Running the Web Backwards: Appliance Data Services

Andrew C. Huang*, Benjamin C. Ling*,
John J. Barton, Armando Fox*
Internet Systems and Applications Laboratory
HP Laboratories Palo Alto
HPL-2000-23
February, 2000

web,
input,
device,
infrastructure

"Appliance" digital devices such as handheld cameras, scanners, and microphones generate data that people want to put on Web pages. Unfortunately, numerous complex steps are required. Contrast this with Web output: handheld web browsers enjoy increasing infrastructral support. We hypothesize that the utility of input appliances will be greatly increased if they too were "infrastructure enabled". Appliance Data Services attempts to systematically describe the task domain of providing seamless and graceful interoperability between input appliances and the Web. We offer an application architecture and a validating prototype that we hope will "open up the playing field" and motivate further work.

Our initial efforts have identified three design challenges: device heterogeneity, user interface, and harnessing infrastructure services. Our architecture isolates device and protocol heterogeneity considerations into a single extensible architectural component, allowing most of the application logic to deal exclusively with Web-friendly protocols and formats. To distribute the user interface between the network infrastructure and user devices, we tag input with commands specifying how the data is to be manipulated once injected into the infrastructure. Additional conceptual contributions include canonicalization and late binding of these commands to applications, two mechanisms that improve the user experience by allowing "natural" extensions of the device's UI to be used for application selection and minimize the amount of configuration required before end-users benefit from Appliance Data Services. Finally, we describe how services can be applied to process data, using simple HTTP connected conversion services as examples; we could also leverage services connected via Jini or CORBA. We also describe an implemented prototype of parts of the architecture and a specific application.

Internal Accession Date Only
* Stanford University, Stanford, CA 94305

Running the Web Backwards: Appliance Data Services

Andrew C. Huang, Benjamin C. Ling, John J. Barton*, Armando Fox
Stanford University and *Hewlett-Packard Laboratories

Abstract

"Appliance" digital devices such as handheld cameras, scanners, and microphones generate data that people want to put on Web pages. Unfortunately, numerous complex steps are required. Contrast this with Web output: handheld web browsers enjoy increasing infrastructural support such as user-transparent transformation proxies, allowing unmodified Web pages to be conveniently viewed on devices not originally designed for the task. We hypothesize that the utility of input appliances will be greatly increased if they too were "infrastructure enabled." Appliance Data Services attempts to systematically describe the task domain of providing seamless and graceful interoperability between input appliances and the Web. We offer an application architecture and a validating prototype that we hope will "open up the playing field" and motivate further work.

Our initial efforts have identified two main design challenges: dealing with device heterogeneity, and providing a "no-futz" out-of-the-box user experience for novices without sacrificing expressive power for advanced users. We address heterogeneity by isolating device and protocol heterogeneity considerations into a single extensible architectural component, allowing most of the application logic to deal exclusively with Web-friendly protocols and formats. We address the user interface issue in two ways: first, by specifying how to tag input with commands that specify how data is to be manipulated once injected into the infrastructure; second, by describing a late-binding mechanism for these command tags, which allows "natural" extensions of the device's UI for application selection and minimizes the amount of configuration required before end-users benefit from Appliance Data Services. Finally, we describe how to leverage existing services in the infrastructure; our prototype is based on HTTP and Java but our architecture could also leverage services connected via Jini or CORBA. We also describe an implemented prototype of parts of the architecture and a specific application.

Keywords

information appliances, infrastructure enablement, ubiquitous computing, Internet services

---

1 Background, Motivation, and Challenges

Much recent work has focused on accessing the Internet from "post-PC" devices, especially "information appliances" such as PDA's, cell phones, and palmtop computers [FGCB98]. Surveying that work, we see that these devices, despite their inherent hardware, software, and network limitations, can interoperate with the rest of the Internet through infrastructure support. In particular, software such as transformation proxies [FGC+97,Pro97,FGG+98] and wireless protocol gateways [Met95, WAP97] enable these devices to leverage the enormous installed infrastructure of servers, content, and interactive services. In the words of one mobile computing project [KB+96], "*Access* is the killer app" for such devices. We capture this effect by saying that the post-PC devices have become more useful because they are now *infrastructure enabled*. Conversely, and partially as a result of infrastructure enablement, the Internet has begun to adapt to these devices and we are now seeing services tailored for their use, such as Yahoo Mobile and a variety of sites that feature "Palm-friendly" pages in addition to their desktop content. These new mobile computing devices are no longer isolated "islands" of computation; they are participants in an Internet system.

1.1 Infrastructure Enablement for Input Centric Devices

Almost all of the work on device access to the Internet has been focused on devices as web-browsers. We can say that it is focused on output from the web. Our goal is to achieve infrastructure enablement for digital input-centric consumer appliances; we believe web-like infrastructure provides a good model for this enablement. Thus we aim to "run the web backwards", adding infrastructure to support input from portable input-centric digital devices.

By input-centric we refer to devices whose primary function is not to extract and browse digital information, but to create it. Examples include digital still cameras and video cameras, handheld scanners, and portable audio recorders. In part because of the pervasiveness and success of the established Internet infrastructure, much of the data created by these devices ends up in the Internet infrastructure, e.g. posted on Web sites. However, although much has been done to simplify the process of *extracting* information from the Internet to a variety of devices, the process for injecting data *from* devices into the infrastructure is extremely painful. We propose a framework and software components for facilitating this process, to stimulate further work that will allow input-centric devices to enjoy the same success as information browsing appliances enjoyed by becoming infrastructure-enabled. We call the resulting system "Appliance Data Services."

1.2 Contributions

The three main contributions of this paper are a systematic identification of the problem space, an architecture for the infrastructure-enablement of input appliances, and an implemented prototype of parts of the architecture. Our discussion of the problem space was informed by experiences with existing web-input approaches and the process of formulating the architecture and prototype described here. The architecture attempts to reconcile the diversity of devices and protocols to be supported, the desired flexibility in

selecting from a wide range of available Web applications, and the requirement of an unobtrusive "no-futz" user experience in operating the device and connecting it to the Web. In particular we identify *command tagging* as a fundamental requirement for Appliance Data Services (ADS), describe an implemented approach to tagging that is extensible to a range of devices and communication protocols, and propose *command canonicalization* and *late binding* as specific architectural mechanisms to address the user experience concerns. We describe our proposed framework in light of the technical and usability challenges it addresses, describe our experience with a working prototype, and use it to motivate further research.

Our prototype implementation sketches all the elements of ADS so that we can experiment with an end-to-end prototype. We tried to avoid dependency on a particular software framework for the application logic; as we describe, a number of such frameworks are either commercially available or under development as research projects, and we wish to enable interoperation with as many of these as possible. The architectural mechanisms we describe can be implemented in the context of any of these frameworks.

The remainder of this section motivates the problem and identifies design challenges through a simple scenario, and introduces command tagging as a fundamental requirement for ADS. The second section describes the architecture and our implemented prototype, identifying ADS as a particular application domain for *infrastructure software frameworks* such as Jini [AOS+99] and Ninja [GW+99], which we describe briefly. We conclude by describing our proposed extensions, lessons learned from the initial prototype, and discussion of work in progress.

1.3 Running the Web Backwards

Imagine that we have taken a photo with a digital still camera and that we would like to publish the photo on the Web. Many current digital cameras are equipped with infrared transceivers, which can be used to communicate photo data to another IR-equipped device such as a laptop or desktop PC. In an ideal world, we would point the camera's IR at the Internet-connected laptop or desktop, press a button to "squirt" the photo data out of the camera, and sit back while infrastructure-deployed software reads the image, converts it to a Web-compatible encoding, authenticates itself to a remote server using your credentials, posts the image file there, and arranges for an HTML link on a designated "photo album" page to point to it.

The reality is considerably uglier, as we found when we helped a California elementary school publish their "Science Fair" results on the Web. We photographed each of 156 student posters with a digital camera and scanned one key item from each poster using a handheld scanner. To upload the data to the Web, we pointed the IR transceivers on the camera and scanner to an IR-equipped laptop connected to the wired Internet. Unfortunately, we had to install and learn to use different IR receiving software for each device. Furthermore, the uploaded images were only identifiable through serial numbers embedded in their file names on the laptop, leading to manual file copying and filename manipulation to coordinate the two data streams. The scanner's TIFF files had to be manually converted to JPEG for Web publishing. Finally, the school's Web server allowed only FTP write-once access, so that changes after uploading required manual intervention by a site administrator.

Taking these obstacles together, it is not surprising that the Web has so far failed to magnify the impact and usefulness of these input devices as it has for (e.g.) PDA's and cell phones. Small scale attempts such as the PhotoNet and Cartogra web sites are attempting to simplify this problem, but they provide only a solution for a very specific application, namely the publishing of photos to a site-maintained album in one of a limited set of presentation formats. We would like a more general solution that

1. handles devices other than cameras such as handheld scanners and digital audio recorders,
2. allows the user to specify what happens to the data (how it is manipulated) once it enters the infrastructure, providing an easy-to-use interface for the novice user without sacrificing expressive power for the advanced user, and
3. is able to coordinate the input from multiple devices.

Such an infrastructure would make it possible to have photos automatically routed to "the science fair application", which would combine them with scanner images from the project's conclusion and add them in a predefined format to the Science Fair website. Selected photos might also be routed to the user's personal photo album, perhaps maintained on a site such as PhotoNet or Cartogra.

1.4 Design Challenges

Recording the science fair was painful for two reasons. First, at a low level, appliances do not understand Internet standards such as HTTP and HTML, although this is slowly changing. Second and more importantly, at a higher level, there is no deployed application infrastructure that automatically performs the data transformations, protocol conversion, and data routing in a way that is largely transparent from the end user's point of view.

This observation leads us to identify two specific design challenges for deploying software infrastructure to address this problem:

1. Dealing with device heterogeneity: This challenge involves being able to handle the various data formats and protocols chosen by device vendors (e.g., JetSend, IRDA, and cradle synchronization). This aspect is further complicated when we want to be able to merge multiple data streams from different devices, as in the "Science Fair" scenario. In addressing this challenge, we leverage significant prior work on dealing with heterogeneity in the context of information delivery to post-PC devices [FGCB98]. As in the prior work, it is important that we provide an extensible solution that will generalize to other devices and protocols.
2. Providing "No-futz", out-of-the-box operation. Task-specific devices such as digital cameras have quite limited user interfaces by design: unobtrusive and familiar interface metaphors, such as the "point and shoot" metaphor for a camera, make the devices easy to use. However, it seems intrinsic to this task domain that users need to be able to specify what should happen to the data they are "injecting" into the infrastructure. The challenge is how to provide a flexible mechanism for specifying this without adding obtrusive features to the device UI, or while accommodating devices like digital microphones whose UI's may be essentially nonextensible. As a corollary, we want to enable devices to exhibit some reasonable "out of the box" behavior with zero user configuration, to allow new users to immediately begin experiencing the value of combining the device with Internet services.

Our answer to these challenges is an Appliance Data Service application architecture

2 Architecture

Before describing the architecture in detail, we clarify our use of two specific terms: "infrastructure" and "service framework".

By "infrastructure" we refer to the deployed collection of hardware and software accessible directly or indirectly via an Internet (usually Web) programmatic interface. In addition to supporting "destination applications" such as web content, Web-accessible services (banking, etc.) and Web-accessible databases, the infrastructure includes "faceless" application components. For example, most Web users never interact directly with Web content caches or accelerators, but they are deployed throughout the infrastructure and programmatically accessed by destination sites. As another example, many portal sites that provide a search or query feature implement the feature by performing the search on a remote machine operated by a different vendor, and reformat the results in HTML for presentation to the user. In other words, the Internet infrastructure is, roughly speaking, the collection of all code that implements or support Internet sites, and the software and hardware mechanisms by which they communicate.

By "service framework" we refer to any programmatic framework for deploying new services in the infrastructure. An example of a somewhat ad-hoc framework that emerged early in the Web's development is the collection of mechanisms used for Web site intercommunication and code execution: HTTP, CGI-bin, SSL handshaking, etc. In this framework, each HTTP-accessible web service can be considered a "service module", and the modules communicate via HTTP and SSL. Various frameworks under development in academic research and industry, including Jini [AOS+99], Ninja[GW+99], ChaiServer [chai] and HP e-speak, feature a richer set of programmatic abstractions. Typically a service framework provides three sets of mechanisms:

1. The ability to compose services, either by writing a meta-service that calls each subservice, or by providing a way to name and execute a pipeline-like composition of services;
2. an inter-service communication mechanism (HTTP, Java Remote Method Invocation, etc.) that may provide specific features such as authentication, security, etc.;
3. a registration/discovery service that tracks which services are available and allows services to be looked up by attribute value.

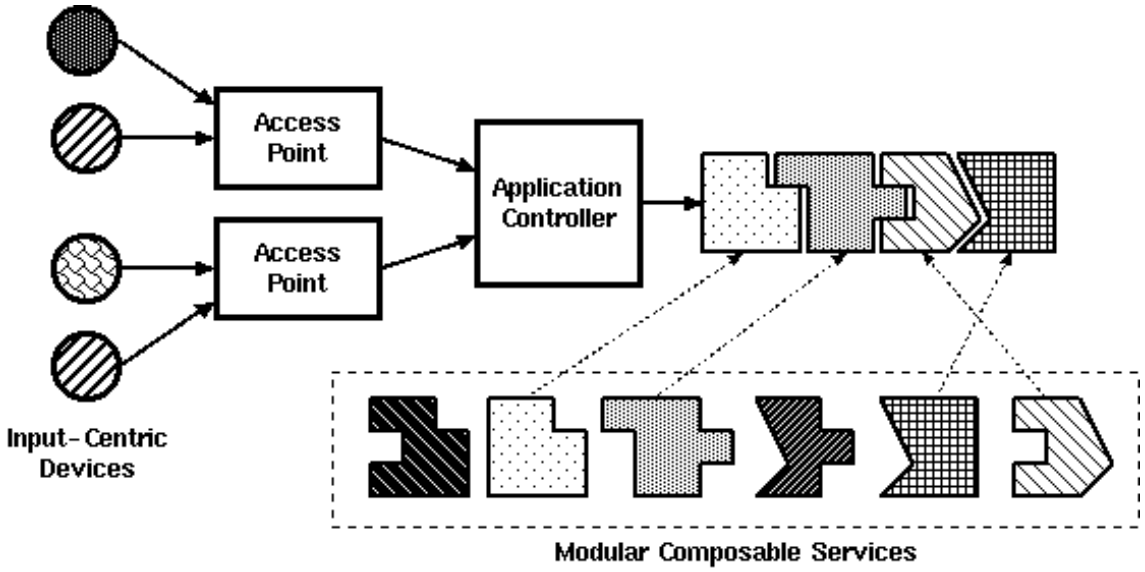


Figure 1: Conceptual diagram of the ADS architecture. Devices and services are joined through Access Points and the Application Controller.

The ADS architecture, shown in Figure 1, was developed to address the challenges outlined in the previous section. The envisioned use of such a framework begins with a user transferring data into the system via an *Access Point*, a network-connected hardware/software gateway that receives data from the user's appliance. The Access Point passes the data, user identifier and command-tag associated with the data to an *Application Controller*, which determines how the data is to be handled. The Application Controller activates the necessary infrastructure-resident *Modular Composable Services* that process and store the data. Each of these components mentioned in this usage model is justified and described in detail in the following sections.

2.1 Access Point

The Access Point is the point of entry for data that is to be pushed into the system. It consists of necessary hardware and software to receive data from appliances. Hardware might include an IR transceiver, RF basestation hardware, or a cradle or cable for "docking" an appliance. Software includes both hardware-specific drivers and the Access Point functionality described below. The Access Point could be implemented as a commodity PC outfitted with the appropriate hardware interfaces, or it could be designed as a special-purpose "network appliance".

The main architectural role of the Access Point is to isolate device heterogeneity considerations in a single architectural component. By presenting a device-independent interface to the user and to the rest of the system, the Access Point converts a potentially large configuration space of handheld digital input devices into a small number of server-digestible data types. To allow *every* wireless device to send data without manually loading device-specific software on the receiver or receiver-specific software on the device, the Access Point must be extensible. Although we have deliberately avoided a design that assumes that appliances can run Java (to avoid

an artificial dependence on a not-yet-deployed technology), a mechanism such as Jini that supports downloading of communication protocol code into the Access Point could be explored to provide such extensibility.

Along with the actual data, the Access Point must obtain a *user identifier* and *command-tag* from the device and attach these to incoming data in a client-protocol-specific manner. These metadata are necessary for a number of reasons:

1. Application selection: The command-tag names the high-level application that the user wants to perform on the data (e.g., "Send picture to my public_html directory"), using a binding mechanism described later. However, the command-tag alone is not sufficient to define the application since different users may have different meanings for the same tag, or result in different semantics in the interpretation of the tag (e.g. "My web site" maps to different URL's for different users). Thus, a user identifier is required to fully specify the desired application.
2. Access control: The user identifier is also required to determine what credentials are to be attached to the application request. For example, a user should be allowed to push data into her own public_html directory, but not necessarily those of other users. Furthermore, some services may be accessible only to authorized users. Thus, the system needs some identifier to attach credentials to the request.
3. Other service features requiring a command-tag and user identifier include billing, security, and personalization. Although we have not investigated the implementation of such features, we have left the user identifier as a necessary "hook" for adding these capabilities later.

A simple example of a command tag is text metadata that results from the user choosing a particular menu item on the device. A more sophisticated tag is possible with recent models of digital cameras that allow the embedding of audio-coded metadata in each image. In the latter example, the Access Point receives images from the camera and extracts the audio metadata for use as the command-tag. We also hope to explore merging of command-tag inputs from one device with data from another device.

Some devices have such limited user interfaces that there is no graceful way to specify a different command-tag with each device input. In this case, the Access Point attaches the special command tag "default" to the incoming data. We describe later why this is important architecturally.

2.2 Service Modules and Application Controller

Once the typed data, command, and user identifier are available from the Access Point, they are handed off to the application infrastructure for application execution. In our system the "application" relies on standalone or composable Internet service modules; for example, an image format translation service might take an image and some parameters as input, and deliver the image in a different encoding. Since ADS is an application architecture and not a service framework *per se*, our architecture *does not define* which service framework should be used to construct and execute the application. We believe there are good engineering reasons to construct the application out of composable building blocks, but nothing in our architecture requires this. Architecturally, it suffices to distinguish one service module that acts as an ADS *application controller*, which must have the following functionality:

- Command-tag canonicalization: Since the command-tag can be one of several MIME-types, format conversion may be required to convert it into a simple string. For example, an audio file may be sent through an audio-to-text conversion service to produce a tag the Controller can use. Note that a composable-modules infrastructure that transforms data can be used to transform command-tags as well, without any modifications to the architecture. This concept is discussed in further detail in the following subsection.
- Command-tag resolution: the canonicalized command-tag is used to select a high-level application, by lookup in a *command database*. The representation of the application to be selected (URL naming a server, list of service modules to compose, etc.) is necessarily framework-specific. We describe later the XML-based "work order" representation used in our prototype.
- Application execution: instantiating the application is, again, framework-specific. For example, in a composable modular framework, execution involves matching the data input types to the needs of the application, finding a composition of available services such as format conversion and data storage that satisfies the data type constraints, and invoking the service modules.

We re-emphasize that there are many possibilities for executing code in the infrastructure, and we do not prescribe any particular method. Depending on the mechanisms used, the Application Controller may be a separate service module, or it may simply be a designated entry point into a piece of monolithic code in a larger service module, perhaps executed directly by an HTTP server as a CGI-bin script. However, the two Application Controller tasks of command canonicalization and command resolution are fundamental to our architecture, so we describe these in more detail before moving on to the description of specific mechanisms used in our implemented prototype.

2.3 Command Canonicalization

The reason that command canonicalization and late binding are fundamental concepts in the architecture, although the implementation media may vary, is that together they effectively address the problem of command-tagging without complicating the user experience. We first describe how each of the two architectural mechanisms works as shown in Figure 2, and then give examples illustrating how they support an unobtrusive user experience.

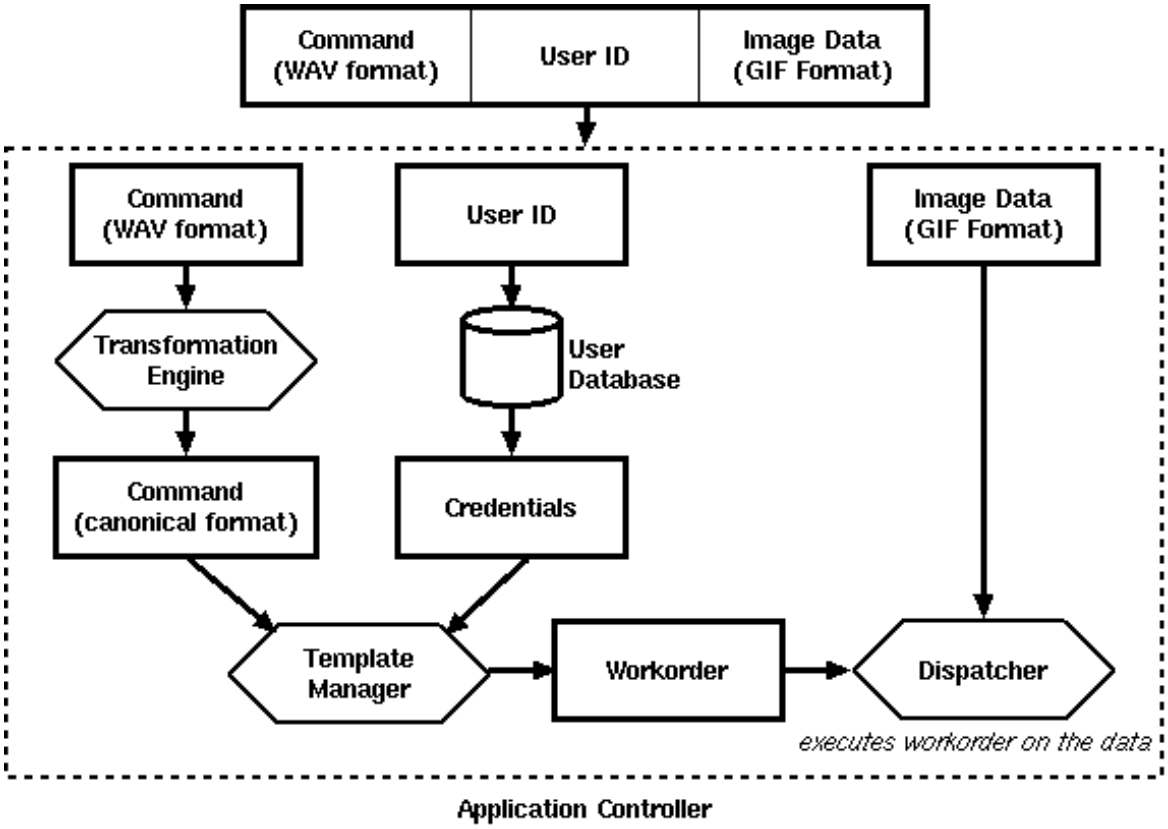


Figure 2: A diagram of command transformation using late binding.

Much recent work on information delivery (web output) has focused on sophisticated data transformation services in the infrastructure. We have already described some ADS applications that make use of data transformation. Less obviously, but advantageously from the point of view of providing natural extensions to device UI's, we can also leverage the transformation infrastructure to transform command tags into a canonical form. For example, recent digital camera models such as the Kodak DC265 allow the embedding of audio-coded metadata in each image, in the form of a short audio clip. We can use an infrastructure speech-to-text service (operating against a fixed and very limited size vocabulary) to transform a spoken command word or phrase into a string, which is then looked up in the template database.

Command transformation is appealing because it decouples the method used to specify commands from the resolution of commands for application selection. Command transformation potentially allows each device's UI to be extended for command-tagging in the way that is most natural for that device.

2.4 Command Resolution Using Late Binding

Commands are "bound" to application descriptions in a separate template database, itself accessible via an infrastructural (e.g. Web) interface. This Web-accessible database is used to resolve a command into a script template ("work order" in our prototype, but machinery may vary according to the infrastructure framework used). Late binding of commands in a separate database contributes to a "no-futz" user experience in at least two ways.

First, users can change or add command behaviors by modifying the database directly through a familiar Web interface. Even if an appliance is Web-configurable (e.g. using vendor-supplied software), centralized configuration frees users from having to configure each device independently. In fact, we envision third-party template databases that free the average user from worrying about how to construct new behaviors. Late binding therefore expands the repertoire of potential behaviors available to services, without burdening each device with the obligation of supporting a UI flexible enough to distinguish among all available commands, perhaps presenting only a subset at any given time.

Second, by modifying the binding of the special command tag "default", a user can modify the behavior of data coming from devices with non-extensible UI's. Thus, for digital cameras that provide no convenient metadata mechanism in which a command can be embedded to accompany a photo, the user can simply re-bind the default behavior to a new application in the command database, which has the same result as redefining the camera's (non-configurable) behavior.

3 Prototype Implementation

This section describes the prototype implementation of the components shown in Figure 3: Access Point, Application Controller, and Modular Services.

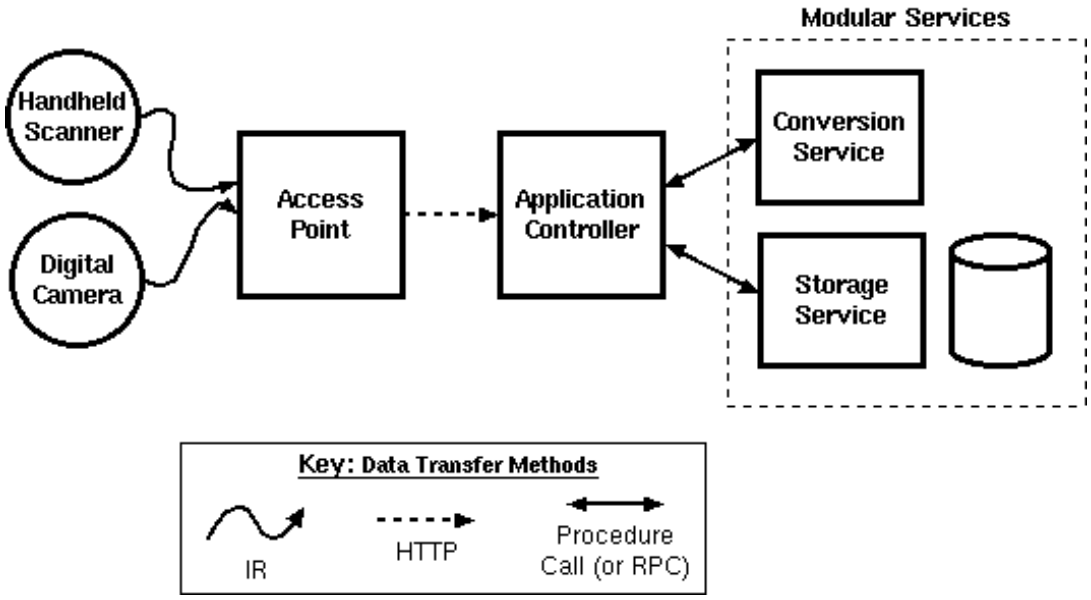


Figure 3: ADS prototype implemenation.

To make the description more concrete, the components are described in the context of the implemented application, a web site displaying attendees of a conference. This application involves taking pictures and scanning business cards of conference attendees to be published on the conference Web site.

Using the ADS system, the user takes a picture of each person attending the conference and scans the attendee's business card. The user then goes to an ADS Access Point, points the camera to the IR port, pushes a button on the camera to transfer the picture, and does the same with the handheld scanner. After repeating this process for each attendee, the appropriate data has been injected into the system. Within ADS, the appropriate transformations are applied and the resulting files are stored on the conference Web site. The end result is a page containing all the attendee's pictures and business cards.

3.1 Access Point

This section describes the implementation of the Access Point and the role it plays in the ADS system. The Access Point is separated into two modules, in order to decouple its two main concerns: device heterogeneity and management of hard state. These two modules, shown in Figure 4, are:

- Information Daemon (InfoD): interacts directly with devices using device-specific adaptors.
- Aggregator: receives data and command-tags from the InfoD, stores a user's pending data, and dispatches service requests upon receipt of a user's command-tag.

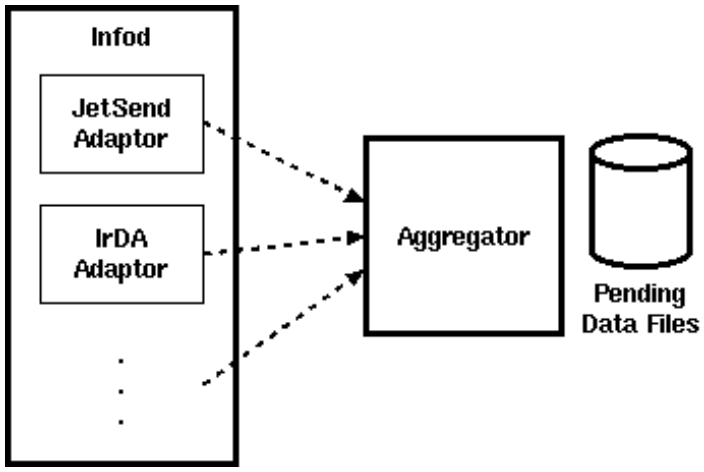


Figure 4: Two components of the Access Point: Information Daemon and Aggregator

How these components -- devices, InfoD, and Aggregator -- interact is better understood in the context of a sample application such as the conference attendee list.

After the user has taken a picture of an attendee, the user points the camera's IR port at the InfoD. In the prototype, the InfoD is implemented on an IR-equipped laptop running Windows 98. The user then pushes a button on the camera to transfer the data into the

system. The InfoD and camera do the necessary handshaking to begin transmission, and the data is passed from the camera to the InfoD. To accomplish this, the InfoD has various device adaptors that handle communication with the appropriate devices. Currently, the InfoD supports digital cameras, handheld scanners, PalmOS, and WinCE devices that use Hewlett-Packard's JetSend IR data transfer protocol.

The InfoD as we have described it is stateless and configuration-less, which makes it appealing to deploy as (e.g.) a publicly available kiosk or Web-centric service. In fact, the InfoD looks very much like a reverse Web-browser, its role being to read device-specific data and write MIME-typed data streams to the rest of the system.

When the InfoD receives the data and extracts the relevant metadata, it uses HTTP POST to send the information to the Aggregator. The aggregator is implemented as a Ninja service running on a Linux workstation. Using HTTP allows the InfoD to include whatever meta information is available. However, the Aggregator does require the following headers:

- Content-Type
- Content-Length
- X-User-Id
- X-Data-Class

Using HTTP POST as the API between Device Adaptors and the Aggregator provides more flexibility than a strongly-typed API. First of all, the HTTP POST API allows the InfoD to send any metadata it receives from the device without knowing what information the Aggregator can handle. Secondly, this API facilitates backwards compatibility. New adaptors can send newly defined parameters in headers or multipart MIME, while old adaptors continue sending the same information as before. Thus, only the Aggregator, rather than all adaptors, needs to be changed to handle both the new and old API.

The Content-Type and Content-Length headers are standard HTTP headers that are used to describe the data. The X-User-Id field is used by the Aggregator to associate the incoming data with the correct user; furthermore, as mentioned earlier, this field is required by other system components to fully define the command-tag received. Finally, the X-Data-Class field is used to differentiate actual data (e.g., jpeg image, text file) from command-tag data.

For each piece of data received (e.g., Data-Class = "data"), the Aggregator archives the entire POST request. When a command-tag arrives, denoted by "X-Data-Class: cmd-tag," the Aggregator attaches the command-tag to each piece of data pending for the current user. Each triple is then sent in an HTTP POST request to the Application Controller.

3.2 Application Controller

The Application Controller receives data from the Access Point Aggregator and executes the requested service command on the data. This component, shown in Figure 5, contains the following three modules:

- HTTP Frontend: receives data posted from the Access Point
- Dispatcher: invokes services on the data
- Template Manager: manages the template database

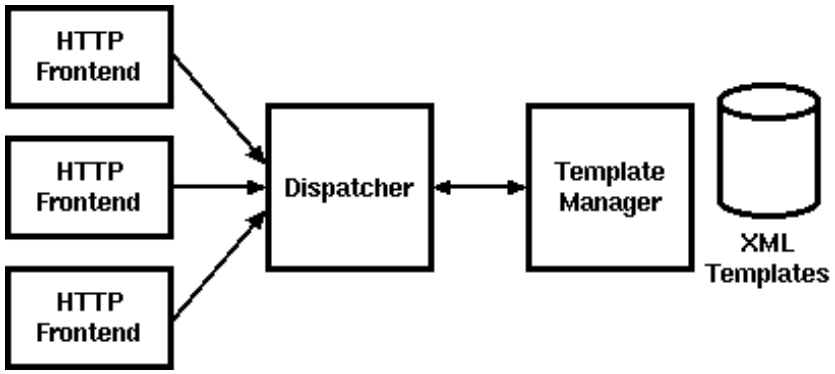


Figure 5: The three components of the Application Controller: HTTP Frontend, Dispatcher, and Template Manager

Again, we return to the attendee list application to describe how these three components interact with one another and how the Application Controller interacts with other components in the ADS system. When the Application Controller receives data from the Access Point it performs the following steps:

1. The HTTP Frontend parses the HTTP POST request to extract the data, user-id, and command-tag. The Frontend then passes this data to the Dispatcher.
2. Upon receipt of this information, the Dispatcher passes the user-id and command-tag to the Template Manager.
3. The Template Manager finds the XML workorder, whose format is described below, that corresponds to the user-id and command-tag given.
4. The Dispatcher parses the workorder using IBM's XML4J package. It extracts the relevant information such as service name, arguments, return type, etc.
5. For each service listed in the workorder, the Dispatcher does the following:
    - Obtains a handle to the remote service using the Ninja service loader

- Uses the Java Reflection API to obtain a handle to the service's main method.
- Calls the main method via Java RMI
- Labels the corresponding entry in the workorder as 'finished.' The labelled workorder can be used for such things as intermediate status information, bookkeeping, billing, and possibly, error detection.

The XML workorder returned by the Template Manager and used by the Dispatcher to execute the correct services on the data is shown in Figure 6.

Workorder Format:

```
<template>

  <command>
    <service>[service name]</service>
    <method>[method name]</method>
    <arglist num="[number of arguments]">
      <arg type="[arg1 type]">[arg1]</arg>
      <arg type="[arg2 type]">[arg2]</arg>
      ...
    </arglist>
    <return>[return mimetype]</return>
  </command>

  <command>
    ...
  </command>

</template>
```

Sample Workorder:

```
<template>

  <command>
    <service>ads.services.Convert</service>
    <method>convert</method>
    <arglist num="3">
      <arg type="byte[]">$1</arg>
      <arg type="java.lang.String">$2</arg>
      <arg type="java.lang.String">gif</arg>
    </arglist>
    <return>image/gif</return>
  </command>

  <command>
    <service>ads.services.Store</service>
    <method>store</method>
    <arglist num="2">
      <arg type="byte[]">$1</arg>
      <arg type="java.lang.String">img.gif</arg>
    </arglist>
  </command>

</template>
```

Figure 6: The workorder format and a sample workorder that converts images from jpg to gif and stores the resulting image.

Note that the only difference between a template and a workorder is that in a workorder, service descriptions are bound to service handles. The reason the current Template Manager returns workorders is that service discovery and lookup mechanisms have not yet been integrated into the ADS system. For this, we intend to explore the possibility of using Ninja's Service Discover Service (SDS) and/or Ninja's automatic path creation mechanisms.

3.3 Composable Services

Our services are simple Ninja services that are invoked using Java RMI. Invoking services via procedure calls makes it easy for service writers to implement new services quickly. Furthermore, Ninja places minimal constraints on the service implementations; all that is required are `init()` and `destroy()` methods. The following services have been implemented for use in the attendee list application:

- ConvertService: uses the UNIX command 'convert' to convert between arbitrary image formats
- ScalingService: scales images by a user-specified amount
- ThumbnailService: resizes images to thumbnail size
- StoringService: stores files on the local drive

For each attendee picture or business card, the Application Controller invokes the appropriate data-formatting services on the data. The Application Controller then uses the StoringService to store the file at the conference Web site.

4 Discussion

We set out to explore the potential for combining the new generation of digital input devices with emerging Internet services but using web infrastructure for input rather than output. Our experience in constructing the Science Fair web site without infrastructure support convinced us that such support would be essential for successful integration. This section presents some observations in trying to build such infrastructure.

The main objective in designing the Access Point was extensibility. Since there are many widely-deployed protocols, it must be relatively simple to write Device Adaptors for the Access Point. We believe that using HTTP POST for the API between device adaptors and the Aggregator provides a good balance of simplicity and flexibility. However, more experience writing device adaptors is required to test this hypothesis.

The Application Controller is still in an early stage; the balance between end-user ease-of-use and application generality will be a great challenge. Our attendee list application revealed many issues in robustness and usability that remain to be tackled, some of which we describe in the Open Issues section.

There is also a larger issue to consider. This web-input experience might be considered inappropriate for setting the requirements for Appliance Data Services. Arguably, a school science fair or conference attendee list is not a typical "killer app", and the constraints imposed by the experience might not represent the bulk of applications of a web-based data-input services systems in the future. However, we argue that this is instead a glimpse at the future on two grounds:

- Science fairs and attendee lists may not have great economic value, but the simplicity and ubiquity of the Web is what motivated the effort to record these events there in the first place. If we design a data input system sufficiently simple and useful, then many marginally economic data input activities suddenly become valuable. For example, consider the construction of auction house catalogs, recording inventory, insurance claim recording, and collaborative content authoring.
- The constraints of these applications, rather than being odd, may reflect constraints imposed on such systems in the future. We choose to tackle a system with heterogenous devices, multiple inputs to coordinate, and "open" services as the application components. A single device with a single communications protocol and a closed application would not explore the potential challenges of deploying and operating real systems.

The Web's simplicity allowed it to quickly evolve from a simple way for researchers to publish static content to a universal interface for sophisticated services such as banking, shopping, and mapping. By analogy, we hope that ADS--which "runs the Web backwards"--will make the 'trivial' task of data-input simple enough that applications that leverage it become more widespread and compelling.

5 Open Issues

ADS is an early prototype first step toward infrastructure support for appliance data services. Although we believe we have identified some fundamental architectural mechanisms for such applications, we have barely begun to explore the issues involved in making ADS "real":

Failure Semantics: An issue we have yet to tackle is how to report success or failure of the application to the user. This is a problem of semantics, not just implementation: only an end-to-end indicator of application success or failure is likely to be useful ("The photo got posted" or "it didn't"), but in some cases the application may be sufficiently long-running that it is unreasonable to expect the user to wait for an end-to-end check. A real concern is making sure the user's expectations are set correctly: in the digital-camera scenario, if the user successfully injects camera data into the infrastructure, the user may feel it is then safe to erase the camera's memory. In fact this is only safe if the application can make *some* guarantee about the persistence of the injected data, if not the success of complete application execution, so that recovery can be attempted later. We speculate that the application logic could include a separate end-to-end acknowledgment delivered to the user "out of band" with respect to ADS; perhaps the user can be sent email upon successful completion of the application.

Security and Privacy: Although we envision the Access Point as a public shared resource, the user identifier accompanying the data entering the Access Point is sufficient to bootstrap a secure connection to the rest of the infrastructure. Because we have not yet investigated how best to provide secure and private service, we have deliberately avoided specifying the format of the user identifier. It might, for example, consist of an identification token accompanied by a challenge/response pair that the AP can use to authenticate itself to the Application Controller. This mechanism, which does not require a user's secret key to be revealed to the AP, is analogous to the mechanism used for roaming in the GSM cellular telephone system. In any case, the user has to trust the Access Point not to maliciously eavesdrop or tamper with the data coming from the device. Since users currently appear to be willing to read their email on shared kiosks in airports and hotels, we do not expect trusting the Access Point to be a major obstacle to deployment.

6 Conclusion

Our overarching goal has been to enable the same level of innovation in connecting input-centric appliances to the Web as has been achieved in the last few years for Web information delivery appliances. To this end, we identified specific design challenges and proposed solutions to them in the context of an enabling architecture:

- Device and protocol heterogeneity: By handling the heterogeneity in the Access Point and producing a uniform representation (typed data, typed command tag, user ID) of device input in a Web-digestible protocol, **we isolate heterogeneity concerns in a single architectural component**. The rest of the software infrastructure can deal exclusively with Web-friendly standard protocols and formats.
- Unobtrusive user experience: We have described how **command canonicalization via transformation** can be used to gracefully extend the functionality of a device's existing UI, and how **late binding of commands** in a database can be used to separate device configuration from the specification of new application behaviors.
- Leveraging Web services: by building an **open system for data-input support** we hope to allow for and thus encourage an economy of service providers to support data input, leading to a spiral of better devices and services to support them.

We hope and anticipate that our initial work on ADS will encourage others to leverage the Web to magnify the usefulness of input centric appliances.

7 Acknowledgments

We thank Patrick Arnold of Hewlett Packard and the rest of the HP Jetsend team for early access to Jetsend protocol code for our Access Point implementations.

References

- [AOS+99] Ken Arnold, Bryan O'Sullivan, Robert W. Scheifler, Jim Waldo, and Ann Wollrath, "The Jini Specification", Addison Wesley, 1999.
- [chai] ChaiServer system described on http://www.chai.hp.com
- [FGCB98] Armando Fox, Steven D. Gribble, Yatin Chawathe, and Eric A. Brewer. Adapting to network and client variation using active proxies: Lessons and perspectives. IEEE Personal Communications (invited submission), Aug 1998. Special issue on adapting to network and client variability.
- [FGC+97] Armando Fox, Steven D. Gribble, Yatin Chawathe, Eric A. Brewer, and Paul Gauthier. Cluster-Based Scalable Network Services. In Proceedings of the 16th ACM Symposium on Operating Systems Principles (SOSP-16), St.- Malo, France, October 1997.
- [FGG+98] Armando Fox, Ian Goldberg, Steven D. Gribble, Anthony Polito, and David C. Lee. Experience with Top Gun Wingman: A proxy-based graphical web browser for the Palm Pilot PDA. In IFIP International Conference on Distributed Systems Platforms and Open Distributed Processing (Middleware '98), Lake District, UK, September 15-18 1998.
- [GW+99] Steven D. Gribble, Matt Welsh, Eric A. Brewer, and David E. Culler. The NINJA project pages, January 1999. http://ninja.cs.berkeley.edu.
- [KB+96] Randy H. Katz and Eric A. Brewer et al. The bay area research wireless access network (barwan). In Proceedings Spring COMPCON Conference 1996, 1996.
- [Met95] Metricom Corp. Ricochet Wireless Modem, 1995. http://www.ricochet.net.
- [Pro97] ProxiNet, Inc. ProxiWeb Thin Client Web Browser, 1997. http://www.proxinet.com.
- See product information for the Kodak DC265 camera on http://www.kodak.com
- [WAP97] WAP Forum. Wireless application protocol (WAP) forum. http://www.wapforum.org.
- [jetsend] The Jetsend communication protocol is described on http://www.jetsend.hp.com
- [irda] http://www.irda.org/
- [sync] For example, see the Intellisync software at http://www.pumatech.com/intellisync.html

---

Vitae

Andrew C. Huang, Stanford University

Gates 252
Computer Science Department
Stanford, CA 94305
Email: ach@cs.stanford.edu

Andrew received his B.S. degree in Electrical Engineering and Computer Sciences from UC Berkeley in 1998. He is currently a second-year Ph.D. student in the Stanford University Computer Science Department and is a member of the Software Infrastructures Group headed by Professor Armando Fox. His current research involves enabling ubiquitous computing devices (such as PDA's, digital cameras, etc.) to access Internet services using software infrastructure support.

Benjamin C. Ling, Stanford University

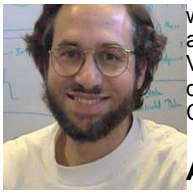
Gates 252
Computer Science Department
Stanford, CA 94305
Email: bling@stanford.edu

Benjamin Ling holds a B.S. in Electrical Engineering and Computer Sciences from UC Berkeley. He received the Bechtel Achievement Award at UC Berkeley, and is currently an Department of Defense Fellow. He is a second-year Computer Science Ph.D. student at Stanford University, and is also a member of the Software Infrastructures Group. His current research involves enabling ubiquitous computing devices to access Internet Services using software infrastructure support.

John J. Barton, Hewlett-Packard Laboratories

1501 Page Mill Road
Palo Alto, California 94304-1126
Email: John_Barton@hpl.hp.com

Dr. Barton works on software infrastructure to support coordinated data input from digital appliances like cameras and PDAs. This work is part of the HP Labs Cooltown project (www.hp.cooltown.com). Before joining HP in 1998, he



worked IBM's T. J. Watson Research Center. There he wrote the Jalapeno Java Virtual Machine boot image writer and managed the Java Technology group. Before that he worked on the "Montana" research project that led to IBM's VisualAge C++ v 4.0 product and co-author "Scientific and Engineering C++" with Lee R. Nackman. He has a Ph.D. degree in Chemistry from the University of California at Berkeley and a Master's degree in Applied Physics from the California Institute of Technology.

Armando Fox, Stanford University

Gates 446
Computer Science Department
Stanford, CA 94305
Email: fox@cs.stanford.edu

Armando Fox joined the Stanford faculty as an Assistant Professor in January 1999, after getting his Ph.D. from UC Berkeley as a researcher in the Daedalus wireless and mobile computing project. His research interests include the design of robust Internet-scale software infrastructure, particularly as it relates to the support of mobile and ubiquitous computing, and user interface issues related to mobile and ubiquitous computing. In previous lives, Armando received a BSEE from M.I.T. and an MSEE from the University of Illinois, and worked as a CPU architect at Intel Corp. He can be reached at fox@cs.stanford.edu. He is also an ACM member and a founder of ProxiNet, Inc. (now a division of Puma Technology), which is commercializing thin client mobile computing technology developed at UC Berkeley.