User Agents and Flexible Messages: A New Approach to Wireless Two-Way Messaging

Thomas Y.C. Woo Thomas F. La Porta Krishan K. Sabnani

Networking Software Research Department
Bell Laboratories
{woo,tlp,kks}@research.bell-labs.com

Abstract

Wireless messaging, in the form of two-way paging, is an integral part of universal Personal Communications Services (PCS). Basic wireless messaging services include providing reliable (acknowledged) message delivery, reply capabilities, and message origination from a messaging device. Many more advanced services can also be envisioned.

Wireless networks and end devices impose many limitations on system design. To overcome the problems caused by such an environment, we have introduced network based proxies, called *user agents*, to assist simple end devices, and a novel way to define messages, called *flexible messages*, so that advanced messaging services may be offered. In this paper, we describe how user agents and flexible messages assist in providing messaging services in the Pigeon two-way messaging research prototype at Bell Laboratories.

1 Introduction

Wireless messaging, specifically one-way paging, is the most popular consumer wireless service [12]. Its growth is expected to continue and new services are expected to be introduced. A new dimension, a return channel, is now being incorporated into wireless messaging systems. This additional component vastly affects the design constraints and services that can be offered by such systems. The trend toward "two-way" wireless messaging is evidenced by the creation of national and international standards [8, 10, 11], and vendor-specific protocol definitions [2, 3, 7].

Pigeon is our proposal of a wireless two-way messaging system. It provides duplex message communication among subscribers carrying self-contained portable wireless messaging devices, to be called *two-way pagers* (or pagers in the short) in the sequel. Unlike traditional pagers, two-way pagers are capable of originating as well as receiving messages.

The basic service offered by Pigeon is reliable end-to-end "flexible" messaging which is a compromise between traditional paging and electronic mail. It improves on traditional paging by adding message acknowledgment, pager initiated messages and direct reply capability; it is not as general as electronic mail in that it does not allow free-form messages.

The challenges with offering useful two-way messaging services arise from the service environment, namely wireless network access and limited end devices. The wireless access links often provide asymmetric bandwidth, with the effective uplink bandwidth limited to one-tenth or less of the downlink bandwidth in many cases. The messaging devices are limited in terms of processing power, battery power, and memory. They are also limited in their input/output capabilities, with only a few buttons and small display screen.

To address these challenges, we have introduced a network element called a user agent in the Pigeon messaging system. A user agent is a network-based proxy that assists simple end devices in service related processing, enabling advanced, flexible messaging services. Each pager device has its own user agent. Together, the pager and user agent cooperate to provide the messaging application. The user agent allows coded messages to be transmitted in the uplink to help alleviate limitations of bandwidth asymmetry; it provides services related processing to allow the pager to act as a "thin" client and operate within its limitations; and it supports "flexible" messages to overcome the input/output limitations of the end devices.

The balance of this paper is organized as follows. In the next section, we lay out the requirements and constraints in the design of Pigeon. In Section 3, we give a description of the key features of Pigeon, focusing on the user agents and flexible messages. In Section 4, we illustrate the operation of Pigeon with

0-8186-8061-X/97 $10.00

a complete service scenario. For a more exhaustive and detailed description of Pigeon's architecture, we refer the readers to [13]. In Section 5, we present the conclusion.

2 Requirements and Constraints

Wireless two-way messaging can take many different forms. For example, running regular wireline messaging applications on a portable computer over a wireless link can be considered a form of two-way messaging, albeit inefficient. Thus, to motivate our design of Pigeon, it is helpful to first lay out the requirements and constraints that must be considered.

2.1 End Device

Among the different forms of wireless messaging, one-way paging is most dominant. In the United States alone, there were more than 20 million active pagers in 1994. This number is expected to double by 2,000 [12].

The success of one-way paging can be attributed to a number of factors: (1) Pagers are completely self-contained and truly portable, providing a good platform for "anywhere, anytime" service. Portable computers or PDAs, though more powerful, have not become widely adopted as platforms for wireless messaging applications. (2) Paging systems are easy to use, for both senders and receivers. (3) Paging, unlike telephony, is asynchronous; the sending and receiving actions are decoupled. A receiver need not be online for a sender to send. (4) Paging service is relatively inexpensive, partly due to its datagram nature, which requires no dedicated connection. (5) Paging provides fairly good performance.

The appeals of traditional paging must not be overlooked. On the other hand, limitations of the end device could constrain the functionalities a system can offer. For example, the lack of a keyboard on a messaging device makes entry of free-form messages impractical.

In Pigeon, the messaging device is assumed to be similar to today's one-way pager. Specifically, it has 6 buttons (4 on the bottom as soft keys and 2 on the side mainly for scrolling), and a 5-line LCD where the top 4 lines are for general text display while the bottom line is reserved for soft key status. Though the pager is programmable, it is not suited for implementing complex processing logic. Its memory storage is minimal, in the order of 1M bytes. Thus, an important design challenge is to define a "maximally" useful form of two-way messaging under the constraints of a pager-like end device.

The user agents in the Pigeon system overcome two of the major constraints of the messaging devices. First, by offloading much of the service specific processing from the pager into the user agent, the software on the messaging devices can be simpler and smaller. This helps alleviate processing and memory constraints of the end device. Second, by mating the pager with a user agent, we effectively run a distributed messaging application, with peer applications executing on the pager and the user agent. This allows the pager and user agent to cooperate and support what we call *flexible messages*. Flexible messages give the subscriber much more freedom in composing messages thus overcoming some of the input/output constraints of the pagers and making the messaging service more natural to use.

2.2 Wireless Communication

The Pigeon two-way messaging system is intended for a wireless environment, and its design and operation are heavily influenced by the characteristics of such an environment. We briefly present these characteristics below.

Limited and Asymmetric Bandwidth. When compared to wireline communication, wireless communication bandwidth is inherently limited. As a result, wireless messaging is more oriented toward short, and mostly text, messages.

In addition, the forward and return channel bandwidth are often asymmetric. Specifically, the forward channel, powered by the network, has significantly higher capacity than the return channel, which is powered by the subscriber end device. Protocol design must take such asymmetry into account.

Disconnection. In a wireless environment, a disconnected state could be as common as a connected state. Disconnection can be long (e.g., device powered off) or intermittent (e.g., device out of range). This requires the system to accommodate disconnected users.

Mobility. A subscriber can move from the coverage of one base station (cell) to another, in which case, messages destined for her must be redirected to the new cell. This procedure is called a *handoff*. Unlike voice applications, the problem of handoffs for messaging is much simplified. Specifically, there is no

connection to be preserved, and the real-time requirement is significantly less stringent.

User agents overcome these considerations imposed by the wireless environment. First, because the pager and user agent execute peer applications, communication between these entities can be highly compressed. This allows the system to greatly reduce uplink bandwidth utilization. Second, because user agents are resident in the network and are persistent, they may be used to overcome effects of end device mobility and disconnection. This increases the reliability of the system.

3 Pigeon System Description

The services offered by a generic two-way messaging system can be very general. In Pigeon, we focus primarily on wireless-related issues. Specifically, we consider only service scenarios where at least one end point is a portable two-way pager device.

Pigeon provides a range of messaging services. Some of these (e.g., *reliable* delivery) are enabled by the availability of a return channel, while some others (e.g., *personalizability*) by the introduction of user agents. Some (e.g., *group addressing*) are used in wireline messaging, while some others (e.g., *transactions*) are new in messaging, wireline or wireless.

Each subscriber maintains a *personal* profile in a user agent, which is a highly available proxy of the subscriber inside the system. The profile contains the subscriber's personal address book, message list, forwarding and filtering instructions (see Section 4), etc. The profile can be dynamically updated by the subscriber, via both wireline or pager (for certain entries only).

In Pigeon, a subscriber can address a message to a *group*. A group can be ad-hoc, pre-defined or pre-registered. An ad-hoc group is formed on the fly by the subscriber. A pre-defined group is set up before by a subscriber and is local to her. A pre-registered group is recognized system-wide, and is assigned a group address. The use of group addressing can lead to better resource utilization, e.g., by exploiting multicast transmission.

A *transaction* refers to a request-response interaction. Most messaging applications are transaction-oriented. Without transaction support, users must correlate a response to its request on their own, and recover if no response is received. Pigeon tracks transactions on behalf of users. It correlate requests, replies and acknowledgments. The support of transactions also indirectly enables query of message status on demand. Pigeon transactions may also be combined with group addressing to provide a transactional multicast service with varying semantics.

In Pigeon, a subscriber can send, receive and reply to messages via a multitude of end devices (e.g., pagers, telephones, computers) and networks (e.g., paging networks, public switched telephone networks, public switched data networks), depending on what is most convenient to her at the time. No prior agreement on the communication format is needed; automatic *format translation* is performed transparently by the system.

This form of universal and unified access is critical for next generation messaging systems.

3.1 User Agents

The concept of user agents is central in the design of Pigeon. In simple terms, a user agent is a representative or a proxy of an end device inside the network. Each end device has its own associated user agent. The two cooperate to facilitate message delivery functions.

The key motivation behind the introduction of user agents are twofold: (1) to address air interface and end device constraints; and (2) to enhance personalizability.

Regarding (1), a user agent allows reduced uplink bandwidth usage by supporting the transmission of coded messages on the uplink. This accounts for the asymmetric nature of wireless links. Second, a user agent serves as a repository for the information about its pager, e.g., its current status and location, and thus can act as the termination point for signaling protocols. This keeps pager processing and state simple, and reduces the amount of signaling over the air interface. Third, the user agent overcomes problems caused by mobility by providing location information to the rest of the system. Fourth, a user agent, being network-resident, is always online, unlike a pager that can be powered off or go out of range. It allows the system to gracefully handle disconnected users. Finally, by performing intelligent processing in the user agent instead of on the end device, we account for the asymmetric nature of processing power between the network and the end device.

Regarding (2), a user agent can perform customized messaging functions, e.g., selective message forwarding and message screening. It does this by maintaining a profile for the subscriber. Pigeon user agents are also programmable, thus making it a true personal

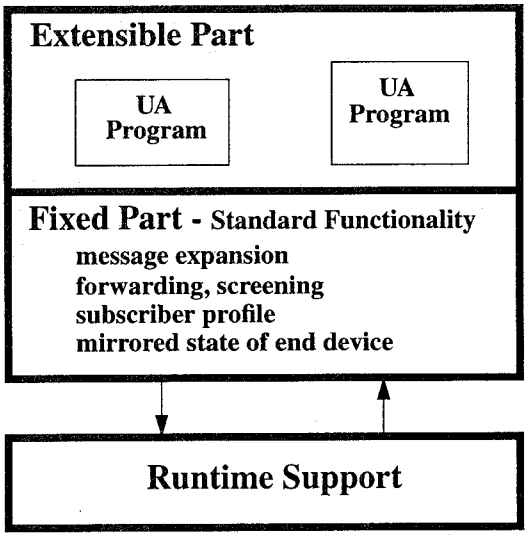

Figure 1: User Agent Structure

server. This adds another dimension to the functionalities that may be provided.

Structure

In our design, a user agent has a fixed and an extensible part (see Figure 1). The fixed part implements the basic messaging functionalities discussed above. These functionalities are generic for all user agents. For its operation, this part mimics the context of the pager device (e.g., the address table, the message table) and keeps information about ongoing message delivery. The extensible part, as the name suggests, can be programmed to perform specific tasks as desired by a subscriber. Some examples include maintaining a personal calendar, retrieving specific information from a WWW page, etc.

The operation of the fixed part is straightforward and is discussed in Section 4. Here, we focus on the extensible part. Strictly speaking, the extensible part specifies only a framework in which additional functions can be added to a user agent in the form of *UA programs*. A UA program is a collection of program blocks (called **on** blocks), each of which contains codes to handle messages of a specific pattern. The framework follows an event-driven model; it contains a kernel that pattern-matches incoming messages and dispatches them to the appropriate program blocks in UA programs. The precise set of primitives (e.g., **send**, **receive**, **forward**) allowed in a program block defines the capabilities available to a subscriber.

A UA program can be used to provide third-party value-added functions to a message flow or it can serve as the destination end point of a message flow itself. An example of the former use is a UA program that serves a supervisory function in an order placement message interaction between a subscriber and a stock trading server. Specifically, it ensures that enough funds are available before an order is sent to the stock server, and that the balance and trade is recorded properly when a confirmation is received.

An example of the latter use is message status query, where query messages sent uplink are addressed to and processed by user agents, possibly with the help of other servers in the system. It is in this regard that the extensible part bears special significance in relationship to signaling. Specifically, by addressing a message to the user agent itself, the extensible part could be used to implement direct user agent signaling functions. This turns a pager into a remote control for its user agent, and a user agent into a personal server for a subscriber.

3.2 Flexible Messages

Pigeon does not directly support free-form messages. Instead, it supports a novel class of messages, called *flexible messages*, that has been specially designed for use in two-way messaging. The flexible messages are supported through cooperation between the user agents and pagers.

Flexible messages are a form of "active" messages. One way to understand them is as small programs that are interpreted when they are sent or replied to. Like programs, flexible messages are constructed from a number of well-defined basic building blocks. Each basic building block provides a distinct form of dynamic customization. They can be combined or mutually nested to achieve maximum flexibility.

Each basic building block has associated with it a unique processing logic and input convention. They have been designed such that their implementation is well suited to a device with limited input capabilities, and their combinations can meet the demand required for most personal and server-based applications. As we will see, the processing logic, as well as the input interface, required to process these building blocks are relatively simple.

Table 1 gives a summary of the major building blocks; they are described in greater detail below. We note that the following description is not exhaustive, many features of flexible messages have been omitted for brevity.

Component	Use	Syntax in MDL	Example
Plain text	regular text	text	Care for lunch?
Rich text	highlight message parts	\<rich text specifier>{...}	Care for lunch \emphasize{now}?
Optional component	include or exclude message parts	\optional{...}	Care for lunch \optional{soon}?
User-defined selection	specifies a list of choices	\choice{... | ... | ...}	Care for \choice{lunch | dinner}?
Pre-defined variable	system defined selections	specific for each variable	Care for lunch at \time?
If conditional	two-way branch	\if{expr} ... \else ... \fi	\if{$r==10} high \else low \fi
Case conditional	pattern branch	\case{expr} ... \or ... \or ... \esac	\case{$c} OK \or Probably Not \esac
Reply component	delineate replies	\reply{...}	\reply{\choice{Yes | No}}

Table 1: Basic Building Blocks for Flexible Messages

Basic Building Blocks

The most primitive building block is *plain text.* As the name suggests, it refers to simple plain alphanumeric text. This corresponds essentially to what is currently available under one-way alphanumeric paging. An example of a plain text message is "Care for lunch?".

Rich text adds text attributes to plain text. Text attributes are typically used to highlight or emphasize the different parts of a message. An example of a rich text message is "Care for lunch \emphasize{now}?", which specifies that the part of the message "now" should be presented in emphasized mode. The available rich text specifiers and their precise implementations varies with end device used.

The notation "\emphasize{}" is part of a high-level language, called *Message Definition Language* (MDL), we have defined for specifying flexible messages. Its syntax should be self-explanatory, we omit its precise definition here. MDL can be used directly by an end user or as the back end language of a GUI-based message definition tool.

An *optional component* provides a way to include or exclude parts of a message. An example of its use is "Care for lunch \optional{soon}?" In essence, this defines two possible messages, "Care for lunch?" and "Care for lunch soon?" that can be dynamically chosen as desired.

A *user-defined selection* allows a subscriber to specify a list of choices among which a selection can be made. An example of the use of user-defined selection is "Care for \choice{ lunch | dinner}?" Again, this can be understood as standing for two possible messages, "Care for lunch?" and "Care for dinner?".

User-defined selections are defined on a per subscriber basis. This gives each subscriber the maximum flexibility to customize for her own needs. Certain selections are sufficiently common that it is advantageous to pre-define them for all subscribers to use. In other cases, it may not be easy to enumerate all the choices in a selection.

An example of the former is the selection "\choice{ Mon | Tue | $\cdots$ | Sun}", which allows a subscriber to choose a specific week day. An example of the latter is the entry of a phone number on demand. These limitations of user-defined selections can be overcome by using *pre-defined variables.*

A pre-defined variable behaves like a variable whose value can be dynamically customized. Unlike user-defined selections where all instances are processed uniformly, the processing logic for each type of pre-defined variable is different. For example, the pre-defined variable phone number is processed by displaying a phone number template of the form (XXX) XXX-XXXX and allowing the user to change each digit successively using up/down keys or the keypad.

In some situations, it may be advantageous to send multiple related requests in a single message. This allows logically related messages to be received as a unit and eliminates the round trip delay between successive interactions. For cases where a request is conditionally dependent on prior responses, *conditional components* can be used to make explicit the dependencies. There are two types of conditional components, namely, *if-conditional* and *case-conditional.* An if-conditional

Message $M1$:

```
Care for \choice{lunch | dinner}?
\case{$c}
    \choice{McDonald's | Taco Bell}
\or
    \choice{Olive Garden | Red Lobster}
\esac
```

Message $M2$:

```
Care for \choice{lunch | dinner} \optional{soon}?
\reply{
    \choice{Sure | I am busy}
    \case{$c}
        at \time
    \or
    \esac
}
```

Figure 2: Flexible Message Examples

takes a mathematical expression as the branch condition and allows a two-way branch. A case-conditional takes a pattern (e.g., a choice response) as the branch condition and allows a n-way branch.

As an example, consider the message $M1$ in Figure 2. It illustrates the use of a case-conditional. The branch condition "$c" denotes the last choice selected; it evaluates to a value between 0 and $n-1$, where n is the number of choices in the last selection. In the case "lunch" is selected, the first branch of the case will be taken, i.e., the user will be asked to select between the choices "McDonald's" and "Taco Bell." The case when "dinner" is selected is similar.

Finally, a *reply component* is used to embed the possible replies along with the request. This reduces the reply step to that of processing the reply component, thus eliminating the need for free-form input. A reply component is denoted by "\reply{}." It is constructed in the same way as a request, except it cannot contain a nested reply component. In other words, the expressive power of a request and a reply is identical.

An example of the use of reply component is provided in message $M2$ in Figure 2. The embedded reply in this case is constructed with a choice component together with a case-conditional. The meaning of the message should be clear, we omit its explanation here.

A flexible message is *two-way* if it contains a reply component, otherwise it is *one-way*. A two-way message is a message that may be replied to. It is used for interactive messaging, either person-to-person or server-based. A one-way message is unidirectional, i.e., no reply can be sent in its response. A one-way message is used in situations wherein a reply is not needed or desired. For example, informational messages (e.g., news) are typically sent as one-way messages.

We now describe the method for generating and sending messages. First, a message must be entered into the system. This requires that the message be stored in the user agent and pager. The messages are stored in a format called *storage encoding*. Once a message is in the pager, it may be selected by the subscriber to be transmitted. This triggers *message processing*. Finally, the message is translated into a highly compressed format called *transfer encoding* and sent uplink into the network. Below, we describe each of these steps.

Message Processing

Once a message is in the pager it may be selected for transmission. A message is processed by a pager when it is being sent or replied to. The former is referred to as the *send-side* processing while the latter *receive-side* processing. Send-side processing is performed before a new message originates from a messaging device. It allows a sender to customize a request. Receive-side processing is performed when forming a reply. It lets a recipient answer each component of a request.

The actions taken in send-side and receive-side processing are similar. In both, each basic building block has associated with it an evaluation procedure that, when invoked, returns a value. For example, the evaluation of a plain text returns itself, while the evaluation of a user-defined selection returns one of the choices. The evaluation of a building block may trigger the evaluation of its component building blocks. In other words, evaluation of building blocks proceed from the outermost level in a recursive fashion.

As an example, the evaluation of message $M2$ in Figure 2 proceeds as follows: In the send-side processing, the request part of the message, i.e., "Care for \choice { lunch | dinner} \optional{soon}?" is evaluated. The evaluation proceeds sequentially from the first component, the plain text block "Care for", then the next component, the user-defined selection "\choice { lunch | dinner}", then the next component, the optional component "\optional{soon}", and finally to the last component, the plain text "?". The values from each of the evaluation are then combined to form the final request.

The receive-side processing is similar, except that it acts only on the reply component and the components contained therein. We omit its description for brevity.

Transfer Encoding

As part of the send-side processing, a message is converted into transfer encoded format. Transfer encoding highly compresses the message and relies on the user agent to decode and expand the message.

As described earlier, a user agent mirrors the state of a pager. In other words, they both store the same set of addresses and messages. Thus, to send a message uplink, all that needs to be transferred is (1) a list of address aliases; (2) reference to the desired message; and (3) a modifier that encodes the customizations to be applied to the message. This is an application of the standard *differencing* technique, where the difference (encoded by the modifier in this case) from a base (the reference in this case) is sent instead of the value itself. The technique is most useful when the size of the difference is small compared to the true value; this is almost always the case in Pigeon.

As an example, consider the message $M2$ in Figure 2. There are two dynamic constructs in the request part of the message, therefore the modifier comprises of two parts, each specifying the customization for one of the constructs. Since each of the constructs can evaluate to two possible values, "lunch" or "dinner" for the user-defined selection and "include" or "exclude" for the optional component, there are four possible modifiers possible depending on how the user chooses to customize the request. Thus, if $M2$ is stored at location 5 in the message table, then the possible transfer encodings for the uplink message are: (The address information is omitted for brevity.)

- 5 0 0 — if the sender chooses "lunch" and to exclude the word "soon" or
- 5 0 1 — if the sender chooses "lunch" and to include the word "soon" or
- 5 1 0 — if the sender chooses "dinner" and to exclude the word "soon" or
- 5 1 1 — if the sender chooses "dinner" and to include the word "soon."

By the same token, the possible replies to this message are encoded as: (We assume 10 to be the unique message identifier for the above request. It must be included to identify the original request for this reply.)

10 0 timeval

if the reply is "Sure", where timeval is some encoding of the time entered by the recipient, or

10 1

if the reply is "I am busy."

Essentially, the modifier encodes the sequence of actions that needs to be taken (by the user agent) to recover the desired message from the stored message. The generation of the modifier is performed in an incremental fashion during receive-side processing.

A study of typical messages sent using the Pigeon system reveals that the average payload size of transfer encoded messages in only 3% of the unencoded message size. When mapped to the IS-136 Short Messaging Service protocol [10], this results in a bandwidth savings of approximately 65% [9].

4 A Complete Service Scenario

In this section, we illustrate the end user view and operation of Pigeon using a common service scenario, namely, person-to-person transactional messaging. To keep the description at a high level, we have abstracted away many of the details. In this example, we discuss the details of the send and receive-side processing, and the operation of the user agents. We illustrate the message coding used in the system. for the coded message formats.

Consider Figure 3. S (message sender) and R (message recipient) are subscribers of Pigeon. User agents S and R represent, respectively, their user agents inside the system. To simplify the discussion, we have lumped the other control functions of Pigeon together, and labeled them "Pigeon Control Functions" (PCF in short). In the following, we use S to refer to both the user S as well as the pager of S, and UA-S to refer to the user agent of S.

In this scenario, S originates message $M2$ (in Figure 2) to R from her pager, the message is delivered via Pigeon to R. R replies directly from her pager, and the reply is delivered via Pigeon back to S.

Message Setup. We assume that previous to sending message $M2$ to R, user S has entered both the message and the address alias for R into her pager and user agent. Thus, both S and UA-S have message $M2$ in their message tables and an alias for R in their address tables. We assume $M2$ and R occupy respectively the 5th and the 3rd entries in their tables.

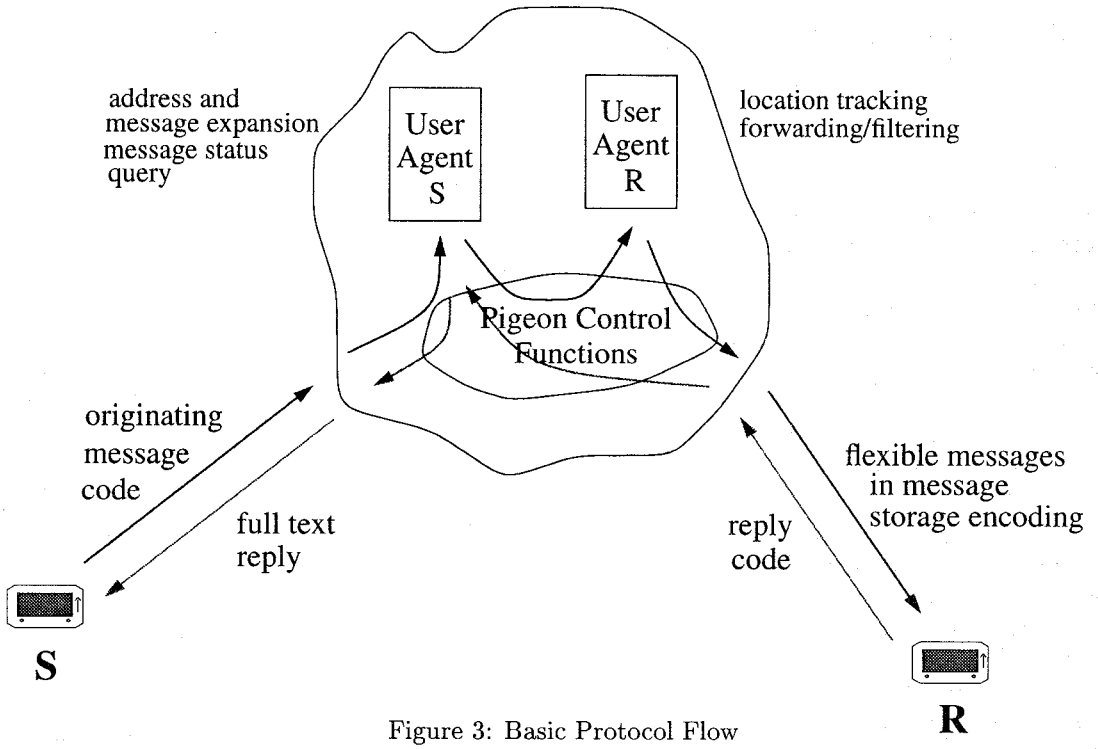

Figure 3: Basic Protocol Flow

Message Origination. To originate the message, S selects the "send" function on the pager. This takes S through a number of interactions:

1. Message Select — S selects the desired message from the message table on the pager. The selection of a message also triggers the send-side processing on the message. In this case, S selects message $M2$. The send-side processing comprises the evaluation of two constructs: (1) the user-defined selection, "\choice{ lunch | dinner}"; and (2) the optional component, "\optional { soon }". The evaluation of (1) prompts S to choose between "lunch" and "dinner," while the evaluation of (2) prompts the user to include or exclude the word "soon." In the present scenario, S chooses "lunch" for (1) and to include "soon" for (2). In other words, the request part of the message is finalized to be "Care for lunch soon?", which is coded with the message number "5" and a modifier (returned by send-side processing) of "0 1" (see Section 3.2).

2. Destination Select — S chooses a destination and a destination format, which specifies the media (e.g., page, email, phone) in which S would like the destination to receive the message. The destination format is only a proposed format from S; it can be overwritten by a format specified by R in her user agent. In case of multicast, S specifies multiple destinations and destination formats. In the present scenario, S desires only a point-to-point messaging to R, so she chooses R as the destination (coded as "3"[1]) and page as the destination format (coded as "0").

3. Reply-to Select — S specifies a reply-to address and a reply-to format. The reply-to address typically refers to the sender herself, and the reply-to format allows a sender to receive the reply in a different format from the request. Multiple reply-to addresses and formats could be specified if the reply needs to be copied to other users; this is treated as a multicast of the reply. In the present scenario, S specifies respectively herself as the reply-to address (coded as "0") and page as the reply-to format (coded as "0").

4. Transaction Timeout — $M2$ is a two-way message, i.e., a reply can be sent in its response. With the transaction support, Pigeon can track the reply and correlate it to the request, if and when it is received. S could specify a timeout beyond which a reply is no longer desired, or

[1]There is actually a use count associated with each destination address. The pager (and the user agent) is intelligent enough to use a short code for a frequently used address.

a undeliverable notification should be returned. In the present scenario, S chooses the system default (coded as "0"), which is 2 hours.

As S is completing the message origination steps, a message origination packet in transfer encoding is incrementally constructed in the pager's send buffer. Upon S's confirmation, this packet is sent uplink to the system.

User Agent Processing. When a message origination packet is received by the system, it is routed first to the user agent of the sender (UA-S in the present scenario) and then to the user agent(S) of the destination(s) (UA-R in the present scenario).[2] We describe the processing at each user agent:

1. User Agent S — The primary function of UA-S is to expand the address alias to its network address and the message number/modifier pair to the full message. The address expansion is straightforward indexing, while the message expansion involves applying the modifier to the message indexed by the message number. The expansion returns a full message ready for delivery in storage encoding format.

   Before returning the full message to PCF, UA-S checks to see if any UA programs resident in its extensible part are enabled. If so, the corresponding UA program(s) is executed.

2. User Agent R — UA-R is consulted for information needed in the delivery of the message. These include: (1) The current status (online or offline) of R. If R is offline, the message could be held for later delivery in a *message storage server*. When R registers, the message will be downloaded; (2) Last known location of R. This provides a starting point in locating R; and (3) Forwarding/filtering instructions. These are R's preferences of when, from who and in what format she would like to receive messages. The forwarding/filtering criteria could be very general, or even programmatic.

Concerning the PCF, it suffices for the present discussion to say that it performs two major functions. First, it creates a transaction for the message. Second, it is responsible for the actual delivery of the message.

[2]To be accurate, the entire packet is not routed. Instead, the PCF keeps the packet and invokes remote operations at the respective user agents.

Message Reply. Message reply is a dual of message origination. It is much simpler, however, because most of the information needed (e.g., destination) is already contained in the message itself. It retains essentially only the message select step. Instead of selecting the message to reply from the message table, it is selected from the receive buffer. When a message is selected, receive-side processing is performed. The receive-side processing results in a modifier that is sent as part of the reply code.

Generally, the reply code is much simpler than the originating message code. In particular, it does not contain any address information. This is possible because the reply id uniquely identifies the original request (cached in the PCF), which in turn contains the address information.

Reply Processing. The reply processing is also very similar to the request processing, except for two differences: (1) The role of the message originator and recipient is reversed. For example, the user agent of S, the recipient of the reply, now performs the functions (described earlier) that UA-R performed in request processing. (2) The user agent of R, now the sender, is bypassed. This shortcut routing is possible because both address and message expansion could be done using the cached copy of the request inside the PCF.

5 Conclusion

Pigeon is our proposal of a wireless two-way messaging system. It is novel from both the end users' as well as the system design perspectives. From the end users' view, Pigeon services are ubiquitous and versatile. In addition to the basic services (e.g., reliable end-to-end messaging), Pigeon provides new messaging functionalities such as multicast message delivery, transaction support, and user extensible services. The concept of flexible messages deserves particular attention. Flexible messages can be understood as small messaging programs (i.e., applets), which when evaluated, produce a set of instantiations for its component constructs. Unlike other types of applets (e.g., Java applets) however, flexible messages are self-contained. That is, each flexible message can be evaluated completely on its own without extra linking steps.

Most of the Pigeon functionalities are not available on existing wireless messaging systems [6]. Their inclusion in Pigeon strikes a careful balance between features and practical (communication, software, interface) constraints.

In terms of system design, Pigeon incorporates many techniques for mitigating the constraints presented by the end device and the wireless operating environment. Two of the key innovations are: (1) the use of asymmetric protocols to accommodate the discrepancy in the capabilities (e.g., bandwidth, processing power) between the pager device and the network; (2) the introduction of network resident user agents as proxies for subscribers inside the network. In addition to its signaling functions, Pigeon user agents are programmable on a per user basis, and thus can perform value-added intelligent services on behalf of their subscribers. In this setup, the end device becomes a remote control for the user agent, which in turn serves as a personal server for a subscriber. This makes our notion of user agents more general than the ones typical in mobile computing [1, 4, 5].

The design of Pigeon has been completed. A prototype has been implemented and is operational at Bell Laboratories. The current prototype makes use of a software pager simulator and a wireless LAN as the air interface. The Pigeon servers are implemented using C++ as Unix processes on the SUN Solaris 2.X platform. A graphical protocol visualization tool has also been developed to provide a real-time view of all message exchanges between the servers.

A second prototype is being developed that integrates Pigeon with the short messaging service under IS-136 [10]. The pager code (including the interpreter for flexible messages) has been ported to an IS-136 digital cellular phone. The phone uses a 8-bit Motorola microcontroller as its CPU, has 256K ROM (for code) and 8K EEPROM (for message and address tables, send and receive buffers, etc.) memory, and is controlled by an embedded message-passing operating system. Its user interface consists of a 3-line LCD with 12 characters per line, 10 scrollable soft keys and a standard keypad. Despite its limitations, our experience with the Pigeon implementation on it has been positive. In particular, the flexible messaging interface has been surprisingly efficient. We are able to navigate through the menus with ease.

Using the messaging services provided by Pigeon for message transport, we are exploring how to provide higher level services, such as remote wireless access to email and a limited form of Web access.

References

[1] A. Athan and D. Duchamp. Agent-mediated message passing for constrained environment. In *Proceedings of Mobile & Location-Independent Computing*, pages 103–107, Cambridge, Massachusetts, August 2–3 1993.

[2] AT&T Wireless Services. *personal Air Communications Technology — pACT Specification Release 1.1*, October 1 1995.

[3] T. Koljonen. A feature-rich Pan-European paging service. *Telecommunications (International Edition)*, 27(10):57–58, 61, October 1993.

[4] M.T. Le, F. Brughart, S. Seshan, and J. Rabaey. InfoNet: The networking infrastructure of InfoPad. In *Proceedings of Compcon*, pages 163–168, San Francisco, California, March 5–9 1995.

[5] G.Y. Liu, A. Danne, A. Marlevi, and G.Q. Maguire, Jr. A mobile-floating agent scheme for wireless distributed computing. In *Proceedings of IEEE Symposium on Personal, Indoor and Mobile Radio Communications*, pages 100–104, Toronto, Canada, September 27-29 1995.

[6] Mobile Telecommunication Technologies Corporation. *Skytel 2-Way Technology Backgrounder*, 1995. Available from http://www.skytel.com/products/st2way.html.

[7] Motorola Advanced Messaging Group. *The FLEX Story — FLEX Technology*, 1995.

[8] M. Mouly and M.B. Pautet. *The GSM System for Mobile Communications*. 1992.

[9] T.F. La Porta, R. Ramjee, T.Y.C. Woo, and K.K. Sabnani. Experiences with network-based user agents for mobile applications. *ACM/Baltzer Mobile Networks and Nomadic Applications*. accepted for publication.

[10] Telecommunications Industry Association. *TIA/EIA IS-136 800 MHz TDMA Cellular — Radio Interface — Mobile Station — Base Station Compatability — Digital Control Channel*, December 1994.

[11] Telecommunications Industry Association. *TIA/EIA IS-41 (Revision C) Cellular Radiotelecommunications Intersystem Operations*, May 4 1995.

[12] M. Tomlinson. *Wireless Data Networks — Emerging Technologies and Market Opportunities*. Telecom Publishing Group, 1995.

[13] T.Y.C. Woo, T.F. La Porta, and K.K. Sabnani. Pigeon: A wireless two-way messaging system. *IEEE Journal on Selected Areas in Communications*, 15(8), October 1997.