Exporting Environment Awareness to Mobile Applications

Girish Welling and B. R. Badrinath
Dept. of Computer Science
Rutgers University, New Brunswick, NJ 08903, USA.
{*welling,badri*}*@cs.rutgers.edu*

Abstract

In mobile computing, factors such as add-on hardware components and heterogeneous networks result in an environment made up of changing resource constraints. An application in such a constrained environment must react to these changes so that available resources are properly utilized. In this paper, we propose an architecture to report changes in the environment to interested applications. The architecture is based on an event delivery mechanism that decouples event detection from delivery, giving the flexibility and extensibility that is necessary in a mobile computing environment. Information associated with the event is delivered as part of the event notification, while delivery latency is reduced by clever thread scheduling. We demonstrate the utility of our architecture by structuring an environment aware networking subsystem around a prototype implementation. The performance of this implementation is competitive with current event delivery mechanisms such as the Unix signal.

1 Introduction

The inherent portability of a mobile computer results in an environment of *constrained resources*. Although these constraints are becoming less noticeable, portability will always induce constraints in mobile computers, when compared to non-mobile computers. For instance, battery powered mobile computers will always face power constraints relative to their fixed counterparts. Since current technology [13] allows hardware components to be added or removed while a mobile computer is still powered on, resources related to such components may change, giving rise to *dynamic constraints*. In such an environment with dynamic constraints, available resources *must* be properly utilized, and the system must adapt when the quality of a resource deteriorates or a resource becomes unavailable.

Network applications running on a mobile computer must also deal with the heterogeneity of different networks. For example, slow cellular or CDPD connectivity may be available outdoors, while faster wired networks, or wireless networks such as *WaveLAN* may be available indoors. Coping with dynamic network-related constraints such as intermittent connectivity and changing bandwidth becomes essential for a network application.

On current systems, resources are allocated among different applications by the underlying operating system. This is justified because there may be applications belonging to several users on the system, and each of these applications is *competing* with others for available resources. Since changes in resource availability are uncommon, boot-time configuration of the system is often sufficient, and an application is kept unaware of any changes. Little or no adaptation is built into the application.

A mobile computer, however, is typically dedicated to a *single user*, who owns all applications on the system. Although resource allocation can be left to the system, better utilization is possible if applications contribute by using resources conservatively. For instance, on a *low-on-battery* condition, an application may disable a graphical user interface, preferring a text based one. This change of user interface may consume less processing power, allowing the processor to operate in a low-power mode; the application has implicitly contributed towards power allocation in the system. Another application can use the iconized state of its display window as a hint to inhibit network activity. No scheduling of network activity by the system can perform better than such voluntary restraint. In general, an

application in our mobile computing model should be (i) aware of resource availability, and (ii) structured so that application specific hints are used to alter functionality and resource usage, thereby contributing towards system resource allocation.

In this paper, we describe our approach to make an application aware of changes in the environment. Since many conditions in a mobile environment can only be determined in user-space, our proposed architecture is based on the facility for *user-level detection of events*. Further, the architecture *separates the detection and delivery of events* so that event producers and consumers are effectively decoupled. This gives the flexibility and extensibility that is necessary for the mobile computing environment. Finally, the architecture *preserves the state of the environment* induced by past events so that a new application can easily determine the current state. We believe that current abstractions which provide similar functionality, such as *signals*, are inadequate for mobile computing environments.

The rest of the paper is organized as follows: Section 2 outlines the design considerations that led to our model. Section 3 presents the overall architecture with Section 4 describing the construction of a network aware subsystem structured around a prototype implementation. Section 5 gives implementation details and presents performance numbers showing that our architecture is competitive. Section 6 compares our work with similar ideas in the community and finally, Section 7 summarizes what we have learned.

2 Design Considerations

Our goal is to build a flexible architecture to export environment awareness to a mobile computing application. In this section, we examine factors that affect the mobile computing environment, list the design objectives of our architecture, and discuss limitations of existing mechanisms when applied to mobile computing.

2.1 The Mobile Computing Environment

The environment associated with mobile computing is characterized by changing resource constraints. Two important factors that contribute to these changes are (i) portable computer technology, and (ii) wireless communication characteristics.

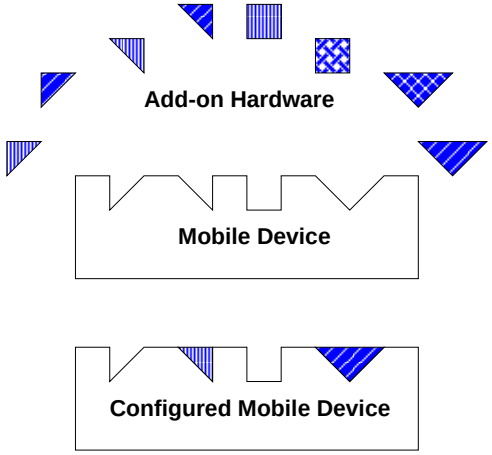

Figure 1: Portable Computer Technology

The principal aspect of current technology that affects the mobile environment is the ease with which hardware can be added to a portable computer (Fig. 1). Diverse components, like disks, memory, modems and network interfaces, can be plugged into a portable computer while it is powered on. Although current operating systems technology has resolved the problem of *hot-swapping* these components, there is no standard method to export the induced changes in the environment to an application.

A mobile computer can be intermittently network connected with untethered wireless links. These wireless links are associated with changing bandwidth, which usually deteriorates with distance (Fig. 2). While network protocols for mobile hosts [7, 12] can transparently maintain network connectivity, the changing characteristics of the link are largely ignored above the transport layer. This makes it difficult for an application to detect alternate network services (name-servers, file-servers, mail-servers, news-servers) while the mobile computer roams, or change functionality depending on the changing link characteristics. Support is therefore necessary to export network awareness to applications that are interested.

2.2 Design Objectives

The contribution of plug-in hardware components to the mobile environment is usually detected within add-on software modules such as loadable device-drivers. Since such a module is usually activated only when a component is plugged in, *the environment aware sub-*

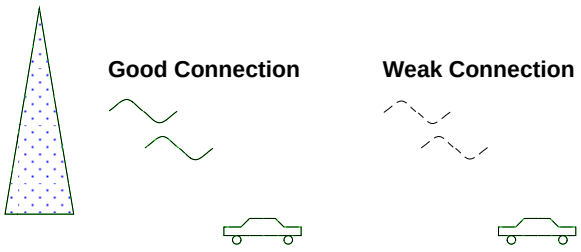

Figure 2: Wireless Communication Characteristics

system on the portable computer must be dynamically extensible so as to include this add-on software as and when it is activated.

Current protocols which maintain network connectivity for mobile computers [7, 12] are partly implemented in user-space. Certain interesting conditions, such as cell handoff, are detected in these user-space modules, making it imperative that *the environment aware subsystem supports user-level detection of changes in the environment.*

In order to export awareness of the changing mobile environment, an application must be be notified of *changes* in the environment state. These changes can be modeled as *asynchronous events* which are delivered to an environment aware application. Our architecture is therefore built around a flexible mechanism for event detection and delivery.

The objectives of our event delivery mechanism are:

1. to allow user-level detection of events.
2. to allow flexible, event specific delivery policies.
3. to integrate event notification and the transfer of information associated with the event.
4. to maintain the current state of the environment.
5. to reduce event delivery latency by clever thread scheduling.

The first objective makes the architecture easily extensible by allowing the generation of arbitrary events, including user defined events to communicate between processes, composition of events, monitored events, and even remote events. Events that are detected within the kernel can also be supported as long as the kernel provides a mechanism to observe them.

The necessity of the second objective can be seen when the delivery of a *low-on-memory* event is considered. Such *low-on-memory* events need only be delivered until sufficient memory has been reclaimed. In general, not all events need be delivered to all registered recipients, and not all registered recipients need be delivered the same information about an event.

Information associated with an event is often required when the event is delivered. For instance, when a new network is detected, associated information like typical latency and bandwidth are useful, especially when network connectivity is intermittent. The third objective suggests that such information can be piggybacked along with event notification making the repeated retrieval of the information unnecessary. The additional cost of transferring this information is usually negligible if the amount of data is small.

A newly created application must have access to the current state of the environment. Since an event is generated only when there is a change in the environment, a separate mechanism is necessary to provide the current state. We integrate such a mechanism with event delivery, as noted by the fourth objective, so that the environment state is effectively maintained. This is accomplished by logging events that have occurred and delivering these to the new application when it is first activated.

It can be argued that our approach of detecting events in user space would result in large delivery latency. The fifth objective is our approach to reduce this latency. The event delivery mechanism gives hints to the task scheduler so that the task to which the event is delivered is scheduled quickly. Such an approach combined with a thread-based environment can reduce delivery latency to below that of a Unix signal, where the event is delivered only when control is transferred back to the destination task user space. We believe this approach is justified in our mobile computing model in which all tasks on the mobile device belong to the same user.

2.3 Problems with Existing Mechanisms

While several asynchronous event notification mechanisms exist, they are inadequate in the mobile computing scenario for a variety of reasons.

The Unix signal mechanism [1] allows a task to be notified about the occurrence of an asynchronous event. The event is usually *detected within the kernel* and is directly associated with the destination task. Event detection and delivery are tightly integrated (Fig. 3a), making dynamic extension difficult. For instance, it is difficult to extend signals so that an occurrence persists

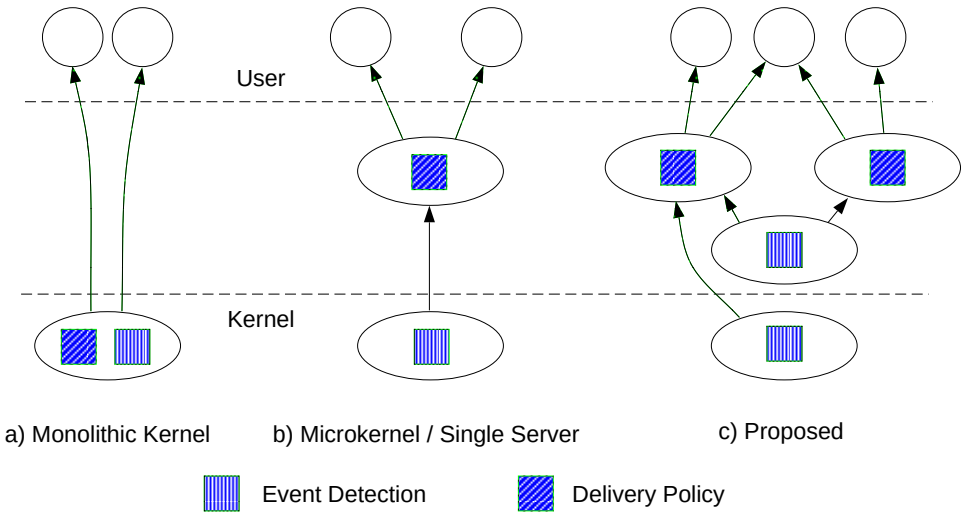

Figure 3: Event Detection and Delivery

and is delivered to a future subscriber. It is also difficult to add a signal describing a new class of asynchronous events. Such extensions are essential in a dynamic environment such as that associated with mobile computing.

A different failing of signals is the inability to transfer information along with delivery. Although the implicit signal type (SIGFPE, SIGSEGV, SIGIO, etc.) may be sufficient for many events, the limitation is easily discerned when programs attempt to communicate using a signal. No information about the signal can be delivered unless it is placed in a predefined mailbox such as a file or a message-queue. Our event delivery mechanism is designed to get around this problem by integrating information related to the event with the notification.

3 Event Delivery Architecture

In this section we outline our event delivery architecture and describe the functionality of its components. The architecture can be considered to generalize the trend in operating systems to extend a *microkernel* that provides minimal functionality. In such an operating system, events may be detected within the microkernel, and a delivery policy implemented in an external server (Fig 3b). Such an architecture is flexible in that alternate delivery policies can be implemented. Windowing systems too, are organized similarly, with primitive events detected within the operating system, and the window server delivering the events to window applications. Our proposed architecture extends this flexibility to allow alternate, user-level event detection mechanisms (Fig 3c).

There are four classes of entities that constitute the architecture: *Factories*, *Channels*, *Event Objects* [1] and *Handlers* (Fig. 4). An independent *Registrar* provides a name-service for the *Channel* name space.

3.1 The Event Channel

A *Channel* is the crux of our event delivery architecture, binding event producers (factories) and consumers (handlers). It both implements the event delivery policy, and preserves the environment state induced by the sequence of events that have occurred. A channel is usually associated with a single component of the mobile environment. For instance, a *network* channel may be associated with all network related events and hence the network environment, while a *hardware* channel may be associated with all events related to changes to the hardware. A *power* channel may be associated with events related to changes in power supply and usage.

Although the architecture does not restrict the type of *Event Objects* that a particular channel can deliver, channels provide a means to classify and separate environment related events. At one extreme, a different

[1] In the discussion, "*Event Object*" represents the object, in contrast to a generic "event".

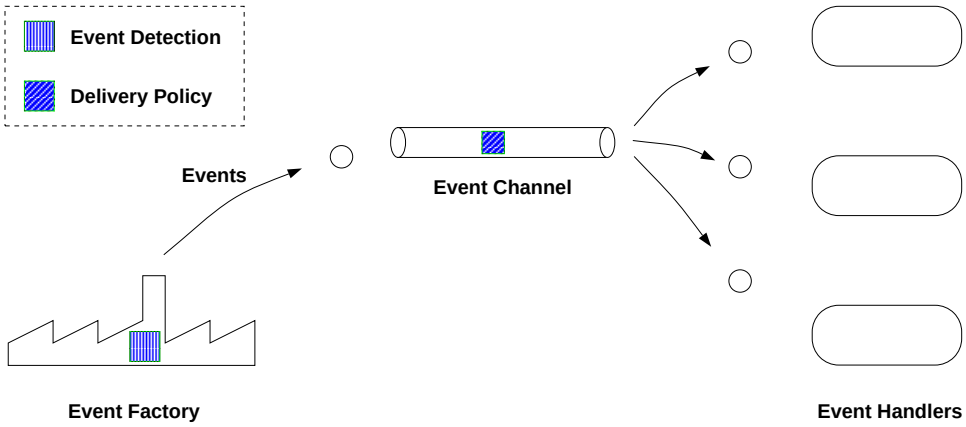

Figure 4: Delivery Architecture

channel may be dedicated to each type of event, while at the other, a single channel may deliver all types of events. Usually, it is harder to implement the delivery policy when a large number of *Event Object* types are delivered on the same channel. However, a large number of channels could increase the task overhead on the system.

The delivery policy determines the order with which an *Event Object* is delivered to registered handlers. The policy is also associated with a termination condition after which the *Event Object* will not be delivered anymore. For instance, the *low-on-memory* event, where the goal is to quickly acquire a certain amount of free memory, can have a delivery policy where the event is delivered to tasks with high memory usage, terminating delivery after sufficient memory has been freed.

A channel preserves the current state of the mobile environment by maintaining a single-event log of *Event Objects*. For each type of events, the most recent *Event Object* persists until a new *Event Object* of the same type arrives. When a handler first registers with the channel, these persistent *Event Objects* are delivered in the order they occurred, so that the handler becomes up to date with the current environment as known to the channel.

The public interfaces of the channel and policy classes are given in Figure 5. When a channel is created, the associated name is registered with the *Registrar* name-service. This name is used to retrieve references to the channel. An optional delivery policy may also be specified when a channel is created. The *Channel::State* member maintains the log of *Event Objects* that have been delivered.

```
class Channel {
   Policy*       policy;
   Event*        State[MAX];
public:
   Channel(char* name);
   Channel(char* name, Policy* p);
   . . .
}

class Policy {
   Handler*      List;
   . . .
public:
         void    Deliver(Event*);
virtual void     Reset();
virtual Handler* NextHandler();
virtual void     AddHandler(Handler*);
   . . .
}
```

Figure 5: The Channel and Policy Classes

The flexibility of a channel is encapsulated within the delivery policy. When an *Event Object* is posted to the channel, it is delivered to handlers based on this policy. The channel uses the *Policy::Deliver()* method to actually deliver the *Event Object*. When a new handler registers with the channel, the *Policy::AddHandler()* method is used to add the handler to the policy.

A delivery policy is implemented as an *iterator* over the set of registered handlers. When called in se-

quence, the *Policy::NextHandler()* method returns handlers one at a time, in the order in which the event must be delivered. This method, in conjunction with the *Policy::Reset()* method which initializes the iterator, is used by the *Policy::Deliver()* method to deliver an *Event Object*.

A default policy, which delivers an event to all registered handlers, is used if no policy is specified when a channel is created. This default policy can be used to derive new delivery policies. A new policy must implement its own *Reset()*, *NextHandler()* and *AddHandler()* methods to maintain the internal list of handlers.

3.2 The Event Factory

Events are detected in an entity we call a *factory*. A factory is usually a software module associated with an aspect of the mobile environment, suitably augmented to produce *Event Objects* in our architecture. For instance, the *mhmicp* module in Columbia Mobile IP [7] is a prospective factory for events relating to network connectivity. When an event is detected, the factory packages information associated with it into an *Event Object*. This object is posted to a channel for delivery to interested environment aware applications.

Much of the flexibility of our architecture is achieved by keeping the event delivery mechanism outside the factory. This separation of event detection and delivery effectively decouples factories from applications allowing active applications to receive event notifications originating in newly activated factories. An application can therefore choose to be environment aware, independent of the existence of factories that contribute to the environment.

```
class Factory {
   Channel*  Chan;
   . . .
public:
   Factory(char* channelName);
   void      PostEvent(Event*);
   . . .
}
```

Figure 6: The Factory Class

The interface of the factory class is presented in Figure 6. When a factory is created, the name of the channel on which it intends to post events is specified. The channel reference is retrieved from the name-service provided by the *Registrar*. All *Event Objects* produced at the factory are posted to this channel using the *Factory::PostEvent()* method.

3.3 The Event Handler

A *handler* forms the end-point for event delivery in our architecture, besides providing the thread of execution for the application's response. An application that chooses to be environment aware creates a handler bound to the appropriate channel. The name of the channel is specified when the handler is created, and the channel reference is obtained from the name-service provided by the *Registrar*. *Event Objects* posted on the channel are now delivered to the handler based on the delivery policy implemented in the channel.

```
class Handler {
   Channel*   channel;
   . . .
public:
   Handler(char* channelName);
   void       Body();
   . . .
}
```

Figure 7: The Handler Class

When an *Event Object* appears at a handler, the response associated with its event type is executed. An application defines a response for each *Event Object* type that is of interest. Type information included in the transferred object is used to determine the *Event Object* type and choose the appropriate response. A default response is provided for all *Event Object* types so that those types that are *not* of interest to an application are ignored.

The interface of the handler class is shown in Figure 7. When a handler is created, an application only specifies the name of the channel to which the handler is bound. The event handler is then activated by calling the *Handler::Body()* method, which can be executed on a separate handler thread. The arrival of an *Event Object* is detected within the handler body, which then in-

vokes the object-type specific response. The definition of event responses is discussed in Section 3.4.

3.4 Event Object Types

Although different kinds of events are associated with different information, certain information is common to all events or may be common to more than one kind of event. This motivates the organization of *Event Object* types as an extensible type hierarchy. We define the basic *Event Object* type, which is extended according to the requirement and functionality of a particular event.

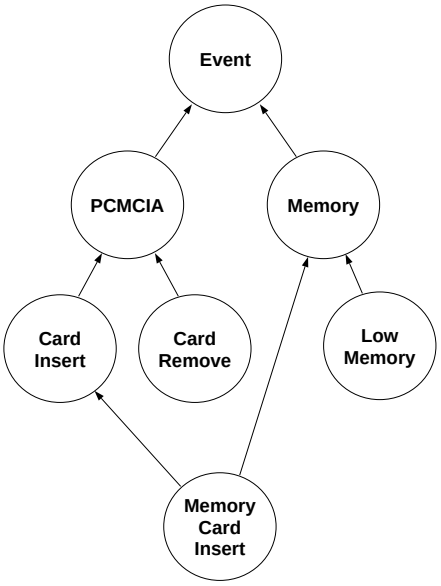

Figure 8: An Event Type Hierarchy

Every *Event Object* type is a subtype of a *root* event type. According to the example type hierarchy of Figure 8, a *Memory-card-insert* event is a *Card-insert* event, which in turn is a *Pcmcia* event. A *Pcmcia* event may only contain the slot on which the event occurred while a *Card-insert* event may provide additional details regarding the card type. A *Memory-card-insert* event is also a *Memory* event, which may include the change in the physical memory available.

It is desirable for an application to respond differently to different types of *Event Objects*. For instance, an application may decrease memory usage when a *low-on-memory* event occurs or enable distributed services when a new network is detected. To support such event specific responses, each *Event Object* type is associated with a response, which an application may define for events of interest. The response is specified in the *<Event_Type>::Handle()* method.

The default response for a particular *Event Object* type unconditionally chains to the responses associated with the immediate supertype *Event Objects*. With reference to Figure 8, if an application has defined responses for *Memory* and *Card-insert* events but not for *Memory-card-insert* events, the *Memory::Handle()* and *Card_insert::Handle()* methods are executed whenever a *Memory-card-insert* event occurs. Chaining of *Event Object* responses by default ensures that the response associated with the *expected* event type in an application is executed, if possible.

```
class Event {
   . . .
public:
virtual size_t Marshal(void*,size_t);
virtual Event* UnMarshal(void*,size_t);
   . . .
virtual void   Handle();
}
```

Figure 9: The Event Class

The public interface of the *root* event type is shown in Figure 9. The *Marshal()* method is used to package an *Event Object* in a factory, so that it can be transferred across address spaces. A typecode is included in the package so that the appropriate *UnMarshal()* method can be invoked in the destination handler address space. In our implementation, we assume that a unique typecode can be assigned to each *Event Object* type. Although it is hard to automatically generate typecodes that are unique across address spaces, the compilers for certain object-oriented languages like Modula-3 compute a *fingerprint*, which can be used as an approximation. With high probability, two types have the same fingerprint only if they are structurally identical. Such fingerprints have been used in systems supporting remote object method invocation [3].

3.5 Discussion

Decoupling event detection and delivery localizes mutual awareness of the participating entities in our archi-

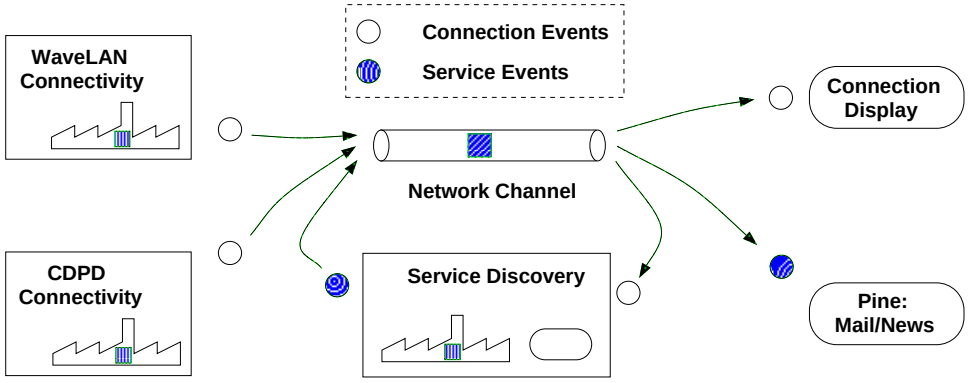

Figure 10: A Network Aware Subsystem

tecture. Awareness of *all* channels is restricted to the *Registrar*. Factories and handlers depend on the name-service provided by the *Registrar* to obtain channel references. A factory only needs the identity of the channel it intends to post events to, while a handler needs the identity of the channel it intends to receive events from. A channel, which requires the identity of all handlers that intend to receive events from it, is provided handler references as part of the handler registration process.

The state of the mobile environment is effectively maintained by the persistence of *Event Objects* within a channel. This feature is unique to our event delivery architecture, and is justified only because of the single-user environment we assume on a mobile computer. A newly activated application is notified of *past* events so that it becomes aware of the current environment.

4 A Network Aware Subsystem

We built a prototype implementation of our ideas on i486 based laptops running the MACH 3.0 microkernel. We used C++ [18] to provide the language level object support, and the capability based Mach IPC [6] to provide intertask communication. Our architecture is encapsulated within a set of C++ classes which are extended according to the needs of a particular system. In this section, we demonstrate the utility of this architecture by structuring a network aware subsystem around it (Fig 10). We describe our network infrastructure, present the requirements of the subsystem, and then show how we construct it within the framework of our architecture.

4.1 The Network Infrastructure

Our network environment consists of pockets of relatively high bandwidth WaveLAN coverage, augmented with low bandwidth ubiquitous CDPD connectivity. WaveLAN is a 2Mbps spread-spectrum wireless technology from the erstwhile NCR, Inc., while CDPD is a packet technology that uses the existing cellular phone network, providing a raw bit rate of 9.6Kbps. A mobile host is equipped with both, a CDPD and a WaveLAN interface.

We use a modified version of Columbia Mobile IP [7], which is itself an extension of the Internetwork Protocol, to provide network packet routing to mobile hosts. The details of the changes made to Mobile IP are beyond the scope of this paper and are irrelevant for this discussion. The point to note is that the mobile host has continuous network connectivity, but the link characteristics are considerably different at different locations. Our modifications to Mobile IP allow packets to be routed to the mobile host irrespective of the interface over which it is connected.

An important consideration of choosing one technology over the other is the cost of utilizing it. CDPD is currently billed on a *per packet and/or per byte* basis, while WaveLAN coverage may be provided on *charge for access* basis. With such a billing policy, it is clearly cost effective for network aware software to utilize WaveLAN connectivity in preference to CDPD connectivity.

4.2 Subsystem Overview

The goal of the subsystem is to export network awareness to mobile applications. The network state is de-

scribed by events which an application can choose to receive. A network aware application uses these events to make application specific decisions which contribute towards effective network utilization.

Remote data-access is an important application in mobile computing. We therefore choose a mail and news based application to demonstrate our architecture. We enhance the *Pine* mail reader so that alternate mail functionality is presented depending on the cost and expected quality of network connectivity. The various support modules are constructed before *Pine* is enhanced to be network aware.

4.3 Definition of the Events

To construct the environment aware subsystem, we first define a set of events that describe the state of the network environment.

```
typedef enum {up, down} LinkState;

class Connection: public Event {
   Address   LocalIP;
   Address   Gateway;
   LinkState State;
public:
   Connection(Address   local,
              Address   gateway,
              LinkState s);
}

class Services: public Event {
   Address   Printer;
   Address   SMTP;
   Address   NNTP;
public:
   Services(Address printer,
            Address smtp;
            Address nntp);
}
```

Figure 11: The Connection and Services Events

In an environment with changing network characteristics and intermittent network connectivity, the current state of the network link is clearly important. We encapsulate this information in a *Connection Event* (Fig. 11), which is generated whenever there is a significant change in the link state. Typical information packaged into this event includes the identity of the gateway through which the mobile host is currently connected, and whether the link is *up* or *down*.

It is usually better in terms of performance, to utilize network services that are close to the mobile client. Since a roaming mobile host is connected through different gateways, it is clear that nearby servers will change when it moves. The availability of such higher level network services is encapsulated in a *Services Event*, which packages information such as nearby printers, mail (*smtp*) and news (*nntp*) servers (Fig. 11).

4.4 The Network Channel

The key entity of our environment aware architecture is the channel. We build a simple *network* channel module using the default policy, where a posted *Event Object* is delivered to all registered handlers.

```
Channel* c;
main() {
   c = new Channel("network");
   c->Body();
}
```

Figure 12: The Network Channel

All that is needed to create the *network* channel is to define a channel object with the appropriate channel name. Since we use the default delivery policy, the code shown in Figure 12 is sufficient. After the channel is created, it is activated by calling the *Channel::Body()* method which arms it for event delivery. The *network* channel is activated at system boot-time, before the activation of any of the modules that use it.

4.5 Constructing the Factories

We next construct the modules where changes to network link state are detected, the factories.

In our system, *Connection Events*, which represent the state of the network link, are generated in modified *mhmicp* and *pumicp* modules of Columbia Mobile IP [7]. The *mhmicp* module is a user level process that monitors the WaveLAN environment and manages the

```
Address    LocalAddr;
Factory*   fact;

void initialize() {
   fact = new Factory("network"); // Create "network" factory
   LocalAddr = GetMyIpAddress();  // Initialize local data
}

void inCell(Address remote) {
   Event* e = new Connection(LocalAddr, remote, up);
   fact->PostEvent(e);            // Create and Post event
   delete e;
}

void lost() {
    Event*  e = new Connection(LocalAddr, NULL, down);
    fact->PostEvent(e);           // Create and Post event
    delete e;
}
```

Figure 13: Code for Connection Event Generation in *mhmicp*

routing tables. The *pumicp* module was intended to be used for a special case in the original Columbia Mobile IP, when a mobile *pop's up* in an administratively foreign network. We use a modified version of *pumicp* to handle the CDPD connection.

The original *mhmicp* module implements a state machine that indicates the current network state. We added exactly three lines of code to *mhmicp* to generate *Connection Events*. An additional twenty or so lines were needed to interface between C, in which the original code was written, and C++ in which our architecture is implemented. The interface code is shown in Figure 13.

The call to *initialize()* is made once, when *mhmicp* begins. A new factory object is created, which binds to the *network* channel, and local data is initialized. We assume that by the time *mhmicp* starts, the *network* channel has already been activated. The calls to *inCell()* and *lost()* are made when *mhmicp* hears a WaveLAN beacon and when the beacon is lost respectively. The appropriate *Connection Event* is created and posted to the *network* channel.

The *pumicp* module is started whenever *mhmicp* determines lost WaveLAN connectivity, and is stopped when the WaveLAN beacon is heard again. This is accomplished by waiting for *Connection Events* indicating that the WaveLAN link is down or up respectively.

4.6 The Service Discovery Module

We now construct the service discovery module, where changes in network services are detected.

The service discovery module generates *Service Events*, which represent the services available at a particular time and location. While a general service discovery mechanism would query for services in the network, we only implement a simple service 'discovery', where services are looked up in a table using the current gateway as the key. The identity of the current gateway is available in the *Connection Event*. Since nearby services would only change when the gateway changes, it is sufficient to generate *Service Events* in response to the notification of a *Connection Event*.

The code for the service discovery module is shown in Figure 14. A handler and a factory, both bound to the *network* channel are first created, and the handler is activated. The *Connection::Handle()* method is the module's response to *Connection Events*, where the gateway (*Gateway* is a member of the *Connection Event*

```
Handler*        eh;
Factory*        sf;
Services*       NoService;
Services*       getServices(Address);

void Connection::Handle() { // The Connection Event Response
   if (State == up)                        // If link is "up"
      sf->PostEvent(getServices(Gateway)); //   Post new services
   else if (State == down)                 // If link is "down"
      sf->PostEvent(NoService);            //   Post no services
}

main() {
   NoService = new Services(NULL, NULL, NULL);
   eh = new Handler("network");       // Create "network" handler
   sf = new Factory("network");       // Create "network" factory
   eh->Body();                        // Arm the handler
}
```

Figure 14: Code for Service Discovery

class) is used to determine the set of nearby services. The *Services Event* obtained is then posted to the *network* channel. If the link state is *down*, a *NoService* object is posted instead.

4.7 Network Aware Subsystem Capability

We have now finished constructing the support modules of our network aware subsystem. To summarize, two types of events are posted onto a *network* channel: a *Connection Event* and a *Services Event*.

A *Connection Event* is posted whenever a change is detected in the network link characteristics. A change occurs whenever the mobile computer moves from WaveLAN connectivity to CDPD connectivity, and vice versa. A change may also occur when the mobile computer moves from one WaveLAN wireless cell to another. An application can choose to be notified of *Connection Events* by creating a handler bound to the *network* channel, and defining the *Connection::Handle()* response for *Connection Events*.

A *Services Event* is posted whenever there is a change in the gateway through which the mobile host is connected. In our simplified service discovery module, this occurs whenever a *Connection Event* is detected. Similar to the notification of *Connection Events*, an application can choose to be notified of *Services Events* by creating a handler bound to the *network* channel, and defining the *Services::Handle()* response.

4.8 The Network Aware Application

We present a high-level version of the environment aware portion of the *Pine* mail and news reader in Figure 15. The ease of writing this environment aware portion is evident from the actual code required. Most of the logic is specific to the functionality required.

Pine mail is enhanced in both, outgoing and incoming paths. Incoming mail arrives at a *home mailbox*, which is constantly monitored from *Pine*. Only mail headers are accessible through a low bandwidth CDPD link, while full mail access is permitted through a high bandwidth WaveLAN link. In the outgoing path, mail is buffered in a *to-send* folder when the mobile host is CDPD connected. This *to-send* folder is flushed when the host becomes WaveLAN connected. *Pine* chooses the *nearest* available *smtp* server to send outgoing mail. The enhancement to the *Pine* news reader is limited to choosing an appropriate *nntp* server to access and post news.

```
Handler*   eh;
Thread*    et;

Connection::Handle() { // The Connection Event Response
   // If CDPD link
   //    Disable full remote mailbox access; Buffer outgoing mail.
   // If WaveLAN link
   //    Enable full remote mailbox access; Flush outgoing mail buffer.
}

Services::Handle() { // The Services Event Response
   // Change servers to those indicated in this event;
}

initialize() {       // Initialization called from Pine
   et = new Thread(eh = new Handler("network"));
}
```

Figure 15: Handler Code in Pine

The *initialize()* function is called once when *Pine* is activated, to create a handler bound to the *network* channel. A separate thread is used for the event handler in this application because the main thread provides *Pine's* interactive user interface. The *Services::Handle()* and *Connection::Handle()* methods define *Pine's* responses to a *Services Event* and a *Connection Event*, respectively.

A *Connection Event* contains the address of the gateway associated with the current connection. In our implementation, the address of this gateway is used to determine whether the link is WaveLAN or CDPD. When a *Connection Event* associated with the CDPD link occurs, full access to the remote mailbox is disabled, and buffering of out-going mail is enabled. When a *Connection Event* associated with the WaveLAN link occurs, the remote mailbox becomes fully accessible and any buffered out-going mail messages are flushed. When a *Services Event* occurs, the servers used by *Pine* are updated to those specified in the event.

5 Implementation Details and Performance

In order to make our architecture portable across different platforms, we have encapsulated operating system dependence within a set of low-level classes. Each of the classes described earlier also implements one or more of a set of abstract classes which form the basic building blocks of our architecture.

The *GlobalObject* class encapsulates a communication endpoint in our architecture. Both *Channels* and *Handlers* form communication endpoints and are *GlobalObjects*. The current implementation of the *GlobalObject* class uses Mach IPC for the inter-task communication. Porting our code to a Unix environment, for instance, will involve rewriting this class to use Unix domain sockets, while a Microsoft Windows port may utilize the clipboard.

Classes of objects that can move across address spaces implement the interface of an abstract *Migratable* class. Besides the *Event* class, the *GlobalObject* class is also *Migratable* albeit by reference. The same mechanism is therefore used to transfer an *Event*, and for instance, a *Handler* reference.

Channels, *Factories* and *Handlers* are all *Runnable*. A *Runnable* object can be executed on a *Thread*, which is an object that provides a thread of execution. While our current implementation assumes thread support for the handler execution, it is not difficult to use an interrupt mechanism such as the signal to execute the handler. Although this involves work to *port* our architecture, an application that *uses* it will be left unchanged.

One goal of this exercise was to determine how typed event delivery performs relative to other event delivery schemes. We therefore compared the latency of event delivery in our implementation with the latency of the Unix signal. Our observations are recorded in Table 1.

CPU	Operating System	Scheme	Latency
i486	Mach 3.0	Ours	1.118 ms
i486	Mach 3.0/UX 42	signal	0.873 ms
i486	Linux 1.3	signal	0.140 ms
SPARC	Solaris SunOS 5.4	signal	0.174 ms

Table 1: Performance Comparison

The method used for measuring the latency of signals is based on that for estimating the overhead of an upcall by Small and Seltzer [17]. An initiating process sends SIGUSR1 to a previously blocked target process, immediately followed by SIGCONT. The target process, which had blocked itself using SIGTSTP, therefore returns from a kill system call and processes the SIGUSR1 signal. The overhead of blocking and waking up is determined in a separate control experiment, and is deducted from the measured time. All measurements are averaged over thirty runs of a thousand iterations each. For our event delivery, an *Event Object* is repeatedly posted to a channel from within its handler. The mean of the measured time for posting and delivering a thousand *Event Objects* over thirty runs is taken.

The latency measurements for Mach and Linux were made on an Intel 486 based PC running at 33Mhz, which is representative of current portable computer technology. The Solaris measurements taken on a 110Mhz SPARC station 5 are included as representative of current RISC technology. The Sparc measurements are marginally worse than the Linux measurements on the i486 probably because of the larger context switch time on RISC processors. The latency of signal delivery on Mach is greater than that on Linux largely because of the overhead of a user-level UX server. The additional latency of our mechanism when compared with Mach/UX signals is probably because of an extra context switch, which is reported to be of the order of 98 microseconds for Linux on the same architecture.

We believe the numbers indicate that it is feasible and useful to support an enhanced event delivery such as ours on current portable computer systems. The delivery latency is small enough to be useful and is not significantly worse than existing delivery mechanisms. Providing the channel abstraction as a kernel primitive will reduce the overhead of the additional context switch, making the performance comparable to signals. In this case, the delivery policy could be specified in an interpreted language using a mechanism similar to the Berkeley Packet Filter [8].

We also believe that the performance of event delivery will not degrade as much as that of a signal when the number of processes increases on a system, although we did not verify this experimentally. The reason is that the presence of a large number of ready processes on a system will increase the latency of rescheduling the target process; the scheduling hints our architecture provides will help reduce this latency. Further, the fact that the handler is always waiting for an event ensures that it is observed as soon as the handler thread is scheduled, in contrast to when a process returns from a system call in signals. Such an approach is also taken in the *QuickLPC* mechanism in Windows NT [5].

6 Related Work

An event in our architecture is an object that represents the occurrence of an asynchronous event in a system. We first compare our mechanism for event delivery with existing mechanisms in both operating systems and programming languages. We then relate our architecture to other event based architectures for system structuring. Finally, we position our work in the context of the work of other researchers in mobile computing.

Operating systems like Unix provide the signal mechanism to deliver asynchronous events to a process. Signals, as pointed out earlier, are limited by the fact that no information related to the event can be easily delivered. Further, the signal delivery policy is fixed and restrictive, with all registered recipients being notified of the event. Modifications to the signal mechanism have been suggested in the context of *synchronous exceptions* by Thekkath and Levy [19], where the signal handling latency is reduced by minimizing the number of cross-domain control transfers. However, to our knowledge, there has been no attempt to extend the flexibility of signals. Extensions have usually been limited to adding signal support for a new condition, such

as the SIGFREEZE and SIGTHAW signals described in [2].

Many object-oriented languages provide language constructs for the delivery of *synchronous* events. These constructs are provided primarily so that exceptional conditions that may occur during code execution can be trapped and recovered from, while preserving as much of the execution state as is possible. The mechanism of such constructs entails knowledge about where in the code an exception is likely to occur. Our event delivery mechanism, on the other hand, aims at providing extensible abstractions for *asynchronous* events. Since it cannot be determined a priori, where in the code such events actually occur, it difficult to design a general language construct for the same.

Upcalls in Swift [4] are suggested to aid in organizing layered software around its natural flow of control. The upcall feature gives a programmer the choice of implementing an upward flow by procedure calls or asynchronous signals. This corresponds to organizing layered software into processes representing vertical or horizontal stripes, respectively. While the flexibility of such alternate organizations is clearly advantageous, the upcall mechanism still has the failing of tight coupling between its two ends. By separating the two ends of event delivery, our event delivery mechanism allows a flexible architecture where participating entities do not depend on the existence of others.

The *Reactor* pattern in ACE (Asynchronous Communication Environment) [16] simplifies the structuring of event-driven applications by integrating the demultiplexing of events and the dispatching of the corresponding handlers. The mechanism is provided so that application-specific event handlers can evolve independently of the event demultiplexing mechanism provided by the operating system (eg. *select()* in Unix). While, the *Reactor* pattern can be considered to decouple the event handler from the event demultiplexing mechanism, the channel abstraction in our architecture can be considered to decouple an event consumer from an event producer.

Certain aspects of the event delivery model in CORBA Event Services [11] are similar to our architecture. CORBA suggests the *push* and *pull* models for event delivery, where the delivery is driven by the event producer and the event consumer, respectively. Our model can be considered to implement the *push* model of CORBA Event Services, where the delivery of environment related events is driven by entities that produce them.

Researchers in mobile computing agree upon the necessity of application adaptation in response to the mobile environment [20, 14, 15, 10]. Montenegro and Drach [9] attempt to minimize application awareness in order to support existing services and applications. Such an approach is limited, and mechanisms have been suggested to convey environment related information to applications. Odyssey [10] provides an API aimed at supporting alternate file access policies. Schilit, Adams and Want [14] suggest servers, where the environment state is maintained as a set of environment variables. Our event delivery mechanism serves the same purpose, but takes into consideration the peculiar features of mobile computing. This results in an architecture that is extensible and flexible, which is essential for a mobile computer.

7 Future Work and Conclusions

We implemented our event delivery mechanism keeping in mind our long-term goal of designing software for mobile computing [21]. It is clear from our experience that such software will be reactive to mobility related events. Our mechanism provides the means to report these events to mobile applications. Aspects of our future work are (i) to consolidate our current implementation, (ii) to develop tools to aid in the utilization of our primitives, and (iii) to investigate how a mobile application should be structured.

Several aspects of our implementation are unsatisfactory from the point of view of using the primitives. For instance, the marshaling and unmarshaling functions for transferring an *Event Object* can be automatically generated from its description. Besides the construction of such tools, we also need to work around several simplifications which enabled us to quickly develop the current implementation.

Based on our experience with mobile application development, we believe that providing abstractions of mobility related factors is important to aid in their construction. These abstractions must provide the functionality that mobile computing entails without significant performance penalty. We have built an event delivery mechanism that provides such an abstraction to report environment related events to an application. The

utility of this mechanism has been demonstrated in the context of typical mobile applications, and its performance is shown to be competitive.

References

[1] M. J. Bach. *The Design of the UNIX Operating System*. Prentice-Hall, 1990.

[2] M. Bender et al. Unix for nomads: Making unix support mobile computing. In *Proceedings of the USENIX Symposium on Mobile & Location-Independent Computing*, August 1993.

[3] A. Birrell, G. Nelson, S. Owicki, and E. Wobber. Network objects. In *Proceedings of the 14th ACM Symposium on Operating Systems Principles*, December 1993.

[4] D. Clark. The structuring of systems with upcalls. In *Proceedings of the 10th ACM Symposium on Operating Systems Principles*, 1985.

[5] H. Custer. *Inside Windows NT*. Microsoft Press, 1993.

[6] R. P. Draves. A revised ipc interface. In *Proceedings of the USENIX Mach Workshop*, 1990.

[7] J. Ioannidis, D. Duchamp, and G. Q. Maguire. Ip-based protocols for mobile internetworking. In *Proceedings of the ACM SIGCOMM Symposium on Communication, Architectures and Protocols*, September 1991.

[8] J. C. Mogul, R. F. Rashid, and M. J. Accetta. The packet filter: An efficient mechanism for user-level network code. In *Proceedings of the 11th ACM Symposium on Operating Systems Principles*, November 1987.

[9] G. Montenegro and S. Drach. System isolation and network fast-fail capability in solaris. In *Proceedings of the 2nd USENIX Symposium on Mobile & Location-Independent Computing*, April 1995.

[10] B. Noble, M. Price, and M. Satyanarayanan. A programming interface for application-aware adaptation in mobile computing. In *Proceedings of the 2nd USENIX Symposium on Mobile & Location-Independent Computing*, April 1995.

[11] Object Management Group, Inc. *CORBA Services: Common Object Services Specification*, March 1995. OMG Document 95-3-31.

[12] C. Perkins. Ip mobility support. Internet Draft, February 1996.

[13] Personal Computer Card Interface Association. *PCMCIA PC Card Standard, Release 2.1*, July 1993.

[14] B. Schilit, N. Adams, and R. Want. Context-aware mobile applications. In *IEEE Workshop on Mobile Computing*, December 1994.

[15] B. N. Schilit, M. M. Theimer, and B. B. Welch. Customizing mobile applications. In *Proceedings of the USENIX Symposium on Mobile & Location-Independent Computing*, August 1993.

[16] D. C. Schmidt. Reactor: An object behavioral pattern for concurrent event demultiplexing and event handler dispatching. In *Pattern Languages of Program Design*. Addison-Wesley, 1995.

[17] C. Small and M. Seltzer. A comparison of os extension technologies. In *Proceedings of the USENIX 1996 Annual Technical Conference*, January 1996.

[18] B. Stroustrup. *The C++ Programming Language*. Addison-Wesley, 1991.

[19] C. A. Thekkath and H. M. Levy. Hardware and software support for efficient exception handling. In *Proceedings of the 7th International Conference on Architectural Support for Programming Languages and Operating Systems*, 1994.

[20] T. Watson. Application design for wireless computing. In *IEEE Workshop on Mobile Computing*, December 1994.

[21] G. Welling and B. R. Badrinath. Mobjects: Programming support for environment directed application policies in mobile computing. In *ECOOP'95 Workshop on Mobility and Replication*, August 1995.