Zippering: Managing Intermittent Connectivity in DIANA

Arthur M. Keller[1], Owen Densmore[2],
Wei Huang[3], and Behfar Razavi[2]

[1] *Stanford University, Computer Science Dept.,*
Gates Building 2A, Stanford, CA 94305-9020, USA
E-mail: ark@cs.stanford.edu

[2] *Sun Microsystems*

[3] *Work done while a student at Stanford University*

This paper describes an approach for handling intermittent connectivity between mobile clients and network-resident applications, which we call zippering. When the client connects with the application, communication between the client and the application is synchronous. When the client intermittently connects with the application, communication becomes asynchronous. The DIANA (Device-Independent, Asynchronous Network Access) approach allows the client to perform a variety of operations while disconnected. Finally, when the client reconnects with the application, the operations performed independently on the client are replayed to the application in the order they were originally done. Zippering allows the user at the client to fix errors detected during reconciliation and continues the transaction gracefully instead of aborting the whole transaction when errors are detected.

1 Introduction

1.1 The Problem

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.

With the advent of portable computers, mobile users want to access distributed information systems on the office network. But limited connectivity is a key problem that prevents the nomadic devices from accessing distributed information systems. Users migrate between periods of direct connectivity, inter-

mittent connectivity, and disconnectivity and have varying amounts of bandwidth available and have varying latencies. For example, users can be in a hotel room connected over a dialup phone line using a laptop, or at a customer meeting connected over a wireless network that can have dead zones, or be on an airplane creating requests to be transmitted later, or even overseas and responding to voice prompts using a touch tone telephone. Users wish to operate in this environment seamlessly using multiple and different access devices with differing connectivity environments.

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.

In earlier work [10], we focused on user interface issues and also considered communication issues. In this paper, we focus on varying connectivity. To address these two issues, we earlier proposed an application architecture for mobile applications. The DIANA architecture—Device-Independence, 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.

We now focus on the need for users to migrate seamlessly between periods of connectivity, intermittent connectivity, and disconnectivity. In particular, users must be able to continue operating during periods of disconnectivity and not have that work lost in case of minor errors. We describe the approach of zippering, a process of resynchronizing between the client system and network-based application once connectivity has been re-established. During zippering, operations performed during disconnectivity are replayed in sequence from the client to the network-based application. If a failure is detected during the replay process, the user at the client is given the opportunity to correct the error and to resume the replay process until reconciliation has been achieved.

The name zippering evokes the process of reconciliation. The two tapes of the zipper and their respective teeth correspond to the client and application operations. Where the teeth are already enmeshed, reconciliation has been achieved. The slide is at the point where resynchronization is currently occurring. Beyond the slide are operations performed during a period of disconnectivity. As the slide moves forward, the enmeshing of teeth represents the synchronization process. Occasionally, the zipper's slide may get stuck due to an error. The slide is re-

versed a short distance, the error is cleared, and the slide is pulled beyond the former failure. If a new failure to zip occurs, the process is repeated, until the zipper is completely closed. Once the zipper is closed, complete reconciliation has been achieved, and continued operation occurs synchronously.

1.2 Related Works

Considering display and network independence, we observe that Mosaic and Telescript [2] 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 DIANA's Form Description Language, 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. In Telescript, itinerant agents travel around interacting with the user and network-based components. All of the functionality of the application's interface with the user is be encoded in the agent. Telescript agents are designed specifically for the type of network connectivity expected. If disconnected support is expected, then the Telescript agent must be written to handle it. Failure handling, such as in our zippering paradigm, must be specifically written into Telescript agents. A simplistic comparison is that DIANA handles the user interface similar to Mosaic and the network interface similar to Telescript.

Kazman et al., [9] present an overview of some user interface architecture models. The Seeheim model [12], the Arch/Slinky Metamodel, the Presentation-

Abstraction-Control (PAC) model [3] and the Serpent model [1] 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 [11].

Imielinski and Badrinath [8] identify some of the issues involved in mobile wireless computing, including location management, configuration management, disconnection, 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 [15] 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 [13, 14, 16].

Java [7] 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 of Paper

In this paper, we describe the architecture and show the benefits of using the DIANA architecture in mobile computing. Section 2 describes an application example using the DIANA architecture. Section 3 gives an overview of the DIANA architecture. Section 4 shows the connected operation. Section 5 focuses on disconnected mode. Section 6 illustrates the design issues for Application Surrogates. Section 7 describes current state of the DIANA architecture. Section 8 discusses future issues about versatile connectivity. We conclude with section 9.

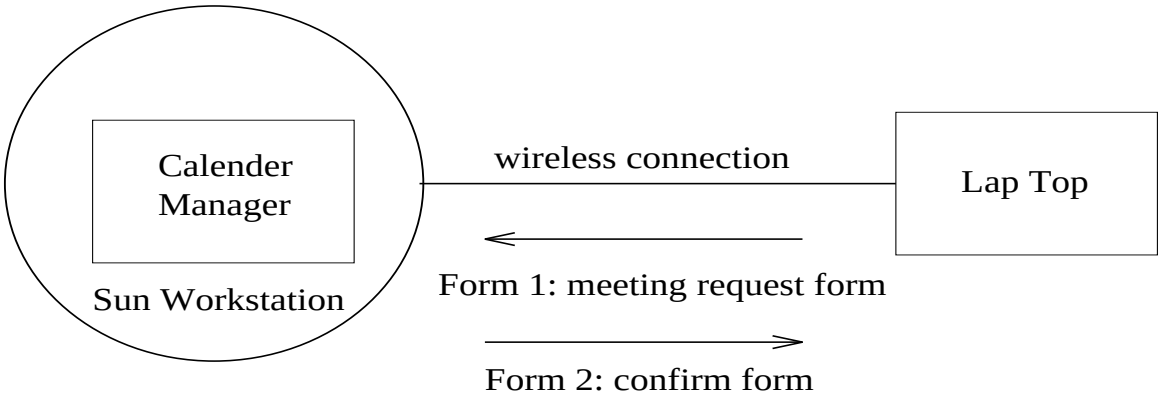

Figure 1: Connected Operation

2 Scenario

Jeff Jones is a sales representative of ABC corporation. Due to the nature of his job, he often travels to meet prospective customers. We will consider the Calendar Manager, an application he uses to make appointments or schedule meetings. This distributed application allows him to maintain his own schedule and to look at the schedules of his partners.

2.1 Connected Operation

When he leaves his office and travels to meet his customer XYZ, he also leaves his network-connected Sun workstation. From his laptop, he can still access the Calendar Manager application running on the Sun Workstation in his office by using a wireless network.

For example, he could set up a meeting with his manager, Sally Smith. Jeff sends a meeting request form synchronously to the Calendar Manager for a meeting with Sally at, say, 2 pm on Tuesday, October 24. Afterwards, Jeff will get a confirmation reply from the Calendar Manager about the meeting with Sally.

2.2 Disconnected Operation

Jeff visits the customer XYZ at a city without a wireless network. He finds himself operating in disconnected mode. Jeff finds that XYZ is very interested in his product and suggests that they could meet in ABC corporation at 2 pm on Wednesday, October 25. In the disconnected environment, Jeff still can check his schedule by using the Calendar Manager Surrogate on his laptop. Jeff fills the time schedule form and requests his favorite meeting and submits the form. The Calendar Manager Surrogate responds that the time is ok and believes the

meeting room available at that time. The form is held for transmission to the network-resident Calendar Manager.

2.3 Zippering

Jeff returns to his hotel room, and connects using a dialup modem over a phone line. His laptop is connected with the Calendar Manager application again. The Calendar Manager application gets the time schedule form and approves the request. But it denies the room request because the room has been reserved in the interim. This response is transmitted to Jeff's laptop. Jeff now has two choices. He may either abort the whole transaction or modify the meeting room to complete the transaction. Jeff chooses an alternative meeting room, which the Calendar Manager approves.

3 DIANA Architecture Overview

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.

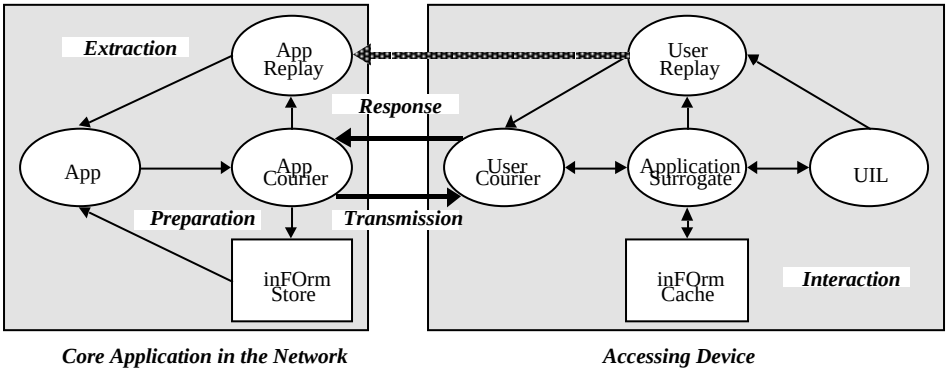

Figure 2: DIANA: Overall Architecture

3.1 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 com-

ponent 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 Personal Digital Assistant (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 be display independent [10].

We have developed UILs that support X-windows displays, PCs, and a simulated touch-tone telephone. Our experiments have shown that it is possible to communicate common data over a variety of interfaces. Supporting a variety of interfaces puts limitations on the type of information that may be communicated. However, since our interfaces so far have been limited to text and numbers, we have been successful in accessing our applications using these various user devices.

3.2 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 detail in Section 5.

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.

3.3 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.

3.4 the Application Surrogate

In disconnected mode, the user courier starts the application surrogate, which is an application-specific agent on the client. After initializing the application surrogate, the user courier re-synchronizes with the application surrogate. Afterwards, the application surrogate will continue the interrupted transaction. The application surrogate is an application-specific agent, which makes decision using the limited state available at the client.

3.5 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 disconnection, then such processing can be embodied in an Application Surrogate that resides on each client device. We will discuss the application surrogate in Section 5.1.

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.

4 Connected Operation

In this section, we will describe each stage in connected operation. The connected operation uses TCP/IP as the synchronous communication protocol. The couriers are used as communication agents in both client and server sides. The couriers use synchronous communication methods by default. If a disconnect occurs, the couriers switch to asynchronous communication mode automatically. Disconnected operation will be described in the next section.

The following subsection illustrates the process and sequence of connected

TransID	User Courier netaddress	Application netaddress

Figure 3: An entry in the transaction table

operation.

4.1 Registration

At first, the user requests the user courier service to start a specific application. The user courier registers this user information, including the user communication address and application name. Afterward, the user courier sets up a TCP/IP connection with the application courier.

4.2 Application Start

After it receives the application name from the user courier, the application courier finds the application and starts it. The application courier also creates the current transaction identification, as TransID. TransID is globally unique number, which includes a version number. The version number begins with 1 and increments by one in each disconnected operation. The application courier registers the following information in its transaction table.

```
TransID: global transaction identification.
User Courier netaddress: network address of user courier
Application netaddress: network address of application
```

4.3 Application Operation

After initialization, the application sends the first form to the application courier. The application courier forwards the form to the requesting user courier. The user courier passes the form to the UIL for interpretation. UIL displays the form on the accessing (i.e., client) device.

The user interacts with UIL and finishes the form. The response form is given to the user courier, which sends the reply to the application courier, which in turn forwards the response form to the application. The application, which has been waiting for the response, now gets the user's input and sends out the next form.

4.4 Application Interaction Continues

The application's form and client's response form are exchanged between the application and accessing device. Because of the long latency, form interaction is

quite slow. In order to reduce latency and limit the need for bandwidth, FDL can support nested forms and the architecture supports a form cache.

A nested form allows each form entry to refer to several subforms. UIL searches the subform in the form cache of the client side. If the form is stored locally, there is no need to request the form from the application.

4.5 Application Completion

After it receives its last replied form, the application decides whether it can commit or needs to abort the transaction. The result of each transaction is sent to the user directly and the connection between the application courier and user courier is dismissed.

5 Disconnected Operation

Disconnected operation can happen at any time during a transaction. If disconnected operation starts at the beginning of a transaction, we say that the client's transaction is totally independent. If disconnection occurs later, the transaction is partially independent.

Both partially and totally independent transactions need zippering to synchronize the client operation with the application. But there are still some differences between them. The totally independent transaction does not have any corresponding application running on the application side and the application courier knows nothing about this transaction.

We will illustrate using an example of disconnection in the middle of a transaction.

5.1 Application Surrogate

In the application continuation stage, the connection may be interrupted. The user courier detects the disconnection and switches to asynchronous mode. In addition, the user courier looks for an application surrogate for the current application in the the application surrogate cache. If one is found, the user courier starts the application surrogate. Now the user courier will replay all the forms completed before the disconnected operation to the application surrogate to bring it up to date. After it synchronizes with the current interrupted transaction, the application surrogate continues the transaction. This replay and synchronization stage is omitted for the totally independent transaction.

If the user courier can not find an application surrogate, the user may proceed with a sequence of cached forms. For partially independent transactions, the next form may be specified by the previous form, and so forth. For totally independent

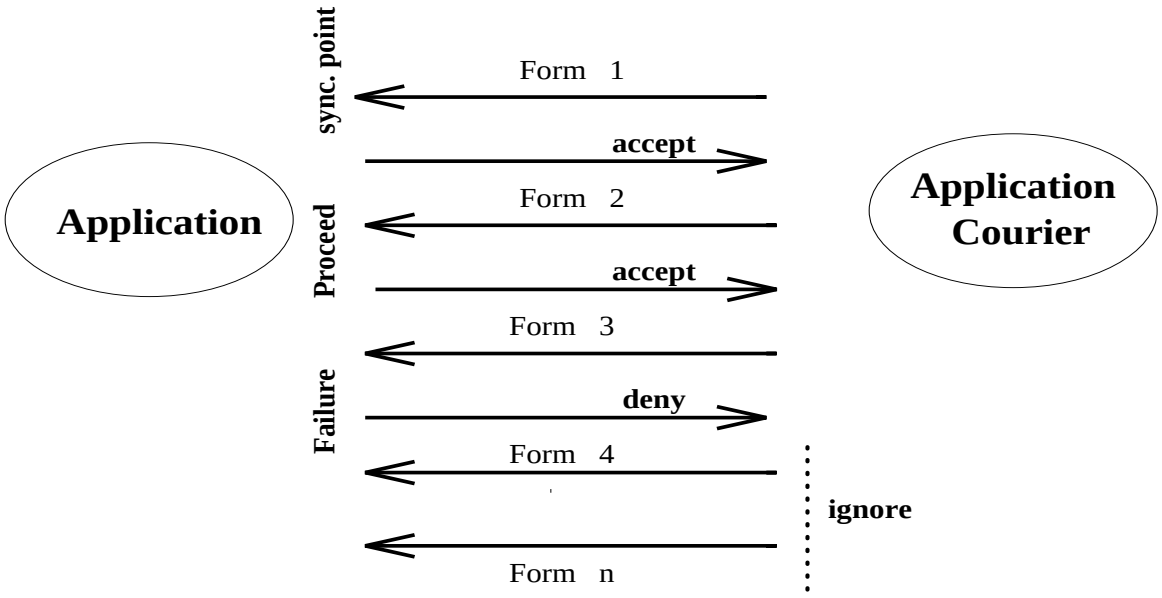

Figure 4: Zippering

transactions, the user must specify the first form to be completed. Otherwise, the transaction must wait for connected operation.

After the application completes the interaction with the user courier, the application surrogate pauses the transaction. The user courier buffers the exchanged forms and sends these forms to the application courier when the client is reconnected to the network. After the application courier receives these forms, it forwards them to the application. The whole disconnected operation is transparent to the application. Zippering begins after the reconnection.

5.2 Synchronization point

At the synchronization point, the queued up reponse forms are transmitted to the application courier. Then zippering begins at the application side. The application does not need to know anything about disconnection or the synchronization point. The application has merely been waiting, potentially a long time, for the remaining forms. Thus the application is given the first queued up form from the application courier and proceeds as in the connected case.

5.3 Proceed

The application repeatedly sends forms to the application courier and gets the response forms queued for it—the "proceed" stage—until it detects an error. Then zippering continues with failure detection, and then failure handling. Once the

failure is handled, the application re-enters the "proceed" stage.

5.4 Failure Detection

Zippering may fail in the proceed stage. Typically, an error is caused by the wrong decisions of the application surrogate. For example, the application surrogate may have been unable to verify that some input was correct. Some correction must now be done, or the transaction may be aborted.

After the failure is detected, the application tells the application courier about the error, and the application courier notifies this error to the user courier. The user courier acknowledges that the error comes from the application surrogate.

5.5 Failure Handling

The user courier presents to the user the original form with the applications' error message. The user courier then asks the user to choose how to continue. The user can either continue the transaction from the failing point by correcting user input or request aborting the whole transaction.

If the user chooses to continue, the user courier sends the incorrect form to the user via the UIL and asks the user to change the form. The response form is then sent to the application. After the failure has been handled, the "proceed" stage is entered again. Other response forms that were to follow the incorrect form are now replayed to the application until the next failure happens. In summary, there is a cycle of three stages: Proceed–Failure Detection–Failure Handling.

A simple example of such a failure handling is when a misspelled password is typed during a totally independent transaction with the Calendar Manager application. Subsequent to this incorrect password form, may be a series of valid forms requesting appointments and rooms. With zippering, the user may retype the password correctly, and the other forms are replayed in order. Without zippering, those other forms would have to be re-entered.

A user interface option we considered but did not implement is to allow the user to scroll through the various forms, from those completed, to the one where the error was detected, to the forms not yet processed by the application. We did not formally evaluate the user interface aspects of zippering.

5.6 Zippering Success

If the application completes the transaction without errors arising from disconnectivity, the application may commit or abort as it chooses. The user courier and application courier dismiss the connection. The application courier removes the current transaction entry from the transaction table.

5.7 Totally Independent Transactions

The totally independent transaction does not get a transID at the application courier side until connectivity is achieved. A new transaction from the perspective of the application courier must be initiated at the zippering start point. After the application courier receives the request from the user courier, the application courier starts the application and allocates transID for this application and then forwards the forms to the application. Zippering may happen afterward as in the partially independent transaction.

In summary, zippering consists of 4 parts: synchronization point, proceed, failure detection, and failure handling.

6 Application Surrogate

This section describes how to write an application surrogate and what functionality needs to be implemented in the application surrogate.

The application surrogate operates on behalf of the application during disconnected mode. Unfortunately, the application surrogate necessarily cannot have access to the totality of the application's state, and it must therefore make educated guesses of what the application would do in the situation. Actually, the application surrogate need only emulate the application by predicting which form the application would request the user to fill out next.

Application surrogates could be written in Tcl or some other scripting languages. Tcl works on most platform machines and has no dependence on operating systems and machine architectures. Furthermore, no compiling is necessary for Tcl and some other scripting languages. Therefore, migrating these codes to other hardware is easier than migrating a programming language like C.

Because of its limited function, no complicated programming techniques are required in writing the application surrogate. The application surrogate merely predicts the behavior of the application from its limited resource on the client side. In most cases, the application surrogate can specify the same input sequence as the application. Therefore, the transaction can be completed. Even when the application surrogate operates incorrectly, by failing to catch an error or by choosing the wrong form to fill out, zippering can often fix the error and continue the transaction.

The application surrogate allows the user to continue working in disconnected mode. It relaxes consistency in the disconnected environment in order to provide availability. Because the application makes the ultimate decisions about the transaction, consistency is achieved for completed transactions.

7 Current Status

We have completed our implementation of the whole DIANA framework. We have implemented a UIL for both OpenLook GUI and Motif GUI. The user courier and the application courier have been implemented and they can support both synchronous and asynchronous communication.

We have previously tested the display independence part of DIANA architecture. UIL for phone interface is written for Microsoft Windows 3.1. This UIL has the same functions as GUI UIL, but it "speaks" the form and asks for the user reply from a simulated touch tone phone.

To test and verify DIANA, we have also implemented a simple travel authorization application which allows users to submit travel requests, review request status and approve requests. This application has successfully worked with both GUI interface and phone interface. The forms included a user login and password form, a main travel request form, and forms for hotel, air, and ground transportation requests. Typical errors include incorrect user name or password, wrong city names, mistyped dates, and dollar amounts that are over limits. We did not experiment with zippering using the travel authorization application.

Furthermore, we have implemented a simplified Calendar Manager using the DIANA architecture. It is a distributed information system that allows user to manages their own schedules and to check their colleagues' schedules. Our Calendar Manager supports disconnected operation. The application surrogate for the Calendar is easily written in Tcl. We demonstrated zippering in the Calendar Manager. The primary error is reserving a room or timeslot on someone's calendar that is already taken. This application allows us to test zippering and demonstrate an architecture that supports seamless operation in both disconnected operation and connected operation.

8 Future Work

While the DIANA project has ended, further work is possible. When the DIANA project started, device independence had not also been demonstrated by Mosaic. It may be interesting to attempt to combine these ideas. DIANA is a stateful network framework that could handle both different kinds of display devices and disconnected operation. WWW browsers, such as Mosaic, are stateless and require internet connectivity. A natural question is how to use a WWW browser in the DIANA framework. DIANA could either translate between FDL and HTML or adopt some extension to HTML in place of FDL. In addition, the issues of form management and application surrogates are still needed during periods of disconnection.

9 Concluding Remarks

The DIANA approach addresses a key problem for Personal Communication Systems (PCS): the communication problems that face mobile users using computing applications especially those with a client-server oriented model. With the long latency and low bandwidth connection, the DIANA approach exchanges forms instead of character-at-a-time echoing. Form exchanges involve the batching of transmission and caching of forms to hide the latency and increase the granularity of client-to-application interaction. Couriers at both client and server side are responsible for communication. They can support both synchronous and asynchronous communication automatically. Communication states can be easily maintained in the couriers and be transparent to the application and user.

To support intermittent connectivity, user courier can run the proper application surrogate to provide application functionality not encoded in the forms. The application surrogate interacts with the user in the place of the network-based application. The response forms are sent to the application courier asynchronously. The application courier uses zippering to implement the final reconciliation with the application.

Zippering consists of 4 parts: synchronization point, proceed, failure detection, and failure handling.

Zippering works well in environments where errors during disconnected operation are relatively infrequent. In extended periods of disconnection, when conflicts are frequent or when subsequent forms depend highly on earlier forms, zippering may not be as useful. In this paper, zippering is used in the context of form-based interaction between users and applications. The World Wide Web illustrates the flexibility and limitations of form-based interfaces. An interesting challenge is to determine how to use to use the zippering paradigm in other kinds of user interaction with relatively large granularity.

10 Acknowledgements

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 on our work. Tahir Ahmad, Mike Clary, Steve Gadol, and Robert Pang previously worked on this project.

References

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

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

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

[4] General Magic, Inc., "The TDE User Guide," Version 1.0 Alpha, October 1995, from URL `http://cnn.genmagic.com/Telescript/TDE/TDEDOCS_HTML/userGuide.html`.

[5] General Magic, Inc., "The Telescript Language Reference," Version 1.0 Alpha, from the URL `http://cnn.genmagic.com/Telescript/TDE/TDEDOCS_HTML/telescript.html`.

[6] General Magic, Inc., "The Telescript Programming Guide," Version 1.0 Alpha, from the URL `http://cnn.genmagic.com/Telescript/TDE/TDEDOCS_HTML/developer.html`.

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

[8] T. Imielinski and B.R. Badrinath, "Mobile Wireless Computing: Solutions and Challenges in Data Management," Department of Computer Science, Rutgers University, Technical Report DCS-TR-296, 1993.

[9] R. Kazman, L. Bass, G. Abowd, and M. Webb, "Analyzing the Properties of User Interface Software," Department of Computer Science, Carnegie-Mellon University, Technical Report CMU-CS-93-201, October 1993.

[10] A.M. Keller, T. Ahmad, M. Clary, O. Densmore, S. Gadol, W. Huang, B. Razavi, and R. Pang, "The DIANA Approach to Mobile Computing," in *Mobile Computing*, Tomasz Imielinski and Henry F. Korth, eds., Kluwer Academic Press, fall 1995, pp. 651–679.

[11] B.A. Myers, "Why are Human-Computer Interfaces Difficult to Design and Implement?" Department of Computer Science, Carnegie-Mellon University, Technical Report CMU-CS-93-183, July 1993.

[12] G. Pfaff (ed.), User Interface Management Systems, New York: Springer-Verlag, 1985.

[13] M. Satyanarayanan, J.J. Kistler, P. Kumar, M.E. Okasaki, E.H. Siegel, and D.C. Steere, "Coda: A Highly Available File System for a Distributed Workstation Environment," IEEE Transactions on Computers **39**:4, April 1990, pp. 447–459.

[14] M. Satyanarayanan, "Scalable, Secure, and Highly Available Distributed File Access," IEEE Computer **23**:5, May 1990, pp. 9–18, 20–21.

[15] M. Satyanarayanan, J.J. Kistler, L.B. Mummert, M.R. Ebling, P. Kumar, and Q. Lu, "Experience with disconnected Operation in a Mobile Computing Environment," in Proc. USENIX Mobile and Location-Independent Computing Symposium, Cambridge, MA, August 1993, pp. 11–28.

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