Washington University in St. Louis
Washington University Open Scholarship

All Computer Science and Engineering Research | Computer Science and Engineering

Report Number: WUCSE-2007-11

2007

# Extending BPEL for Interoperable Pervasive Computing

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

The widespread deployment of mobile devices like PDAs and mobile phones has created a vast computation and communication platform for pervasive computing applications. However, these devices feature an array of incompatible hardware and software architectures, discouraging ad-hoc interactions among devices. The Business Process Execution Language (BPEL) allows users in wired computing settings to model applications of significant complexity, leveraging Web standards to guarantee interoperability. However, BPEL's inflexible communication model effectively prohibits its deployment on the kinds of dynamic wireless networks used by most pervasive computing devices. This paper presents extensions to BPEL that address these restrictions, transforming BPEL into... **Read complete abstract on page 2.**

Follow this and additional works at: https://openscholarship.wustl.edu/cse_research

Part of the Computer Engineering Commons, and the Computer Sciences Commons

## Recommended Citation

Hackmann, Gregory; Gill, Christopher; Gill, Christopher; and Roman, Gruia-Catalin, "Extending BPEL for Interoperable Pervasive Computing" Report Number: WUCSE-2007-11 (2007). *All Computer Science and Engineering Research.*
https://openscholarship.wustl.edu/cse_research/116

Department of Computer Science & Engineering - Washington University in St. Louis
Campus Box 1045 - St. Louis, MO - 63130 - ph: (314) 935-6160.

This technical report is available at Washington University Open Scholarship: https://openscholarship.wustl.edu/cse_research/116

# Extending BPEL for Interoperable Pervasive Computing

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

Complete Abstract:

The widespread deployment of mobile devices like PDAs and mobile phones has created a vast computation and communication platform for pervasive computing applications. However, these devices feature an array of incompatible hardware and software architectures, discouraging ad-hoc interactions among devices. The Business Process Execution Language (BPEL) allows users in wired computing settings to model applications of significant complexity, leveraging Web standards to guarantee interoperability. However, BPEL's inflexible communication model effectively prohibits its deployment on the kinds of dynamic wireless networks used by most pervasive computing devices. This paper presents extensions to BPEL that address these restrictions, transforming BPEL into a versatile platform for interoperable pervasive computing applications. We discuss our implementation of these extensions in Sliver, a lightweight BPEL execution engine that we have developed for mobile devices. We also evaluate a pervasive computing application prototype implemented in BPEL, running on Sliver.

Department of Computer Science & Engineering

Washington University in St. Louis
School of Engineering & Applied Science

2007-11

# Extending BPEL for Interoperable Pervasive Computing

Authors: Gregory Hackmann, Christopher Gill, and Gruia-Catalin Roman

Abstract: The widespread deployment of mobile devices like PDAs and mobile phones has created a vast computation and communication platform for pervasive computing applications. However, these devices feature an array of incompatible hardware and software architectures, discouraging ad-hoc interactions among devices. The Business Process Execution Language (BPEL) allows users in wired computing settings to model applications of significant complexity, leveraging Web standards to guarantee interoperability. However, BPEL's inflexible communication model effectively prohibits its deployment on the kinds of dynamic wireless networks used by most pervasive computing devices. This paper presents extensions to BPEL that address these restrictions, transforming BPEL into a versatile platform for interoperable pervasive computing applications. We discuss our implementation of these extensions in Sliver, a lightweight BPEL execution engine that we have developed for mobile devices. We also evaluate a pervasive computing application prototype implemented in BPEL, running on Sliver.

Type of Report: Other

Department of Computer Science & Engineering - Washington University in St. Louis
Campus Box 1045 - St. Louis, MO - 63130 - ph: (314) 935-6160

protocol-agnostic, it makes several assumptions about the underlying network which limit its ability to model pervasive computing applications. BPEL assumes that the number of participants in the application is known at design time, and cannot be changed at runtime. While this assumption is reasonable in traditional wired networks, it does not fit typical pervasive computing settings, where participants may come and go at any time. Further, BPEL only supports a small set of communication patterns, which hinders its use in dynamic networks. Nevertheless, despite BPEL's limited *communication* features, its *processing* constructs are highly expressive. Our experience developing the Sliver middleware [4] shows that BPEL already can be used to develop complex, standards-based applications on lightweight devices.

In this paper, we show that by augmenting BPEL's communication capabilities, it can be adapted further into a powerful standard for executing pervasive applications, even in mobile settings. Section II describes the BPEL standard and outlines its potential for pervasive computing. In Section III, we discuss several extensions to BPEL which support group communication and re-use of partner links in pervasive computing applications. We discuss the Sliver middleware, which incorporates these extensions, in Section IV. A prototype over-the-air deployment system using the Sliver middleware is described in Section V. We evaluate a sample application deployed using this system in Section VI. Finally, we discuss work on related systems in Section VII, and give concluding remarks in Section VIII.

## II. Problem Statement

BPEL is a powerful language which leverages software services to model and execute applications as workflow processes. However, current limitations in BPEL's communication model hinder its ability to interact with other hosts in a dynamic network. In this section, we discuss briefly features of the SOAP and BPEL standards as they relate to pervasive computing. We also highlight the need for BPEL extensions to support pervasive computing adequately.

When deploying applications that span multiple hosts, the need for standardized message formats is paramount. This is especially true when interactions among hosts are unplanned, as is the case in many pervasive computing settings. The SOAP standard addresses this need, by describing a XML-based encoding for messages. SOAP provides an object serialization scheme that converts objects into architecture-independent, language-independent XML strings. Like most object-oriented languages, SOAP builds complex messages by aggregating primitive types like integers and strings.

In addition to specifying a message encoding scheme, SOAP provides a simple framework for service invocation. Traditionally, this invocation framework has been used as a standards-based RPC mechanism, where each remote method is exposed as a SOAP service. However, SOAP supports a rich range of interaction patterns beyond simple request/reply pairs. In the interest of portability, SOAP does not mandate an underlying network transport. Though SOAP is most frequently coupled with HTTP, some SOAP implementations support a wide array of transports ranging from Bluetooth to SMTP. When coupled with an advertisement mechanism like Zeroconf [5] or Bluetooth service discovery, users can discover and invoke remote SOAP services without prior knowledge of their existence.

At the most abstract level, a BPEL process is essentially a list of service invocations, with the ability to perform sets of invocations in parallel or in a specific sequence. Much of BPEL's power comes from its ability to store the results of these invocations in global variables, which processes can pass as inputs to other Web services. Between service invocations, BPEL processes optionally may examine or change parts of these variables using the XPath [6] query language. Processes also may change their behavior at runtime based on variables' contents, by using branching constructs and `while` loops. Finally, processes may indicate series of actions that should be carried out asynchronously when events *fire*, such as when specific messages arrive at the process or when a timer expires. Using these simple constructs, workflow designers can string simple services into complex, adaptive, and interoperable applications.

Because BPEL is a Turing-complete language [7], the computations that are performed in between service invocations can be extremely powerful. However, this powerful computational model is coupled with an inflexible communication model. BPEL represents all communication links — whether incoming or outgoing — as abstract *partner links*. From the process's point of view, a partner link is simply a communication channel which allows the initiator to send exactly one request, and the recipient to reply with at most one response. Each partner link is bound to a single remote endpoint at a time. In the interest of being transport-agnostic, BPEL processes have very limited control over the bindings of these partner links: the current version of BPEL does not allow processes to inspect or modify a partner link's binding, except by copying bindings directly between two partner links. It is also assumed that the BPEL middleware has some policy for pairing partner links to endpoints; this mechanism is not exposed to the process. Most BPEL middleware have an API for mapping outgoing partner links to endpoints at deployment time, and automatically bind incoming partner links to the source of the incoming messages. Because partner links are described in terms of their *types* and not their *endpoints*, workflow designers can model interactions with remote services that may not be identified until runtime — an important feature in pervasive computing settings.

Unfortunately, though process designers do not need to predict the identity of partners at design time, they must declare a fixed set of partner links with well-known types nevertheless. So, the process designer must effectively predict at design time the number and kinds of partners that will participate in the workflow. In wired network settings, this assumption is often reasonable. However, in pervasive settings, many applications assume — and often benefit from — the dynamic nature of the network. Peers may enter or leave the

network at practically any time. This dynamic behavior is both an asset and a liability: new peers may provide additional data or services when they arrive, but existing peers may sever communication links unexpectedly if they leave the network.

This discrepancy currently forces process designers to use one of two inadequate options for dealing with interactions among hosts. First, a designer may estimate the maximum number of participants during the process's lifetime, and create a single-use partner link for each. For example, a workflow for an auction would have a single partner link for the seller, and a partner link for each potential bidder. This approach places artificial constraints on the process's scope, and quickly leads to unwieldy code duplication.

Second, a designer may declare one partner link for each *type* of interaction. Incoming messages are handled as asynchronous events, so that the process designer does not have to predict at design time when or how often these interactions will occur. For example, in the auction application, there would be two partner links: one for the seller, and one shared among all the bidders. However, this approach assumes that the underlying BPEL middleware allows partner links to be bound to different hosts during the process's lifespan: e.g., that the link shared among the bidders is bound to one host between the time that a bid is received and a response is sent, after which another host can be bound to the same link. As noted above, BPEL provides no guarantees about how partner links are bound to remote endpoints, much less the lifespan of these bindings. Moreover, this approach only works if no two participants will ever issue the same kind of request concurrently, since each partner link can only be bound to one endpoint at a time. This constraint effectively prohibits applications from using long-lived transactions.

BPEL's simple message interaction patterns further compromise its communication power. Though SOAP allows service providers to send and receive any number of messages in an arbitrary order, BPEL is much more restrictive. BPEL processes only support one-way request (the partner sends exactly one message and immediately disconnects) and request-response (the partner sends exactly one message and then receives exactly one message) interaction patterns. As we discuss later in Section III, these simple interaction patterns are insufficient in pervasive computing settings.

Despite BPEL's shortcomings, its characteristics fit many important needs of pervasive computing applications. When BPEL is coupled with SOAP, any device equipped with a standard SOAP middleware can participate in the application. Because BPEL models applications in a standardized XML format, new applications can be deployed over-the-air to devices equipped with a general-purpose BPEL execution engine. Finally, because of BPEL's widespread acceptance in wired settings, many modeling and verification tools (such as JDeveloper BPEL Designer [8] and NetBeans Enterprise Pack [9]) exist to assist developers with constructing applications in BPEL. In the next section, we will describe several extensions we propose for the BPEL language to permit its deployment and use in pervasive computing settings.

```
<partnerLinks>
  <partnerLink ... />*
  <ext:partnerGroup name="ncname"
    partnerLinkType="qname" />*
</partnerLinks>
```

Fig. 1. Partner Group Declarations

```
<ext:add partnerGroup="ncname" partnerLink="ncname"
  mustNotBeMember="no|yes"? />
<ext:remove partnerGroup="ncname"
  partnerLink="ncname" mustBeMember="no|yes"? />
```

Fig. 2. <add> and <remove> Activity Semantics for Partner Groups

## III. BPEL Extensions

To address the issues highlighted above, we propose several extensions to the BPEL language. Briefly, these extensions make the following changes:

1) Processes may declare *partner groups*, or partner links that are bound to multiple incoming endpoints simultaneously.
2) Processes may sent multicast messages to all members in a partner group.
3) Processes may send or receive an arbitrary number of messages over a partner link or partner group.
4) Processes may make limited changes to the bindings of partner links, with well-defined behavior.

For the sake of consistency, we describe our extensions with the same notation used in [2]. Tags with the `ext:` prefix are part of our extensions; all other tags are part of the standard BPEL specification.

### A. BPEL Extensions for Partner Groups

As we discussed in Section II, BPEL processes communicate with remote hosts over partner links. These partner links are declared in a single <partnerLinks> delimited section located at the beginning of the process description. We extend this <partnerLinks> section to introduce the notion of *partner groups*, as Figure 1 illustrates. We define partner groups to be unbounded lists of partner links. Like a partner link, each partner group has a unique name and an associated *type* (i.e., the kinds of services that the links can invoke on the workflow, and vice versa). Unlike partner links, partner groups can be bound to any number of endpoints simultaneously, and can be manipulated by the process at runtime using additional operations discussed below. These two traits are essential in mobile settings: they allow a process to refer to any number of remote hosts, without requiring the process designer to predict this number at design time.

Initially, partner groups are not bound to any endpoints. Figure 2 describes two new BPEL activities (`add` and `remove`) which change the membership of partner groups. The `add` activity adds an endpoint (taken from a specified partner link) to a partner group. By default, if the endpoint is already a member of the group, then the operation will do nothing. If the `mustNotBeMember` attribute is set to `yes`, then the process will throw a `mustNotBeMember` fault instead. Likewise, the

```
<ext:reply
  (partnerGroup="ncname" | partnerLink="ncname")
  moreMessages="no|yes"? ... />
```

Fig. 3. Extended <reply> Activity Semantics

```
<ext:close partnerLink="ncname" />
<ext:unbind partnerLink="ncname" />
```

Fig. 4. <close> and <unbind> Activity Semantics

remove activity removes an endpoints from a partner group. If the mustBeMember attribute is set to yes, then attempting to remove a non-existent member from a group will throw a mustBeMember fault.

To leverage these partner groups, we extend the definition of the reply activity, as shown in Figure 3. This extended reply activity has two additions that differentiate it from a normal BPEL reply activity. First, the reply may be sent to a partner group instead of to a partner link. In this case, the same message is sent to all members of the group. Second, the moreMessages attribute indicates whether or not the partner is allowed to continue sending or receiving messages over the link after this message is sent. By default, moreMessages is set to no, and the process uses the simple single-request, single-response interaction pattern normally used by BPEL. If moreMessages is sent to yes, then the process can continue to send and receive messages over the link, permitting complex interaction patterns.

The draft specification for WS-BPEL 2.0 [10] will allow processes to emulate multicast behavior using standard activities: processes can declare variables which effectively act as lists of bindings; copy bindings from partner links into these variables; and then iterate over the contents of these variables using a forEach activity. (As of this writing, the current version of BPEL is WS-BPEL 1.1; WS-BPEL 1.1 does not allow processes to copy partner link bindings into variables, and does not support the forEach activity.) Nevertheless, the extensions proposed here are simpler for process designers to use, since multicast is treated as a first-order activity. Moreover, WS-BPEL 2.0 will not address the other issues highlighted in Section II: namely, BPEL's inability to support complex message interaction patterns or to effectively handle an unbounded number of participants.

### B. BPEL Extensions for Partner Link Re-Use

As we discussed in Section II, in order for BPEL processes to effectively support an unbounded number of participants, they must expect multiple hosts to use the same incoming partner link at different times. However, BPEL does not provide a clear semantics for the lifespan of bindings between hosts and partner links. To address this issue, we propose the following semantics for incoming partner links:

1) Partner links can accept incoming connections if and only if they are not already being used, i.e., if and only if they are not currently bound to any host.
2) When a reply task is invoked and its moreMessages attribute is no, the corresponding partner link is unbound immediately after the message is sent, and the communication channel is closed[1].
3) Processes may explicitly unbind partner links or close their underlying communication channels as needed.

These rules permit processes to reuse partner links efficiently, while allowing process designers to predict and control how this reuse occurs. Note that many BPEL engines, such as Sliver and ActiveBPEL, already exhibit the first two behaviors; but only Sliver (with the extensions presented in this paper) provides the third behavior.

Figure 4 shows two new BPEL activities (close and unbind) which support the third behavior. The close activity allows the process to close a partner link's communication channel, and then unbind the partner link programmatically. The unbind activity unbinds a partner link, but does not close the underlying connection. As we discuss below, these activities are useful for implementing multicast and publish-subscribe communication schemes, which BPEL previously did not support.

### C. Impact of BPEL Extensions

Though these extensions are conceptually simple, they add substantial power to BPEL's existing communication model. In [11], Wohed evaluates BPEL's capabilities with respect to six communication patterns commonly used by collaborative applications. Wohed concludes that BPEL cannot support two of these six patterns: publish-subscribe and multicast. These patterns are especially important in pervasive and mobile applications, since they are the only patterns out of the six which do not require the process designer to specify a fixed set of endpoints.

The BPEL extensions described in this section address this issue. As we illustrate later in Section VI, processes can implement multicast messaging by maintaining a partner group whose membership reflects the set of multicast listeners. When new listeners connect to the process, they are added to this partner group, and the partner link is explicitly unbound using the unbind activity. Because the partner link is unbound after each new listener connects, an unlimited number of listeners can re-use the same link. Disconnection messages are handled in a similar fashion, by using the remove activity to remove the listener from the group, and the close activity to close the corresponding communication channel. Messages then can be sent to all the members of this group using the extended reply activity.

Publish-subscribe behavior can be implemented similarly, by using a different partner group for each kind of message subscription (in the interest of brevity, we omit further discussion). Thus, the extensions proposed in this section support

[1]For the purpose of explanation, we assume in this section that the underlying communication protocol is connection-oriented. In situations where this is not true, attempts to "close" the communication channel are ignored.

the development of sophisticated pervasive and mobile applications which use any of the six important communication patterns noted in [11].

## IV. Sliver Middleware

We have implemented the extensions described above as part of the Sliver middleware. Sliver is a lightweight SOAP and BPEL execution engine designed for deployment on mobile devices. Sliver supports a wide range of computing platforms, ranging from mobile phones to desktop PCs. In this section, we will discuss briefly the design and implementation of Sliver.

Sliver's architecture features several characteristics motivated by the specific needs of mobile devices. First, Sliver provides a clean separation between communication and processing. Communication components can be interchanged without affecting the processing components. This separation allows Sliver to support a wide variety of communication media and protocols, ranging from HTTP to Bluetooth. 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 5. 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 consumes the requests that arrive from the transport layer and routes them to the appropriate processes.

In its current version, Sliver supports BPEL's core feature set and has a total code base of 190 KB including all dependencies (excluding an optional HTTP library). Most applications will only use a subset of Sliver (e.g., because they only use one of Sliver's included transport providers). Hence, Sliver's effective footprint may be reduced by bundling applications with only the components of Sliver which each application uses. This procedure can be automated by post-processing application bundles with utilities such as ProGuard [12], which remove unused bytecode and assign compact names to classes and methods.

In the remainder of this section, we will briefly discuss the design of Sliver's constituent layers. More detailed design information, including sample code, may be found in [4].

### A. Transport Layer

The transport layer is responsible for the transmission of messages produced by the upper layers. Because mobile devices support a wide range of communication media and protocols, Sliver's transport layer uses pluggable communication providers. This design is inspired by the pluggable protocol layers used by TAO [13] and Apache Axis 2.0 [14]. Sliver's transport layer features a streamlined API specifically designed to facilitate the support of new protocol providers. 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 and Bluetooth L2CAP on J2SE and MIDP devices, as well as HTTP on J2SE devices.

### B. 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 [15] and kSOAP [16] packages to parse XML and SOAP documents. These packages are designed with mobile devices in mind: they have a small combined footprint (47 KB of storage space) and operate on most available Java runtimes.

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 analogous to Java packages or C++ namespaces.

### C. SOAP Server

The SOAP protocol provides a standard for service invocation in addition to message encoding. Sliver's `SOAPServer` class implements a SOAP service handler, which dispatches incoming service invocations to the corresponding user-provider service. Again, in the interest of brevity, we do not discuss in detail how these service invocations 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, request messages have an associated type name and namespace, which are used to direct requests to the appropriate service.

### D. BPEL Parser

Unlike standard SOAP services — which are generally stand-alone entities implemented in any of a wide variety of languages — BPEL processes use a standardized XML schema to describe interactions among other SOAP services. Sliver

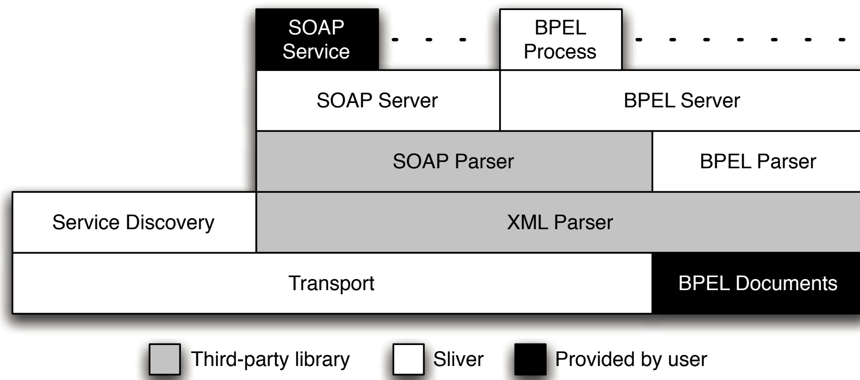


Fig. 5. The architecture of the Sliver execution engine

represents each BPEL tag with a corresponding Java class; e.g., the `reply` tag is represented by the `Reply` class. Each class has a constructor which uses the kXML library described in Section IV-B to tokenize and parse the corresponding part of process descriptions. These constructors make local validation decisions which verify the BPEL document's validity; e.g., the `Reply` class's constructor throws an exception if it encounters a `reply` tag without a `partnerLink` or `partnerGroup` attribute. These local decisions eliminate much of the need for a heavy-weight, fully-validating XML parser library.

BPEL subdivides processes into nested *scopes*; state information (like variables and communication links to Web services) is shared among all the activities within a scope. Sliver maintains all the information known about each scope at parse time (e.g., the types and names of variables) in `ScopeData` objects. A `ProcessInstance` object is also instantiated for each workflow instance; it tracks information which varies across instances (e.g., the values of variables). [4] discusses how these constructs are maintained and used in further detail.

Sliver supports all of the basic and structured activity constructs in BPEL, with the exception of the *compensate* activity, and supports basic data queries and transformations expressed using the XPath language [6]. Sliver also supports the use of BPEL Scopes and allows for local variables, fault handlers, and event handlers to be defined within them.

### E. 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. The `addProcess` method creates a BPEL process in the specified namespace; it reads the processs BPEL specification from the provided InputStream. This method invokes the BPEL parser described above, which encapsulates the entire workflow in a `Process` object.

The `bindIncomingLink` method maps incoming partner links to the kinds of messages that they accept as input. Likewise, the `bindOutgoingLink` method maps outgoing partner links to concrete endpoints. These methods allow applications which embed Sliver to apply a wide variety of partner link mapping policies flexibly, according to the application's needs. Such policies may range in complexity from using simple hard-coded mappings, to dynamically remapping partner links at runtime using the service discovery layer described below.

Sliver's BPEL server layer is able to host a wide variety of useful workflow processes. A framework has been proposed which allows for the analysis of workflow languages in terms of a set of 20 commonly recurring workflow patterns [17]. Sliver currently supports 14 of these 20 patterns in full, and partially supports one other pattern. Even on resource-limited PDA and mobile phone hardware, the cost of executing most patterns in Sliver is on the order of 100 ms. The interested reader may consult [4] for a full performance evaluation.

### F. Service Discovery

In pervasive computing environments, remote services occasionally may connect and disconnect as their hosts move in and out of range. To support such environments, Sliver includes a framework for discovering remote services at runtime using Bluetooth service discovery. The services discovered using this mechanism can be used to update the BPEL server's service bindings programmatically as new service providers are discovered at runtime. Though this framework currently only provides service discovery using Bluetooth, it could be adapted easily to support other discovery protocols, such as Zeroconf.

## V. Over-the-Air Process Deployment

Apart from middleware suitability, software provisioning is another key concern in mobile and pervasive applications. In these environments, interactions among devices may occur both frequently and unexpectedly. It is unreasonable to expect each device to store all of the software that it may ever need, or for device owners to predict which software to deploy ahead-of-time. For example, consider the auction scenario described in Section I. Though the artist will likely have had foresight to

deploy an auction application on his phone, it is unlikely that the art collectors would have deployed a compatible client for this auction application ahead-of-time.

MIDP addresses this issue using over-the-air provisioning (OTA), as described in [18]. Using this mechanism, developers can package software as self-contained applications, which mobile devices can download and deploy over a wireless connection. Unfortunately, MIDP's existing OTA scheme has significant infrastructure requirements which are impractical to fulfill in most pervasive environments. Notably, MIDP requires developers to host their applications on an HTTP server; it also assumes that clients can somehow locate this server at runtime, but does not specify a concrete discovery mechanism. Moreover, MIDP does not permit applications to share code: any common libraries must be duplicated in each application. This policy is especially wasteful in pervasive computing applications, where communication middleware comprises a substantial portion of the codebase, and bandwidth and storage space are limited.

BPEL processes, on the other hand, are highly compact: complex applications can be modeled in a few kilobytes of text. A general-purpose BPEL execution platform would offer considerable storage and bandwidth savings: though the initial cost of deploying the engine would be relatively high, the incremental cost of each additional application would be negligible. Moreover, since BPEL relies on Web standards like SOAP, it would be possible to create a single, general-purpose client for interacting with these applications. As a proof-of-concept, we have created a prototype system which provisions BPEL processes over-the-air using Sliver. In the remainder of this section, we will discuss the key components of this system in further detail.

### A. Process Repository

The first major component of our system is the process repository. This component advertises and distributes user-provided BPEL process files. The repository is exposed to other devices in the local Bluetooth network as a SOAP service. Other devices can connect to the SOAP service using the Bluetooth L2CAP protocol, and invoke methods which list the available processes and retrieve the corresponding BPEL code from the repository.

To support runtime discovery, this SOAP service is also advertised as a Bluetooth service. Bluetooth services are advertised using a well-known 128-bit identifier. This identifier is unique for each *type* of service, but shared across all *providers* of that service: i.e., there is a single 128-bit ID which all process repositories share. Interested devices can locate any number of process repositories in the network at runtime by searching for this 128-bit ID, and then interact with them using the well-known SOAP interface.

Figure 6 shows a screenshot from the process repository application. Using a simple graphical interface, users can browse the local device's filesystem and select which BPEL processes to advertise. As we discuss later in this section, each BPEL process must also have a corresponding user-provided WSDL definition file. On demand, the user can also start and stop the SOAP service which exposes the repository to the Bluetooth network.

### B. Process Server

The second component of the OTA system is the server which executes BPEL processes. This application uses Bluetooth service discovery to compile a list of all service repositories in range, as described above. The user is presented with a GUI which lists all the repositories that were discovered.

Once the user selects a repository to explore, the server application uses the SOAP interface described above to obtain a list of processes stored in the repository. The user is presented this list, from which she may select the process to download. After a process is selected, the server application uses the repository's SOAP service to download the process's BPEL code. Once the BPEL code is downloaded, it is parsed and executed by Sliver's BPEL server. The server application GUI includes a subset of the client GUI described below, so that the user can interact with processes hosted on the local device. This procedure is illustrated in Figure 7.

As we discussed in Section II, the BPEL code does not indicate how to map partner links to concrete endpoints. However, these mappings can be defined using a separate WSDL specification file. Hence, the process server advertises each process's user-supplied WSDL specification along with its BPEL markup. After submitting the BPEL code to the Sliver middleware, the server application parses the WSDL file to obtain the links' mappings. It then programmatically maps these links using the `BPELServer`'s `bindIncomingLinks` method, allowing the process to be deployed without any extra user input. Note that for the sake of simplicity, our server application does not currently handle outgoing link bindings. It is conceptually straightforward to add this support, by using Sliver's service discovery framework to locate external services.

As with the repository service, these processes are exposed to the Bluetooth network as SOAP services and advertised using Bluetooth service advertisements. The process server also creates a separate SOAP service which allows the client to retrieve the WSDL description of processes with which it wishes to interact. Like the repository's SOAP service, this SOAP service has a well-known ID and interface.

### C. Process Client

The final major component of this system is the client application. This client is fully generic: users can discover and participate in workflow processes without any prior knowledge of them. On startup, processes hosted on local servers are discovered using Bluetooth service discovery and are presented to the user in a list. Once the user selects a process from a list, he may select a message type to send to the process and input the message's contents. After the initial message has been sent, any responses from the process are displayed on the screen.

As we described in Section II, BPEL uses SOAP as its underlying message-passing mechanism. This trait allows the

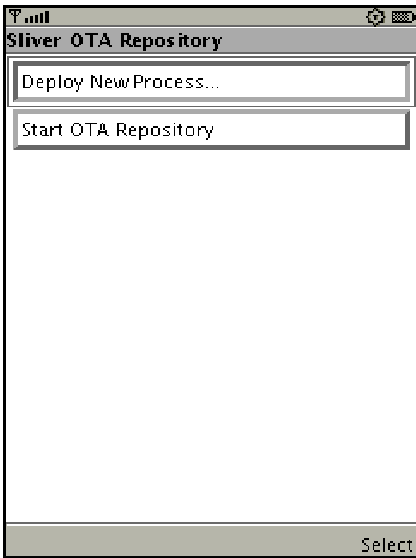


Fig. 6. Sliver Process Repository application

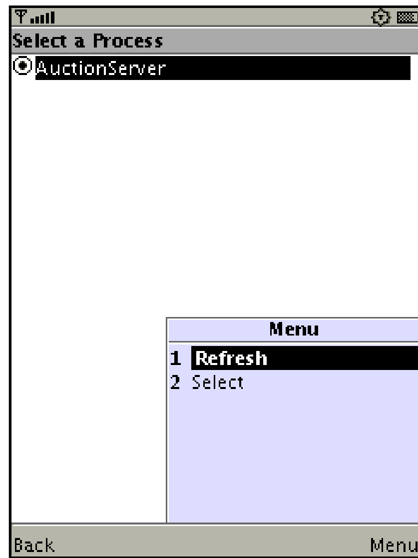


Fig. 7. Sliver Process Server application

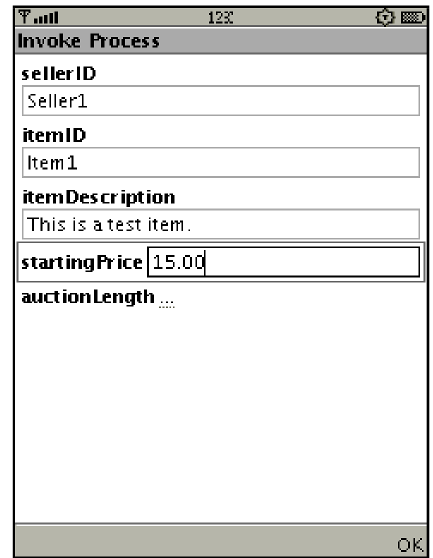


Fig. 8. Sliver Client application

client application to be fully generic. SOAP defines a consistent structural representation for all messages that the BPEL process will produce and consume. Hence, Sliver's SOAP serialization components are suitable for interacting with any remote BPEL process.

However, the client not only must consider the *structure* of the messages, but also their *contents*. That is, SOAP standardizes how to express a message object as structured XML; but it does not mandate which data types are composed to create that message. The latter is expressed using the process's WSDL description, which describes message contents using an embedded XSD schema. Using this WSDL document, the client application can assemble a customized form-style UI on the fly for each process, as is shown in Figure 8.

Unfortunately, WSDL and XSD are complex standards, and mobile devices cannot reasonably be expected to host a full WSDL and XSD parser. We address this issue by deploying a custom WSDL parser, which includes a lightweight XSD parser adapted from Xydra [19]. These two lightweight parsers support a useful subset of WSDL and XSD. Namely, the combination does not support WSDL imports; XSD derived types; or XSD restrictions. All three features are convenient, but are significantly more complex to parse than other parts of the standards. Also, the first two features do not offer any additional expressive power, and could be removed from WSDL documents using a series of static transformations. We feel this trade-off is reasonable: while these omissions may create a slight burden for the process developer, they also make OTA deployment feasible, greatly increasing the processes' utility.

## VI. Sample Application

To demonstrate the effectiveness of our BPEL deployment and execution environment in pervasive computing settings, we have used Sliver to implement the auction application described in Section I. We have deployed this application on a Linux PC using Sun's Java Wireless Toolkit [20], and a Nokia 6682 mobile phone[2]. Each component of our application is distributed as monolithic .JAR file, which has been post-processed with ProGuard in order to reduce bytecode size. In this section, we discuss how this auction application leverages the BPEL extensions described in Section III and the deployment mechanism described in Section V to interact with hosts in a pervasive computing environment. This application, along with the Sliver middleware which it embeds, is available as open-source software at [21].

The auction workflow consists of a 4.5KB BPEL document and a 4.2KB WSDL descriptor. Despite the application's small size, it offers a considerable amount of functionality. Sellers may create a new auction with a user-specified description, price, and length. Interested buyers may submit bids for the auction, or simply request information about the auction's status (e.g., the item's current price). The seller and all buyers are automatically notified when the auction's price changes, and when the auction ends.

The process's functionality and compactness draws heavily from the extensions described in Section III. As new buyers participate in the auction, they are collected into a single partner group. This permits an unbounded number of buyers to participate in the auction, and allows the process to send multicast updates to all buyers with a single line of BPEL code. The process also takes advantage of the ability to specify when partner links close, in order to send an unlimited number of update messages back to the seller and buyers. The code snippet in Figure 9 illustrates the use of these extensions in the auction process. In all, the entire process is modeled in under 130 lines of BPEL code, including abundant whitespace.

[2]This particular application uses the MIDP UI toolkit, which prevents its native deployment on J2SE devices. This is not a limitation of Sliver, which natively supports both J2SE and MIDP devices.

```
<partnerLinks>
  <ext:partnerLink name="buyer" ... />
  <ext:partnerGroup name="buyers" ... />
  ...
</partnerLinks>
...
<eventHandlers>
  <onMessage operation="bid" partnerLink="buyer" ...>
    <sequence name="handleBid">
      <ext:add partnerGroup="buyers"
         partnerLink="buyer" />
      ...
      <ext:reply partnerGroup="buyers"
         operation="bid" variable="auctionStatus"
         moreMessages="yes" />
      <ext:unbind partnerLink="buyer" />
  </onMessage>
  ...
</eventHandlers>
```

Fig. 9. Code Fragment to Send Multicast Auction Updates

Owing to Sliver's compactness and modularity, the entire system can be deployed with low storage space and bandwidth impact. The auction process is initially deployed using the repository application discussed in Section V-A. This repository application requires 48KB of storage space initially, and an additional 8.7KB of space to store the process.

After being deployed on the repository application, the auction process can be discovered and downloaded by the process server application described in Section V-B. The process server application consumes 181KB of storage space, and uses 8.7KB of bandwidth to download the auction process. Once the auction process is deployed, the client described in Section V-C can discover and interact with it. The client application consumes 88KB of storage space, and only needs to download the 4.2KB WSDL file in order to participate in the auction. Though this system is lightweight, it is not specific to the auction application: it can be reused for any other BPEL process stored in the process repository.

Our implementation currently assumes that the repository, server, and clients are all located on different devices. As we discussed in Section V-B, we also assume that the processes hosted on the server will not initiate outgoing connections to other services. These assumptions do not represent inherent limitations of Sliver's design or of BPEL: rather, they were made to streamline the user interface, and could be lifted with additional UI code.

## VII. Related Work

[22] proposes HSN-SOA, a decentralized system for managing networked appliances in pervasive computing settings. Using the *service scenario scripting ($S^3$) editor*, users generate workflow graphs which describe reactive interactions among appliances (e.g., when the user turns on the TV, the TV turns on the speakers and dims the lights). The $S^3$ editor autonomously subdivides these graphs and distributes the plan fragments to the corresponding devices in the local network. After the plan is fragmented and distributed, no centralized coordinating server is required to execute the plan. Though HSN-SOA does not use BPEL, it relies heavily on other Web standards: appliances are exposed to the local network as SOAP services, and their capabilities are advertised using WSDL descriptions. However, HSN-SOA implicitly assumes a stable network, and requires users to describe plans in terms of specific appliances. Hence, it does not address environments where devices are added to or removed from the network after the plans have been written.

[23] describes a prototype system that autonomously generates BPEL workflows in pervasive computing settings. The user issues a description of a goal or task to a Task Selection Service residing on his mobile device. This service invokes a Task Planner Service on a centralized server, which generates and executes a customized BPEL process. In turn, this custom-generated process invokes Web services on the local network, including services which reside on the user's device, in order to carry out the plan. The server also generates a Web front-end to the process, so that the user may monitor its execution from his mobile device using a Web browser. This approach avoids many of the shortcomings in BPEL's communication scheme, since processes can be generated on-the-fly based on what devices are available at the time. However, it requires that a heavyweight centralized server be available to all devices in the network, which may be unreasonable in many pervasive networks. Scalability is also a concern, since all processes are hosted and executed on the centralized server rather than on the mobile devices.

## VIII. Conclusions and Future Work

In this paper, we have proposed a series of BPEL extensions that support the creation of flexible, standards-based pervasive computing applications, even when the devices involved are mobile. As a proof-of-concept, we have used the Sliver middleware to implement a system for deploying, executing, and participating in BPEL processes over-the-air. This system comprises three Java applications ranging from 48KB to 181KB in size, and has been deployed on MIDP-compatible devices. Sliver does not currently support some of BPEL's most advanced features, including *Compensation*. Many of these advanced features are intended for long-running business transactions and will be used rarely in pervasive and mobile applications. Nevertheless, we plan to address Sliver's remaining BPEL compliance issues in future work, and consider ways to further modularize Sliver.

Standardization efforts like BPEL play a significant role in developing pervasive computing applications. These standards are one part of a larger effort to deploy robust collaborative applications in mobile settings. This effort encompasses other important topics, ranging from user interface design to data routing mechanisms, that have yet to be fully resolved. However, standards-based middleware like Sliver demonstrate the feasibility of deploying sophisticated collaborative applications on mobile devices, and offer a concrete platform on which these other challenges can be explored.

## REFERENCES


[1] C. E. Ortiz, "J2ME technology turns 5!" http://developers.sun.com/techtopics/mobility/j2me/articles/5anniversary.html, 2004.

[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] D. Box and et. al., "Simple object access protocol (SOAP) 1.1," W3C, Tech. Rep. 08 May 2000, 2000. [Online]. Available: http://www.w3.org/TR/2000/NOTE-SOAP-20000508/

[4] G. Hackmann, M. Haitjema, C. Gill, and G.-C. Roman, "Sliver: A BPEL workflow process execution engine for mobile devices," Washington University, Department of Computer Science and Engineering, Tech. Rep. WUCSE-06-37, 2006.

[5] E. Guttman, "Autoconfiguration for IP networking: Enabling local communication," *IEEE Internet Computing*, vol. 05, no. 3, pp. 81–86, 2001.

[6] J. Clark and S. DeRose, "XML path language (XPath) version 1.0," W3C, Tech. Rep. 16 November 1999, 1999. [Online]. Available: http://www.w3.org/TR/1999/REC-xpath-19991116

[7] W. Emmerich and et. al., "Grid service orchestration using the business process execution language (BPEL)," *Journal of Grid Computing*, vol. 3, no. 3, pp. 283–304, 2005.

[8] Oracle, "Oracle BPEL process manager," http://www.oracle.com/technology/products/ias/bpel/index.html, 2006.

[9] Sun Microsystems, Inc., "NetBeans enterprise pack," http://www.netbeans.org/products/enterprise/, 2006.

[10] OASIS Open, "Web services business process execution language version 2.0 public review draft," http://docs.oasis-open.org/wsbpel/2.0/wsbpel-specification-draft.html, August 2006.

[11] P. Wohed and et. al., "Pattern based analysis of BPEL4WS," Queensland University of Technology, Tech. Rep. FIT-TR-2002-04, 2002.

[12] E. Lafortune, "ProGuard," http://proguard.sourceforge.net/, 2006.

[13] F. Kuhns and 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, pp. 81–98.

[14] Apache Software Foundation, "Axis 2.0 - axis2 architecture guide," http://ws.apache.org/axis2/1_0/Axis2ArchitectureGuide.html, 2006.

[15] S. Haustein, "kXML 2," http://kxml.sourceforge.net/kxml2/, 2005.

[16] S. Haustein and J. Seigel, "kSOAP 2," http://ksoap.org/, 2006.

[17] W. M. P. van der Aalst, A. H. M. ter Hofstede, B. Kiepuszewski, and A. P. Barros, "Workflow patterns," *Distributed and Parallel Databases*, vol. 14, no. 1, pp. 5–51, 2003.

[18] C. E. Ortiz, "Introduction to OTA application provisioning," http://developers.sun.com/techtopics/mobility/midp/articles/ota/, November 2002.

[19] O. Chipara and A. Slominski, "Xydra — an automatic form generator for web services," http://www.extreme.indiana.edu/xgws/xydra/, 2003.

[20] I. Sun Microsystems, "Sun Java wireless toolkit for CLDC," http://java.sun.com/products/sjwtoolkit/, 2006.

[21] G. Hackmann, "Sliver," http://mobilab.wustl.edu/projects/sliver/, 2006.

[22] M. Nakamura and et. al., "Implementing integrated services of networked home appliances using service oriented architecture," in *Proceedings of the 2nd International Conference on Service Oriented Computing (ICSOC '04)*, 2004, pp. 269–278.

[23] A. Ranganathan and S. McFaddin, "Using workflows to coordinate web services in pervasive computing environments," in *Proceedings of the 2004 IEEE International Conference on Web Services (ICWS 2004)*, 2004.