Customizing Mobile Applications

Bill N. Schilit *
Computer Science Department
Columbia University
schilit@parc.xerox.com

Marvin M. Theimer
Palo Alto Research Center
Xerox Corporation
theimer@parc.xerox.com

Brent B. Welch
Palo Alto Research Center
Xerox Corporation
welch@parc.xerox.com

Abstract

The dynamics of mobile systems require applications to intelligently adapt to changes in system configurations and to their environment. We describe a workplace in which users interact with a number of stationary and mobile systems through the course of a day. The relationship between systems and devices is constantly changing due to user mobility. We present a facility for mobile application customization called "dynamic environment variables." The facility allows flexible sharing of customization contexts, supports short interactions with long term services, and provides efficient notification of environment changes to applications. A sample application built in PARC's mobile computing environment and initial performance evaluations are described.

1 Introduction

Mobile computing differs from desktop computing because of the dynamic nature of system state. In our research lab mobile users interact with mobile computers as well as a world of stationary and embedded systems [8]. As users move, the sets of stationary and mobile objects they deal with changes. This dynamic state is not just a problem, it is also an opportunity for applications to intelligently adapt to their environment.

This paper describes a facility for *dynamic customization* of applications. In contrast to the Unix practice of one time initialization at program start up, dynamic customization supports changing application preferences at any time during program execution. In particular, the system modifies application preferences in response to changes in the physical computing environment.

Most software is designed with the assumption that the system it operates in will not change much over time. Computers are configured according to the hardware on the desktop, the printers down the hall, and the file servers on the local network. Each computer is tightly knit into a particular environment. It is therefore not surprising that Unix programs, for the most part, have configurations that are difficult to change. Frequently, the only time that a configuration can be set is at program startup.

In contrast, here are some examples where changing an application's customization values after startup is necessary:

- Selection of proximate devices such as printers and displays.
- Application response to migrating X windows among different kinds (i.e. color and monochrome) of displays.
- Application response to being "docked" at high speed network taps versus operating over slow speed links.
- Managing and moderating resources, such as local disk space depending on network performance.
- Supporting security in an application by means of auto-lockout based on the user's location relative to his mobile computing devices.

[1] This work was supported by Xerox. Portions were also paid for by ARPA under contract DABT63-91-C-0027.

* Currently visiting Xerox Palo Alto Research Center, 3333 Coyote Hill Rd., Palo Alto, CA 94304

We believe that existing mechanisms for program customization limit the usefulness of mobile applications. Consequently in this paper we introduce a general design for dynamic reconfiguration of applications. The next section summarizes Unix techniques for initialization and customization and explains why these are not adequate for many mobile computing applications. Section three presents our approach based on a community of environment variable servers. This is followed by a more detailed description of interfaces, commands, server interaction and performance.

2 Unix Initialization and Customization

Application *customization* is the setting of parameters (*preferences*) by users in order to control the appearance and function of an application. Often, user preferences are expressed at *initialization time* when the application starts running. Unix programs employ a few different facilities for program customization:

- command line arguments
- environment variables
- "rc" initialization files
- databases like Xrdb [1]

Customization by environment variable dates to the era of "dumb terminals" before window based workstations. In this situation a user has one screen and runs one process at a time. The inheritance of environments from parent process to child was sufficient because all work was carried out in a subprocess of the interactive shell. This mechanism has survived into the "windowing workstation era" but its limitations are evident to anyone who has tried to maintain a uniform environment among multiple shell windows. To some extent a workstation wide database, like Xrdb, acts as a replacement for environment variables in a windowing system.

We are entering a third era of computing in which users interact with a multitude of stationary and mobile systems through the course of a day. Our lab has built electronic whiteboards, mobile stylus-based notebooks, and personal communication devices that are in a constantly changing relationship to each other because of user mobility. Mobile systems need to maintain and share a context that is not limited to the boundary of process hierarchy, window manager, or host.

In addition, applications must separate customization from initialization if they intend to support environment changes during program execution. Historically most applications have run from start to stop in a short time frame. Today, however, we run applications, such as mail and emacs, that persist over very long time periods. We evolve a context of windows, editor buffers, command history, et cetera, which is expensive to recreate. As a consequence, mobile computing requires quick instantiation of a context as well as persistence across sessions. For example, a context that follows the user is useful for *migrating application windows*, where windows move from one display to another following a mobile user. For security reasons it is desirable that a user walking away from a device will cause the session on that device to become suspended, logged-out or locked out.

Unix applications provide an approach to separating initialization from configuration. Non-interactive servers, by convention, re-read configuration files when they receive the `HUP` signal; X window managers (like twm) support the same facility through a menu command. In both cases, reconfiguration requires manual intervention and is often heavy-handed, with the entire internal state being destroyed and recreated. There are two problems with this approach. First there is no facility for incremental changes and therefore reinitialization resets all state where perhaps only some should change, and second, it is based on configuration files that need to be changed, and hence constantly monitored by interested parties.

The X windowing system provides a server-based resource manager, Xrdb [3], for communicating user preferences to applications. The advantage of Xrdb is that clients accessing a central server do not need a common or duplicate collection of initialization files on all hosts. Although the database-server approach has the advantage that it allows changing parameters without editing files on disk, it does not notify existing applications of those changes. One could imagine extending the notion of X-events or Xrdb to solve these problems. A new type of X-event, for example, could indicate what environment information has changed. However, this solution implies that applications would require X servers on any hosts on which they are run.

[1] Xrdb, the X resource manager, is the part of X11 window system that manages user preferences about colors, fonts, etc. Each X11 screen has an associated Xrdb server process.

Rather than focus on a limited solution, we describe a general RPC-based one in the following section.

3 A Community of Dynamic Environment Servers

A *dynamic environment server* is a program that manages a set of variable names and values (an *environment*) and delivers updates to clients that have previously shown interest by *subscribing* to the server. Typically there is one environment for each user, plus environments for rooms, work groups, et cetera. Clients use RPC to get variables from servers, and servers use RPC to provide callbacks to clients. The key feature of an environment server is that environment changes are quickly seen by multiple processes through the use of RPC callbacks. The environment servers maintain a shared context for mobile applications that is not limited to a process hierarchy or a particular window server. More important, however, is that environment servers can help model the dynamics of a mobile computing environment by providing a simple and standard interface for change notification.

We've designed an architecture that includes a community of environment servers used in conjunction with a *user agent* [7] managing user information. Environment variables describing a user are maintained by the user agent, which also keeps track of the user's current location. Applications subscribing to the user's environment will automatically be notified of changes, including their owner's location.

As an example, consider two environment variables `LOCATION` and `NEAREST_PRINTER` and how they might be used. As a mobile user moves, his location is tracked by the user agent, which updates `LOCATION` in the user's environment. Applications subscribing to this environment get notified of, among other things, location changes. Applications generally subscribe to the user's environment as well as a number of environments based on the user's current location, for example, a room. As the room changes, an application cancels subscriptions from the previous room and subscribes to new environment servers for the current room. An environment variable in such a server might be `NEAREST_PRINTER`.

It is important to remember that environments are dynamic and their contents may be frequently changing. For example, the environment for a room might include a variable for the occupants:

```
OCCUPANTS=adams:schilit:theimer:weiser:welch
```

This variable would be maintained by updates from user agents since they are the source of location information about a particular user. As the `OCCUPANTS` variable changes in the room's environment server, programs subscribing to the room are notified and may perform any re-customization that is applicable.

In summary, we are building a system of environment servers that has the following structure:

- Multiple environment servers run on the network. There is one environment for each mobile user and other environments are maintained for locations (e.g. rooms and offices).
- For efficiency there may be more than one environment per environment server.
- The location of the user is stored in the user's environment variable `LOCATION` maintained by the user agent.
- When the `LOCATION` variable changes applications are notified and they may choose to subscribe and/or drop existing subscriptions to environments based on the changed user environment .
- Subscribing to new environments is facilitated by attributed-based names. Environments that depend on a location are registered with attributes for the specific location.

4 Dynamic Environment Service

The generic interface presented here is similar to Unix environment variables that have a key and a value. The interface differs from Xrdb and Unix environment variables in two ways: (1) distribution: sharing is possible between process, jobs, and machines (2) asynchronous notification of change.

4.1 RPC Interface

The dynamic environment interface is an RPC based service consisting of the following functions:

- `subscribe(e,cl)`

  Request notification, via a callback to closure `cl`, when changes occur in the specified environment `e`. The closure specifies the procedure to be called and its parameters.

- `cancel(e)`

  Cancel a subscription and stop receiving notifications about changes to the environment.

- `get(e)`

  Fetch the entire environment `e`.

- `setenv(e,k,v)`, `getenv(e,k)`, `unsetenv(e,k)`

  Similar to the shell functions for environment variables.

4.2 Attribute-based Naming

This section describes how connections are established between clients and environment servers. A name server, described below, is used by clients to locate an environment server. Clients then issue a subscribe RPC and start receiving all values from the server. In our system configuration we expect a small number of variables per server, however, for scaleability it might be necessary to add an extra step to the subscription process whereby clients select the subset of variables that are desired.

The system design of mobile clients and many servers requires a flexible way to bind one to the other. Our approach is for environment servers to be found using associative matching on attributes. For this purpose we use an attribute based naming system. This works as follows. Dynamic environment servers are assigned a set of attributes. If items maintained by the server can be thought of as having a location, then one attribute for the server is "location," whose values are a canononical form used throughout our lab. Subscribing to servers is accomplished by matching a subset of these attributes.

Attribute based naming has a number of advantages over other styles of naming for mobile computing. First, naming by a client's requirements rather than a server's host name makes for more "portable" systems. This allows flexible binding to servers by attributes instead of by host names or other physical characteristic that may change over time. Also, attributes allow matching by as few or many properties as is best suited to the application at hand.

The lookup specification and the registration for an item have the same form: an expression of keys, values, and nested keys and values. The specification, used for looking up an item, may contain fewer key-value pairs than the registration. In this case only key-values present in the specification determine the correctness of the match. More formally, the expression syntax is:

```
EXPR    ::= value | ( LIST )
LIST    ::= LIST PAIR
PAIR    ::= ( key EXPR )
```

where `key` and `value` are text strings. For example:

```
((Type environment) (Kind printing) (Location ((ID "35-2172"))))
```

can be used as a query to find a server exporting a set of "printing" environment variables for a room with ID 35-2172. These might include:

```
PRINTER=Snoball
NEARBY_PRINTERS=Snoball:BerkeleyBarb
```

4.3 Shell Commands

This section describes a set of Unix commands for maintaining dynamic environment variables. These commands are similar to and modeled on the shell's environment variable commands. The differences are that dynamic environment commands can specify the server (defaults to the user's server), and the command to print a dynamic environment has an option like `tail(1)` for following the changes to a dynamic environment.

- `setdenv [ -env EXPR ] [ VAR [ = word ] ]`

  With no arguments, `setdenv` displays all dynamic environment variables from the environment server registered as `$USER`. With the `VAR` argument, it sets the environment variable `VAR` to have an empty (null) value. With both `VAR` and `word` arguments, `setdenv` sets the environment variable `VAR` to the value word, which must be either a single word or a quoted string.

  After setting the dynamic environment variable all processes subscribing to the environment are notified. [2]

  If the `-env` switch is specified then an environment server matching the attributes specified by `EXPR` (as described in section 4.2) is used. For example:

  ```
  setdenv -env '((User brown) (Location Library))' biff = n
  setdenv -env '((User brown) (Location Library))' menu = lookbib
  ```

  will set the variable `biff` and set the variable `menu` in all environments that have the `(User brown)` and `(Location Library)` attributes. These commands might be used to disable mail notices and to add the `lookbib` program to a menu set while user brown is in the library.

- `unsetdenv [ -env EXPR ] VAR`

  Removes a variable from the environment. After unsetting the dynamic environment variable all processes importing the environment are notified.

  If the `-env` switch is specified then an environment server matching the attributes specified by `EXPR` is used.

- `printdenv [ -f ] [ -env EXPR ] VAR`

  `printdenv` prints out the values of the variables in the environment. If a variable is specified, only its value is printed.

  If the `-f` switch is specified then the program enters an endless loop, reporting any changes to environment variables as they occur.

  If the `-env` switch is specified then an environment server matching the attributes specified by `EXPR` is used.

4.4 Dynamic Application Design

In this section we describe the types of modifications that are necessary for applications to support our approach. The way dynamic environment variables are introduced into applications depends on the existing application framework. Environment changes occur asynchronously to the executing application; however, applications unprepared for concurrency (e.g. lacking locks) cannot easily field asynchronous notifications. To address this we provide different interfaces for applications designed with concurrency in mind and those that are single threaded.

Figure 1. categorizes the software mechanisms available to Unix applications for inter-process communication. These reflect the design choices for supporting a Unix based dynamic environment that relies on communicating values over the network or between processes. The figure is divided into techniques for multi-threaded applications capable of concurrency, and applications built with a polling model and not capable of concurrency. Single threaded clients can access the dynamic environment by polling for changed values from shared memory, by invoking a system call (e.g. to read a file) or by issuing a remote procedure call. Concurrency-capable applications can be notified asynchronously by software interrupt, remote procedure callback, or similarly from input available on a non-blocking file descriptor.

Asynchronous notification has the advantage that the application is immediately informed of changed values and no cycles are wasted on polling. However, for this to work, applications must be multi-threaded. For concurrent

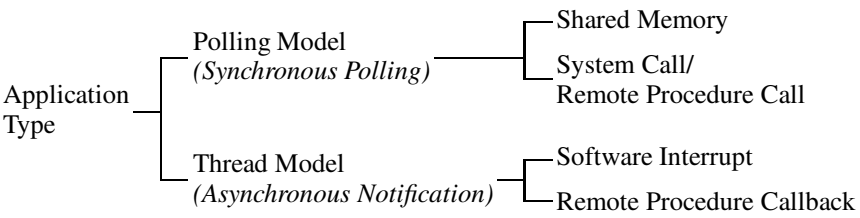

Figure 1: Dynamic Environment Interaction

applications our system provides an RPC callback interface. Non-concurrent applications use a polling interface based on a shared memory server. These approaches are described below.

Concurrent applications access the dynamic environment through an RPC connection and an RPC callback connection. A small library hides the RPC layer from applications. We support both C/C++ and Modula-3 programming with separate versions of this library. One issue addressed by the library is how to integrate a potentially blocking RPC callback service with an applications. The Modula-3 language [6] has built-in thread support that we take advantage of by dedicating a thread to the callback RPC service. Since there is no standard threads definition for C/C++ a main loop containing the `select` or `poll` system calls is often used to provide a poor man's multi-tasking. Our C/C++ library supports this approach by allowing the main loop in either the library or in the application.

Single threaded applications access the dynamic environment through shared memory [5, 4]. An application maps a memory segment containing the dynamic environment into its address space and periodically polls the contents for changes. A separate process is used to receive dynamic environment updates from the network and write them into the shared memory. The cost of this extra process can be amortized over multiple clients. Furthermore, polling shared memory can be much more efficient than polling techniques requiring a system or remote procedure call. As in the concurrent model, a small library hides the shared memory layer from the application. The non-threaded library supports C/C++ programming.

4.5 Example Application

One of the applications we have built uses dynamic environment variables for presenting a proximate device selection dialog to the user and then migrating the user's editor window upon selection. This application encorporates two uses of the dynamic environment. The dialog employs the dynamic environment to update a presentation list of nearby displays, and separately, the editor window changes displays whenever the the `DISPLAY` dynamic variable changes value. This process is described more fully below.

A PARCTAB [2, 1] based application, the *proximate chooser* presents a selection of nearby displays for migration. The PARCTAB is a hand held wireless device that communicates with applications running on back-end hosts.

The chooser initially shows a proximity ordered list of displays from the `NEARBY_DISPLAYS` dynamic environment variable. This variable is part of a room's dynamic environment. The application library monitors the user's location and withdraws subscriptions that were based on the old location and subscribe to the appropriate servers based on the new location. This process causes the `NEARBY_DISPLAY` variable to change as the user moves from room to room. The chooser, a Modula-3 application, uses an RPC callback thread to obtain environment notifications. After the dialog starts, movement between offices or common areas causes updates to the `NEARBY_DISPLAYS` variable. Changes to this variable in turn update the chooser dialog with a new proximate selection ordering. When the user decides they want to setup shop at a display they make a selection which the chooser uses to set the `DISPLAY` dynamic environment variable.

Changing the `DISPLAY` variable has the effect of propogating the new value to applications that are interested in display migration. We use an extended Emacs editor supporting a command for moving the editor's X window to another display. Our Emacs interface to dynamic environment variables uses a subprocess to run `printdenv` in the mode where it enters a loop reporting variables changes as they occur. The variables are read by Emacs from a pipe and converted into calls to user supplied elisp functions. The use of filter procedures invoked on process output is a

[2]Notification is asynchronous so completing the `setdenv` command does not imply that clients have received notification.

standard Emacs technique. The mechanism is based on polling a file descriptor for available input that is classified in a polling model. One filter function for processing dynamic environment variables watches for changes to the `DISPLAY` variable and causes the editor window to migrate to the new display.

4.6 Performance

The architecture of our system encourages *cliques* of clients and servers within a larger client-server community. A clique of clients, for example a set of applications in the same room, will subscribe to change notifications from a single server. To determine the performance of the system we have measured a single clique. For the shared memory server, we believe the system load is determined by the same costs as the callback server. This is because the additional expense of a local procedure call to poll variable changes stored in shared memory is small compared to the cost of system and remote procedure calls as long as the polling rate is suitably low.

The most important parameters that determine an environment server's performance are the number of client subscriptions and the frequency of variable changes. For a timing experiment, we setup a system configuration consisting of a remote environment server, a local process setting environment variables at a Poisson distribution around a fixed rate, and one or more local subscribing processes. Both hosts are Sun SPARCStation 2 class machines. We measured the variable propogation time between a set and the resulting subscribe notification. The information flow for this measurement includes a remote RPC to the environment server and another remote RPC from the environment server to each subscriber.

Figure 2. shows average notification times for an environment server supporting a collection of between 1 and 64 subscribers. The average variable update rate was between 1 and 4 updates per second with one variable being stored in the server. Similar runs with 1000 and 5000 variables at the server showed no significant timing difference indicating that RPC processing overshadows table lookups at the server. The delay for a single subscriber receiving notifications of a variable update is around 10 ms. The slope of the line shows a constant factor of around 3 ms for each additional subscriber.

There are two limits of the server, both are a result of the decision to use Sun RPC instead of a special purpose protocol. First, the number of subscriptions cannot exceed the process file descriptor limit, currently 256. Second, since unicast is used for outgoing messages, the product of update rate times subscribers is limited. Measurements show the server process capable of delivering around 80 messages/second.

Our system requires an initial associative name lookup before a subscription is made. The time for creating a subscription (including the associative name lookup of the server) is around 350 milliseconds and about half that, 150 milliseconds, if the server being subscribed is on the local host.

5 Conclusions

We have presented a simple and generic approach to handling the dynamics of a mobile computing environment. The system consists of a community of servers that each provide a set of variables and values as well as notification of changes to the set. Servers can be organized to represent physical environments, such as rooms, or work contexts such as meetings and project groups with equal ease. A source of location information feeds into the system permitting applications to recustomize according to changes in the physical environment.

The system structure we support is flexible. Unlike other approaches for sharing environment information we do not limit cooperation. Traditional environment variables limit cooperation based on process hierarchy, and Xrdb limits cooperation based on window system. These criteria restrict the class of application permitted in a mobile system that can easily span multiple machines. Alternatives such as extending X window system events might be suitable for some classes of applications. However, this requires that applications using the system must all include the X window system. In addition, this alternative does not include a way to link information about the changing physical world (such as people's locations) to changes in the values used for application customization.

Our system is scalable. Using logically distinct servers that each manage a small part of the environment means that the system can grow gradually. Moreover, the technique of binding and unbinding to environment servers permits transparent movement between administrative domains. Whether the servers are implemented as distinct processes is not exposed thereby making the potential benefit of consolidation possible. The performance measurements shows that large variable sets with reasonably high update rates can be supported. The least scalable aspect of the implementation is the additional cost per client-subscription at an environment server. Because our implementation is based on Sun

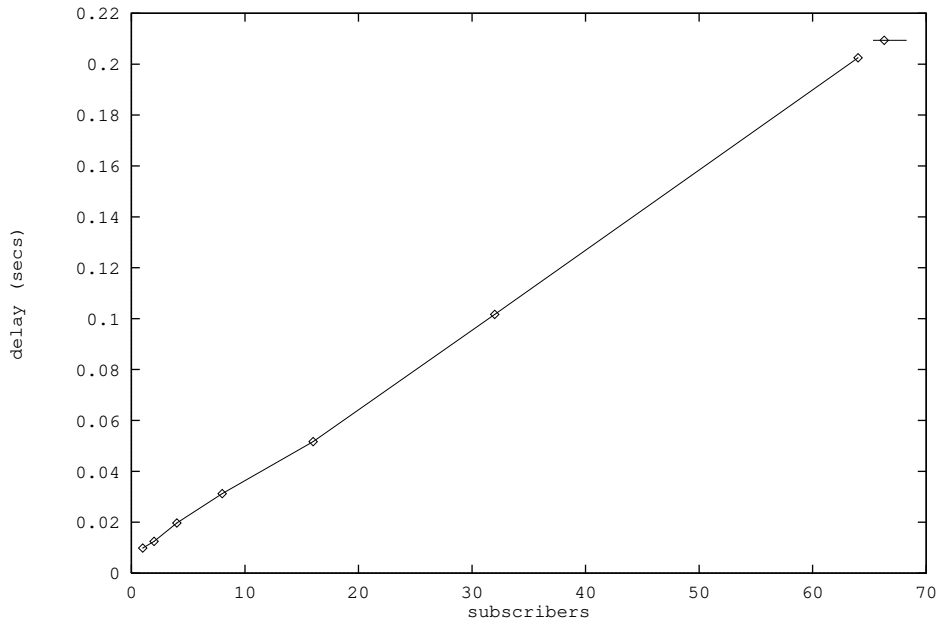

Figure 2: Variable propogation delays

RPC we incur the overhead of an RPC connection and the cost of an RPC message. A larger system might explore lighter weight alternatives. However, in terms of human perception the overhead is acceptable: an environment server maintaining 60 callbacks subscriptions with a variable update rate between 1 and 4 times per second has a little over a fifth of a second variable propogation delay.

Finally, our facility supports short interactions with long term services. When starting programs it is often necessary to setup a context. This may involve reading and processing shell login and other initialization files. In a mobile setting, interactions may be very short lived as a user moves between one system and another. Dynamic environments help alleviate the startup costs by preserving program context across machines and logins.

References

[1] N. Adams, R. Gold, B.N. Schilit, M. Tso, and R. Want. An infrared network for mobile computers. In *Proceedings USENIX Symposium on Mobile & Location-independent Computing*. USENIX, August 1993. To Appear.

[2] N. Adams, R. Gold, B.N. Schilit, M. Tso, and R. Want. The PARCTAB mobile computing system. Publication in preparation, 1993.

[3] Paul J. Asente and Ralph R. Swick. *X Window system toolkit : the complete programmer's guide and specification*. Digital Press X-motif series. Digital Press, Rockport, MA, 1990.

[4] M. Young et al. The duality of memory and communication in the implementation of a multiprocessor operating system. In *Proc. Eleventh ACM Symp. on Operating System Principles*, pages 63–76, November 1987.

[5] R. A. Gingell, J. P. Moran, and W. A. Shannon. Virtual memory architecture in sunos. In *Proc. Summer 1987 USENIX Technical Conf.*, pages 81–94. USENIX, June 1987.

[6] Greg Nelson, editor. *System Programming with Modula-3*. Prentice Hall Series in Innovative Technology. Prentice Hall, 1991.

[7] Mike Spreitzer and Marvin Theimer. Architectural considerations for scalable, secure, mobile computing with location information. Publication in preparation.

[8] M. Weiser. The computer for the 21st century. *Scientific American*, 265(3):94–104, September 1991.