Pushing the Envelope of Pervasive Access

Badrish Chandramouli[*1], Hui Lei[2], Kumar Bhaskaran, Henry Chang, Michael Dikun, Terry Heath
[*]*Duke University, Durham, North Carolina 27708-0129*
badrish@cs.duke.edu
IBM T. J. Watson Research Center, Yorktown Heights, NY 10598, USA
{hlei, bha, hychang, mdikun, theath}@us.ibm.com

Abstract

This paper presents the design and implementation of the Puma middleware system. Puma enables pervasive access to Web applications from a wide range of clients. In addition to traditional, browser-equipped client devices such as laptops and PDAs, Puma supports the use of peer collaboration tools such as instant messengers, SMS devices, email clients and telephones. While those collaboration tools were initially intended for free-form interaction between people, Puma leverages them for structured interaction between people and computers in order to offer more flexibility, convenience and intimacy to end users. In addition to user-initiated, or pull-based, interactions, Puma allows an application to proactively push an interaction to a user, in a manner sensitive to the application's needs and the user's current context. Architecturally, Puma employs various Modality Bots to mediate between application servers and heterogeneous clients. The Modality Bots also serve as the initial point of contact for application-initiated interactions. As an experiment, Puma has been used to mobilize the Human Tasks Application, which supports the creation, processing, and management of the manual steps in business processes.

1. Introduction

The World-Wide Web has evolved into a prominent infrastructure for the provisioning and access of computer applications, thanks to the ubiquity of the Web and widely-adopted technologies like J2EE, JSP, and Struts. The so-called Web applications are hosted on networked application servers and interface with end users through Web browsers on client devices. Web browsers and Web application servers converse in HTTP, with the user interaction model encoded in some presentation markup language such as HTML.

Person mobility is an inherent part of everyday life. It is highly desirable to allow users to access their applications and data at any time, regardless of their locations and the devices they use. Despite a long history of research efforts, pervasive access remains a very active research topic. It has gained even more importance lately due to the rapid growth of the mobile workforce and the deep penetration of pervasive clients in the market [20, 21].

The Web application model bolts well with the vision of pervasive application access because the application state is completely maintained on the server side. Many Web application servers are also able to adapt the application interface to the characteristics of the client device and represent the resultant interface in a device-specific format such as WML [22] and cHTML [23]. Nevertheless, the Web application model requires that some form of browser exist on the client device. Further, the Web is fundamentally a *pull* technology. The user must therefore explicitly initiate all interactions with the application.

We have been investigating how to exploit peer collaboration tools to extend the reach of mobile Web applications. Various collaboration technologies – including cell phones, email, instant messaging (IM), the short message service (SMS), and pagers – have emerged that people can use to interact with each other even when they are on the move or far away. Using collaboration tools as the interface to Web applications eliminates the applications' dependency on Web browsers and thus allows applications to be accessed even when a Web browser is not available. Many built-in features of peer collaboration tools can also be very handy for human-computer interaction. For example, collaboration tools are capable of

[1] This work was performed while the author was visiting IBM Watson Research.
[2] Correspondence author.

receiving "calls", which can be exploited by applications to initiate and *push* an interaction to end users. It is also conceivable to transfer an on-going user interaction session to another device or to another user, analogous to the way people transfer a phone call today. Using the email modality, people may be able to interact with applications in an asynchronous and disconnectable fashion. Finally, the collaboration tools allow for a hybrid interaction scenario, where the server-side participant can be seamlessly switched between a machine and a human being, without the end user on the client-side taking any explicit action.

Enabling pervasive application access from collaboration mechanisms presents a number of challenges. First, collaboration tools were initially designed for free-form, person-to-person interaction. There is no native support for controlling or structuring the messages being exchanged. The interaction between the user and the application, on the other hand, must be based on well-formed messages and exchange sequences. Second, collaboration mechanisms come in a large number of varieties, and have widely varying capabilities. Such client heterogeneity imposes the requirement for target-specific user interfaces. In spite of the wide spectrum of clients, the complexity of application development must be under control. Third, although a user typically has multiple collaboration mechanisms, she may have access to only a subset of them at a particular time. Depending on the circumstance, she may also have a preference for which of the available tools to use. Thus, there is a need to dynamically select an appropriate tool to engage the user for a particular interaction.

We have designed and implemented a Web application extension framework, dubbed *Puma*[3]. Puma provides pervasive access to Web applications from a wide range of clients, including both Web browsers and collaboration tools. In addition to user-initiated, or pull-based, interactions, Puma allows an application to proactively push an interaction to a user, in a manner sensitive to the application's needs and the user's current context. Puma employs various Modality Bots to mediate between application servers and collaboration clients. The Modality Bots interpret UI specifications obtained from the application and render them in a modality-appropriate fashion. They also serve as the initial point of contact for application-initiated interactions.

[3] "Puma" stands for Push-enabled Mobile Applications.

Puma is a successor of the PerCollab system [2, 3], but differs in significant ways. PerCollab is middleware that bridges workflow systems and peer collaboration tools. It proactively engages users in business processes by pushing the manual process steps to a convenient collaboration mechanism of the users. Puma generalizes that idea to enable pervasive access to applications beyond workflow systems. Puma also supports both push-based and pull-based interactions, as opposed to push only in PerCollab. While user interaction in PerCollab is based on a simple message-exchanges model, Puma allows the explicit handcrafting and customization of the UI, which enables more contextual information to be presented to the user and can result in a more friendly UI.

To our knowledge, Puma is the first generic system for accessing Web applications from arbitrary collaboration modalities. It is also the first system that offers the capability of application-initiated, two-way interactions.

Next we discuss the considerations that influenced the Puma design. We then present the salient aspects of the system design and current implementation. We report on our experience of using Puma to mobilize a particular application – the Human Tasks Application. Finally, we survey related work and close with a summary of our ideas.

2. Design Rationale

The design of Puma is driven by four requirements: support of application-initiated interactions, dynamic selection of an appropriate interaction modality, accommodation of heterogeneous clients, and structured user interfaces on collaboration tools.

Traditionally user interactions with applications are initiated by end users. Although this kind of interaction model is appropriate for applications with a short duration, it is not suitable for long-running applications that can last days or even longer. Examples of such long-running applications include many business processes, activity management, monitoring and surveillance. Constant user presence in these applications is typically not necessary; neither is it feasible. User involvement is needed only when certain events happen and/or when the application state satisfies a predefined condition. With a pull approach, the burden is placed on the user to periodically poll the application for the purpose of determining whether her participation is required. Needless to say, such an approach can be very inefficient and may cause critical opportunity loss.

In comparison, a push-based approach allows the application to engage a user at the right time by proactively pushing an interaction session to the user. This can substantially reduce the demand for user attention and in the meantime promises to improve the efficiency of the application. We advocate pushing two-way interaction sessions to users and argue that simply pushing a notification message may not be adequate. Some applications compensate the limitations of a pull approach by sending users a one-way message on demand. The users can then start an interaction session with the application, from a client browser. In this case, the users have to make an extra effort to switch from a messaging mechanism to a browser. Still, there is no guarantee that a browser is immediately available at the time the notification message is received.

Intended as a generic Web application framework, Puma supports both client-initiated and application-initiated interactions. It allows interaction sessions to be pushed to users' collaboration tools, which all come with a native receiving capability. Because Puma integrates multiple collaboration modalities, it must decide which of the collaboration modalities should be used to push the interaction to. As people move from place to place, their connectivity and accessibility to various collaboration tools may change. Depending on the circumstance, some types of tools may also be more preferable than others. For example, if the user is in a meeting, he may not want to receive any phone calls. When he is giving a presentation, he may not want to be interrupted by any instant messages. Generally speaking, the best means of engaging a particular person at a particular moment depends on the person's current context, such as the person's location, activity, connectivity and personal preferences [24]. Therefore, Puma allows the application to specify a modality policy that is predicated on user context when pushing an interaction. In addition, different modality policies can be used in different sections of the application for different interactions, reflecting the needs of the application itself.

A fundamental challenge in mobile and pervasive computing is creating and managing applications for a large diversity of clients. Applications should be able to render an interface on any client at the user's disposal. Given the wide range of device capabilities, form factors, and user input and output methods, it is unscalable for the application developers to create interfaces for each possible kind of clients. Although the use of collaboration tools adds to the disparity of clients, fortunately many technologies developed to tackle client heterogeneity remain applicable. Puma has adopted some of the more mature solutions. The applications in Puma are based on the Model-View-Controller (MVC) design pattern [25]. The Model in the MVC pattern encapsulates the business logic as well as the business domain state knowledge. Each View component within an application describes the presentation and user interaction elements that logically belong together. The Controller represents the application flow, including the navigation between the View components, validation of user requests, and error handling. By associating clear and distinct responsibilities with different components, the MVC pattern maximizes the extensibility and reusability of application components, which has proven effective in reducing the complexity of application development.

Puma allows the View aspects of an application to be represented in modality-independent format. The modality-independent representation describes the intent behind the user interaction rather than the actual physical representation of UI controls. It is rendered at runtime based on the characteristics of the particular modality. The use of modality-independent representation in fact serves two purposes. First, it allows an application to capture the basic interaction structures that can be reused across multiple devices and modalities as appropriate. Second, it provides a way to describe the structure and content of the user interface for originally unstructured collaboration modalities, without resorting to some home-grown or modality-specific representation.

Further, Puma allows the layout and style of the user interface to be manually customized to specific client modalities if so desired, leading to highly customized and usable interfaces.

3. System Design

In this section, we give an architectural overview of Puma, followed by a discussion of the major system components and system operations.

3.1. System Architecture

The Puma architecture is shown in Figure 1 (the direction of arrows in the figure describes the flow of control between components). The system accommodates a diverse and extensible set of clients, including conventional Web browsers and collaboration tools. The collaboration tools are integrated into the system via *Modality Bots* (e.g., IM Bot, Email Bot, and Phone Bot). Each Modality Bot connects one collaboration modality (e.g., IM, email and phone) with Puma, controlling the user interactions through that particular modality. The *Dispatcher* serves as a single point of entry for user

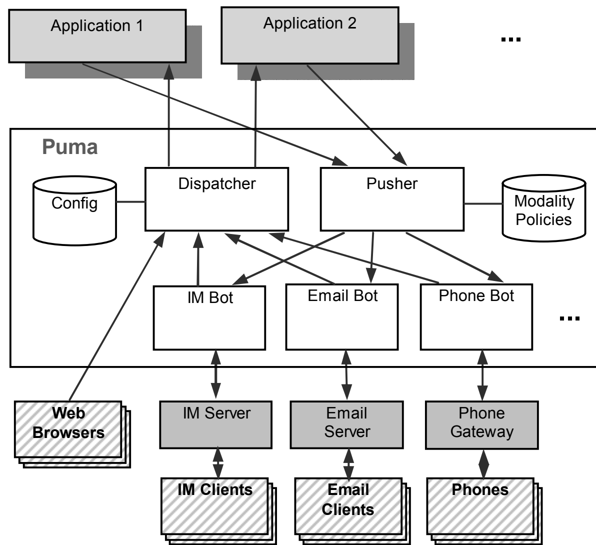

Figure 1: Puma Architecture

requests from various clients. It interprets those requests and routes them to corresponding applications, based on navigation information contained in the configuration file. The *Pusher* component handles application-initiated user interactions. It receives user interaction specifications from applications and forwards them to appropriate Modality Bots for the purpose of proactively engaging users via a callable client (i.e., a collaboration tool).

3.2. Modality Bots

The Modality Bots allow disparate collaboration modalities to be integrated into the system. Each Modality Bot handles one category of collaboration tools and is addressable in the network of the corresponding modality. For example, the IM Modality Bot is a user in the instant messaging system. The user may start an instant messaging session with the Modality Bot and send it various messages. Similarly, the Phone Modality Bot may be reached at a standard telephone number. The user dials this number to use Puma.

A Modality Bot performs three kinds of functions. First, it manages user interactions that go through collaboration tools of a particular type. It receives one view specification at a time and renders it in a modality-specific manner. Depending on the modality, the Bot may or may not need to establish a connection with the collaboration tool before an interaction session starts. Second, the Modality Bot acts as a Web client and communicates with the Dispatcher. Each request from the Modality Bot is a bundling or composition of user input collected from the collaboration tool. The response from the application contains a new view specification. Third, the Modality Bot receives descriptors of application-initiated interactions via the Pusher. It obtains an initial view specification through the Dispatcher so as to trigger an interaction with the user on his collaboration tool.

It is possible that a Modality Bot may be conducting multiple interaction sessions with an end user at the same time. For example, one interaction may be initiated by the user, while another one is triggered by an application. Mingling messages from different interactions can be very confusing to the user. Depending on the modality, the Bot can handle the situation in one of two ways. It can tag each message with an interaction session ID or description so that the user can correlate messages properly. Or it can ban concurrent sessions altogether, requiring the user to suspend or exit one session in order to enter another.

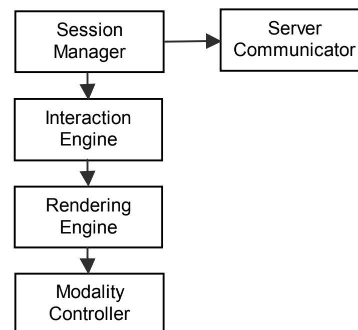

Figure 2: Modality Bot

The Modality Bots are architected as shown in Figure 2. The *Session Manager* maintains all user interaction sessions. Each session object communicates with the application server via the *Server Communicator* component. The modality-independent view representation is handed off to the *Interaction Engine*, which extracts low-level presentation elements (e.g., labels and data) and interaction elements (e.g., type-in fields or selection lists) and passes them to the *Rendering Engine*. The Rendering Engine is modality specific, and renders the presentation and interaction elements appropriately. The Rendering Engine uses the *Modality Controller* to communicate with a user's collaboration tool.

3.3. Pusher

The architecture of the Pusher is shown in Figure 3. At the core of the Pusher is the *Push Engine*. The Push Engine receives and validates descriptors of application-initiated user interactions, determines the appropriate collaboration tools for engaging the user by consulting with the *Modality Resolver* and the

Address Resolver, and delivers the interaction descriptors to the corresponding Modality Bots.

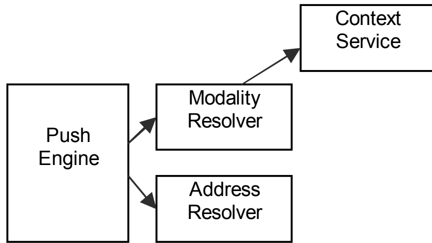

Figure 3: Pusher

The Modality Resolver determines the proper modalities given a user ID and a modality policy ID. A modality policy may be predicated on temporal attributes (e.g., time of day) and the user's context conditions (connectivity, location, current activity, availability etc). In an extreme case, a modality policy can simply enumerate applicable modalities without a qualifying condition. There are two alternatives to the representation of the modality policies. One alternative is to represent each policy as a set of rules. In this case, the Modality Resolver serves as an interpreter of the policy rules. The other alternative is to implement each policy as a Java class that implements all the policy logic. The Modality Resolver then instantiates and executes the Java object for the specified policy.

The *Context Service* [6] is an infrastructure service developed in an earlier project for gathering and disseminating heterogeneous context information. It allows the Modality Resolver to obtain user context information without having to worry about the details of context derivation and context management. Information currently provided by the Context Service includes IM online status, activities and contact means derived from calendar entries, desktop activities, as well as user locations reported from a variety of sources such as cellular providers, wireless LANs, GPS devices, and RIM blackberry devices.

The Address Resolver returns the modality-specific address of a user, such as the user's telephone number or email address. Internally it uses a registry that maintains the mappings from user IDs to their modality-specific addresses.

3.4. User-Initiated Interaction

The sequence of a user-initiated interaction is shown in Figure 4. We assume that the application is constructed based on the MVC design pattern. The client is supposed to be a connection-oriented collaboration mechanism such as a telephone or an IM client. Other clients work in a similar fashion. The interaction consists of the following steps.

1. The user calls the Modality Bot from a collaboration client.
2. The Modality Bot composes an HTTP request based on user input and sends the request to the Dispatcher.
3. The Dispatcher routes the request to the appropriate application by calling the application's Controller.
4. The Controller invokes the business logic by calling the Model component.
5. The Controller then forwards the control to an appropriate View component.
6. The View component generates the initial view markup and returns it to the Modality Bot.
7. The Modality Bot conducts a dialogue with the user according to the view markup.
8. If the view markup contains elements for user input, the Modality Bot bundles the user input in another request and repeats Steps 2 to 7 (not shown).
9. The user leaves the call with the Modality Bot.

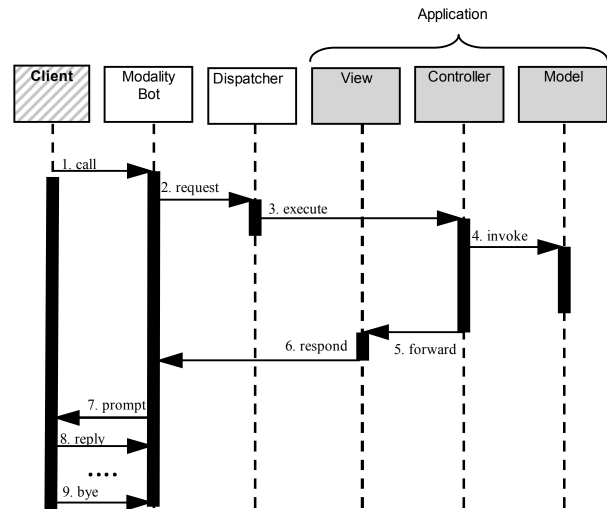

Figure 4: User-Initiated Interaction

It should be clear from the above action sequence that the Modality Bot plays the role of a dual proxy. To the server-side application, it acts like a Web client and communicates with the application server in standard HTTP. To users on a collaboration tool, on the other hand, it represents the Web application and appears as a peer on the modality-specific network of collaborators.

3.5. Application-Initiated Interaction

Figure 5 shows the sequence diagram for a typical application-initiated interaction. Again the client is assumed to be a connection-oriented collaboration

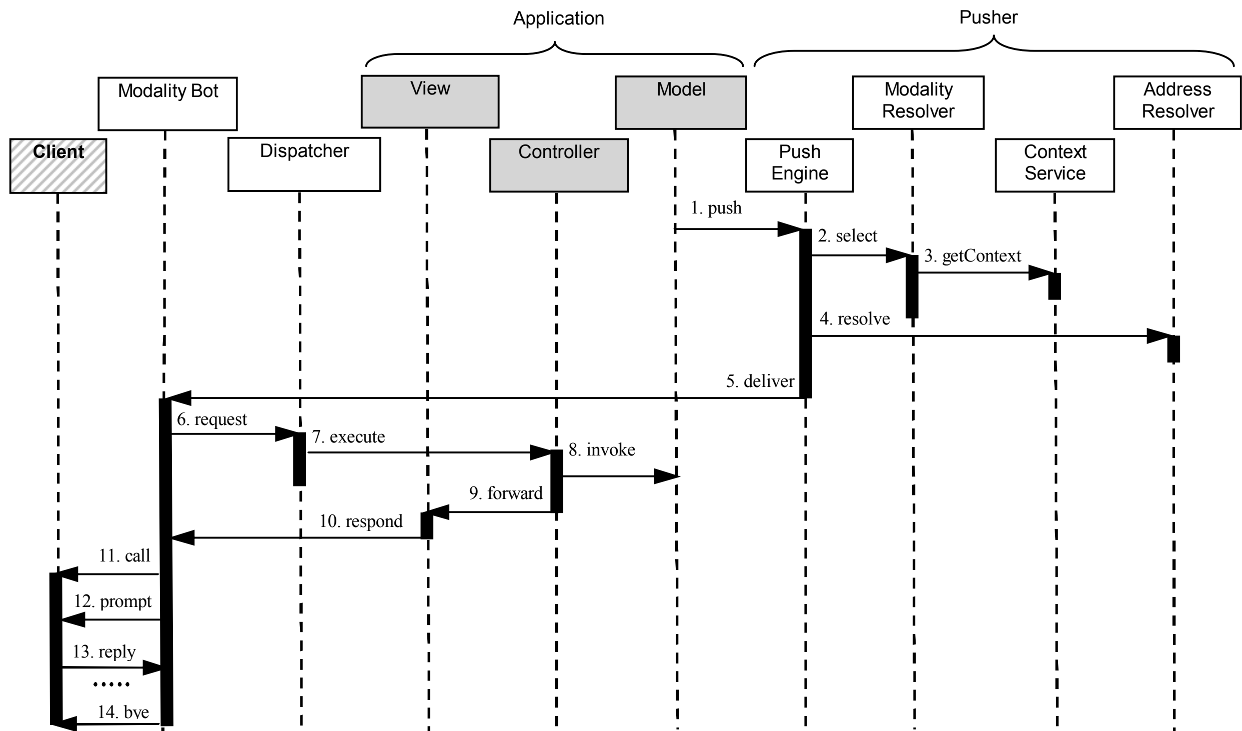

Figure 5: Application-Initiated Interaction

mechanism. The interaction consists of the following steps.

In Step 1, the business logic in the application sends an interaction descriptor to the Push Engine, along with the ID of the user that should be engaged, and the ID of the applicable modality selection policy. The interaction descriptor identifies the application itself and the interaction parameters. In Step 2, the Push Engine calls the Modality Resolver to determine the appropriate modalities for the user. The Modality Resolver optionally retrieves the user' current context from the Context Service (Step 3). The Push Engine also calls the Address Resolver to obtain the modality-specific addresses of the user (Step 4). It then delivers the interaction descriptor to the corresponding Modality Bots, along with the ID and the modality-specific address of the user (Step 5).

Each Modality Bot contacted constructs an HTTP request based on the information received and sends the request to the Dispatcher (Step 6). The Dispatcher processes the request in the same way as it processes a user-initiated request, before it returns an appropriate view to the Modality Bot (Steps 7–10). The Modality Bots then calls the user in question to start a dialogue (Step 11). It then mediates the traffic between the user and the application as usual (Steps 12-13). When either the view markup or the user indicates that no further interaction is needed, the Modality Bot terminates the call (Step 14).

Application-initiated interaction requests, like user-initiated ones, are first sent to the Modality Bot. They are then mapped to HTTP requests and eventually to view markups via the same path of Dispatcher -> Application Controller -> Model -> View. This allows the existing Web application framework, which was initially designed for pull-based interactions only, to be retained for the new push-based interaction paradigm. This in turn preserves prior investments in the IT infrastructure and offers new application capabilities without undue development efforts.

4. Implementation

We have implemented a prototype of Puma on top of the WebSphere Application Server (WAS) V5.0.1 [26]. The implementation is depicted in Figure 6. We use the WebSphere Portal Server (WPS) V5.0.1 [4] as the Dispatcher, which itself installs as an enterprise application in WAS. WPS is chosen as the application platform because it naturally supports the MVC application model and heterogeneous client types.

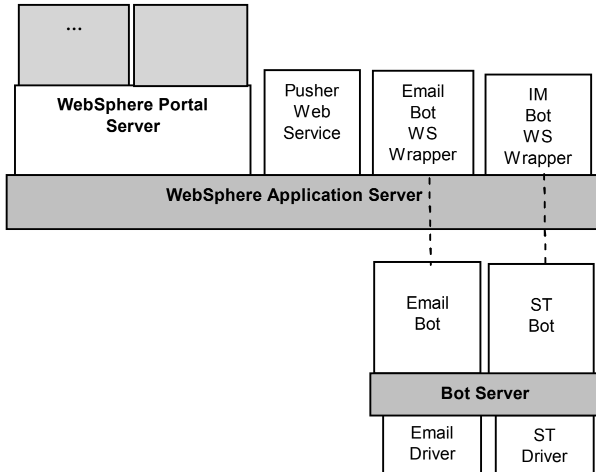

Figure 6: Puma Implementation

The Pusher is implemented as a Web service in WAS. It consists of three sub-components: the Push Engine, the Modality Resolver and the Address Resolver. The APIs for the sub-components are given below:

- Push Engine:
```
void push(InteractionDescriptor id, UserID user,
          String modalityPolicyID);
```
- Modality Resolver:
```
Modality[ ] select (String modalityPolicyID,
                    UserID user);
```
- Address Resolver:
```
Address resolve(UserID user, Modality modality);
```

The push() method on the Push Engine is exposed in the Web service interface of the Pusher. Modality policies are represented as Java classes in our system, allowing flexible and expressive policies to be specified. Each modality policy class implements the following interface:

```
Interface ModalityPolicy {
        public Modality[ ] select(UserID user);
}
```

Puma uses XForms [1] for the modality-independent representation of the View components of an application. Although XForms is a modality-independent language, the view it represents does not have to be modality-independent. Using the WPS framework, an application developer can adapt a view to a particular modality by either supplying an XSLT stylesheet to tailor the layout and style of the view, or handcrafting a separate view for the modality. The modality-specific view, still encoded in XForms, can then be rendered by the corresponding Modality Bot.

The Modality Bot are implemented as bots on IBM's BotServer [27], which is a system that enables the creation and administration of intelligent action agents (i.e., bots) in various message-based environments. Each Modality Bot is wrapped in a Web service to facilitate invocation by the Push Engine. The following method is exposed in the Web service interface of each Modality Bot:

```
void deliver(InteractionDescriptor id, UserID user,
             Address clientAddress);
```

The current implementation includes Modality Bots for email and Sametime instant messaging [8]. These two modalities are selected for the initial implementation because they have very different characteristics: email is for asynchronous interaction while IM is for synchronous interaction. In the rest of this sub-section, we discuss the implementation details of the Modality Bots.

When a new interaction session, which can be requested by either a user or an application, is established with the user, the Modality Bot creates a new session object and executes it on a thread taken from a pool of available threads. The session object issues an HTTP GET request to the WPS application and obtains the view to be rendered. The view is sent in XHTML with embedded XForms and is extracted from the application's response. The XForms data is then passed to the Interaction Engine (c.f. Figure 2). The Interaction Engine builds upon the IBM XML Forms package [5]. It loads the XForms, parses it, and calls various *writers* to process the XForms elements (e.g. input, output, select, switch). The writers are common across all modalities and use a modality-specific Rendering Engine to actually present the elements. The writers populate the XForms instance data based on user input. They also perform schema validation and check for mandatory and relevant form fields before populating the instance data. When the form is to be submitted, the session object sends the instance data to the application server in an HTTP POST request along with the action. The application responds with the next view in the interaction sequence and the process repeats.

The Rendering Engine for email batches all XForms elements of a view. It sends a single email to the user at the end, prompting the user to fill in

various input fields. The email is sent via the email communication driver, which is implemented using the JavaMail API [12]. The session ID is embedded in the email body, in order for the Rendering Engine to correlate user replies. The input fields are also numbered for correlating user input with form fields. When the response is received, the data entered by the user are sent to the appropriate writers. The same user can be engaged in multiple sessions at the same time with the application.

The Rendering Engine for Sametime instant messaging is more interactive. It presents the XForms elements as it receives them from the writers. User input is immediately sent to the writers in order to validate and update the instance data. In Sametime, since there is only one chat window for each correspondent, it would be confusing for the user to be engaged in multiple interaction sessions concurrently, as they would all be rendered in the single chat window with the Sametime Bot. Hence, only one session is allowed to be active. For example, if an application-triggered interaction occurs while the user is already in an active session, the Bot would inform the user about the new session and give the user an option to suspend or exit the current session and work on the new session. When an active session is finished, the user is presented with a menu of pending sessions to switch into.

5. Experiment

In order to validate the design and implementation of Puma, we have used Puma to mobilize a particular application – the Human Tasks Application (HTA) that is being developed by IBM's product division. HTA supports the creation, management and processing of manual tasks. Such functionality is useful for many business integration solutions and business processes. The original HTA allows a task participant to perform the following operations from a Web browser:

- Query tasks: retrieve information on tasks assigned to a participant
- Claim a task: gain exclusive ownership of an assigned task.
- Process a task: obtain corresponding task input data and provide task output data.
- Mark a task complete: declare completion of a task to prevent further editing of task output data.
- Unclaim a task: release the task and have it assigned to all potential participants again.

Our mobile extension of the HTA provides the above functions on pervasive clients such as PDAs and collaboration tools. In addition, it proactively pushes newly assigned tasks to users by contacting them on appropriate collaboration modalities.

As shown in Figure 7, the HTA consists of the Human Task Manager (HTM) service, the task list portlet, a collection of task processing portlets (one for each type of human tasks defined), and a collection of JSPs (one for each portlet). The HTM is the Model part of the application, encapsulates the logic of human task management and maintains the state of human tasks. The portlets constitute the Controller part of the applications, and the JSPs serve as the View components.

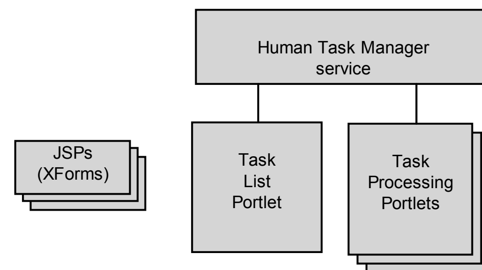

Figure 7: The Human Tasks Application

The task list portlet queries the HTM for the user's tasks, and passes control to the corresponding JSP which generates the XForms for operating on the list of tasks. The XForms first shows a list of tasks available to the user and allows the selection of a task to work on. Once the task has been selected, it allows the user to claim, unclaim, process, and mark complete the selected task. If the user chooses to claim, unclaim, or mark complete the task, the requested action is performed by invoking the corresponding HTM API call and the user is returned back to the list of tasks. If the user chooses to process a task, the control is passed to the relevant task processing portlet.

There is one task processing portlet for each task in the system. The task processing portlet interacts with the HTM to retrieve task state and data, bundles data into a Java bean, and passes control to the corresponding JSP which generates the XForms markup. The XForms layout may differ by tasks. But in a typical case, the XForms first checks if the task has been claimed. If not, the user is prompted to claim it. If the task is claimed, the XForms displays relevant task information to the user and prompts for user input if necessary. Finally, the user is given the option of completing the task, unless the task has the auto-complete feature turned on.

XForms have greater expressive power than traditional web forms. We exploit this ability to send a single XForms document with control flow embedded

within the document. This allows the user interaction to take place via a disconnectable modality like email even when there is no network connectivity.

In summary, the mobilization of the HTA includes the following changes to the original application. First, the HTM service was made to access the Pusher Web service and push new tasks to users. The original JSPs, which generated HTML markups, were re-written to generate XForms instead. The portlets were also modified to call these new JSPs.

5.1. Example Use Case

We have also implemented a hypothetical travel request approval business process using the HTA. The process involves an employee and his manager. We present the following example use case to illustrate the functionality of Puma and HTA.

When the employee accesses the HTA from his IM client, he sees that a travel request task has been assigned to him (presumably due to the instantiation of the business process). The employee chooses to claim and process this task and is prompted to provide travel details (see Figure 8). When he completes the

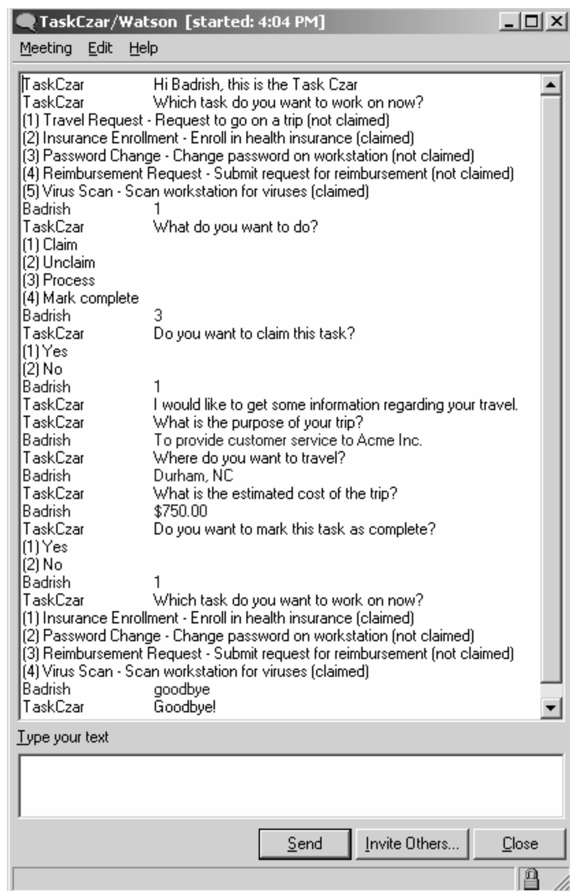

Figure 8: Travel Request via IM

task, the business process creates a new task of travel approval in HTA and assigns it to the manager. The HTA then pushes this new task to the manager via the Pusher. The Pusher realizes that the manager is in a meeting and does not want to be interrupted. So the Pusher delivers the task by email (see Figure 9). The manager finds the message in his mailbox after returning from the meeting. He grants the request by replying the email message. Finally, a new notification task is created in HTA to inform the employee of the approval. Because the employee is still available on IM, the notification task is pushed as an instant message. This completes the travel approval process.

The same business process was also implemented on Puma's predecessor – PerCollab [2]. A comparison of the two implementations shows Puma's additional capabilities of supporting pull-based interactions and fully customized user interfaces. More importantly, Puma supports pervasive access to applications other than workflow systems.

6. Related Work

The idea of exploiting peer collaboration tools to interface with computer applications is not a new one. People have long been enjoying the convenience and flexibility of telephone-based voice applications. There is a very large and profitable market for voice applications, spanning the domains of supply chain, finance, travel, healthcare, government and education. WebSphere Voice Server [9] and Microsoft Speech Server [10] are commercial offerings for enabling telephone access to applications and data. There is even a proposed standard, VoiceXML, for representing a voice interface [11]. Puma has taken this idea much further and developed a generic and extensible framework for pervasive access from any collaboration tools. Compared to voice, textual modalities such as email, SMS and IM offer additional benefits. They allow access when interaction via voice is not feasible (for example, when in a meeting). They also avoid the intrinsic difficulties and ambiguities in voice recognition. In addition, Puma allows interactions to be pushed to users by applications, which is not typically supported by current voice systems and applications.

Some applications are capable of a simple form of push. They can send notifications to users via some collaboration mechanism. Email is a common choice of e-commerce Web sites for sending order confirmations. The last couple of years have seen a growing use of SMS. While most applications send notifications via a modality-specific API (e.g., JavaMail [12]), the Notification Dispatcher provides a

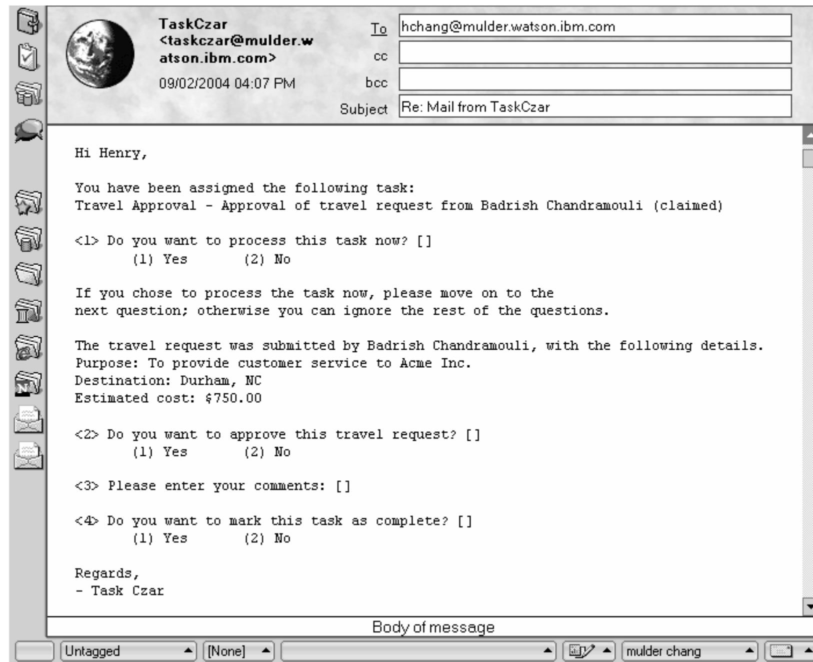

Figure 9: Travel Approval via Email

uniform API for delivering messages to a variety of channels including telephones, WAP devices, SMS devices, IM, and email [6]. It also allows delivery preferences to be specified that are predicated on the user's dynamic context. Nevertheless, existing systems support the push of one-way messages only. In comparison, Puma enables the push of two-way interaction sessions, which introduces a whole set of design challenges.

A large body of work has addressed the generation of user interfaces for heterogeneous devices [13, 14, 15]. A widely-adopted approach is model-driven UI development [28]. The idea is to represent the user interface using a device-independent representation, which specifies the kinds and structure of the information to be exchanged between the user and the application. The device-independent representation is then converted to a device-specific representation via some form of automatic adaptation. Several device-independent UI representations have evolved over the years, including UIML [16], XForms [1] and Microsoft Mobile Controls [17]. Examples of device-specific UI representations are HTML for PCs and PDAs, WML for WAP cell phones, and cHTML for Japanese i-mode phones. Runtime systems that support some variation of UI targeting include WebSphere Portal [4], Microsoft .Net [18], and Volantis Multi-Channel Server [19]. These systems target client devices that accept certain form of UI markup. Unfortunately, peer collaboration tools, which were intended for free-form people-to-people interaction, do not fit this description. Therefore Puma has to resort to Modality Bots that manifest as peers in the collaboration system but control user interactions according to XForms-based UI representations. With WebSphere Portal as part of the implementation, Puma also allows the application UI to be adapted to client characteristics.

In the domain of unified communication, a number of projects have developed extensible frameworks that enable people-to-people communication across heterogeneous end-points and route calls to a convenient callee device based on user preferences and context. These include the Mobile People Architecture [29], Universal Inbox [30], and our own Mercury system [24]. As mentioned before, the interaction between people is ad hoc and unstructured. The key aspect that sets Puma apart is the additional support for representing and rendering a structured user interface for interaction between people and applications.

7. Conclusions

Pervasive access to computer applications and services, at any time and from any location, is an old but enduring vision in computer science. Recently, there have been renewed interests from universities and industry labs in researching new technologies for pervasive access, due to the requirements of on-demand business practices and the ever growing market of mobile offerings. We have been investigating this issue in the context of the

omnipresent Web applications and developed the Puma system. Puma bridges Web applications and pervasive clients, extending application access from conventional browser-equipped devices to peer-collaboration mechanisms. Puma supports both user-initiated and application-initiated interactions. It represents the user interaction model in platform-independent XForms and renders it on heterogeneous clients. Observing that the choice of an appropriate user interaction modality depends on the user's current context, Puma also provides for client selection policies that are predicated on dynamic user context information such as location, activity and connectivity. Puma is the first system we are aware of that provides a generic framework for pervasive application access from arbitrary collaboration modalities. It is also the first known system that enables the proactive pushing of interaction sessions from applications to users. Although the presentation and interaction capabilities of collaboration tools may appear primitive compared to those of a full-fledged graphical user interface, anecdotal experiences indicate that application access using collaboration mechanisms can go a long way in offering flexibility and convenience to users. The great market success of phone-based voice applications also confirms that.

8. References

[1] W3C. XForms – The Next Generation of Web Forms. http://www.w3.org/Markup/Forms.

[2] D. Chakraborty and H. Lei. Pervasive Enablement of Business Processes. *2nd IEEE International Conference on Pervasive Computing and Communications (PerCom 2004)*, Orlando, Florida, March 2004.

[3] D. Chakraborty and H. Lei. Extending the Reach of Business Processes. *IEEE Computer,* 37(4), April 2004.

[4] IBM. WebSphere Portal for Multiplatforms. http://www-306.ibm.com/software/genservers/portal/

[5] The IBM XML Forms Package, April 2003. *http://www .alphaworks.ibm.com/tech/xmlforms*.

[6] H. Lei, D. Sow, J. Davis II, G. Banaduth and M. Ebling. The Design and Applications of a Context Service. *ACM Mobile Computing and Communications Review (MC2R), 6(4), October 2002.*

[7] Guruduth Banavar et al, An Authoring Technology for Multidevice Web Applications. *IEEE Pervasive Computing*, Jul.-Sept. 2004 issue.

[8] Lotus. Sametime. http://www.lotus.com/products/ lotussametime.nsf/wdocs/homepage.

[9] IBM. WebSphere Voice Server. http://www-306.ibm. com/software/pervasive/voice_server/

[10] Microsoft. Speech Server. http://www.microsoft.com/ speech/

[11] W3C. VoiceXML. http://www.w3.org/TR/voicexml20/

[12] Sun. JavaMail. http://java.sun.com/products/javamail/

[13] Shankar Ponnekanti, Luis Alberto Robles and Armando Fox. User Interfaces for Network Services: What, from Where, and How. *4th IEEE Workshop on Mobile Computing Systems and Applications (WMCSA 2002)*, 20-21 June 2002, Callicoon, NY, USA.

[14] Krzysztof Gajos and Daniel S. Weld. SUPPLE: Automatically Generating User Interfaces. *International conference on Intelligent user interface*, Funchal, Madeira, Portugal January 13 - 16, 2004.

[15] Karin Coninx, Kris Luyten, Chris Vandervelpen, Jan Van den Bergh and Bert Creemers. Dygimes: Dynamically Generating Interfaces for Mobile Computing Devices and Embedded Systems. *Fifth International Symposium on Human Computer Interaction with Mobile Devices and Services (MobileHCI)*, Udine, Italy, September 2003.

[16] User Interface Markup Language, http://www.uiml. org/index.php.

[17] Microsoft. ASP.Net Mobile Controls. http://msdn. microsoft.com/mobility/othertech/asp.netmc/default.aspx.

[18] Microsoft. .Net framework. http://msdn.microsoft.com/ netframework/technologyinfo/default.aspx.

[19] Volantis. Multi-Channel Server. http://www.volantis. com/story.jsp?story=volmcs&snav=voltech&tnav=volmcs.

[20] Gartner. Startegic Planning Report. http://regionals4. gartner.com/regionalization/img/gpress/pdf/2004_chapter_ mobile.pdf

[21] Mobile Tech News. http://www.mobiletechnews.com/ info/2004/02/03/122211.html.

[22] Open Mobile Alliance. WAP Wireless Markup Language Specification. http://www.oasis-open.org/ cover/wap-wml.html.

[23] W3C. Compact HTML for Small Information Appliances. http://www.w3.org/TR/1998/NOTE-compactHTML-19980209/.

[24] H. Lei and A. Ranganathan. Context-Aware Unified Communication. *IEEE International Conference on Mobile Data Management (MDM 2004)*, Berkeley, CA, January 2004.

[25] Sun. Designing Enterprise Applications with the J2EE Platform. http://java.sun.com/blueprints/guidelines/ designing_enterprise_applications_2e/index.html.

[26] IBM. WebSphere Application Server. http://www-306.ibm.com/software/webservers/.

[27] IBM. BotServer. Proprietary system.

[28] P. Szekely. Retrospective and challenges for model-based interface development. In F. Bodart and J. Vanderdonckt, editors, *Design, Specification and Verification of Interactive Systems '96*, Wien, 1996.

[29] M. Roussopoulos, P. Maniatis, E. Swierk, K. Lai, G. Appenzeller and M. Baker. Personal-level Routing in Mobile People Architecture. *USENIX Symposium on Internet Technologies and System,* October 1999.

[30] B. Raman, R. Katz and A. Joseph. Universal Inbox: Providing Extensible Personal Mobility and Service Mobility in an Integrated Communication Network. *3rd IEEE Workshop on Mobile Computing Systems and Applications*, Monterey, CA, December 2000.