Event Delivery Abstractions for Mobile Computing

B. R. Badrinath
Dept. of Computer Science
Rutgers University
New Brunswick, NJ 08903

Girish Welling
Dept. of Computer Science
Rutgers University
New Brunswick, NJ 08903

Abstract

An application intended for a mobile computing environment is different from more traditional ones in that it is more *event driven*. The reason is that mobile computing is associated with *constraints*, both in terms of mobile host hardware and the network environment. The effect of these constraints is complicated by the fact that the environment in which a mobile host operates changes as it moves. An application in such an environment will have to metamorphose as changes occur, in order to make the best possible use of the constrained environment and thus provide the best possible quality of service to a user. We choose to model these changes in the environment as *events* which are delivered to each application that is interested. This paper describes a language level abstraction to deliver events along with the run-time support required. The idea itself is not restricted to the mobile computing environment: it could be used in all such environments where an application may be interested in altering its behavior in response to changes in the state of its environment.

1. Introduction

Mobile computing is characterized by resource constraints of various kinds including battery power, memory, disk space, network bandwidth, and processing power. While it may be argued that these constraints are becoming less noticeable, it is undeniable that the inherent portability of mobile computing devices will always induce constraints relative to a non-mobile computer. An application intended for such a mobile computing device will need to take into account all such constraints so that best possible use is made of available resources.

The problem of resource constraints is compounded by the fact that they change with time as the mobile device moves. An application finds that there is more memory available when a memory card is inserted, or that there is network connectivity when a network card is inserted. It is important that an application adapts to these changes so that newly available resources can be put to the best possible use by existing applications. A more complicated problem occurs when there is a reduction in available resources. An application on a fixed computer would do nothing — the system would perform all the necessary reallocation. This is justified because, in general, there may be applications belonging to several users on the system and each of these applications is *competing* with others for available resources. The mobile computer scenario, however, is different because all the applications belong to a *single user*. The system cannot make reallocation decisions without suitable hints from the user. These decisions are best left to each application which may use application specific hints from the user to perform any reallocation of resources. Applications must *cooperate* with one another in order to provide the best possible quality of service from the *perspective of the mobile user*. There is a need for an application program to be aware of the availability of resources so that application specific activity can be initiated and performed.

Distributed applications in a mobile computing environment will also depend on other param-

eters such as the location or available services. An application may need to present different views in terms of both functionality and quality, depending on the location of the mobile computing device. Another application may provide different interfaces depending on services that are currently available. In order to do this, it is imperative that an application is aware of changes to the location of the device or the services available. In general, it is necessary for an application to be aware of any change in the state of the mobile environment.

In this paper, we describe our approach to making an application aware of changes to the environment. We model these changes as *events*, some of which an application may choose to receive. We do *not* suggest any policies or strategies to handle these events — that is left to the application. We concentrate on delivering mobility related events to the application.

Although the idea of delivering events to an application is not new (UNIX® provides the *signal* mechanism), there is no universal mechanism to do this. Each operating system provides it's own event delivery abstraction and interface. Our proposed solution to this problem is to present abstract views of events at the *programming language level*. This allows an application to be implemented without committing to a particular operating system mechanism for event delivery, and makes it easy to port the application to different platforms. We describe in this paper, the implementation of our abstraction on Mach 3.0, along with the kernel enhancements we made in order to support the delivery of key events typical of a mobile computing environment.

The rest of the paper is organized as follows: Section 2 gives the design of our event delivery abstraction and outlines some of the factors that led to it. Section 3 presents the overall architecture of the event delivery mechanism and the run-time support required. Section 4 gives the details of the implementation of a prototype we built to test our ideas. Section 5 describes an extended example of user-level buffering of network communication using our abstraction. Section 6 compares our work with other related work. Finally, Section 7 presents the conclusions, fits our ideas into our broader vision, and outlines the direction of our future work.

2. Design Considerations

Various events that affect an application are possible in the mobile computing environment. Typical events may indicate that:

- the Mobile Host has moved,
- a cell hand-off has occurred,
- a PCMCIA card has been added or removed,
- the radio signal quality has deteriorated,
- the battery has become weak,
- a new network has been detected
- better bandwidth is available
- the bandwidth has deteriorated

An application that chooses to be aware of any of these events must use event delivery services offered by the underlying platform. The underlying platform includes the operating system and possibly, some run-time support. Different platforms provide different services and abstractions for event delivery, each with it's own advantages and disadvantages. We discuss in this section, the different options we had in designing our abstractions for events in the mobile computing environment, and the reasons we made our decisions.

Language Abstractions vs Operating System Primitives: The idea of delivering events to user space is not new. The *signal* mechanism in UNIX® and the *evc_wait* trap provided by Mach are examples of existing event delivery mechanisms. The Microsoft Windows® programming environment is entirely event based, where a typical application program is in a loop, waiting for the occurrence of an event and then processing it. The problem is that each operating system

provides a different abstraction of event delivery. Our motivation for providing a language level abstraction for event delivery is rooted on the idea of making an application *portable across different platforms*. This is important with the variety of portable computing devices available today. Any operating system peculiarity is encapsulated in our language level abstraction and its run-time support.

Providing a language abstraction also increases flexibility in terms of the set of events delivered. For instance, radio devices often provide information about radio signal quality while most UNIX-like operating systems do not provide a *signal* for the *bad-signal-quality* event. The option of adding a new *signal* to the existing *signal* mechanism is arduous because it involves modifying the operating system and it would have to be done for every new event. It is easier to introduce run-time support to monitor the radio signal quality and deliver the event through our language abstraction.

Monitored and Triggered Events: The various events possible can broadly be classified into *triggered events* and *monitored events*. A *triggered event* is one that is caused by the change of some low-level state. Examples include the change in state of the PCMCIA slot when a card is inserted or removed, or the change in the location of the computing device when a cell hand-off has occurred. We introduce the term *monitored event* to describe one that is generated when a measured quantity crosses a boundary value. Examples of *monitored events* are those generated when the battery becomes *weak*, when the signal-to-noise ratio (*SNR*) becomes *low*, when the processor load crosses some value, or when the amount of free memory becomes low. The motivation for this classification is that we believe an application needs different functionality in each of these cases.

Consider a *monitored event*: an application may want to define it's own notion of *weak* in the case of the battery, *low* in the case of the *SNR* or *reasonable* in the case of bandwidth. No similar application specific redefinition is needed in the case of *triggered events*. *EventObjects* and *MonitorObjects* are abstractions of the functionality required when *triggered events* and *monitored events* respectively, are delivered to an application program.

It may be argued that a *monitored event* is a programmable *triggered event* where the exact condition for event generation is programmable; also a *triggered event* can be generated using a *monitored event* to monitor a low-level state — why then is there a need for different abstractions for the two? The reason is two fold: efficiency and flexibility. A *triggered event* is more efficient because there is no unnecessary polling; a *monitored event* incurs the overhead of polling. On the other hand, a *monitored event* is more flexible because it can be generated with only a mechanism to observe state; a *triggered event* needs support from the underlying platform in order to deliver change of state. Further, a *monitored event* allows a qualitative decision to be made about when the event actually occurs while a *triggered event* does not. Support for both kinds of events is provided and the choice of using one or the other is left partly to the implementor of the run-time support for a particular platform, and partly to the application programmer.

Granularity of Event Delivery: An application program may define several *EventObjects* or *MonitorObjects* — an event is delivered once to each object. This is at a finer granularity than the UNIX *signal* which is delivered once to each process. The motivation for delivery to an object lies in the fact that each module in an application may need to handle events in it's own way. It is conceivable that the display module wants to shut off the display in response to the *low-on-battery* event, while the communications module of the same application wants to disconnect an open connection. In order to do this using a UNIX

signal, both modules would need to insert a module specific handler into the UNIX *signal* handler. This gives the effect of delivering events at a finer granularity, but makes the event-handler static in that a module cannot reprogram it's event-handler easily. With the new *EventObject* and *MonitorObject* abstractions, each module just defines its own object with a customized event-handler.

From a software engineering perspective, different modules of a large application are usually developed by separate groups who get together only to define how modules interact with one another. By delivering events at a granularity smaller than a process, there is support for a module to receive events and program an event-handler independent of other modules. This makes modules less dependent on each other making the design and development process that much easier.

Object Oriented Approach: It is possible to provide a mechanism that delivers events at a finer granularity than a process without using object-oriented techniques. For instance, it is possible to allow multiple signal-handlers for the same *signal* in a process so that when the *signal* is delivered, each handler is called, one at a time. However, this approach would either be bound to existing event delivery mechanisms or would involve infrastructure to actually deliver the event. The approach is platform dependent and it is difficult to conceive of any similar mechanism which is platform independent. By using object-oriented techniques, code associated with the event-delivery mechanism is shared by all event-handlers, and the event-delivery interface is maintained across different platforms.

Event Related Information: Events are often associated with information — for instance, when a PCMCIA card is inserted, the type of the card that was inserted and the slot into which it was inserted are very likely to be used by any card-insert event-handler. Such an event-handler will first retrieve this information and then perform a series of actions. Although most current event delivery mechanisms do not deliver any event-specific information with the event, we believe that small amounts of information can be piggy-backed along with event delivery. This makes it easier to implement an event-handler at little or no extra cost; no event-specific information needs to be retrieved in the event-handler since it is all ready and available. In our abstraction, event-specific information is made available at the event delivery endpoint in the form of parameters to the event-handler function. This event-specific information is defined in the signature of the event-handler. In the case of the PCMCIA card-insert event-handler, both the card-type and the slot are provided as formal parameters, and these are initialized to the actual values when the event occurs and the event-handler is called.

Event Handler Execution: Any useful event delivery mechanism must provide infrastructure to perform actions in response to the event. Such an entity is usually an imperative program segment (event-handler) that would execute either on an existing thread by preemption or on a different thread created specifically for this purpose. If the event-handler preempts an existing thread as is the case with the UNIX *signal*, the preempted thread cannot be active at the same time. This makes it difficult for the preempted thread to negotiate with the event-handler in order to alter it's behavior. For instance, suppose a thread is within a critical section for memory allocation when a *more-memory-available* event preempts it. A deadlock condition arises because the event-handler cannot add the newly available memory without waiting for the preempted thread to leave the critical section, and the preempted thread cannot continue until the event-handler completes. Such situations can be prevented by providing a separate event thread on which the event-handler executes — any state that is shared with other threads must be accessed through suitable locks.

In our *EventObject* and *MonitorObject* abstractions, the event-handler is a *method* associated with the object, and this *method* executes on a separate thread pre-created within the object. This is similar to the interrupt handling mechanism in Solaris 2, where unlike earlier versions of Solaris (and SunOS), an interrupt handler executes on pre-allocated interrupt threads maintained specifically for interrupt handling [1]. The *method* signature in our abstraction specifies formal parameters through which event-specific information is provided as actual parameters. The task of an event handler implementor is now reduced to defining an object of the required type and providing the appropriate *methods* to handle the events.

Run-time Support: In order to provide abstractions such as those we propose, there is a necessity for run-time support in the form of functionality implemented within a library and in a server daemon. The *EventObject* abstraction depends on events generated by the system: there is a need for a daemon that can register requests for event delivery, trap the occurrence of the event, and then deliver it after collecting the associated information. All operating system peculiarities of event delivery are encapsulated in this daemon, which we call the *Event Registrar*. What remains is the delivery of the event to each end point. Operating system peculiarities of the inter-task communication required to achieve this are encapsulated within the library implementation of our abstractions. Since the *MonitorObject* abstraction only depends on the mechanisms provided to observe the environment variable in question, no equivalent daemon is necessary, and all functionality is encapsulated within it's library implementation.

3. Event Delivery Architecture

Our event delivery mechanism delivers events to C++ objects. We provide a *base class*, which both encodes endpoint delivery and provides the

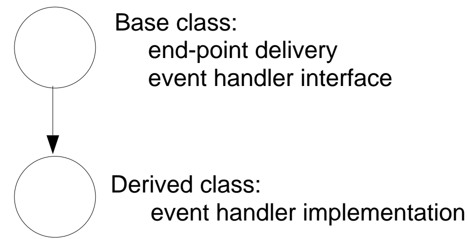

Figure 1: Inheritance Graph and Functionality

event-handler interface. The event-handler interface is defined as a *virtual function* in the *base class*; this function is called when the event occurs. An implementor uses this *base class* to define a new *derived class* in which the actual event-handler is a *method*. Since the event-handler had been defined as a *virtual function* in the *base class*, control is passed to the actual event-handler in the *derived class* when the event occurs. The reader is referred to [2] for a complete description on the concept of *virtual functions*.

We have chosen the C++ language binding [2] for our implementation because of (i) the availability of the GNU C++ compiler, (ii) the fact that C++ can easily be integrated with existing libraries and (iii) the acceptance of C++ in the program development community. We believe, though, that our ideas can be implemented in any object oriented programming language with support for indirect function invocation and a system with support for threads along with a means for inter-task communication.

The EventObject Architecture: The public interface of a general *EventObject* base class is shown in Figure 2 and the architecture of the *EventObject* and it's run-time support is shown in Figure 3. It is natural to report any event that is directly generated by the operating system, as a *triggered event* through an *EventObject*. The *Event Registrar* is the run-time support module in which all *triggered events* are detected and forwarded.

The *EventObject* constructor takes a bit-mask as a parameter which defines the set of events

```
class EventObject {
public:
      EventObject(unsigned EventMask);     // The Creator
      ~EventObject();                      // The Destructor
virtual void EventX(int Parameter);        // Event Handler for X
virtual void EventY(int Parameter1, int Parameter2);
virtual void EventZ(/* Formal parameters for Z-related information */);
... // Other Event Handler Function Signatures
}
class EventTest :public EventObject {
public:
      EventTest() :EventObject(EVENT_X | EVENT_Y) {};
                                         // Interested in Events X and Y
      void EventX(int Parameter);        // Redefinition of Event Handlers
      void EventY(int Parameter1, int Parameter2);
}
void EventTest::EventX(int Parameter) {
      // Code for EventX Handler
}
void EventTest::EventY(int Parameter1, int Parameter2) {
      // Code for EventY Handler
}
```

Figure 2: Public Interface of the EventObject Class and its Use

that will be delivered to the object that is created. When an *EventObject* is created, it registers itself with the *Event Registrar*, and part of this registration process includes specifying the set of events that must be delivered to it. Whenever an event notification is received by an *EventObject*, control is passed to the appropriate event handler. Corresponding to each possible event, there is one event-handler interface — only three of these are shown. The complete *EventObject* class would have event handler interfaces for every *triggered event* that is supported, besides private members of the class.

We henceforth use the term *EventObject* to denote all instances of classes derived from the

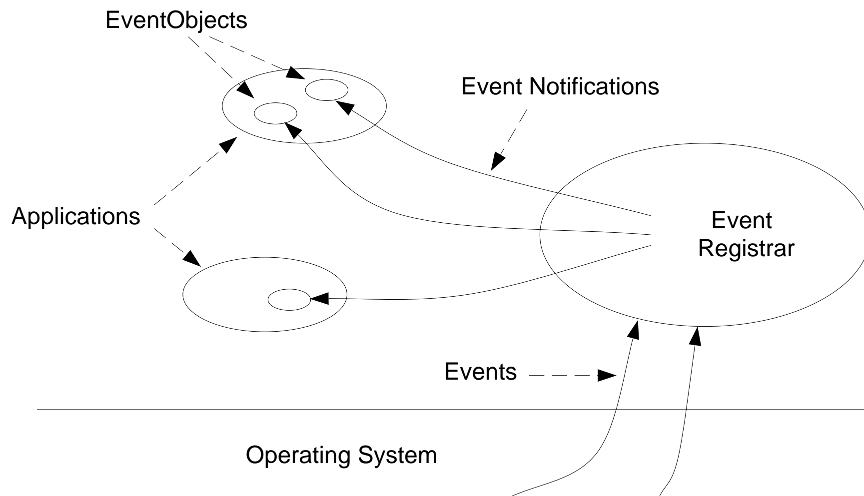

Figure 3: EventObject Architecture

```
class MonitorObject {
public:
      MonitorObject(int TimeOut, int LoWaterMark, int HiWaterMark);
virtual int MonitorFunction();
virtual void MonitorLoWaterMarkFn();
virtual void MonitorHiWaterMarkFn();
}
```

Figure 4: Public Interface of the MonitorObject Class

EventObject base class, and the term *EventObject Class* to denote the base class itself.

In order to correctly forward an event, the *Event Registrar* maintains the identity of all *EventObjects* that exist in the system. Along with this identity, the set of events in which the *EventObject* is interested is also maintained. Appropriate event notifications can hence be sent to every *EventObject* that is interested in a particular event.

Also shown in Figure 2 is the *EventTest* class which shows how to define the event handlers for the events *X* and *Y*. The *EventTest* constructor calls the *EventObject* class constructor indicating interest only in the *X* and *Y* events. The event handler functions for the two events are then defined. Since the event handler functions are defined as virtual functions in the *EventObject Class*, control is passed to *EventTest::EventX* in lieu of *EventObject::EventX* when event notification *X* is received by an instance of the *EventTest Class*.

The MonitorObject Architecture: The public interface of the *MonitorObject* class is shown in Figure 4. The *MonitorObject* class only provides a framework for monitoring an environment variable. The constructor takes three parameters: a time-out value, a low watermark, and a high watermark. The time-out value defines a time-period indicating the frequency with which the *MonitorFunction* method is called, internally. This function is defined in a *derived class* and would make the actual observation of the required environment variable. The high and low watermarks are values of the environment variable which when crossed generate an event. The appropriate function is called as shown in Figure 5.

In its most basic form, the *MonitorObject* class only defines the interface that should be provided by a *MonitorFunction* besides that of the event handler functions. A family of derived classes can thus be defined, by providing a *MonitorFunction* which returns an integer value that is significant in the mobile environment (signal-strength, processor load, etc.). We expect to provide a few such derived classes where an implementor would only need to add her event handler functions. However, an implementor could use the basic *MonitorObject* class to monitor a different environment variable.

4. Implementation Details

We have implemented our ideas for event delivery on an Intel 80486 based Toshiba portable running the Mach 3.0 microkernel. We use the PCMCIA card insert and remove events as examples of *triggered events*, and the signal-quality as an example of a *monitored event*. All device driver

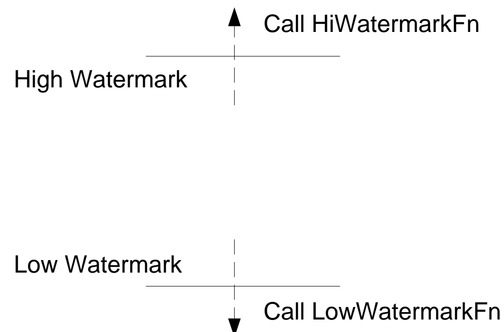

Figure 5: MonitorObject Functionality

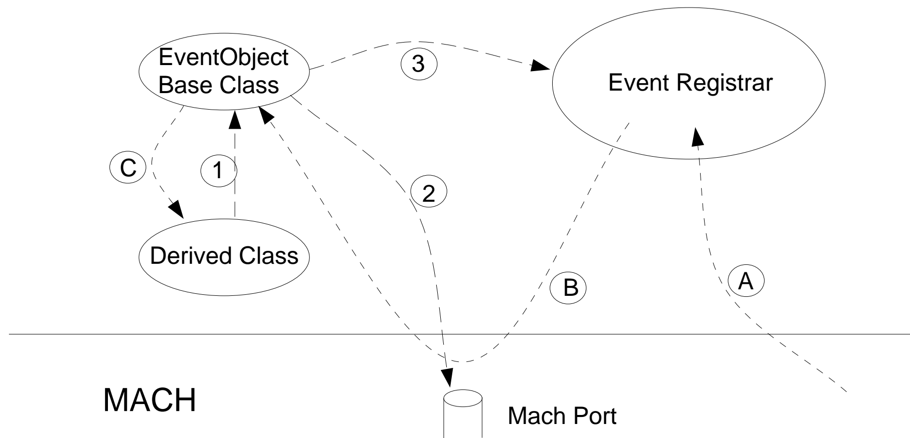

Figure 6: EventObject Creation and Delivery

support for handling the PCMCIA card interrupts was developed as part of this work.

4.1 The EventObject Abstraction

When an *EventObject* is declared (Figure 6–1), a Mach port is created (2) and registered with the *Event Registrar* (3). A port is a Mach abstraction which defines a communication endpoint. All event notifications arrive as Mach messages through this port (B). Event specific information arrives as part of the event notification and is passed on to the event handler as actual parameters. Since the event handler interface is defined as a virtual function, the call is made though a pointer, and the effect is that the implementor-defined event handler in the derived class is called on event delivery (C).

An *EventObject* is *active* in that there is an associated internal thread of control. This thread is created along with the *EventObject* and all activity associated with the *EventObject* is performed on this thread. The thread is in a forever loop that waits for event notifications on the port, determines the type of the event, and then executes the appropriate event handler. An implementor of the event handler may therefore, need to access any external objects through suitable locks so as to prevent any inconsistencies caused by concurrent access.

The *Event Registrar* is a separate Mach task in our implementation. In order to allow an *EventObject* to register itself, the *Event Registrar* advertises a special well-known port through the Mach Environment Manager and all registration requests are addressed to this port.

The *Event Registrar* is implemented as a set of threads of which one waits for registration requests from *EventObjects*. A registration request

```
class EventObject {
      mach_port_t    EventPort;        // Mach port created by object
      mach_port_t     EventThread;     // Thread created by object
      EventThreadLoop();               // Thread Loop
public:
      EventObject(unsigned EventMask);
      ~EventObject();
virtual void EventCardInsert(unsigned slot, unsigned card);
virtual void EventCardRemove(unsigned slot, unsigned card);
}
```

Figure 7: Example EventObject Class Implementation

is accompanied by the port defined by the *EventObject* and the set of events that the object is interested in. The *Event Registrar* organizes this information as a table with the set of ports (*EventObjects*) to which event notifications are to be sent for each event type. An internal *EventReport* function is provided in order to send an event notification, given the event type and the associated information. This function looks up the table maintained by the *Event Registrar* to determine the *EventObjects* to which a notification must be sent. An event notification is therefore sent to an *EventObject* only if it is contained in the set of events for which the object had registered.

The *Event Registrar* creates a thread for each event type and mechanisms provided by the host operating system are used to detect an event. On detecting an event, the thread collects information associated with the event and calls the *EventReport* function. In order to demonstrate the flexibility of the *EventObject* abstraction and build a prototype *Event Registrar*, we added support to our Mach PCMCIA device driver to handle the card insertion and removal interrupts, and use these as examples of *triggered events*.

The Intel 82365 chip controls the PCMCIA bus interface on many portable computers. This chip can be programmed to generate an interrupt whenever a change is detected on the state of one of the PCMCIA slots that it controls. Whenever a PCMCIA card is inserted or removed the change of state can be determined in the interrupt service routine by reading a status register on the chip. This change is communicated to user-space through the event count service provided by Mach. A thread in user-space waits for an event using the *evc_wait* call, which returns when the change of state is detected and reported by the device driver. Any state change is recorded in the driver in case multiple associated events occur close to each other, and the semantics of the event count service guarantees that no such event is lost. The thread in the *Event Registrar* that detects the card insertion and removal events waits for either using the *evc_wait* call. When it returns, the thread initiates actions to report the event.

```
class Enabler :public EventObject {
public:
      Enabler() :EventObject(EVENT_CARD_INSERT | EVENT_CARD_REMOVE) {};
      void EventCardInsert(unsigned slot, unsigned card);
      void EventCardRemove(unsigned slot, unsigned card);
};
void Enabler::EventCardInsert(unsigned slot, unsigned card) {
      if (card == PCMCIA_WAVELAN)
         system("/etc/wavelan.start");
      else
         cout << "Unsupported card\n";
}
void Enabler::EventCardRemove(unsigned slot, unsigned card) {
      if (card == PCMCIA_WAVELAN)
         system("/etc/wavelan.stop");
      else
         cout << "Unsupported card\n";
}
main() {
      Enabler e;
      thread_suspend(mach_thread_self()); // Main thread suspends execution
}
```

Figure 8: Inheriting from the EventObject Class

Our current PCMCIA device driver architecture requires all card specific drivers to be part of the Mach kernel (This is an initial version and one of our goals is to allow a card driver to reside outside the kernel and to load it from user-space after the type of the card is determined). At boot-time, if a card exists in the PCMCIA slot, it is probed by each card driver to determine if it can be supported. In case it can be supported, the appropriate driver initializes and sets up the card so that it can be eventually configured from user-space.

In order to support the insertion and removal of a card while the system is still up and running, we added support to the PCMCIA controller device driver to probe the card and set it up through the *device_get_status* and *device_set_status* calls. If a card exists at boot-time, the same earlier sequence of actions occurs, but if a card is absent, nothing happens until a card is inserted. When a card is inserted, the PCMCIA device driver detects the event and reports it to user space. The *Event Registrar*, which is waiting for the event in *evc_wait*, wakes up and determines the kind of card that has been inserted, and the slot into which it has be inserted using appropriate *device_get_status* calls. The *EventReport* function is now called to notify all *EventObjects* that are interested in the event.

Figure 8 shows a small working program that we use to detect the card insertion and removal events and start the network. The *Enabler* class constructor calls the parent *EventObject Class* constructor with a parameter indicating its interest in the two PCMCIA interrupts. The corresponding event handler functions are declared and defined. The event handlers determine if the card inserted or removed was a WaveLAN card, and start or bring down the network appropriately. The file */etc/wavelan.start* is a shell-script similar to */etc/netstart* which configures the IP-address using the familiar *ifconfig* program and also sets up the routing tables appropriately. */etc/wavelan.stop* is a similar shell-script used to shut down the network when the card is removed. The only thing left to do is to declare an object of the new *Enabler Class*, which is all that is done in the main procedure.

This shows the ease with which useful event-driven programs can be written using our *EventObject* abstraction. A complete example of application-level buffering using *EventObjects* is presented later, in Section 5 of this paper.

4.2 The MonitorObject Abstraction

A *MonitorObject* also creates a thread when declared. This thread wakes up periodically, after sleeping for a period of time specified by the time-out value. When it wakes up, the thread calls the *MonitorFunction* method to determine the current value of the environment variable being monitored. If the new value is such that one of the watermarks has been crossed, the appropriate event handler function is called similar to the case of an *EventObject*. Again, these event handler functions are defined by an implementor in a derived class.

To show the flexibility of the *MonitorObject Class*, we build the *SignalMonitor Class* (Figure 9) to observe and react to the recorded signal-quality. Information maintained by our WaveLAN device driver is used to determine the signal-quality. We will use this *SignalMonitor Class* in our extended example presented in Section 4.

Here, *MonitorFunction* is called every *TimeOut* time units and depending on the change in the signal-quality, the appropriate event handler is called. The values corresponding to the low and high watermarks are defined when the *SignalMonitor* is created. Whenever the high watermark is crossed from below, *MonitorHiWaterMarkFn* is called, and when the low watermark is crossed from above, *MonitorLoWaterMarkFn* is called. The event handlers shown just print out whether the signal-quality is good or bad.

```
class SignalMonitor :public MonitorObject {
public:
      SignalMonitor(int TimeOut,int Min,int Max)
                   :MonitorObject(TimeOut,Min,Max) {};
      int MonitorFunction();
virtual void MonitorLoWaterMarkFn();
virtual void MonitorHiWaterMarkFn();
};
int SignalMonitor::MonitorFunction() {
      int signal;
      get_signal_strength(&signal);
      return signal;
}
void SignalMonitor::MonitorLoWaterMarkFn() {
      cout << "Bad Signal\n";
}
void SignalMonitor::MonitorHiWaterMarkFn() {
      cout << "Good Signal\n";
}
```

Figure 9: Inheriting from the MonitorObject Class

5. An Extended Example

In order to demonstrate the power of our event delivery abstractions, we present the implementation of a user-level deferred-send primitive. The idea is to detect the existence of network connectivity at user-level, and to judiciously choose to send or buffer based on whether the chance of a successful send is high or low.

The new *DeferredUdp Class* (Figure 10) we define makes use of the *SignalMonitor* and *EventObject* classes we defined earlier, to keep track of the observed signal-quality and the existence of the network card. An additional *UdpObject Class* is assumed to exist, so as to maintain the overall object-oriented approach (we have implemented this class, but choose not to present it here as it is irrelevant). This class offers primitives to create a UDP endpoint given a port number, and to send and receive a packet through the endpoint.

The *DeferredUdp* class constructor takes as parameter, the port number which will identify the UDP endpoint. This endpoint is created using the *UdpObject* base class constructor. The *SignalMonitor* base class constructor is called with the required parameters. The time-out value of 1000 time units defines the frequency with which the signal-quality will be monitored. The high and low watermarks are set to be the same value because we intend to actually transmit a packet if the signal-quality is greater than 14, and buffer packets if it is less. This value was chosen as the cutoff because we observed that when the signal-quality reported by the WaveLAN device was lower than 14, a large number of packets were lost. The *EventObject* class constructor is called with the appropriate parameter so that only card insertion and card removal events are reported.

The next step is to construct the event-handlers. We maintain the state of the network in the boolean member, *NetIsUp*, and maintain unsent packets as a list of packets in the *SendQueue* member. The card-insert event-handler checks if the card that was inserted is a WaveLAN card: if it is, *NetIsUp* is marked appropriately, and packets are extracted from the *SendQueue* and actually sent out. The card remove handler just marks *NetIsUp* to be FALSE so that incoming packets are now buffered. The *MonitorHiWaterMarkFn* is called when the signal-strength becomes greater than 14, which is our cutoff value. *NetIsUp* is set to TRUE, and all buffered packets are flushed out.

When *MonitorLoWaterMarkFn* is called, there is a high probability of packet loss and so *NetIsUp* is set to FALSE.

Finally, the *DeferredSend* method is built. When *NetIsUp* is TRUE, the network is up and the signal-quality is good — the packet is sent out over the network. Otherwise, the packet is buffered in *SendQueue* and will be flushed out when conditions become favorable for transmission.

The implementation of the *DefferedUdp Class* demonstrates the use of our event delivery abstractions. The ease with which an application can monitor the state of the mobile environment is evident. It can also be seen that the code is portable onto any platform which provides the run-time

```
class DeferredUdp :public SignalMonitor, EventObject, UdpObject {
      boolean_t      NetIsUp;
      SLList<Packet> SendQueue;
public:
      DeferredUdp(u_short PortNum):
                     SignalMonitor(1000, 14, 14),
                     EventObject(EVENT_CARD_INSERT | EVENT_CARD_REMOVE),
                     UdpObject(PortNum)  {NetIsUp = FALSE;}
      void DeferredUdpSend(Packet& SendPkt);
      void EventCardInsert(unsigned slot, unsigned card);
      void EventCardRemove(unsigned slot, unsigned card);
      void MonitorHiWaterMarkFn();
      void MonitorLoWaterMarkFn();
};
void DeferredUdp::EventCardInsert(unsigned slot, unsigned card) {
      Packet pkt;
      if (card == PCMCIA_WAVELAN) {
            NetIsUp = TRUE;
            while (SendQueue.remove_front(pkt))
                  UdpSendTo(pkt);
      }
}
void DeferredUdp::EventCardRemove(unsigned slot, unsigned card) {
      if (card == PCMCIA_WAVELAN)
            NetIsUp =  FALSE;
}
void DeferredUdp::MonitorHiWaterMarkFn() {
      Packet pkt;
      NetIsUp = TRUE;
      while (SendQueue.remove_front(pkt))
            UdpSendTo(pkt);
}
void DeferredUdp::MonitorLoWaterMarkFn() {
      NetIsUp = FALSE;
}
void DeferredUdp::DeferredUdpSend(Packet& SendPkt) {
      if (NetIsUp)
            UdpSendTo(SendPkt);
      else
            SendQueue.append(SendPkt);
}
```

Figure 10: An Implementation of a User-Level Deferred UDP Send

support required.

6. Related Work

The work presented in this paper is the initial portion of our broader goal of providing programming language support for mobile computing. Our experience shows us that one peculiarity of the mobile computing environment is the necessity for an application to react to changes in the environment. This fact seems to be agreed upon by other researchers as well. Noble, Price and Satyanarayanan [3] from CMU talk about application-aware adaptation for file access. Zenel and Duchamp [4] from Columbia talk about application level communication filtering to reduce wireless traffic. Neuman, Augart and Upasani [5] from ISI, USC discuss the role of directory services in support of location-independent computing. Each of these require the monitoring of mobility related events of some form and reacting to them. All of them involve run-time support to aid in this task. In this paper, we try and abstract out the functionality required to make an application aware of changes associated with the mobile environment.

Terri Watson from University of Washington, Seattle presents strategies for designing applications for a wireless environment [6]. Among the design strategies suggested are *"expose network costs, utilize workstation resources, and adapt to variations in network connectivity"*. Each of these strategies require an application to be aware of an aspect of the environment and then react to a change in it. Our event delivery abstractions provide a uniform mechanism for this and can be the basis for implementing all such strategies.

Schilit, Theimer and Welch from Xerox PARC describe a system [7, 8] for a dynamically changing environment where applications need to be dynamically customized. A high level RPC based service is provided to obtain current environment related information synchronously and a change in this environment though a callback. The emphasis is on maintaining environment variables similar to shell environment variables such as the NEAREST_PRINTER or NEARBY_DISPLAYS. Our ideas, on the other hand, work towards providing programming language level abstractions of low-level events that occur in the system like the addition of a network interface or the degradation of the transmission signal strength. These events are more fundamental than a change in the nearest printer, and would affect the performance of the system and the observed quality of service. Our aim is to give an application access to even the most primitive events that may be generated in the environment so that the application can customize itself and give best possible quality of service to the user.

The idea of delivering events associated with change in the state of remote objects in a distributed system is described by Jim Waldo and others [9] from Sun Microsystems Laboratories. The motivation is to extend the RPC paradigm to provide a callback mechanism in the form of an event. Our approach is more restrictive and is tuned to detecting changes in the local state of a mobile environment. The focus is on providing a platform independent method of delivering such changes in the form of events.

Michael Bender and others from SUN Microsystems suggest various enhancements to the operating system to support nomadic computing [10]. Among these are kernel changes to support power management and system state checkpointing, and drivers and other kernel support for the PCMCIA bus standard. The system support we suggest is similar to this but involves propagating events right up to the user, and provides language abstractions for this so that event delivery becomes independent of the platform.

Gabriel Montenegro and Steve Drach, also from SUN Microsystems argue that *"the approach of making an application aware of the characteristics of the underlying communication mechanism is*

expensive and time consuming" [11]. They suggest an alternative, implemented entirely within Solaris 2.4, where network operations which are guaranteed to fail do so fast. In doing so, however, they still needed to make changes to network service modules and some utilities. Although their approach is to minimize application awareness, they could not eliminate it. Our approach is quite the opposite: we believe we *must* make an application aware of the environment. We can manage with minimum changes to the operating system.

Our larger goal is to provide platform independent programming support for mobile computing. The event delivery abstractions described in this paper provide the awareness portion of this support. We are currently working on programming level abstractions to support actions we believe will be common in mobile computing environments. We believe these abstractions will be useful in implementing an application's reaction to the events it is interested in.

7. Future Work and Conclusions

Distributed systems are hard to design and implement; distributed systems for mobile hosts are harder still because of constraints, dynamic change, and mobility. One approach to simplify the design and implementation of a mobile computing application is to provide a uniform programming language level abstraction through which all environment related events and actions are reported and performed. Functionality is encapsulated within this high level abstraction and makes an application easily portable to different platforms. We have presented in this paper a prototype implementation of our event delivery abstractions. Mobility and environment related events appear as methods to which control is passed when the events occur. A uniform programming view is therefore provided, and this view is the same no matter what the underlying operating system is. One aspect of our future work is to port our prototype onto other platforms so that we have the same event delivery abstractions available on them. In doing so, we also hope to determine what minimal functionality is required of the underlying operating system, and how several *"real"* operating systems actually compare in providing it.

Assuming that it is possible to support our event delivery abstractions on a variety of platforms, it is necessary to show that useful user-level policies can be implemented using them. It should be possible for applications to alter their behavior so as to make the best possible use of available resources. This is essential in a mobile computing environment where resources are limited and intermittent. We have presented in this paper the implementation of a policy whereby an application can defer network communication until such a point that it is most likely to succeed. This policy is simple to implement using our event delivery abstractions, and can be very easily altered if necessary. Other environment dependent policies such as those described by Huston and Honeyman [12] can also be easily implemented using our event delivery abstractions.

The event delivery abstractions proposed in this paper give an application the basic mechanism to be aware of the mobile computing environment. A major focus of our future work is to investigate and construct general mechanisms that will aid the application in reacting to changes in the mobile environment. We are not certain as yet about the exact functionality that is necessary or useful, but we expect that a mechanism to restrictively *migrate* portions of an application and one to make portions *persistent* will be useful in the environment. We intend to eventually encapsulate all such mechanisms within programming language abstractions similar to our event delivery abstractions.

We believe that with the variety of hardware and platforms available today, it is essential that applications are easily portable. Since mobile computing adds a new dimension to the distributed

systems problem, it is necessary to identify and package these specific problems in a manner independent of the underlying platform. One aspect of all that must be packaged is the event delivery mechanism. We believe that our abstractions for event delivery will evolve into useful primitives to build applications for the mobile computing environment. Keeping in mind our broader vision, our event delivery mechanism is the eyes for the mobility aware portion of an application; the hands are under construction.

Bibliography

[1] Steve Kleiman and Joe Eykholt. Interrupts as threads. In *Operating Systems Review Vol. 29, No. 2*, April 1995.

[2] Bjarne Stroustrup. *The C++ Programming Language*. Addison Wesley, 1991.

[3] Brian Noble, Morgan 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.

[4] Bruce Zenel and Dan Duchamp. Intelligent communication filtering for limited bandwidth environments. In *5th Workshop on Hot Topics in Operating Systems*, May 1995.

[5] Clifford Neuman, Steven Augart, and Shantaprasad Upasani. Using prospero to support integrated location-independent computing. In *Proceedings of the USENIX Symposium on Mobile & Location-Independent Computing*, August 1993.

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

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

[8] Bill Schilit, Norman Adams, and Roy Want. Context-aware computing applications. In *IEEE Workshop on Mobile Computing*, December 1994.

[9] Jim Waldo, Ann Wollarth, Geoff Wyant, and Sam Kendall. Events in an rpc based distributed system. In *Proceedings of the Winter 1995 USENIX Conference*, January 1995.

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

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

[12] L. B. Huston and P. Honeyman. Partially connected operation. In *Proceedings of the 2nd USENIX Symposium on Mobile & Location-Independent Computing*, April 1995.