The DIANA Approach to Mobile Computing

Arthur M. Keller, Stanford University
Tahir Ahmad, Stanford University
Mike Clary, Sun Microsystems
Owen Densmore, Sun Microsystems
Steve Gadol, Sun Microsystems
Wei Huang, Stanford University
Behfar Razavi, Sun Microsystems
Robert Pang, Stanford University

Abstract

DIANA (Device-Independent, Asynchronous Network Access) is a new application architecture to solve two major difficulties in developing software for mobile computing — diversity of user interface and varied communication patterns. Our architecture achieves display and network independence by decoupling the user interface logic and communication logic from the processing logic of each application. Such separation allows applications to operate in the workplace as well as in a mobile environment in which multiple display devices are used and communication can be synchronous or asynchronous. Operation during disconnection is also supported.

1. Introduction

People want to access their applications not only while in the workplace but also while they travel, and while using a variety of devices. Although much current computer software can provide sophisticated functionality to ordinary users, the software rarely addresses the special needs of mobile users.

A number of issues limit the usability of software for mobile users. We consider two particular issues: user interface and communication. Conventional software is usually developed with implicit assumptions about the kinds of display devices users have and the communication media available between them and the users. For example, applications written for the X-windows protocol can only be used by those users who run X-windows across high-speed computer networks. Such software is useless to mobile computer users with a different kind of communication environment.

Various computing devices tailored for mobile users are currently available, such as notebook computers, palmtop computers, and personal digital assistants (PDA). The user interface on these devices differs greatly. To make software accessible to these devices, application developers currently have to customize their software for each individual type of device. We anticipate an even greater variety of such mobile devices in the future and such diversity can put a significant burden on software developers.

On the other hand, the communication networks available to mobile users are still relatively slow and unreliable, as compared with the high-speed local area networks that connect workstations, servers, and mainframes. Also, various communication protocols are used for mobile computing and handling all these different protocols can be a significant software development cost. Moreover, mobile users may need to communicate intermittently because of the high cost or unavailability of communication.

To overcome these two hurdles in software development for mobile computing, we propose a new application architecture for mobile applications. The DIANA architecture — Display Independence and Asynchronous Network Access — de-couples the display and communication logic of applications from their processing logic. The resultant applications will enjoy the benefits of being display and transport independent. Not only will new applications be able to take advantage of this architecture, but also we envision that existing applications designed for directly connected users on particular devices will migrate to our approach.

1.1 Background

Our work on this project started by looking at workflow systems closely and trying to understand why people had failed to use them extensively in the industry. Our study raised the following issues as hindrances in the applicability of such systems.

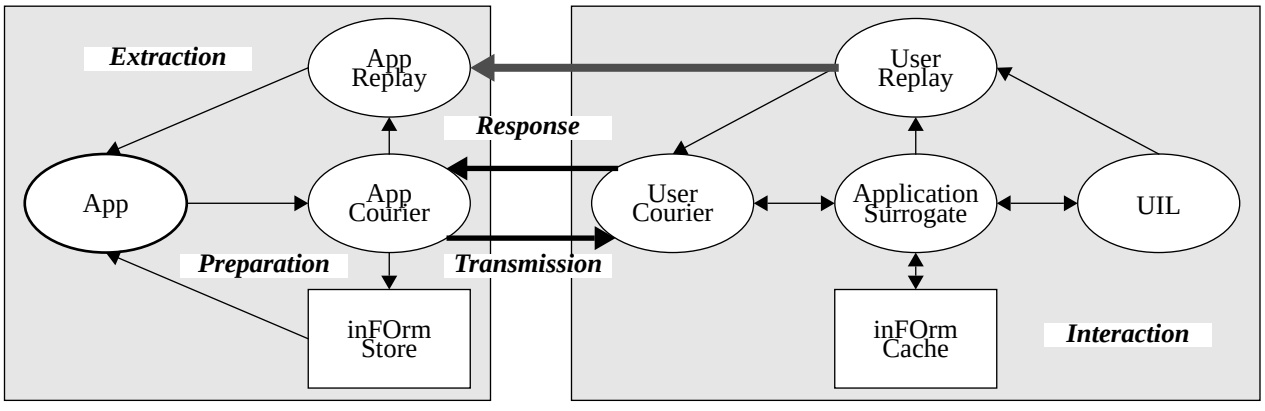
Platform Bindings. One desirable aspect of these systems is that they should be able to make legacy applications work together in a broader context than they were originally designed for. However, most of these legacy applications are bound very strictly to the platform for which they were designed. Porting these applications to multiple platforms incurs significant development and maintenance costs.

Network Management. Both the legacy applications and the workflow management systems have embedded assumptions about the underlying networks and the available communication protocols that they will be using. These assumptions provide significant inflexibilities and limitations. Furthermore, networking across hardware platforms introduces complications such as security. Also, there may be the unavailability of similar operating system features on different platforms, e.g., a workflow system relying on RPC (remote procedure call) to implement triggers becomes ineffective if one participant system does not provide support for RPC.

Design Complexity. An inherent problem with workflow systems is the complexity of designing working workflow models. Most of the systems, in their bid for providing the designers maximum flexibility, put a lot of responsibilities on the designers. Designers have to manage the conceptual entities in their workflow model, such as roles, tasks, jobs. Designers also have to be aware of the heterogeneity of the physical resources their system will be using and the restrictions this heterogeneity will impose on their model.

Different Data Representation. Multiple applications comprising a workflow system often have different views of data, which makes it more difficult to

FIGURE 1. DIANA − Overall Architecture and inFOrm Life Cycle



develop a generic solution for inter-application workflow.

To address these issues, we developed the DIANA architecture. DIANA uses a universal, dynamic language understood by all the entities in the system (applications, users, and system components) that is based on the semantics of interaction. This language can be interpreted on multiple platforms, so that the same core application becomes accessible from a variety of heterogeneous devices. The network components of DIANA are capable of switching between synchronous (e.g., direct connect) and asynchronous (e.g., e-mail based) communication modes. Applications can access the responses from clients by using a simple information-level Application Programming Interface. By providing a single DIANA client to legacy applications, they can be made to interact with all other entities using the DIANA architecture.

1.2 Related Work

Considering display and network independence, we observe that Mosaic and Telescript [Caruso93] have close similarity with DIANA.

DIANA has a significant overlap with Mosaic as far as the interface is concerned. Mosaic uses Fill-Out Forms to express user interactions and query certain information sources distributed throughout the network. We have implemented the sample travel authorization application using Fill-Out Forms for a comparison between DIANA and Mosaic. From our experience of using Fill-Out Forms, we observe that one key difference between the DIANA approach and the Mosaic approach is that DIANA uses an interface language based on the semantics of the user interaction. On the other hand, Fill-Out Form language part of HTML assumes the presence of an access device with certain layout characteristics. This semantic representation of interaction in DIANA makes the system extensible to non-conventional access devices, such as telephones. However, we can extend FDL by incorporating layout preferences, such as those in HTML, without sacrificing display independence.

Considering network communication, the World Wide Web, and browsers such as Mosaic, adopts stateless communication between users and the application. Where transaction states are needed, they are kept by the application. In contrast, DIANA's Courier provides connection-based communication, and it keeps the states for the client applications. We believe this stateful communication can not only reduce setup time and bandwidth requirements but also facilitates caching information locally. In addition, the Courier provides both synchronous and asynchronous communication while Mosaic provides only synchronous communication currently.

General Magic's Telescript promises network and device independence. Unfortunately, very little information about Telescript is publicly available to permit a meaningful comparison with DIANA. A simplistic comparison is that DIANA handles the user interface similar to Mosaic and the network interface similar to Telescript.

[Kazman93] presents an overview of some user interface architecture models. The Seeheim model [Pfaff85], the Arch/Slinky Metamodel, the Presentation-Abstraction-Control (PAC) model [Coutaz87] and the Serpent model [Bass90] have some overlap with the DIANA model in that they all attempt to separate the presentation of a user interaction from its functionality within the application logic. DIANA proposes a form based solution to implementing a user interface architecture in which the core application logic is de-coupled from the interface logic. The applications in DIANA use a form based approach in which applications do not have to be responsible for fine grained rendering details as user's input or retrieve information to and from the forms. The fact that existing interfaces require most applications to control the presentation on a very interactive and fine scale is regarded as one of the reasons why human computer interfaces are hard to design and implement [Myers93].

[Imielinski93] identifies some of the issues involved in mobile wireless computing, including location management, configuration management, disconnec-

tion, cache-consistency, recovery, scale, efficiency, security and integrity. The paper also presents a model of a system to support Mobility. In this system, Mobile Units are assumed to have wireless e-mail access to Mobile Support Stations (fixed network hosts with a wireless e-mail interface to communicate with the mobile units). Most of our discussion assumes the presence of a similar system for the purposes of wireless e-mail communication.

The Coda File System [Satya93] is a highly available file system designed to suit distributed and mobile computing. It uses an optimistic replica control strategy to provide high availability and relies on a dynamic cache manager to provide disconnected operation. DIANA proposes to use a similar caching strategy to minimize the network utilization and improve efficiency in addition to support disconnected operation. Details of various aspects of the Coda File System are available in [Satya90a], [Satya90b] and [Steere90].

Java [Gosling95] is a language for portable applications that may be disseminated among a variety of platforms. HotJava is a browser that allows dynamic fetch of Java applets over the Internet in a manner that extends the notion of the World Wide Web. HotJava works best in a connected network environment; it has no special features to handle disconnected or intermittently connection operation.

1.3 Outline

In this paper, we will describe the proposed architecture and explore the advantages of using DIANA architecture in mobile computing. We will also report on the status of our implementation and our experience in using DIANA. The organization of the rest of this paper is as follows. Section 2 describes the overall architecture of DIANA along with an illustrative application. Section 3 discusses how DIANA addresses the display independence problem. Section 4 focuses on the connectivity issues in DIANA. Section 5 describes the application development methodology. Section 6 discusses the current implementation of DIANA. Section 7 discusses future work, followed by concluding remarks in Section 8.

2. DIANA – The Overall Architecture

This section presents an overview of the DIANA architecture. We define the key components of the system, describe their responsibilities and explain how they work together. Subsequent sections of the paper discuss these components in further detail.

2.1 User Interface Logic

In order to de-couple the user interface logic from the processing logic of applications, we define a component called the ***User Interface Logic*** (UIL) to handle generic user interface operation on the client on behalf of applications. As a component of the DIANA system, this UIL is independent of the specific applications. There is a different UIL for each different type of display devices. Each different UIL will implement the semantics of forms using the display characteristics appropriate for the particular display device. For example, applications will be able to communicate with the UIL for a X-windows display device as they will do so with the UIL for a PDA. The end result will be that users can use different display devices as clients to the same application without requiring the application to handle each of those devices separately.

In order to allow applications to communicate with the UIL in a generic way, we design a ***Form Description Language*** (FDL) for applications to express the types of user interface to UILs. This language is based on the form paradigm and focuses on the semantics of the information exchange between the applications and the users rather than its aesthetics. A Form or script written in FDL is called an ***inFOrm*** (standing for *in**fo-fo**rm*, hence the capitals). Since FDL focuses on the semantics of information exchange rather than the characteristics of display devices, applications which use FDL to describe their user interface can truly be display independent.

2.2 The Courier

The network manager in DIANA is called the ***Courier.*** It supports network independence through the support of multiple network protocols and both synchronous and asynchronous modes. The synchronous communication mode represents direct communication with potentially high bandwidth and low latency. On the other hand, asynchronous communication is used in the situation when communication is intermittent or has high latency. For asynchrony, a store-and-forward information exchange paradigm is more appropriate. Courier makes this kind of asynchronous communication transparent to applications. We will discuss these modes in more details in a subsequent section.

There are two components of the Courier: one resides on the network with the application (Application Courier) and the other resides on the access device (User Courier). The purpose of having these couriers is to provide an encapsulation of the underlying communication medium so that both users and applications can have the same processing logic independent of whether they are communicating in synchronous or asynchronous modes.

2.3 The Application Replay

In asynchronous communication mode, users may work asynchronously with applications during disconnection. In order to deliver inFOrms from users to applications in the sequence as they are generated, we define *Application Replay,* an agent on the applications' side that presents the inFOrms that were collected during disconnection to applications as if they were generated while continuously connected.

2.4 Other components

In order to reduce communication traffic, we add an inFOrm cache on the user device to store those inFOrms which are frequently used. It can also cache those inFOrms which the user may need during disconnection, thus allowing the user to continue to work during those periods.

If the application has application-specific processing not encodeable in FDL that should occur on the client during periods of disconnectivity, then such processing can be embodied in an Application Surrogate that resides on each client device. We will discuss the application surrogate later.

The inFOrms for all applications are identified in a central registry and are stored in an inFOrm store. This approach allows users to search for inFOrms for specific purposes.

2.5 The Life Cycle of an inFOrm

Figure 1 describes the overall DIANA architecture. The following are the different phases an inFOrm goes through during its life cycle from preparation, transfer and processing.

Preparation. When a transaction starts, the application first obtains an inFOrm from a repository (inFOrm Store) and optionally fills in some default data on it such as the user's employee number and his or her department code.

Transmission. The next phase involves the physical transmission of the inFOrm from the application to the user by the Courier. The Courier is responsible for connection detection and switching between directly connected operation and disconnected operation using e-mail. The details of the mechanisms used during this transmission phase are hidden from the application.

Interaction. The user interacts with the inFOrm in this phase, providing the necessary information by filling in the form. This process involves the UIL interpreting the inFOrm script and controlling the user interaction. The user responds to the application by entering his inputs in the inFOrm rendered by the UIL on the user's display device.

Response. This phase is similar to the transmission phase, except that the flow is from the user's device to the application's machine. It involves transmitting the inFOrm replies to the Application Replay Agent.

Extraction. In this phase, the application receives the inFOrm reply from the Application Replay Agent and extracts the information from that form using the DIANA API. The transmission and Extraction phases are asynchronous from the core application's perspective. The application may then proceed to process the reply inFOrm and optionally reply to the user through another inFOrm cycle.

2.6 Application Example

Jeff Jones is a sales representative of ABC corporation. Due to the nature of his job, he often travels to meet prospective customers. Before going on a trip, Mr. Jones must use travel authorization software to submit a travel request. He uses the same application to view the results of his requests and clarify on his travel details if necessary.

By using the travel authorization software (we will simply call it the "application" in the following discussion), he first submits his travel request using a workstation running the X-windows system. The application first sends a travel request inFOrm to him via the Courier. In this situation, the Courier uses TCP/IP to transfer the inFOrm from the application to the client.

The UIL for X-windows on his workstation interprets the inFOrm and graphically renders it. He then enters his travel details and returns the reply form back to the application via the Courier using TCP/IP.

When he leaves his office to travel to his customer, he disconnects his User Courier from the Application Courier. The Application Courier, noticing this disconnection, forwards the inFOrms to him via electronic mail without affecting the application.

While away from his office, he uses a palmtop computer, a character-only device, to retrieve the status of his travel request. The User Courier first retrieves those inFOrms sent to him during disconnection from his mail box. They are interpreted by the UIL for this character-only device. Then the User Courier resumes normal communication with the Application Courier over a serial line.

In this example, we have seen how a user access the same application with multiple display devices and connects to the application sometimes continuously and sometimes intermittently.

3. User Interface and Display Independence

We consider applications that interact with users through a question and answer dialog. In this question-and-answer paradigm, applications are considered to send out questions organized as forms, called inFOrms in DIANA, and users reply to these forms. The notion of question forms provides an abstraction of user interfaces to applications. Instead of seeing the heterogeneity of access devices and user interface, applications see only an abstract interface for information exchange, but not the actual display devices. This form-based interaction helps to separate display mechanism from applications so that they are display-independent.

In fact, form-based interaction is analogous to form-based service provided by public service offices like the Department of Motor Vehicles. Someone applying for a driver's license will fill out an application form. In a similar way, users are requested to fill out inFOrms to provide inputs to applications that use the DIANA framework.

To accomplish display independence, inFOrms have no relationship with the display capability of user devices. They focus on the semantics of the information being exchanged, rather than the representation of the information on users' devices. By removing any ties they may have to any display devices, they become truly display-independent. The final representation of the information is left to the User Interface Logic which will be discussed in details later.

3.1 Form Description Language

To achieve display independence through the use of generic forms, DIANA provides FDL for applications to define their user interface as inFOrms. In Table 1, we show the list of input items provided by FDL for applications to choose the items which best

suit their input needs. More specifically, they specify the questions and the nature of the replies they expect. For example, an application may require text input to one question and an integer input within a range in another one. Notice the difference between the interaction items listed below and their representation by the GUI. For example, a one-from-many interaction describes the semantics of a question while a radio button is a possible implementation for displaying such a question.

Notice that some of the interaction items might not be renderable on some devices. Users who wish to see or hear such items will need to use suitable devices.

3.2 User Interface Logic

Rendering. The User Interface Logic (UIL) interprets inFOrms received from applications on behalf of end users. UIL renders inFOrms on users' devices according to the display characteristics of the devices. UIL is also responsible for getting responses from users and sending these back as user inputs to applications. In other words, UIL can be considered as a local translation tool of FDL for display devices while FDL is the universal language for information exchange. On the other hand, inFOrms are the scripts written in FDL to express a unit of interaction between two DIANA agents.

Each UIL is device-specific. There is an instance of a UIL on each category of access device. When UIL

TABLE 1. Device-specific interpretations of the types of user interaction

Type	Properties	X-windows Interface	DOS Interface	Telephone Interface
Text input	Allows user to enter a string	Text Input Widget	String printf and scanf, or text input fields	Touch-tone key encoding (or speech recognition)
Command	UI Logic returns to application when user chooses a command	Command Buttons in Xview	Initial character represents the command	Use dial buttons (or speech recognition) to choose command
Menu	A group of commands for user to execute	Menu in Xview	Initial character as command	Use dial buttons (or speech recognition) to choose command
One-from-many input	User can choose only one option. Options may change dynamically	Radio Button in Xview for short list, scroll list for long list	Radio Buttons on an ASCII screen for short list, scroll list for long; highlight chosen option	Use dial buttons (or speech recognition) to choose option
Multiple-options input	User can choose multiple options; options may change dynamically	Option Button in Xview for short list, scroll list for long list	Option Buttons for short list, scroll list for long list; highlight chosen options	Use dial buttons (or speech recognition) to choose option
Yes/No	User answer yes or no	Yes and No buttons	Y or N key input	2 special dial buttons
Bounded Input	User can only input a value within a range	Scroll Bar in Xview	Text input w/bound, or graphical scroll bar	Use dial buttons with "#" sign as "ENTER" (or speech recognition) with range-checking
Date/Time Input	Enter date/time; UI Logic check for validity of entry	Text Input, with arrows, or menus	Input fields with arrows, or menus	Use dial buttons to choose from a list (or speech recognition)
Simple Display	For display only	Text display in Xview	Text display	Speech Synthesis

renders inFOrms, it maps each different type of interaction item to the corresponding user interface feature available on that display device. For instance, UIL may use radio buttons to represent questions that present to users a list of options from which to choose one. The implementation of UIL for each display device is independent of which applications the UIL serves.

Grouping. Besides rendering, UIL is also responsible for managing the interaction process with the user. Applications may send multiple inFOrms together in a single interaction, with dependencies described among the various inFOrms. For example, a travel authorization application may send a top-level menu form with another travel request form to the user in one interaction. When the user chooses the travel request submission function in the top-level menu form, she is directly given the travel request form. Thus, the flow of interaction between the user and the application can be specified within the inFOrms. When UIL interprets inFOrms, it understands the flow of interaction and it guides users through the process.

Extensibility. When applications are being developed in DIANA, they make no distinctions regarding the particular display devices to be used. Therefore, when a new access device is used, all that needs to be done is to implement a UIL for this new device. After that, all applications can work with this device requiring neither modification nor recompilation. This portability is achieved by having the applications and UIL be separate entities that communicate using FDL, which is device independent.

3.3 Implementation

Broadly speaking, the structure of a UIL can be divided into 3 parts: a parser which parses inFOrms into internal structure, a renderer which maps the input items listed in Table 1 into the user interface available on the target display device, and a communication module which talks with User Courier. The parser and the communication module are the same for all instances of UIL. To implement a UIL for a new display device, the only implementation needed is of a new renderer for the user interface on that device.

4. The DIANA Network Architecture

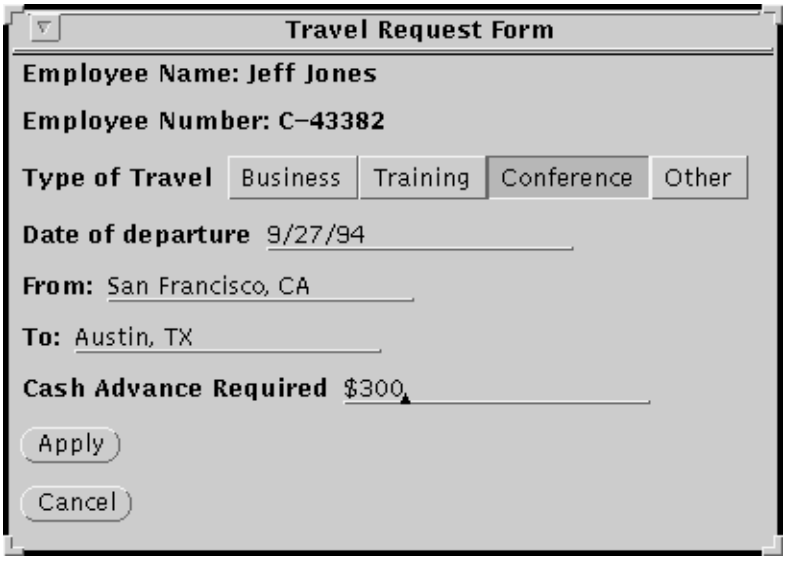
The network components of DIANA are the Couriers and the Replay Agents. Each Courier is responsible for the actual transmission of information across the network, while each Replay Agent deals with managing the inFOrm replies and passing them to the applications when desired. Before delving into the architectural details, we will explain our approach of hiding the network details from the applications.

A significant part of the design and development cycles of many existing distributed applications is spent on dealing with networks and communications. The network logic is deeply embedded in these applications, which incurs significant costs for maintenance, upgradability and porting of these applications to other user platforms.

For example, in a client-server model, the basic client is designed to work with particular network protocols, such as TCP/IP, UDP, and so on. Even though the flexibility of choosing a particular network protocol is critical for some applications, there are many applications that are not time critical and would benefit from independence from the heterogeneity of the networks. These are the applications that fit very well into our model. We must also mention that in the DIANA architecture, the performance penalty incurred by a reliable protocol is mitigated through caching to reduce network traffic. By grouping multiple one inFOrms in a single interaction, we further reduce network utilization.

Another reason behind our decision to de-couple communication logic from applications was the commonality of communication logic among a variety of applications. The DIANA architecture allows application developers to model their applications at a higher level — information exchange — and relieves them from having to deal with the intricacies of network communication.

FIGURE 2. A Travel Request inFOrm on an OpenLook Interface




```
BEGIN FORM newRequestForm
    HEADING “Travel Request Form”
    BEGIN SIMPLEOUTPUT empName
        HEADING “Employee Name”
        DATA <TEXT> “Jeff Jones”
    END SIMPLEOUTPUT
    BEGIN SIMPLEOUTPUT empNum
        HEADING “Employee Number”
        DATA <TEXT> “C-43382”
    END SIMPLEOUTPUT
    BEGIN ONEFROMMANYFIXED travelType
        HEADING “Type of Travel”
        BEGIN OPTION business
            HEADING “Business”
        END OPTION
        BEGIN OPTION training
            HEADING “Training”
        END OPTION
        BEGIN OPTION seminar
            HEADING “Conference”
        END OPTION
        BEGIN OPTION other
            HEADING “Other”
        END OPTION
        DEFAULT business
    END ONEFROMMANYFIXED
    BEGIN SIMPLEINPUT date
        HEADING “Date of departure”
        TYPE <DATE>
    END SIMPLEINPUT
    BEGIN SIMPLEINPUT fromPlace
        HEADING “From:”
        TYPE <TEXT>
        DEFAULT <TEXT> “”
    END SIMPLEINPUT
    BEGIN SIMPLEINPUT toPlace
        HEADING “To:”
        TYPE <TEXT>
        DEFAULT <TEXT> “”
    END SIMPLEINPUT
    BEGIN SIMPLEINPUT cashAdv
        HEADING “Cash Advance Required”
        TYPE <MONEY>
        DEFAULT <MONEY> $0.00
    END SIMPLEINPUT
    BEGIN COMMAND apply
        HEADING “Apply”
    END COMMAND
    BEGIN COMMAND cancel
        HEADING “Cancel”
    END COMMAND
    BEGIN COMMAND quit
        HEADING “Quit”
    END COMMAND
END FORM
```

Our approach also takes into account mobile computing and intermittent connectivity. Many computer users are moving to smaller, portable devices. This migration has led to the evolution of a large variety of portable computing devices from notebooks to palmtops to PDA's. Some of these devices are connected to the network through wireless communications, such as wireless e-mail. Such connectivity is often asynchronous. Other mobile communication can support a continuous connection but with high latency. In this case, if responses do not arrive in a timely fashion, DIANA will fall back to the intermittently connected case transparent to the application. By hiding how the information was transmitted between the user device and the application, DIANA supports a variety of network protocols between nomadic devices and network resident applications.

4.1 Issues in Network Management

We begin the design discussion with some of the questions that we answered in the process of modeling the network components.

Naming. The first question we considered is how different entities in our system address each other. There are two type of entities that interact with DIANA ─ end user clients and applications (we will use the term "agent" unless we wish to distinguish between these two). These agents can appear at different places in the network.

Everything directed to a user is directed to the UIL component of DIANA residing on the user's device, and everything directed to an application is directed to the Application Replay Agent at the application's host machine. Also, all connections between two nodes are made between the User Courier and the Application Courier, which in turn forward the inFOrms to the appropriate agents. We use the existing *port@host* naming conventions by assigning well known ports to the DIANA components and using the Internet addressing mechanisms to identify the hosts (when the application and user device are directly connected over the Internet).

The entities interacting with DIANA can address a particular agent using a notation similar to *agent@host*. This information is stored in the meta-information headers of the inFOrms discussed later in this section. The Courier can look at the meta-information and route the forms locally to the appropriate DIANA component.

DIANA resolves the issue of the heterogeneity of application naming in the following manner. In DIANA, the sender application must obtain an inFOrm script from the inFOrm Repository before sending it to the destination. Applications can obtain only those inFOrms, which they are prepared to interpret and for which they have already registered the scripts with the Courier. These applications provide their local identifiers at the time of registration and these identifiers are stored in the meta-information of the inFOrms. Hence the Courier at the sender's machine resolves the *host* part of the address and the Courier at the receiver's machine resolves the *agent* part.

Delegation. In a corporate environment, the end users of a system perform a lot of routine work, such as generating weekly progress reports and scheduling meetings. In many cases, it is desirable to write an application that does some regular work on the users' behalf.

This notion requires that a system like DIANA provide mechanisms for sorting inFOrms according to *roles* or functionalities. The end users should be allowed to *delegate* inFOrms to other authorized agents to do some work on their behalf. This delegation is transparent to the agent sending the inFOrm.

We suggest that the end users provide delegation information to the Courier. The Courier goes through an extra indirection before determining the actual destination of an inFOrm. As an example, suppose that the corporate VP has an automatic meeting scheduler that sends inFOrms to other meeting attendees requesting that they confirm the schedule. The receivers of these inFOrms can either complete the inFOrms by themselves or delegate this job to their secretaries or their own meeting schedulers.

Ultimately, supporting a system based on <*agent, role*> pairs to resolve the delegation of inFOrms seems promising. This convention is very important from the point of view of corporate computing where different people can have different responsibilities under different roles. The details of such a system have been left for future consideration.

4.2 Meta-Information on InFOrms

Each inFOrm contains a meta-information header, which is designed so that different components of DIANA can monitor the flow of an inFOrm through the system. The source and destination addresses are included in the meta-information header. It is desirable to keep this header as small as possible. Table 2 presents some of the logical fields that are an essential part of the meta-information. This list is not complete; we expect it to evolve with time.

4.3 The User and Application Couriers

In DIANA, the Courier is divided into Application Courier and User Courier, which handle communication issues for application and users respectively. The communication between the Application Courier and the User Courier can happen in two basic modes: connected and disconnected.

The Courier components always attempt to establish a direct connection if possible. The Courier has the capability to run on top of multiple network protocols, such as TCP/IP and UDP, in connected mode. Once the connection is established, the Courier components exchange a series of commands as a result of which the desired action is performed, e.g., the inFOrm is delivered to the UIL from the application, or an inFOrm reply is sent from the UIL to the Application Replay agent.

In the disconnected mode, the Courier currently uses e-mail as the medium of communication. The e-mail messages contain special subject headers and can be filtered and passed onto the corresponding Courier component at the destination. Time-critical applications which rely on the network for a guaranteed minimum transfer rate are not compatible with this mode of communication. However, other protocols for disconnected or intermittently connected operation and may be implemented in the DIANA architecture without change to applications or other user device software. The Courier provides a mechanism to determine the communication mode (i.e., connectedness) at any particular time.

4.4 Application Surrogates

It is important to explore how much useful work can be done on the user's access device while the accessor is disconnected from the network, or when the system is operating in the asynchronous mode.

The introduction of an application surrogate to deal with the disconnectivity logic of the user interaction is presented as an attempt to address this problem. A careful study of the behavior of these surrogates will reveal the benefits of this technique. By caching the inFOrms required for an application and having a

TABLE 2. The Meta-Information Headers for FDL

Name	Function	Agent Access
TransferID	identifier specifying the flow of a particular inFOrm	read only
Source	address of the sender	read only
Destination	address of the receiver	read/write
Delegated to	address of the actual destination	read only
Time	time fields to monitor flow intervals	read only
Priority	urgent/normal/... to determine how time critical this transfer is	read/write

surrogate for that application, the users will be able to do useful work while disconnected. The Coda File System relies on caching of objects (including the applications which users are expected to use during the disconnected period) to support disconnected operation [Satya93]. We expect that an application surrogate-based approach will be able to use the limited resources on the nomadic devices in a more efficient manner. Also, the use of application surrogates facilitates support for multiple platforms as only the surrogates will have to be implemented on each user platform, whereas the bulk of the application logic will be network resident and most of the user interface will be handled by the UIL. A scripting language, such as TCL, appears useful for constructing Application Surrogates.

Ideally, we would like to be able to generalize the functionality of the application surrogates and include them in the UIL. Whether the applications should be aware of the presence of application surrogates on the remote devices also remains to be answered.

4.5 Implementation

The Application and the User Courier consist of 2 parts: a communication manager which handles both synchronous and asynchronous communication modes, and a connection table which records the information on the pairs of communicating parties. At this stage, the Replay Agent simply maintains a first-in-first-out queue of reply forms. In subsequent work, we will explore the operation of the Replay Agent and its use in disconnected operation.

5. Application development Methodology

Application development using DIANA framework differs from conventional application development approach. In conventional application development, the developer first designs the functionality and infrastructure of the applications. Then she works out the finer details as well as the interface to the external world. But in the DIANA approach, when the developer designs an application, she first designs the application's interaction with users in terms of inFOrms. These inFOrms define inputs requested from users as well as the application's responses to those users.

After defining inFOrms, the developer can then design the structure of the application and code it. When coding the application, the developer should assume only an asynchronous communication channel with the users, even though the Courier might establish a synchronous channel with the user.

This approach of first defining the interface with the users before coding allows the developer to mock-up the application and test the correctness of the interface independent of the functionality of the application. Testing can be carried out at an earlier stage so that some design flaws can be detected earlier.

5.1 Benefits of DIANA to Application Developers

The goal of DIANA is to achieve display independence and communication independence in software development. These are achieved by the separation of display logic and communication logic from applications. When communicating with human users or other applications using DIANA, applications do not see user interfaces and communication channels. Instead, they merely send to the Courier inFOrms, which contain the information being exchanged, and the Courier will take care of the delivery of the inFOrms. In summary, DIANA provides an abstract message exchange mechanism (the Courier) with a universal language for describing the messages (FDL).

There are significant advantages to using DIANA in application development. First, by separating display logic and communication logic from applications, developers can be freed from handling display and communication issues and can concentrate their efforts instead on the design and coding of the application logic. Thus, application development is speeded up and the quality is enhanced.

Second, with the provision of these display and communication facilities, even novice programmers can

develop sophisticated applications with GUI and communication. Without DIANA, the development of these applications will require knowledge of multiple user interfaces and communication mechanisms to have the same flexibility.

Third, DIANA provides application portability across heterogeneous display devices. While a number of portable GUI facilities currently available also provide portability, they limit the usability of applications to bitmap-based and character-based display devices. These portable GUI facilities focus on the presentation format of information exchanged between human users and applications, and enforce strict representation of such information using graphical objects like buttons and scroll bars. On the other hand, DIANA focuses on the semantics of information exchanged but does not presume its representation. Therefore, it is able to support a greater variety of devices with a varied level of display capabilities.

Note that it is easy for DIANA to support future display devices that are presently unforeseen, as DIANA focuses on the semantics of information being exchanged. Once a new device is supported, all applications using DIANA can work with that new device, once the common UIL, User Courier, etc., are developed. Application Surrogates are not required for initial use of the new device.

As FDL focuses on the semantics of information being exchanged, it serves as a universal language for applications. Applications can use FDL to talk with one another. The consequence is that FDL provide a common ground for applications to communicate and collaborate. FDL can be used as a communication language for distributed applications. Applications share a common protocol, namely inFOrms, and thus can achieve a greater degree of cooperation not achievable without a common protocol.

One of DIANA's main goals is to facilitate mobile computing, where users have only intermittent communication channels with network-resident applications. The asynchronous communication mechanism provided by DIANA favors such intermittent connectivity. Consider the situation when a mobile user has only 5 minutes to connect to a host application but the whole interaction will take 10 minutes. Without DIANA, the user will be unable to continue work after the first 5 minutes. However, with DIANA, the user can use the first 5 minutes to send an inFOrm to request some job to be done by the host application. He can then check the results of his work in another inFOrm at a later time.

DIANA supports disconnected operation. When users are disconnected from host applications, they can still work with the inFOrms stored on their client devices. There are agents which will record their operations and replay the whole operations to the applications automatically once the users are connected again.

Finally, DIANA minimizes network traffic by using inFOrms to exchange information. The inFOrms do not contain extra data on how to represent the information. Without DIANA, applications will have to send extra detail to a mobile display device, for example, to render the windows and buttons. The saving in communication traffic is especially important to mobile computing as the communication channels have limited bandwidth.

5.2 DIANA's Application Programming Interface

From an application's perspective, the whole processing of communication with the users can be divided into a series of steps. First, the application prepares the inFOrm it wants to send to the user. At this point, the application can update the inFOrm by adding a dynamic data part to the inFOrm. Then it sends the inFOrm to the user. After that, the application can continue with other operations and asynchronously wait for the reply form from the user. Alternatively, the application can choose to wait synchronously. After receiving the reply form, it can access the user's replies from the reply form.

Table 3 describes the API provided by DIANA to do the above actions.

TABLE 3. The description of the DIANA API

Call	Description
initDiana()	initializes DIANA and should be called before any DIANA facilities are used.
prepareForm()	reads in an inFOrm given its name. The prepareForm() procedure should be called first before any dynamic data can be apply to the inFOrm and the inFOrm can be sent.
getItem()	gets a pointer to an item within another item, given a string specifying the item from the other item. This procedure is useful for navigating through an inFOrm.
setAttr()	sets the attribute of an item. Various attributes of an item can be set, for example, the name, the heading, the options, the default value. The setAttr() procedure is particularly useful for setting the dynamic data part of an inFOrm before the inFOrm is sent.
sendForm()	sends an inFOrm to an agent, either a human user or an application.
getReplyForm()	waits for a reply from the agent. Applications can specify whether the waiting should be blocking or non-blocking. A handle to the reply form is returned, which the application uses to access the user's reply.
getAttr()	gets a particular field inside the reply inFOrm given the field name. The return type is a pointer to void, which the application should cast to the proper pointer type before accessing the value.

Readers should notice that the list of API calls exported by DIANA is simple and small. This simplicity compares favorably with the complexity of the API required to for applications to handle GUI and communication directly. This simplicity reflects the design goal of DIANA — to provide application developers with a simple and uniform interface to a great variety of access devices.

5.3 Procedural vs. Object-Oriented Interface

Currently, DIANA uses a conventional procedural programming approach in both the view exported to application developers and its implementation. Alternatively, an object-oriented approach can better model the whole information exchange mechanism tackled by DIANA. It is possible to develop an object hierarchy to represent the interaction items and inFOrms; however, recompilation of application programs might be required if the object hierarchy is changed to encompass new types of data potentially being transmitted. The overloading feature of some object-oriented programming languages like C++ avoid the typing problem in the return value of the getAttr() procedure mentioned above. Based on the type of interaction items passed as the parameter, the right type of return value can be returned.

6. Current Implementation

At the current stage, we have implemented only a portion of the whole DIANA framework. We have implemented a UIL for OpenLook environment. We have also implemented another UIL for the telephone, which is simulated by a software whose interface consists of 12 buttons mimicking the dial buttons of a telephone for input and dialogs are spoken out by a speech synthesizer for output. The Courier currently supports only TCP/IP synchronous communication. The Replay Agent is a simple first-in-first-out queue of reply forms.

To test and verify DIANA, we have also implemented a travel authorization application which allows users to submit travel requests, review request status and approve requests. This application has successfully worked with both OpenLook and telephone interface without special adaptation to either display device. The OpenLook interface is a good representative interface in the workplace while the telephone interface is common in a mobile environment. Though tested in a simulated environment, we believe that the idea of display independence for mobile computing has been successfully demonstrated in this exercise.

We have implemented and tested disconnected operation, but the results of this experiment are beyond the scope of this paper.

7. Future Issues

This section discusses some of the outstanding issues we have not yet addressed.

7.1 Workflow Support

The DIANA infrastructure has important features to support business workflow processing. We feel that one dimension of the evolution of DIANA can lead us into studying workflow systems and incorporating basic features of workflow systems into DIANA as default services provided by the Courier. e.g. the Courier already has the notion of delegation based on different roles of end users (see section 4.1).

The device and network independence supported by DIANA is essential for workflow systems where a variety of heterogeneous systems need to interact with each other.

The universal FDL on the other hand can make all the applications understand each other. Legacy applications can be incorporated into a workflow system by providing a single DIANA client which can translate between the FDL and the applications input and output parameters.

However, minimal work has been done in this direction so far and further study of workflow systems and their overlap with DIANA is essential.

7.2 Disconnectivity Logic

It is important to explore how much useful work can be done on the user's access device while the accessor is disconnected from the network, or when the system is operating in the asynchronous mode.

The introduction of an application surrogate to deal with the disconnectivity logic of the user interaction is presented as an attempt to address this problem. A careful study of the behavior of these surrogates will reveal the benefits of this technique. By caching the inFOrms required for an application and having a surrogate for that application, the users will be able to do useful work while disconnected. The Coda File System relies on caching of objects (including the applications which users are expected to use during the disconnected period) to support disconnected operation [Satya93]. We expect that an application surrogate-based approach will be able to use the limited resources on the nomadic devices in a more efficient manner. Also, this approach seems more suitable to support multiple platforms as only the surrogates will have to be implemented across platforms, whereas the bulk of the application logic will be network resident and most of the user interaction will be handled by the UIL.

Ideally, we would like to be able to generalize the functionality of the application surrogates and include them in the UIL. Whether the applications should be aware of the presence of application surrogates on the remote devices also remains to be answered.

7.3 Replaying Disconnectivity Interaction

A simple scheme can be adapted to record interaction occurring between the application surrogate and the end user while the network is down. This logged interaction can be "replayed" to the host application after the network connection is re-established or when this interaction is forwarded through e-mail.

The user may complete several inFOrms on a disconnected device before they can be transmitted to the application. In this case, the inFOrms need to be given to the application in the order that the application expects them. It is possible that an exception arises, so that the application next requests information not contained in the inFOrms already completed. In particular, the Application Surrogate has made a different decision from the actual Application due to a lack of information. The User Replay and Application Replay agents will allow the additional information to be supplied and the communication to be resynchronized. If necessary, the inFOrms already

completed may be edited. We call this synchronization process *zippering*. We have implemented zippering, but the results of this experiment are beyond the scope of this paper. The goal of zippering is for the application not to be able to detect such inconsistencies, particularly when it does not even know about the existence of application surrogates.

7.4 Customizing Interfaces

An interesting problem is to let the applications customize their interface, but without re-introducing the device-dependence problem.

According to application developers, 70% to 95% of their efforts are concentrated on developing and customizing the look and feel of their applications. This statistic implies that DIANA must address the interface layout and customization issue. One approach can be that the UIL provide a big percentage of the above functionality and let the developers write their own surrogates to customize the remaining part as desirable. In particular, the UIL could utilize user and application profiles to support user-, device-, and application-specific customizations.

Extending FDL to include some layout hints about the grouping of several items on an inFOrm is also possible.

8. Concluding Remarks

DIANA addresses the issues of platform dependence, network binding, and interoperability by proposing a simple, semantic-based system. DIANA will eliminate the effort required to implement application-specific user interfaces and communication modules, thus significantly reducing the design and implementation cycle of applications. Applications can be implemented and tested on a simple text interface and then plugged into the DIANA interface. Using a complete and reliable implementation of DIANA, the application developers will be able to concentrate on testing their applications' processing logic without having to worry about the intricacies in user interface code or communications code. Also, the design of the user interface can proceed in parallel with core application development and can easily be changed to adapt to any new requirements placed on the application. We believe that the ideas introduced in this paper will address some of the drawbacks in current workflow systems, as well as enhance interoperability between incompatible systems and increase the customer base of applications.

A current prototype of the system exists which supports the inFOrm life cycle. The network component supports direct connectivity using a TCP/IP connection and a UIL for the OpenLook environment exists. We have developed a telephone interface to FDL. We have also experimented with disconnected operation and will describe that experiment in future writing. We expect that many of these ideas will be very important in the world of mobile computing and heterogeneous systems in the near future.

9. Acknowledgments

This research was funded in part by Sun Microsystems. We acknowledge the support and encouragement of Terry Keeley, Bert Sutherland, and Emil Sarpa.

We thank Bob Sproull, and Xinhua Zhao for their thoughtful comments. We thank Marianne Siroker for her help in preparing this paper.

10.Bibliography

[Bass90] Bass, L., Clapper, B., Hardy, E., Kazman, R. and Seacord, R., "Serpent: A User Interface Management System," *Proceedings of the Winter 1990 USENIX Conference*, Berkeley, CA, January 1990, 245-248.

[Bennet90] Bennet, A., "FormIKA: A Form-Based User Interface Management System for Knowledge Acquisition," Knowledge Systems Laboratory, Stanford University, June 1990.

[Caruso93] Caruso, D., "General Magic Got Quite a Start," *Digital Media*, March 29, 1993.

[Coutaz87] Coutaz, J., "PAC, An Implementation Model for Dialog Design," *Proceedings of Interact '87*, Stuttgart, September, 1987, 431-436.

[de Souza92] de Souza C. S., "A Semiotic Approach to User Interface Language Design," Center for the Study of Language and Information, Stanford University, May 1992.

[Gosling95] Gosling, J., and McGilton, H., "The Java Language Environment, A White Paper," Sun Microsystems Computer Company, May 1995.

[Guarna88] Guarna, V. A., Jr. and Gaur Y., "A Portable User Interface for a Scientific Programming Environment," Center for Supercomputing Research and Development, University of Illinois, Urbana-Champaign, February 1988.

[Imielinski93] Imielinski, T. and Badrinath, B.R., "Mobile Wireless Computing: Solutions and Challenges in Data Management," Department of Computer Science, Rutgers University, 1993.

[Kazman93] Kazman, R., Bass, L., Abowd, G. and Webb, M., "Analyzing the Properties of User Interface Software," Department of Computer Science, Carnegie-Mellon University, October 1993.

[McBryan91] McBryan, O. A., "Software Issues at the User Interface," Department of Computer Science, University of Colorado, May 1991.

[Myers92] Myers, B. A. and Rosson, M. B., "Survey on User Interface Programming," Department of Computer Science, Carnegie-Mellon University, February 1992.

[Myers93] Myers, B. A., "Why are Human-Computer Interfaces Difficult to Design and Implement?" Department of Computer Science, Carnegie-Mellon University, July 1993.

[Pfaff85] Pfaff, G., (ed.), *User Interface Management Systems*, Newyork: Springer-Verlag, 1985.

[Satya90a] Satyanarayanan, M. Kistler, J. J., Kumar, P., Okasaki, M. E., Siegel, E. H. and Steere, D. C., "Coda: A Highly Available File System for a Distributed Workstation Environment," *IEEE Transactions on Computers*, April 1990.

[Satya90b] Satyanarayanan, M., "Scalable, Secure, and Highly Available Distributed File Access," *IEEE Computer*, May 1990.

[Satya93] Satyanarayanan, M., Kistler, J. J., Mummert, L. B., Ebling, M. R., Kumar, P. and Lu, Q., "Experience with disconnected Operation in a Mobile Computing Environment," Department of Computer Science, Carnegie-Mellon University, June 1993.

[Steere90] Steere, D.C., Kistler, J. J. and Satyanarayanan, M., "Efficient User-Level Cache File Management on the Sun Vnode Interface," In *Summer Usenix Conference Proceedings*, Anaheim, June 1990.

[Tso93] Tso, M., "Using Propoerty Specifications to Achieve Graceful Disconnected Operation in an Intermittent Mobile Computing Environment," Xerox Corporation, Palo Alto Research Center, June 1993.

[Watanabe86] Watanabe, T. and Oketani, I., "Functional Design of Cooperatively Integrated Information System," Data Processing Center, Kyoto University, October 1986.

[Zarmer90] Zarmer, C. and Canning, P., "Using C++ to Implement an Advanced User-Interface Architecture," HP Laboratories, Hewlett Packard Company, April 1990.