocketTransport(9001);
        BPELServer server = new BPELServer(transport);
        server.addProcess(namespace,
            new FileInputStream("ExampleWorkflow.bpel"));
        server.bindIncomingLink("ClientLink", namespace, "executeWorkflow");
        SocketAddress endpoint = new SocketAddress("127.0.0.1", 9000);
        server.bindOutgoingLink("ExternalSOAPServiceLink", endpoint);
        server.start();
    }
}
```

Fig. 9. Code for a simple BPEL process

As processes execute, the `BPELServer` also assists each process with sending and receiving SOAP messages. Unlike the `SOAPServer`, the `BPELServer` parses these messages using the kDOM parser, i.e., as nested `Element` objects. As noted in Section 3.2, this approach preserves the requests' structural value, but ignores their semantic value. We do this because the BPEL process is effectively a middleman between SOAP services: it does not care about the semantic meaning of the data that it handles. However, BPEL processes are allowed to query structural information about variables; e.g., part of a process may repeat until a

variable has a certain value. Keeping the variables in `Element` form is ideal for this, since this allows us to query the variables' structure in a generic way.

Unfortunately, this approach has one drawback: the `BPELServer` cannot ensure the validity of the messages that it handles. Many processes receive messages from remote services and eventually forward them to other services. If such a process receives a malformed message or a message of the wrong type, then it will blindly forward this invalid message to another service. According to the BPEL specification, the BPEL process should instead generate an error as soon as a non-conforming message is received.

However, we argue that this drawback is not a major concern. Web services are advertised using the Web Service Description Language [19] (WSDL), and their message formats are described using the XML Schema [20] (XSD) language. Many existing BPEL engines use these advertisements to verify that incoming messages are well-formed and of the correct type. However, WSDL and XSD are complex standards; it is impractical to deploy a WSDL or XSD parser on most mobile devices, let alone to deploy a verifier on top of such a parser. Fortunately, most verification can be done statically at design time rather than dynamically at run time. BPEL design tools like JDeveloper BPEL Designer [10] can verify statically that a process will never forward messages between two incompatible services. In effect, Sliver shifts the burden of verification to more capable devices (i.e., the process designer's workstation). This allows Sliver to be deployed on devices which cannot be expected to perform these verification steps.

Sliver cannot yet handle cases that must be verified at runtime (e.g., when a Web service sends a message that contradicts its advertisement). Again, we argue that this is not a major practical problem. Sliver's behavior and the "correct" behavior have the same end result: the process fails. However, Sliver will generate a failure when the malformed message is rejected by its ultimate destination, rather than when the process first receives the message; this is not compliant with the BPEL standard. In future work, we will consider how to provide an optional, lightweight verification process which would handle these cases.

4 Evaluation

In its current state, Sliver supports BPEL's core feature set and has a total code base of 114 KB including all dependencies (excluding an optional HTTP library). Sliver supports all of the basic and structured activity constructs with the exception of the *compensate* activity, and supports basic data queries and transformations expressed using the XPath language [21]. Sliver also supports the use of BPEL Scopes and allows for local variables and fault handlers to be defined within them. However, Sliver does not currently support some of BPEL's most advanced features, including *Serializable Scopes*, *Compensation*, and *Event Handlers*. In future work, we will extend Sliver to support these features.

In order to provide an adequate evaluation of Sliver, it is important not only to benchmark its performance against an existing BPEL engine, but also to demonstrate to what extent the expressive power of BPEL is preserved under

Sliver's present implementation. A framework has been proposed which allows for the analysis of workflow languages in terms of a set of 20 commonly reoccurring workflow patterns [22]. A study of the BPEL language in terms of this framework shows that BPEL can support in full 16 of these 20 workflow patterns [23]. Sliver presently supports all but 2 of these 16 patterns. As part of our evaluation, we implemented and tested these patterns as BPEL processes using the solutions described in [23]. For consistency, all of the basic activities in these processes are represented by the invocation of a trivial Web service which adds two numbers and returns the result. These example processes also provide a convenient test suite with which to benchmark Sliver's performance.

Our benchmark consists of 12 of the 20 patterns listed in [22]. The Multi-Merge, Discriminator, and Arbitrary Cycle patterns are excluded because BPEL does not support them. Sliver also does not presently support all of the BPEL features used by two of the Multiple Instances patterns. Additionally, some of the patterns supported by Sliver were excluded, as their implementation in BPEL is either redundant or is implicitly included in our solution for other patterns. For example, the Implicit Termination pattern is included in all other patterns with the exception of Cancel Case. Finally, the patterns Deferred Choice, Interleaved Parallel Routing, and Milestone are non-deterministic and therefore do not make practical benchmarks.

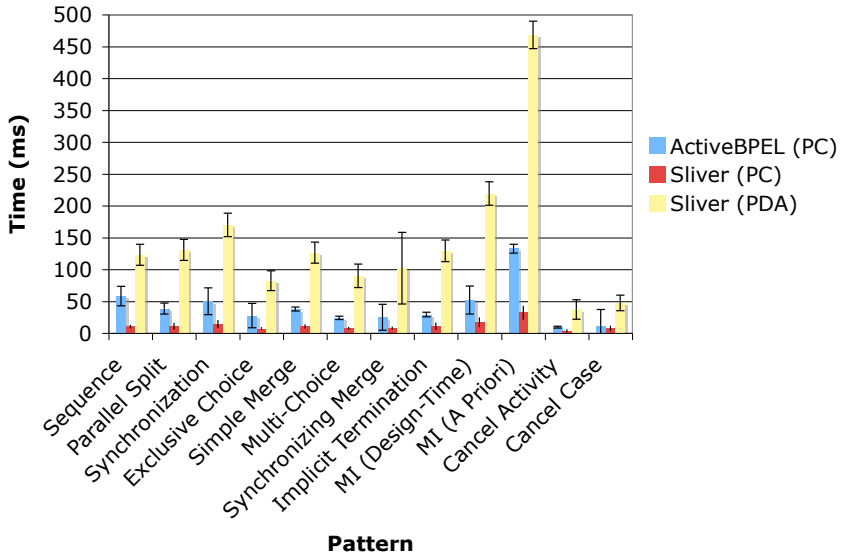

Fig. 10. The cost of executing BPEL patterns; all results are the mean of 100 runs

In Figure 10 above, we compare Sliver's execution of these 12 patterns versus that of ActiveBPEL 2.0.1.1 and Apache Axis 1.4, popular open source engines

for BPEL and SOAP respectively[2]. Our test platform for this comparison is a desktop computer equipped with a 3.2 GHz Pentium 4 CPU, 512 MB of RAM, Linux 2.6.16, and Sun Java 1.5.0_07. Both ActiveBPEL and Apache Axis are hosted on Apache Tomcat 5.5.15. Additionally, this figure shows Sliver's performance running these processes on a Dell Axim X30 PDA which is equipped with a 624 MHz XScale CPU, Windows Mobile 2003, and IBM WebSphere Micro Environment 5.7. To isolate the cost of process execution from network delays, the BPEL process and SOAP service are colocated.

105 runs of each benchmark were used to generate Figure 10. The first few runs of each benchmark have unusually high costs (often 5 to 10 times the mean) due to class loading, etc. For this reason, we discarded the first 5 runs of each benchmark and computed the mean of the remaining 100 runs. The error bars indicate the standard deviation of these runs.

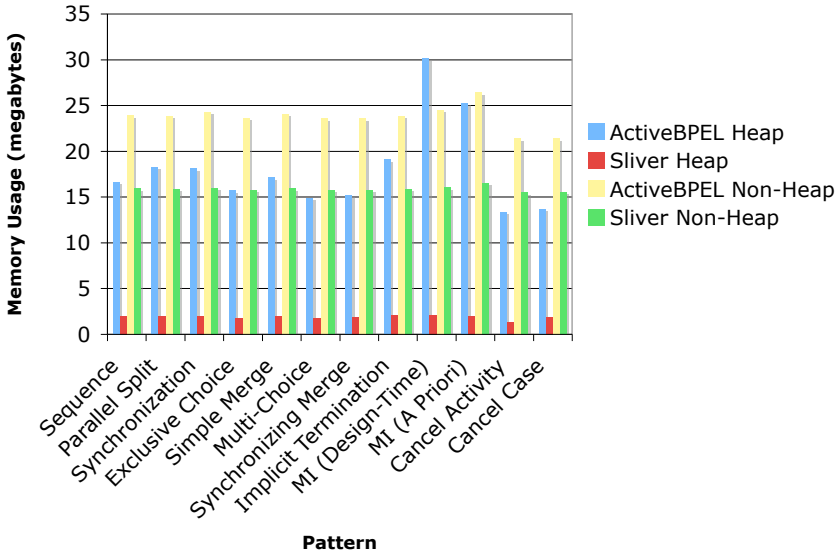

Fig. 11. The peak memory usage of each BPEL pattern

In addition, we used Sun's Java Management Extensions [24] (JMX) API to track the memory consumed while executing each pattern. Figure 11 illustrates the peak memory usage of ActiveBPEL and Sliver on our desktop PC. Unfortunately, we were unable to accurately monitor memory usage on the Dell Axim, because IBM WebSphere Micro Environment does not currently support

[2] We once again emphasize that Sliver is not intended to replace feature-rich SOAP and BPEL engines on capable hardware, but rather to support the execution of BPEL processes on resource-limited devices. Our comparison is only intended to provide a metric for acceptable performance.

JMX. We expect that it will have significantly reduced memory usage compared to the desktop platform. In particular, the non-heap memory usage (which consists largely of loaded classfiles) will be greatly reduced on mobile devices, since they will use Java runtimes with much more compact class libraries. These runtimes may also garbage-collect more aggressively when deployed on devices with reduced memory capacity, further reducing peak consumption.

We draw two important conclusions from our benchmark results. First, Sliver consistently outperformed ActiveBPEL on comparable hardware. Sliver also consumed approximately 7%–13% as much heap memory as ActiveBPEL, and approximately 66%-73% as much non-heap memory on our desktop computer. The increase in performance and decrease in memory usage over ActiveBPEL can be primarily attributed to the extra overhead incurred by ActiveBPEL's validation and administrative services, such as real time graphical status display of active processes. The results of Sliver's execution on the Dell Axim PDA show that while Sliver's performance is naturally lower than on our desktop, it is nevertheless feasible to deploy BPEL processes on such limited hardware.

Second, BPEL process execution often has negligible cost. Even on the resource-limited PDA platform, the cost of carrying out most processes is on the order of 100 ms. (The only exceptions are the Multiple Instances patterns, which contain loops that make them inherently slower than the other patterns.) As noted above, in order to isolate the costs of the BPEL engine, we evaluated processes which invoke a trivial SOAP service located on the same host. Realistically, the cost of executing non-trivial SOAP services (including network delays) is expected to dwarf the cost of supporting the BPEL process in Sliver.

5 Related Work

In [25], Gehlen describes a rule-based, peer-to-peer middleware targetted for mobile devices. Like Sliver, Gehlen's middleware system is based on the SOAP protocol, features pluggable communication providers, and can be deployed on MIDP devices like mobile phones. Sliver uses its SOAP provisioning layer to host traditional BPEL processes. In constrast, [25] supports reactive, context-aware applications described by the RuleML [26] language. Because these two systems feature such vastly different programming models, they are suited for different classes of applications, and are not directly comparable.

[6] proposes a partitioning scheme to support the execution of BPEL processes on mobile devices. This scheme divides large BPEL processes into multiple, smaller processes. This way, complex applications can be distributed across multiple mobile devices, even if no single device is powerful enough to host the entire application. Sliver is complementary to this effort; it provides a platform on which these simple processes can be hosted.

[27] describes an effort to support Web service integration in pervasive networks using BPEL and mobile agent technology. The BPEL process is executed by a mobile agent rather than a centralized BPEL engine. This agent may clone itself and migrate directly to the devices on which services are hosted. It may

also carry "packed" copies of the service along with it as it migrates. The current prototype system relies on the proprietary Bee-gent [28] mobile agent framework, which is not suitable for mobile devices because it uses parts of the Java API not supported by MIDP. However, many of the concepts described in [27] could be reimplemented on top of light-weight middleware framework such as Sliver.

6 Conclusion

In this paper, we have presented Sliver, a middleware engine that supports BPEL process execution on mobile devices. Our design flexibly supports many different communication protocols and media, while still maintaining a minimal footprint. Sliver uses a series of small, hand-written parsers in place of a heavyweight, fully-validating XML parser. These parsers keep Sliver's code size and runtime overhead suitably low for deployment on even the most resource-limited mobile devices. In its current implementation, which is available as open-source software at [29], Sliver can host many useful processes on hardware ranging from mobile phones to desktop computers. In future work, we plan to address the remaining BPEL compliance issues and consider ways to further modularize Sliver.

The development of middleware engines like Sliver is an important step toward the long-term goal of bringing groupware to mobile devices. Other important challenges — including task allocation, data distribution, and user interface design — still remain. Nevertheless, Sliver's runtime performance demonstrates that today's mobile devices are already capable of hosting sophisticated groupware applications, and that this ultimate goal is practical as well as desirable.

References

1. Wikipedia: Workflow. `http://en.wikipedia.org/wiki/Workflow` (2006)
2. OASIS Open: OASIS web services business process execution language (WSBPEL) TC. `http://www.oasis-open.org/committees/tc_home.php?wg_abbrev=wsbpel` (2006)
3. Box, D., Ehnebuske, D., Kakivaya, G., Layman, A., Mendelsohn, N., Nielsen, H.F., Thatte, S., Winer, D.: Simple object access protocol (SOAP) 1.1. Technical Report 08 May 2000, W3C (2000)
4. Ortiz, C.E.: J2ME technology turns 5! `http://developers.sun.com/techtopics/mobility/j2me/articles/5anniversary.html` (2004)
5. PlatformX: Stargate. `http://platformx.sourceforge.net/` (2006)
6. Maurino, A., Modafferi, S.: Partitioning rules for orchestrating mobile information systems. Personal Ubiquitous Comput. **9**(5) (2005) 291–300
7. Apache Software Foundation: Webservices - axis. `http://ws.apache.org/axis/` (2005)
8. ActiveBPEL LLC: ActiveBPEL engine. `http://www.activebpel.org/` (2006)
9. IBM: WebSphere process server. `http://www-306.ibm.com/software/integration/wps/` (2006)
10. Oracle: Oracle BPEL process manager. `http://www.oracle.com/technology/products/ias/bpel/index.html` (2006)

11. Apache Software Foundation: Apache tomcat. http://tomcat.apache.org/ (2006)
12. JBoss, Inc.: JBoss application server. http://www.jboss.com/products/jbossas (2006)
13. Ortiz, C.E.: The generic connection framework. http://developers.sun.com/techtopics/mobility/midp/articles/genericframework/ (2003)
14. Kuhns, F., et. al.: The design and performance of a pluggable protocols framework for corba middleware. In: Proceedings of the Sixth International Workshop on Protocols for High Speed Networks (PfHSN '99), Kluwer, B.V. (2000) 81–98
15. Apache Software Foundation: Axis 2.0 - axis2 architecture guide. http://ws.apache.org/axis2/1_0/Axis2ArchitectureGuide.html (2006)
16. Haustein, S.: kXML 2. http://kxml.sourceforge.net/kxml2/ (2005)
17. Slominski, A.A.: XML pull parser (XPP). http://www.extreme.indiana.edu/xgws/xsoap/xpp/ (2004)
18. Haustein, S., Seigel, J.: kSOAP 2. http://ksoap.org/ (2006)
19. Christensen, E., Curbera, F., Meredith, G., Weerawarana, S.: Web services description language (WSDL) 1.1. Technical Report 15 March 2001, W3C (2001)
20. Fallside, D.C., Walmsley, P.: Xml schema part 0: Primer second edition. Technical Report 28 October 2004, W3C (2004)
21. Clark, J., DeRose, S.: XML path language (XPath) version 1.0. Technical Report 16 November 1999, W3C (1999)
22. van der Aalst, W.M.P., ter Hofstede, A.H.M., Kiepuszewski, B., Barros, A.P.: Workflow patterns. Distributed and Parallel Databases **14**(1) (2003) 5–51
23. Wohed, P., van der Aalst, W.M.P., Dumas, M., ter Hofstede, A.H.M.: Pattern based analysis of BPEL4WS. Technical Report FIT-TR-2002-04, Queensland University of Technology (2002)
24. Sun Microsystems, Inc.: Java management extensions (JMX). http://java.sun.com/products/JavaManagement/ (2006)
25. Gehlen, G., Mavromatis, G.: Mobile web service based middleware for context-aware applications. In: Proceedings of the 11th European Wireless Conference 2005. Volume 2., Nicosia, Cyprus, VDE Verlag (2005) 784–790
26. Boley, H., Tabet, S., Wagner, G.: Design rationale of RuleML: A markup language for semantic web rules. In: Proceedings of the first Semantic Web Working Symposium (SWWS'01). (2001) 381–401
27. Ishikawa, F., Yoshioka, N., Tahara, Y., Honiden, S.: Mobile agent system for web service integration in pervasive networks. In: 1st International Workshop on Ubiquitous Computing. (2004) 38–47
28. Toshiba Corp.: Bee-gent multi agent framework. http://www.toshiba.co.jp/rdc/beegent/index.htm (2005)
29. Hackmann, G.: Sliver. http://mobilab.wustl.edu/projects/sliver/ (2006)