

# Sensor-enhanced Mobile Web Clients: an XForms Approach

John Barton (John\_Barton@hpl.hp.com)

Tim Kindberg (timothy@hpl.hp.com)

Hui Dai (huid@cs.colorado.edu)

Nissanka B. Priyantha (bodhi@copley.lcs.mit.edu)

Fahd Al-bin-ali (albinali@cs.arizona.edu)

Hewlett Packard Labs  
1501 Page Mill Road  
Palo Alto California, USA 94304

## ABSTRACT

This paper describes methods for service selection and service access for mobile, sensor-enhanced web clients such as wireless cameras or wireless PDAs with sensor devices attached. The clients announce their data-creating capabilities in "Produce" headers sent to servers; servers respond with forms that match these capabilities. Clients fill in these forms with sensor data as well as text or file data. The resultant system enables clients to access dynamically discovered services spontaneously, as their users engage in everyday nomadic activities.

## Categories and Subject Descriptors

H.3.5 [Online Information Services]: Web-based services. H.5.2 [User Interfaces] Input devices and strategies. C.2.4 [Distributed Systems]: Client-server, distributed applications.

## General Terms

Design, Standardization

## Keywords

Mobile Computing, Ubiquitous Computing, Sensors, Browsers, Forms, MIME types.

## 1. INTRODUCTION

Currently some mobile phones come equipped with digital cameras and color displays for sharing images. This mobile phone/display/camera device and its use model is a special case that can be generalized. The camera is a particular type of *sensor*: a device that reads data from the user's physical environment. And the users may submit their sensed data — the images — over the wireless connection to only a limited set of services that are the same everywhere the user goes: they send the images to other users or upload them to a web site. The general case has much more potential than the camera-enhanced phone. First, the camera

could generalize to several types of sensors. Second, the services for processing the sensed data could be open: in addition to existing services, users could send them for photo printing, image processing, optical-character recognition, or whatever services arise on the Internet. Third, services could be specialized to the user's location or circumstances.

Consider how to utilize a world of open services specialized for a nomadic user's physical environment, services that process values from a variety of sensors integrated into a range of devices that the user carries. How can a user of a device with a digital camera load images into a photo printing service they encounter while traveling? How can this user send the same image to their digital display at home? Or to the digital display in a friend's house? Will the answer change when the user's device includes global positioning values, the ability to co-record temperature, orientation, and the values in nearby electronic beacons and object identifiers [18] such as bar-codes that may be read via a camera or a standard scanner? Will the answer change when the photo-printing service or digital displays change? In general, how can a user with a wirelessly networked, mobile, digital, data-capture appliance interact with electronic services in a way that works in many different places, and as devices and services change over time?

These questions arose from our effort to develop a system for supporting "nomadic computing" [19]. We have in mind a system of the future with advanced wireless digital cameras, "badges" with sensors [22], PDAs equipped with cameras [13] or position and orientation devices [21], and so forth, carried by people as they work, play, or shop. These future nomads use these devices routinely in a variety of services associated with places and other entities in the physical world, which they discover and utilize as they move around. For example, they could be recording notes from a meeting with work colleagues, then in a store uploading a photograph of a chair that might look good in their home, then helping to construct a record of their child's school project.

These activities are technically realizable today. However, such a realization would be difficult or expensive even if the hardware and communications infrastructure was already in place. While all the uses involve both data capture and communication, the destination of the data and its compatibility with the sensors varies. Actual use would require, at one extreme, specialized interworking of the destination service and sensors or, at the other

extreme, global uniformity so that all sensors and services fit a common model. Neither solution can be realistic for nomadic users, who interact with different systems, in different places, and with devices that change over time.

## 1.1 Contribution and scope

In this paper we explore an HTTP- and XForms-based solution to the problem of nomadic service discovery and service invocation. We'll call our networked, mobile, digital data-capture appliance a "sensor-enhanced web client", thinking of it as a new kind of input to web servers while minimizing the differences from present-day browsers. Conceptually a "sensor" captures data, but often the word sensor conjures up simple measuring devices and the term "sensor network" has come to mean large numbers of simple devices. We use the term in a broader sense, to include cameras (image sensors), bar-code scanners, RFID readers and so on. We'll restrict our electronic services to Web-based services, meaning that the primary communications channel will use HTTP over TCP/IP. We describe the protocol that allows sensor-enhanced handheld web clients to upload data to spontaneously discovered location-dependent services. We also describe our implementation that allows the potential for these clients to be explored.

Nomadic, sensor-enhanced computing is our main interest, but our web-centric approach has the potential to benefit other users as well. A desktop PC might also be equipped with sensors such as bar-code readers and it certainly can run a similar web client to the one we put on a mobile device. Sensors are not the only sources of data that a web client can supply to a service — why not data from applications, such as entries from address books? And while we normally consider there to be a "human in the loop", a ubiquitous computing environment might include devices that supply sensed values such as temperatures to services autonomously.

## 2. TYPE-DEPENDENT DISCOVERY

Our approach begins with two analogs from the current Web:

1. Multimedia content negotiation on the Web allows clients to notify servers of their media capabilities. These servers interact with a wide variety of clients and gracefully move to improved media technologies over time. The HTTP `Accept` header drives the simplest version of this content negotiation.
2. Humans with keyboards interact with server-side applications via HTML forms. This model allows a vast variety of applications to be fronted by Web-based user interfaces and those UI's can change dramatically between applications and over time.

Those considerations lead us, first, to make the linchpin of our service selector the same one used by the Web, i.e. MIME types. Second, they lead us to retain the simplicity of the `Accept` paradigm by straightforwardly reversing it: to think of services as sinks of standard MIME (media) types and sensors as sources of standard MIME types.

Our solution adapts these two aspects to nomadic handheld clients with sensors.

1. Our client announces to services that it can provide particular types of data by supplying a `Produce` header that parallels the `Accept` header of HTTP 1.1.

The service can, for example, reply with a web page containing links to services that can consume these types. These links return forms, but the fields in the form are not (just) text or file fields. They also accept multimedia data that the client can supply from its sensors.

2. Thus the second part of our solution: form-based support for MIME-typed data upload from sensors in devices. Subsequent use of the device, for example taking a picture, loads the form with data matching the MIME type, and sends it to the service, resulting in, for this example, the upload of an image. In filling out forms, users are not concerned with intermediate files: as far as they are concerned, data flows directly from sensors into forms and thus to Web services and applications.

The `Produce` header works in tandem with the existing `Accept` header for devices that include both sensors and a display or other media playback capabilities. Moreover, the form model for uploading sensed data allows the client to access heterogeneous services without deployment of service-specific client code. These services need not be hosted in general-purpose server machines; they might be services like printing or projecting embedded in devices [20]. Our aim is to support either extreme equally.

Concretely we propose:

1. a new HTTP header, `Produce`
2. an analysis of the W3C XForms [26] candidate recommendation applied to sensor data from mobile clients.

These two items are covered in the next two major sections of this paper. Then we discuss our implementation that allows the implications of these proposals to be explored.

### 2.1 The Produce Header

The `Produce` header mirrors the `Accept` header in form but not in function. Like the `Accept` header, its purpose is to specify a set of MIME types, but ones that the client can produce instead of consume. Unlike the `Accept` header, its role is not *media type* selection but rather *service* selection. Depending on the use model, the content that the client receives when it does an HTTP POST or GET with a `Produce` header is either a form that a service has selected for the client based on its capabilities (assuming at least one match was found), or a page giving links to services that match the client's capabilities. For example, a wireless camera that can produce images of type `image/jpeg` retrieves an upload form for the user's Web album when the user points it at *myPortal.com*. But when the user points the same camera at the dynamically discovered home portal of their friend's house, they see (on the back of the camera) links to the digital picture frames and printer in that home.

Note that utilizing the `Produce` header may be only the first step in service selection. In general, we expect a returned service page to show customization, quality, and pricing options. The user can then select from among the options or seek another service with the required data-consuming characteristics. The `Produce` header prefilters the service but the user makes the selection.

```

Produce          = "Produce" ":" #( media-range [ produce-params ] )
media-range      = ( "*"/*" | ( type "/" "*" ) | ( type "/" subtype ) ) *( ";" parameter )
produce-params    = ";" "q" "=" qvalue *( produce-extension )
produce-extension = ";" token [ "=" ( token | quoted-string ) ]

```

Figure 1. Syntax of the proposed Produce header

## 2.2 Syntax of the Produce Header

The syntax of the Produce header is shown in Fig. 1. This specification of the Produce Header is isomorphic to the Accept Header defined in the HTTP /1.1. Using the convention defined in the HTTP protocol, the "\*" character is used to refer to a set of media types, e.g. "\*"/\*" indicates all the defined media types and "foo/\*" means all the subtypes of the type "foo". The media-range describes the types of data that the client device can produce. The device may have the ability to produce either a single type or a set of types.

The requester can specify preferences between media types through specificity and "qvalues". As in the Accept header definition in HTTP /1.1 [9], "the media-range can be overridden by more specific media ranges or specific media types. If more than one media range applies to a given type, the most specific reference has precedence." For example:

```
Produce: image/*, image/jpeg, */*
```

has the precedence (1) image/jpeg, (2) image/\*, (3) \*/\*. As with the Accept header, the "quality factor" parameter allows the client to indicate relative preferences with a weighting, "q", between 0 and 1 (the default). For example:

```
Produce: image/*;q=0.2, image/jpeg;q=0.6,
        image/gif;level=1, */*;q=0.5
```

would cause the following values to be associated:

```
image/gif;level=1    image/jpeg = 0.6
image/bmp           = 0.2    image/png = 0.5
```

Another example, for a digital camera, is:

```
Produce: image/jpeg; q=0.6, image/jpeg2000;
        q=0.4, video/quicktime
```

This camera prefers to produce quicktime video (with a default q value of 1). Otherwise, it prefers to send a jpeg-encoded image but can also produce the jpeg2000 format.

## 2.3 Evaluating Service Precedence with the Produce Header

Consider a set of services, each of which requires a vector of data of one or more media types to be submitted. Each service is associated with the equivalent of a compound Accept header specifying one or more required and/or acceptable media types, each with an effective qvalue. For example, a service might require both (video/quicktime) and (image/jpeg, image/\*; 0.2). That is, it requires a video of type video/quicktime in addition to an image, preferably of type image/jpeg.

We rank the "preference" of the different services against the Produce header from the point of view of the service

providers; we ignore the requester's Accept headers for the sake of simplicity in the explanation. Depending on the use model, the preference could be used to select the "best" service or to send back a ranked list to the requester.

1. Check the appropriate Produce header field. If no Produce header is found, terminate with a "no available service" indication. Otherwise proceed to step 2.
2. For each service, if the requester does not produce one or more media types required by the service, eliminate this service. Otherwise:
  - a. For each media type accepted by the service entity and contained in the Produce header, multiply the quality factor from the Produce header with the quality factor for this media type specified for the service, and choose the largest result if there are alternatives. Add all the values together. The final value is the final preference factor for this service.
  - b. Add this service to a list, sorted by preference factor.
3. The algorithm ends with a list of the available services that are sorted by preference.

We expect that an algorithm like this would be included in any standard for a Produce header.

## 3. XFORMS FOR SENSOR DATA

Once we have the ability to select a service appropriate to the data-creating sensors of our device, we need a mechanism to couple the service and the sensors. Form filling is the web-based method used to couple keyboard input from humans in to services. Therefore we chose to explore extensions of this approach for sensor-enhanced devices.

In a web-based service-access model, a client downloads a "form" that has two roles: it contains markup for presentation on the client device, including controls and related information; and it has *fields* that accept data values from the client device, which may include values from sensors as well as more conventional values such as text entered by a human. Once the form has been completed, it is submitted to the URL it specifies.

To realize form-based service access, we explored the utilization of the proposed XML-based form standard called XForms [26]. This choice seem to be the lowest-effort way for these capabilities to be added into the W3C protocol suite. We considered working with conventional HTML forms, but XForms overcomes several limitations of HTML forms valuable for mobile sensor systems.

We describe the XForms advantages by example. The XML fragment shown in Fig. 2, to be embedded in the head of an XHTML document, solicits one or two images and a comment on the images:

```

<xforms:model xmlns:my="http://purl.org/net/TimKindberg/xmlns/imageUpload">
  <xforms:instance>
    <my:data>
      <my:image/>
      <my:comment/>
    </my:data>
  </xforms:instance>
  <xforms:input ref="my:data/my:comment">
    <my:caption>Add a comment</my:caption>
  </xforms:input>
  <xforms:bind ref="my:data/my:comment"
    required="true()"
    relevant="/my:data/my:image = true()" />
  <xforms:input ref="my:data/my:image">
    <caption>Snap an image or two</caption>
  </xforms:input>
  <xforms:bind ref="my:data/my:image" required="true()" maxOccurs="2" />
  <xforms:submission action="http://example.com/submit" method="post" />
</xforms:model>

```

**Figure 2. Part of an XML document illustrating the Xforms elements**

The XML specifies a logical model for the items of data to be collected, the "controls" used to collect the data, and constraints on the data collection. This is textually separate from presentation markup in an XForms-based document, and both model and presentation are in turn separate from the instance data to be collected and submitted to the "action" address. The example shows an `instance` XML element, which defines the values that are to be filled in and submitted. XForms authors also specify abstract "controls" such as the example's `input` element, used for obtaining values for an instance of the form. XForms authors can also use the `bind` element to specify constraints that the form-filling client must interpret and satisfy when values are filled in.

The following are the main XForms features in relation to the sensor-enhanced clients we are considering:

**Separate presentation logic.** Since we are dealing with client devices of various types and sizes, form input widgets should be tailored appropriately to the device's capabilities—rather than, as in HTML, rendering them with a standard appearance. XForms already provides for this, allowing the client to render abstract controls with appropriate concrete representations. The example's `input` element for the comment may be rendered as any text-input widget appropriate for the device. That might be a keyboard or a microphone used to record speech that is then converted to text.

**Constraint constructs.** Xforms `bind` elements provide functionality not found in HTML forms, such as specifying whether a given field is optional or not, and the type of data that may be filled in. In general, constraints enable the client to give feedback to the user about invalid entries, without the time delay of a return trip to the server for validation. In the example, the `bind` element refers to the comment field and one of the specifications is that comments are required. The user-agent (web client) should thus insist on this field being filled rather than wastefully submitting an invalid empty field to the server.

**Selective rendering of input widgets.** The "relevant" constraint enables the form's author to schedule the filling of different fields in a form through preconditions. This is useful when dealing with devices of limited rendering capabilities, and when seeing controls for operating several sensors concurrently might confuse the user.

In the example, the comment field becomes relevant only when the image field evaluates to `true()`—that is, is non-empty.

**Multiple instance values per field.** Clients sometimes need to capture several values—for example, a set of images—whose number is unknown in advance of form-filling. XForms supports multiple instance entries per field, with constraints on the allowed number. In the example, the user is allowed to enter up to two images.

### 3.1 Sensor-filled fields

In our devices, form fields may be filled in with data directly captured from sensors—in addition to data from files and data entered by humans. Furthermore, sensed data should be specified by media type only—not by sensor type. In our prototype implementation we extended the values allowed in the "type" attribute in the `bind` element of XForms to MIME media types. As we explained in the above example, the `bind` element specifies constraints on input data. But the current specification allows only types from the XML Schema [26], and XForms-specific types for list items and duration values. We allow MIME media types [10] as type values. In the following we augment the example above by specifying that the field whose instance data is "my:data/my:image" is to be filled with a value of type "image/jpeg":

```

<xforms:bind ref="my:data/my:image"
  required="true" maxOccurs="2"
  type="image/jpeg" />

```

(Note this is not XForms-compliant, see below.) Our client interprets the `bind` element and associates the MIME type with

the referenced instance element. Then when the `input` element is rendered, the MIME type binding causes the client to render this input control as an imaging sensor dialogue rather than, for example, a keyboard input widget.

This approach satisfies our most important requirement: by requiring a media type rather than specifying a device type, services are freed of concern with which sensors the client possesses. For example, if the form requires `image/jpeg` then a client might supply an image from a camera or a scanner or whatever image sensor it has available. The form should not specify "camera", since the client may not have a camera but may have another image-producing device for which the service still has value. Therefore, the point of agreement between a client and server is the media type—which, by hypothesis, belongs to the extant MIME standard—rather than a device description, for which there is much less agreement on vocabulary and semantics. We expect the range of sensors producing any given media type to increase, leading to more and more lost opportunities for service access unless agreement is by media type rather than device type.

This MIME-based approach naturally leads to a number of issues in the connection between the media type and sensors. Because the service's specification for a given field is only the required media type, the client might be able to choose which of possibly several sensors produce data of that type. Consequently the client will have to provide a means of activating the appropriate sensors. If the client has a user interface, then the choice can be offered to the user with a selection widget. Otherwise, it can activate any suitable sensor.

Another issue is that the types of sensed values relevant to nomadic computing go beyond those in the MIME set. For example, to our knowledge there is no MIME type corresponding to the value from a Global Positioning System (GPS) sensor, or for the value from a bar-code reader.

Finally, feedback is an important issue. Users may want to review the sensed values that they have filled into the form, just as they can see the values they have entered into text widgets before clicking the "submit" button. Fields filled from sensors should be rendered appropriately, but only when the user wishes to examine them, because of the limited output capabilities of many clients. We shall return to these points in the final discussion.

Our work proceeded concurrently with the evolution of the XForms specification. Recent versions of the specification contain an "upload" control that allows for input of data from a sensor. The specification uses this example

```
<upload ref="mail/attach1"
mediatype="image/*">
  <label>Select image:</label>
</upload>
```

This syntax has a similar effect as our *ad-hoc* approach; the upload control would be rendered into a sensor-activation widget. However, the XForms specification introduced a new element rather than trying to overload the form `input` element, whereas we experimented with the simpler approach of type extension—a method that would be applicable to uploading data from files as well as sensors. Both approaches suffer from potential ambiguity: a form could be created with two bind elements or two upload elements referencing the same point in the XML instance. One alternative without this problem would place the MIME type

specification in the `model` element of the form as we suggested in an online comment on XForms [2].

## 3.2 New issues with Sensors

**Client-Side Coordination.** We considered two aspects of coordination in form-filling with sensors. The first is that there is sometimes a requirement to fill several fields at the same time. For example, an image upload service might present its clients with a form containing one field for the image and the other for a location, e.g. from a GPS unit attached to a camera, so it can annotate the picture with where it was taken. The location field should be filled in at the time the image was taken, because doing so later or earlier might produce misleading results. Although we have not yet implemented this facility, the first type of coordination seems to be achievable through the XForms "relevant" constraint, which would enable activation of an appropriate sensor when a target field has been filled in.

The second aspect of coordination is that sometimes it is convenient to tie field-filling to submission. For example, to save a user from an extra interaction with awkwardly small controls on a PDA, a form could specify that it was to be submitted as soon as an identifier had been read — e.g. from a bar-code. The user's interaction model, once the form has been downloaded, is then "scan and view": the user presses a button to scan a bar-code on, say, a product and then looks at the resultant Web page that the server provided. The XForms specification 1.0 does not appear to support this second type of coordination.

**Multi-client form processing.** Multiple mobile devices should be able to contribute to the filling of a single form. This roughly corresponds to various approaches to "multibrowsing" [12, 17]. A simple approach is to pass a partially filled-in form between multiple clients with different capabilities. For example, a form could be passed between several personal devices such as a camera and a PDA, each of which fills in some subset of fields in the form. Indeed the conceptual simplicity of this approach leads to the suggestion that, ideally, a submitted form should also be form, albeit one with, in general, more fields filled in than when it arrived at the client. Then the form becomes at all stages a completely self-describing data structure, not just at the XML level but also at the markup level; "at all stages" may even include form processing among multiple services acting as "clients" to other services. The XForms 1.0 specification would not allow this as clients submit only instance data, without a model or enclosing markup.

## 4. IMPLEMENTATION

To explore our proposal and to gain some experience in its potential we have implemented a set of sensor-enhanced web clients and some form-based services.

### 4.1 Sensor Enhanced Web Client Simulation

To develop our software rapidly and to be able to examine a wide variety of application scenarios without overcoming the complexities of actual sensors in this prototype, we used a prototyping tool for ubiquitous computing called Ubiwise [4]. This program simulates the visual appearance of electronic devices and their physical environment using a combination of two programs, one to show devices and one with a 3D model of the use scenario. While the sensors, device screens, and controls are simulated, the web client and web service code is not. Consequently this approach aids in the development of the client-



Figure 3. Screenshot of a simulated digital camera about to sense a bar-code to find location-specific services.

server connection, but it allows only rudimentary experiments on user interface design.

## 4.2 Scenario

To demonstrate our implementation, we simulated a nomadic computing scenario. In our scenario, a user arrives in a room she has never visited before, with her web-browser enhanced camera equipped with bar-code conversion software. She uses that device:

1. to obtain links to the services in the room that she can take advantage of with her client device, and
2. to access services that work with the sensed values she can produce.

Both service discovery and service access take place via forms. While we don't know of a digital camera with these capabilities currently, there are commercially available PDAs and phones that can read bar-codes via a camera attachment [15]; such software saves the bulk and expense of a specialized scanner.

## 4.3 Bootstrapping Service Discovery via the web/id Application

To begin our nomadic user needs to find services available in the space she has just entered. She uses her bar-code scanner to discover the available services via a "you-are-here" bar-code placed by the entrance to the room (see Fig. 3). This bar-code is a "physical hyperlink": the code values can be mapped to a web page with the "web/id" service [18]. The user selects the web/id service URL from her bookmarks. Her client fetches the page in a GET operation that specifies the Produce header "image/jpeg; text/id" (we made up the latter for this experiment). This particular implementation of web/id ignores the first media type but uses the second to return a form specifying a field to be filled from an text/id value. In the absence of the Produce header, a form with a simple text input widget for the

identifier would have been sent and the user would have to read the code off the wall and enter it manually.

The user's device renders the web/id form into a tiny web browser on back of the camera. The browser, while too small for typical web pages, can show simple text like "Snap an ID for Local Information" (see Fig. 4). On seeing web/id's form, the user snaps an image of the "you are here" bar-code. The camera software images the bar-code, recovers the identifier, fills the resultant identifier (a string) into the form and submits the form back to web/id. Web/id takes the identifier and looks up the URL that the room's administrators have bound to it: the URL of a "service selector" page giving a customized list of the available services. Web/id redirects the client to that page.

## 4.4 Service selection and service access

At this point in our scenario, the user has a web client requesting a list of services. Again the Produce header, `image/jpeg; text/id` is added to the request. The service selector examines the client's capabilities from the Produce header. It sends back a page that is customized to the client, including, along with more general information, links to a digital picture frame and a printer in the room, both of which can consume `image/jpeg`. It omits the audio service that would be able to play audio from the client through speakers if the client were able to produce it.

## 4.5 Produce Header Analysis

The server side of our system has a new web-based selector meta-service for analyzing Produce headers. The analysis is needed in two cases: for a service that returns forms for sensor fill-in and for a directory service that lists those services. Both cases share the code for analyzing the Produce header itself; the latter case also requires a registry of the form-based services. We implemented our example services as Java Servlets running in the Apache/Tomcat servlet engine. The registry is currently a "service map" file directly analogous to Apache's type-map file [1].



Figure 4. Screenshot of the back of a simulated bar-code-reading camera equipped with a miniature web browser.

#### 4.6 Forms Proxy

We implemented a sensor-enhanced web client by interposing a "form-proxy" on the client between a standard web browser and

web services (see Fig. 5). The form-proxy acts in part as a web proxy. It intercepts XForms that the browser downloads from services, renders them into HTML to send on to the browser, and it coordinates the filling of those forms. To fill the forms, the

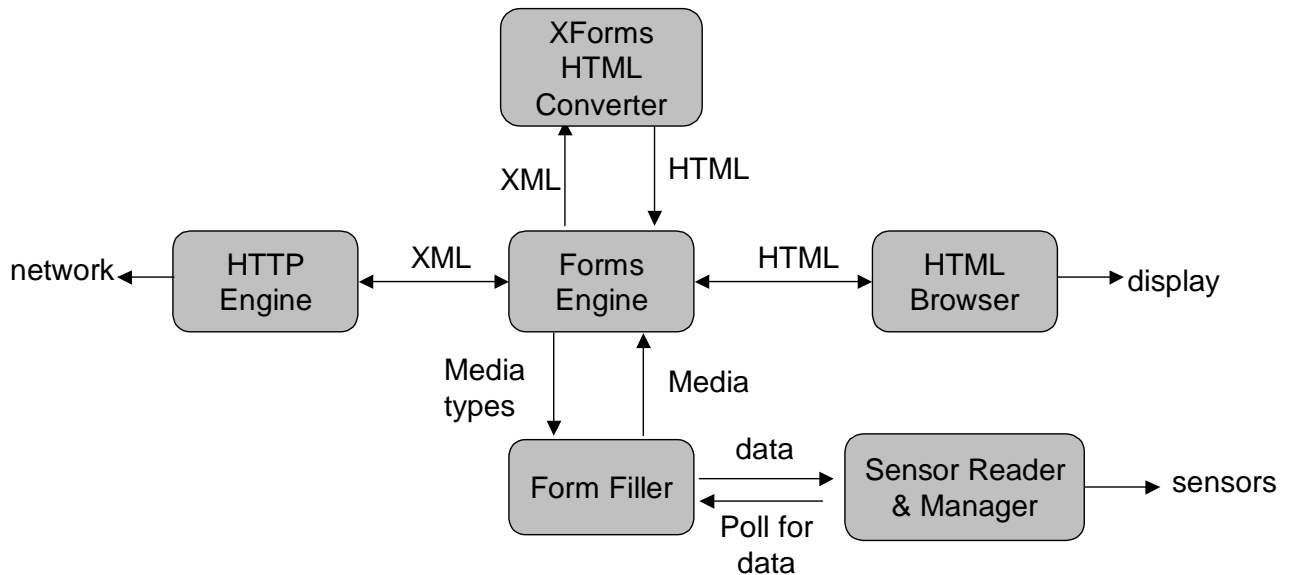


Figure 5. Client-side transforming XForms proxy. The Form Engine acts as an event dispatcher to coordinate the other components. Requests from the HTML browser are sent to the HTTP engine. If HTML comes back it is sent to the browser; if XML comes back it is sent to the Converter. If the Converter finds XForms, the media types requested are sent to the Form Filler to (possibly) activate sensors. HTML comes back from the Converter and is sent to the browser for user interactions.

form-proxy activates sensors and intercepts data sent back from the browser. The form-proxy is written in Java and it has been used with both desktop HTML browsers and the Java based browser in the Ubiwise simulator.

Our implementation includes an abstraction of sensors wherein each is associated with the media type it can produce and each is classified as "blocking" or "non-blocking". For example, a human-operated camera is a blocking sensor since the human has to click the shutter. A GPS sensor, on the other hand, potentially always has a value to produce and is classified as non-blocking. Client implementers have to examine the type requirements for a given form and determine how to map the sensors:

- A form that maps to a set of non-blocking sensors should be rendered with a submit button to trigger the sensor measurements and immediate subsequent upload.
- A form that maps to a set of sensors that includes one blocking sensor could render without a submit button, allowing the operation of the blocking sensor by the user to trigger both sensor measurements and upload. This model would allow a digital appliance-like operating mode where the user is simply using the device without detailed attention to the web page rendered for the form.
- A form that maps to more than one blocking sensor requires a more complex user interface design. Note that this is the normal case for current Web HTML forms with keyboard inputs.

For our simple scenario we have two forms, one for web/id and one for uploading an image to a digital picture frame. Both have a single blocking sensor, the imaging sensor. In our first implementation, both forms were submitted after the user takes a picture. Thinking of ourselves as users, this worked well for the web/id step: we selected that service and want immediate results. However, for image transfer from the camera to the digital picture frame, immediate transfer when the image is capture may or may not be the behavior users expect. Our experimental set up allows us to explore approaches for how web clients can gracefully adapt to these cases in future work.

## 5. RELATED WORK

Service discovery systems such as those of UDDI [23], Jini [16], SLP [11], and UPnP [24] enable clients to discover services based upon arbitrary attributes. In principle, these discovery services allow for more fine-grained service selection than we have described. In practice, such services have proved to have little utility except in largely static—that is, non-nomadic—environments. The level of agreement on vocabulary, syntax and semantics required between the developers of clients and the developers of services is too onerous, especially as the set of possible services (and thus the set of possible attribute values) is a rapidly moving target. Moreover, detailed service specification tends to defeat nomadic interoperation through lost opportunities: if the user's wireless camera searches automatically for a "printer", then it does not thereby search for the new type of image upload service that someone will invent tomorrow, even though in principle it could take advantage of it.

Sensing systems include specialized applications in which sensors are configured to transmit their data to a specific custom service or application. This is the approach used in surveillance systems for example. Aiming for greater applicability in ubiquitous

computing environments, Dey et al [6] describe a "Context Toolkit" for processing sensed data in the acquisition of application context. Their toolkit has a specialized purpose but uses HTTP to maximize interoperability.

Architecturally the critical preceding work for us is the HTML/HTTP system underlying the Web and the peer-to-peer data-transfer protocol called Jetsend that was supplied in a number of HP products.

As the analysis in Fielding [8] argues, HTTP supports distributed hypermedia through a design optimized for requests moving from clients to servers and data returning in replies. HTML Forms added the ability for clients to reply to services with data. With HTML pages of links acting as directories and web crawler/search engines providing service discovery, the Web system for spontaneous interactions was created.

Jetsend [25] allowed data to be transferred between digital "appliances", that is peer devices with wireless connectivity and consumer electronics interfaces. Implementations sold in HP products used IrDA for the wireless transport. The Jetsend datatype system resembled MIME types in having sets of functionally similar types like "image/\*" containing only a few widely used types like "image/jpeg" or "image/tiff". The Jetsend content-negotiation focused on data transfer like we do with our Produce header. However, Jetsend was a push-based protocol with the content negotiation driven by the client and no markup language or hypertext was involved.

In previous work [19], we investigated HTTP-based protocols for "content exchange", whereby the nomadic user can push content to services in the pervasive infrastructure from their portable device. In that work, the nomadic user selects a service such as a printer, which supplies its interface as an HTML form; the user fills out the form and sends the form and the content that they wish to upload from their sensor as multipart MIME encoded data over HTTP. This type of spontaneous interaction with dynamically discovered services enables the nomadic user to interact with services without the need to reconfigure their device when they enter new environments. But there is no support for client- (sensor-) specific service selection; and as we observed above, HTML forms do not have sufficient structure to meet the requirements of multimedia data upload.

The SpeakEasy project [7] has goals similar to ours for spontaneous interoperation between devices. However, rather than being web-based, SpeakEasy uses mobile code to implement user interfaces and transport-protocol endpoints, which are migrated between the service and the client. Users do not have to reconfigure their devices in order to take advantage of a dynamically discovered service; however, they do have concerns about the security and resource challenges that mobile code raises, which have yet to be overcome, especially for portable devices. Applets have been relatively unsuccessful on the desktop web, while HTTP continues to serve us well.

## 6. DISCUSSION

Up to this point we have the tools to explore the potential for direct interaction between mobile sensors and spontaneously encountered services based on web technologies. Our first experiments indicate that the approach has promise as well as a number of open issues and directions that would have to be pursued to make reach our goal.

On the client side we have already found three areas that require deeper examination:

1. how to allow media types to be extensible and yet sufficiently standard,
2. how client software can map media types to sensors in ways that users find effective, and
3. how client software can allow review before submission when users want it but be also easy to use when they do not.

The first issue is systemic and in part lies outside of any one designer's ability to affect. The web-based approach already allows extensibility by not mandating any particular media types beyond some lower bound like `text/xhtml`; it already points to a set of standards through the MIME registration. Missing in the XForms model now however is a means for identifying when input that can be in character text (object identifiers, GPS coordinates, orientations, temperature, and so forth) should be held in self-describing XML or in MIME-like prescribed standards.

The other two issues concern client designers. For mobile sensor-based web clients these designers are more likely to be digital device engineers than software engineers with a background in web-browser design. We hope our work will simulate more studies to guide these designers in future.

On the service side, we can take the `Produce` header idea in two directions. First we intend to provide a formal proposal for the `Produce` header syntax to appropriate standards bodies. Then we can develop a module for web servers like Apache that makes `Produce` header analysis easily accessible to developers. Finally we can embed the `Produce` header module in a meta-service that acts as a directory for form-based services. This meta-service could have front-end content-adapters for the variety of sensor-based appliances and back-end adapters for the variety of other services that list services, such as UDDI.

Systemically we have the interesting possibility of lifting forms to be "first-class" objects sent between clients and between services. The current asymmetric form model with forms arriving at clients and data returning means that the service that supplied the form is the only one that can interpret the reply. Indeed it may be quite difficult for even an expert in the service to debug form fill-in failures. If forms were themselves submitted to services, with the blanks filled it, then they would be self-describing. This is exactly the model adopted in most paper-based form systems whenever the technology allows.

Every client-server system has issues of privacy and security. Since our proposal extends XForms and parallels the `Accept` header, these issues will be easily analyzed by analogy to existing practice.

Finally, we have implemented client and service components that enable experimentation with the form paradigm, as our proof-of-concept application for image-upload shows. In other directions we can explore services for building web-site content [14] like on-line merchandising. With just web/id and image upload we can implement a service for "physical registration" as described in [3], which builds a list of physically-based services in an environment, like printing, projecting, displaying, and communicating through operations with handheld sensor clients. Our implementation, which runs on the desktop and uses simulations of mobile devices,

also enables investigation of the other use-models we described in our introduction: desktop vs. mobile, application-data vs. sensed data, and autonomous vs. human-supervised form fill-in. Many of the necessary mechanisms exist; now it is a question of pursuing the open issues we have identified.

## 7. References

- [1] Apache HTTP Server Version 1.3 "Module `mod_negotiation`", [http://httpd.apache.org/docs/mod/mod\\_negotiation.html](http://httpd.apache.org/docs/mod/mod_negotiation.html).
- [2] Barton, J. "Markup Support for Sensor Data Upload in Forms", [http://www.hpl.hp.com/personal/John\\_Barton/XFO\\_RMs/ExtendingInputElement.htm](http://www.hpl.hp.com/personal/John_Barton/XFO_RMs/ExtendingInputElement.htm), 2001.
- [3] Barton, J., Kindberg, T., and Sadalgi, S. "Physical Registration: Configuring Electronic Directories using Handheld Devices", IEEE Wireless Communications, Vol. 9, No. 1, Feb. 2002 and HP Labs Technical Report HPL-2001-119.
- [4] Barton, J. and Vijayaraghavan, V., "UBIWISE, A Ubiquitous Wireless Infrastructure Simulation Environment". HP Labs Technical Report HPL-2002-302, and [http://www.hpl.hp.com/personal/John\\_Barton/ur/ubiwise/](http://www.hpl.hp.com/personal/John_Barton/ur/ubiwise/).
- [5] Biron, P. and Malhotra, A, Eds., "XML Schema Part 2: Datatypes", <http://www.w3.org/TR/xmlschema-2/>.
- [6] Dey, A., Salber, D., and Abowd, G. "A Conceptual Framework and a Toolkit for Supporting the Rapid Prototyping of Context-Aware Applications". Human-Computer Interaction (HCI) Journal, Volume 16 (2-4), 2001, pp. 97-166.
- [7] Edwards, K., Newman, M., Sedivy, J., Smith, T., Izad, S. "Recombinant Computing and the Speakeasy Approach". In proceedings MobiCom '02, ACM, Sept. 2002, pp. 279-286.
- [8] Fielding, R. "Architectural Styles and the Design of Network-based Software Architectures", PhD Thesis, Univ. Calif. Irvine, 2000, [http://www.ics.uci.edu/~fielding/pubs/dissertation/to\\_p.htm](http://www.ics.uci.edu/~fielding/pubs/dissertation/to_p.htm).
- [9] Fielding, R., Gettys, J., Mogul, J., Frystyk, H., Masinter, L., Leach, P., and Berners-Lee, T., "Hypertext Transfer Protocol — HTTP 1.1". RFC 2616, June 1999.
- [10] Freed, N. and Borenstein, N., "Multipurpose Internet Mail Extensions (MIME) Part Two: Media Types", Request for Comments: 2046, November 1996.
- [11] Guttman, E., Perkins, C., Veizades, J., and Day, M., "Service Location Protocol, Version 2", Internet RFC: 2608. 1999.

- [12] Han, R., Perret, V. and Naghshineh, M., "WebSplitter: A Unified {XML} Framework for Multi-device Collaborative Web Browsing", in Computer Supported Cooperative Work, p21-230, 2000.
- [13] HP Cambridge Research Lab's "Project Mercury" <http://crl.research.compaq.com/projects/mercury/>, and MIT's Project Oxygen "Handy 21 Project" <http://oxygen.lcs.mit.edu/H21.html>.
- [14] Huang, A., Ling, B.C., Barton, J., and Fox, A., "Running the Web Backwards: Appliance Data Services", Proceedings of the 9th International World Wide Web Conference, May 2000, pp. 619-631 and HP Labs Technical Report HPL-2000-23.
- [15] International Wireless, <http://www.mitigo.com/> (as of Feb. 2003).
- [16] Jini <http://www.jini.org/>.
- [17] Johanson, B., Ponnekanti, S., Sengupta, C., Fox, A., "Multibrowsing: Moving Web Content Across Multiple Displays", Proceedings of Ubicomp 2001, September 30-October 2, 2001 and [http://graphics.stanford.edu/papers/mb\\_ubicomp01/mb\\_cam2.pdf](http://graphics.stanford.edu/papers/mb_ubicomp01/mb_cam2.pdf)
- [18] Kindberg, T., "Implementing Physical Hyperlinks Using Ubiquitous Identifier Resolution", Proceeding of 11th International World Wide Web Conference, July 2002.
- [19] Kindberg, T. and Barton, J., "A Web-Based Nomadic Computing System", Computer Networks, Elsevier, vol 35, no. 4, March 2001, and HP Labs Technical Report HPL-2000-110.
- [20] Kindberg, T., Barton, J., Morgan, J., Becker, G., Bedner, I., Caswell, D., Debaty, P., Gopal, G., Frid, M., Krishnan, V., Morris, H., Pering, C., Schettino, J., Serra, B., and Spasojevic, M., "People, Places, Things: Web Presence for the Real World", MONET Vol. 7, No. 5 (October 2002) and HPL-2000-16.
- [21] Pradhan, S., Brignone, C., Cui, J.-H., McReynolds, A., and Smith, M., "Websign: Hyperlinks from a Physical Location to the Web", IEEE Computer special issue on location-based computing, August, 2001 and HP Labs Technical Report HPL-2001-140.
- [22] Smith, M., and Maguire Jr., G., "SmartBadge/BadgePad version 4", HP Labs and Royal Institute of Technology (KTH), <http://www.it.kth.se/~maguire/badge4.html>, date of access 2002-11-15.
- [23] UDDI Home page. <http://www.uddi.org/>.
- [24] Universal Plug and Play <http://www.upnp.org/>.
- [25] Williams, P. "JetSend: An Appliance Communication Protocol". In proceedings IEEE International Workshop on Networked Appliances, IEEE IWNA '98, IEEE Press, Kyoto, Japan, November 1998.
- 26] XForms homepage <http://www.w3.org/MarkUp/Forms/>.