Mobile User Recovery in the Context of Internet Transactions

Debra VanderMeer, *Student Member*, *IEEE*, Anindya Datta, *Member*, *IEEE Computer Society*, Kaushik Dutta, *Student Member*, *IEEE Computer Society*, Krithi Ramamritham, *Fellow*, *IEEE*, and Shamkant B. Navathe, *Member*, *IEEE*

Abstract—With the expansion of Web sites to include business functions, a user interfaces with e-businesses through an interactive and multistep process, which is often time-consuming. For mobile users accessing the Web over digital cellular networks, the failure of the wireless link, a frequent occurrence, can result in the loss of work accomplished prior to the disruption. This work must then be repeated upon subsequent reconnection—often at significant cost in time and computation. This "disconnection-reconnection-repeat work" cycle may cause mobile clients to incur substantial monetary as well as resource (such as battery power) costs. In this paper, we propose a protocol for "recovering" a user to an appropriate recent interaction state after such a failure. The objective is to minimize the amount of work that needs to be redone upon restart after failure. Whereas classical database recovery focuses on recovering the system, i.e., all transactions, our work considers the problem of recovering a particular user interaction with the system. This recovery problem encompasses several interesting subproblems: 1) modeling user interaction in a way that is useful for recovery, 2) characterizing a user's "recovery state," 3) determining the state to which a user should be recovered, and 4) defining a recovery mechanism. We describe the user interaction with one or more Web sites using intuitive and familiar concepts from database transactions. We call this interaction an Internet Transaction (iTX), distinguish this notion from extant transaction models, and develop a model for it, as well as for a user's state on a Web site. Based on the twin foundations of our iTX and state models, we finally describe an effective protocol for recovering users to valid states in Internet interactions.

Index Terms—Mobile, Internet, transaction, user recovery, interaction model, parametric dependencies, dependent component action graph, ACID properties.

1 Introduction

Web sites are increasingly making use of dynamic scripting techniques, which allow greater interactivity than static HTML. Here, users interact with dynamic sites to achieve specific goals, typically through a sequence of actions. A disruption in the sequence due to the failure of one of the participating or intermediary systems typically results in the user having to restart the sequence, often at significant expense to both the user and to e-businesses. For mobile users accessing the Web over digital cellular networks, such disruptions occur frequently, as the wireless link is much less reliable than wired connections. In this paper, we focus on the case of the mobile Internet user.

Consider a scenario in which a user is buying an airline ticket over his wireless Internet connection. He first logs on to the airline site with his frequent flier number, then checks his frequent flier mileage. He then enters his preferred travel dates and destination, and chooses among the itineraries offered by the site, selects his seat, enters his credit card information, and receives a confirmation of the purchase. Note that this interaction can span multiple sites, as may be the case, for example, where an airline passenger's initial reservation request is entered on the airline's site, but processed on a different site to whom the airline has outsourced online reservation processing.

In both these cases, if the user's wireless connection drops (as occurs frequently [22]) during the purchase step, he must reconnect to the Internet and restart the sequence of actions. Significant amounts of time and effort are wasted in redoing previously completed work. The client incurs significant costs in battery resources and airtime, as well as the user's own time. On the server side, this includes expensive steps—such as processor-intensive login and authentication and I/O intensive database lookups. Redoing this work can contribute to scalability problems on Web sites, particularly when redundant processing causes first-time jobs to wait for computational resources to become available.

1.1 The Problem

In the scenario described above, we would like to be able to avoid the repetition of work (computation, communication, I/O) required after a connection disruption, i.e., we would like *to reduce the cost of recovering a user's interaction*.

Much like a database transaction, in many ways this interaction consists of a number of actions aimed at achieving a particular goal or set of goals. However, the

- *D. VanderMeer and S.B. Navathe are with the College of Computing, Georgia Institute of Technology, Atlanta, GA 30332. Email: {deb, sham}@cc.gatech.edu.*
- *A. Datta and K. Dutta are with the DuPree College of Management, Georgia Institute of Technology, Atlanta, GA 30332. Email: anindya.datta@mgt.gatech.edu and gte314q@prism.gatech.edu.*
- *K. Ramamritham is with the Department of Computer Science and Engineering, IIT-Bombay, Powai, Mumbai 400076, India. Email: krithi@cse.iitb.ac.in.*

Manuscript received 18 Oct. 2002; revised 11 Feb. 2003; accepted 22 Feb. 2003.
For information on obtaining reprints of this article, please send e-mail to: tmc@computer.org, and reference IEEECS Log Number 8-102002.

1536-1233/03/$17.00 © 2003 IEEE Published by the IEEE CS, CASS, ComSoc, IES, & SPS

recovery of such interactions is quite different from the classical notion of transaction recovery. In recovering transactions, the focus is on ensuring that the underlying database system is rendered consistent. Here, if a transaction prematurely aborts, the transaction is rolled back and is resubmitted after the database system recovers. In our case, we assume that the database and application systems have the necessary recovery components in place—this is *not* our concern. We *are* concerned with the *significant cost to the user of resubmitting the transaction all over again*, and so our goal is to devise a mechanism where users can efficiently and quickly restart from an appropriate point prior to disconnection (yet subsequent to the beginning of the interaction) so that the interactions can resume from that point. Thus, our concern is not "system recovery," but rather "user recovery."

1.2 Goals, Tasks, and Contributions

The purpose of this paper is twofold: 1) to propose ways for users to "recover" from interaction failures by minimizing the amount of work that must be redone upon reconnection, and 2) to demonstrate that this recovery problem is interesting, not only because of the application domain, i.e., the Internet and the World Wide Web, but also because of a number of interesting connections to and differences between this scenario and the recovery scenarios considered in extant (transaction) literature.

1.3 The Interaction Recovery Problem

In the context of a specific user interaction, a user interacts with one or more Web sites through a sequence of actions aimed at achieving a specific goal or set of goals. Since these characteristics resemble those of a transaction, in what follows, invoking the transaction metaphor, we describe user interaction with a Web site as an *Internet Transaction* (**iTX**). The goal here is to recover a user's **iTX** in the case of failure. We now discuss the overall scope of the **iTX** recovery problem by identifying several interesting subproblems.

1.3.1 Describing and Capturing a User's "State" during the Course of an *iTX*

Determining the components of a user's state is a prerequisite to understanding exactly what it is that we are recovering for a user. We are not aware of any usable and implementable models of interaction states of users and, consequently, propose our own state model—which is simple and generic (making it useful beyond the scope of this work). Finally, we describe mechanisms for capturing a user's state using data structures and algorithms.

1.3.2 Choosing a User State for Recovery

Given a set of user states produced by a user's **iTX**, how can we decide to which state to recover a user? Ideally, we would simply choose the user's most recent state before failure. However, this is not always possible or correct, since a user's state may have become invalid, for instance, if the data in the underlying database is modified between failure and reconnection.

Another complexity that arises is the potential for multiobjective interactions in an **iTX**. Here, the sequence of user actions as observed by a Web site (or multiple sites) may not correspond to a linear sequence of actions toward a single objective, but rather may consist of interleaved progress toward multiple user objectives. Consider, for instance, the example of purchasing an airline ticket online. Here, after entering his preferred travel plan, but before confirming the purchase of his ticket, our user might check his frequent flier mileage. Effectively, in the same interaction session, the user is performing two tasks: mileage checking and ticket purchase, where the mileage-checking action is not directly related to the ticket-buying action. The recovery protocol must deal with the complexities arising from a user expressing more than one objective in the same **iTX**.

1.3.3 Recovering a User after Connection Failure

Given a means of describing **iTXs** and user states, how can we use this information to recover a user to a useful state in his **iTX**? We have identified the necessary properties of such a protocol and developed a recovery protocol satisfying these properties, taking into account architectural considerations of wireless connections to the Internet.

We note here that a server may fail while processing a user request. This is particularly problematic in the case where the server fails while processing a transaction-oriented request, such as a credit card payment. In this scenario, the user, whether he accesses the network over a wired or wireless link, has no means of knowing whether his payment action completed or not. In this paper, we are interested in developing protocols that provide the *same quality of service guarantees* for users of both mobile that are available to users of wired connections. In other words, *our protocols address the problems that arise due to the failure of a user's wireless connection, rather than addressing the more general problem of HTTP reliability*.

1.4 Objectives of the Paper

Having described the problem space considered here, we summarize the objectives of this paper as follows:

1. to model an **iTX**, i.e., a user's interaction with one or more Web sites, and describe the properties of **iTXs**;
2. to model user *states* relative to the Web site (or sites), where the state encapsulates a set of information useful for recovery,
3. to develop an algorithm to determine which states are valid in an **iTX**,
4. to find ways of extracting different user objectives and the corresponding sequences of user actions from a single **iTX**,
5. to propose a user action recovery technique to handle failures, and
6. to show the expected behavior of the protocol in possible failure cases.

The remainder of this paper is organized as follows: Section 2 describes related work. Section 3 describes the basic notion of an **iTX**. Section 3.3 describes the properties of **iTXs**. Section 4 describes an **iTX** model and discusses how an **iTX** can be decomposed into a set of component **sub-iTXs**, where each **sub-iTX** represents a logically separate objective in an interaction. Section 5 describes our recovery protocol and Section 6 describes how the

protocol will behave in possible failure cases. Section 7 concludes the paper.

2 Related Work

The theory and application of transaction-based processing in the context of database systems are well-researched topics. Early work such as [1] and [14] describe the basic tenets of transaction processing and compare the performance of transaction processing protocols. Long transactions (e.g., Sagas) are discussed in [12] and [4].

The metaphor of transaction processing has since been extended to other application areas. The ACTA framework [4] describes a formal framework for extending the idea of a transaction to other areas. For example, [2], [13], [19] describe workflow processing using transactions and workflow templates, while [23] utilizes transactions to model mobile interactions. The notion of transactions has even been applied to electronic commerce, as in [18] and [21].

All the works cited above have one idea in common: The activity modeled as a transaction consists of a predefined sequence of operations (typically called a *template* in the workflow literature). However, user activity with the Web often progresses in a "stream-of-consciousness" fashion, where neither the user nor the site knows which operation will come next in the interaction. While this paper describes a user's interaction with one or more Web sites using the metaphor of a transaction, we relax the restriction that the transaction models an interaction with a predefined sequence of operations. Rather than using predefined templates, our proposed approach *observes* the sequence of operations as they occur, logs a "state" corresponding to each operation, saves these "states" in a log, and utilizes these "states" to recover the user to a useful point in his interaction.

We also draw on previous work in recovery and graph theory. Database recovery is described clearly in [16]. In [17], the authors describe a protocol for mobile recovery based on basic transaction recovery. Our work differs from this in that we consider the validity of stored user states, as well as multiobjective transactions. In [3], the authors describe an approach to recovering database-backed applications. Again, this work takes a server-centric view, while we consider recovery from the point of view of a single user. In terms of graph theory, we draw on work described in [5].

3 iTX: Definition and Properties

We define the notion of an **iTX** as follows:

Definition 1. *An **iTX** I is a sequence of user actions $\langle A_1, A_2, \ldots, A_n \rangle$ (called* component actions*) initiated by a particular user in the context of a single user session on a particular Web site or a set of Web sites, to achieve one or more user objectives.*

The reader will note that we consider user interaction in the context of a *user session*, rather than in the more familiar context of a *Web session*. Clearly, this notion requires further explanation. In this section, we first use an example to convey the basic idea of an **iTX**. We then clarify the notion of user sessions as opposed to Web sessions and then consider the properties of **iTXs**.

3.1 An iTX Example

We now describe an example scenario, which we also use as a running example through the remainder of this paper. In this example, as well as through the remainder of this paper, we assume that the Web sites we discuss are well-built (i.e., database transactions are atomic and supported by a transaction processing monitor, such as BEA's Tuxedo software) and error-free (i.e., the Web site application code produces expected outputs, given acceptable inputs).

Consider a scenario in which a user interacts with an airline reservation Web site. Here, our example user logs on to the airline site with his frequent flier number and proceeds to interact with the site. He first enters his desired travel plan for an upcoming trip and receives a set of itineraries. Then, using a link on a navigation bar, he checks his other (previously planned) itineraries for upcoming trips to check for potential conflicts with his current travel plans. Using yet another link on the navigation bar, he then checks his frequent flier mileage to verify that the mileage for his recent trips appears in his account. He then returns to his itineraries (using the back button) and cancels a previously planned trip. Then, returning to his mileage account (through the History list), he notices that a recent trip is not reflected in his account, and so he submits a request to update his mileage account to reflect that trip. He then rechecks his mileage. Then, he returns to the itinerary page (through the History list) and selects his preferred flight. After selecting a seat on the flight from the available seats, he submits his credit card information for payment. Finally, he checks his itineraries again, to verify that his purchase appears there.

We denote this **iTX** I as the sequence

$$\langle A_1, A_2, A_3, A_4, A_5, A_6, A_7, A_8, A_9, A_{10}, A_{11} \rangle,$$

as shown below:

I: A_1 (login);
A_2 (plan entry);
A_3 (previous itinerary check);
A_4 (mileage check);
A_5 (itinerary cancel);
A_6 (request mileage update);
A_7 (mileage re-check);
A_8 (itinerary selection);
A_9 (seat selection);
A_{10} (ticket payment);
A_{11} (itinerary verify).

Here, we assume that the use of the Back button and the History list will replay the previous response received, rather than reposting the previous request to the site. This can occur in a number of ways, e.g., if a user is interacting with a site through multiple instances of the browser. Thus, the action sequence as observed by the Web site does not necessarily contain Back or History actions.

3.2 The Notion of a User Session

The reader may have noted that Definition 1 makes reference to the notion of a *user session*. We now explain why we introduce this concept, rather than employing the more familiar idea of a *Web session*.

A Web session represents a single user's interaction with a single Web site in a single visit. During this interaction, the site recognizes the continuity of the Web session through the use of a session identifier. Web sessions have limited time validity; after a site-specified time period, inactive Web sessions "time out," i.e., become invalid on the site. Clearly, the notion of a Web session has meaning only to the Web site on which the Web session is taking place.

It is possible, however, that a user may interact with more than one Web site in pursuit of his objectives. Consider, for example, a scenario in which an airline has chosen to outsource its online reservation system to a third party. Here, the airline's Web site might provide a form for entering travel reservation data, while subsequent interactions in the ticket-purchase process (e.g., itinerary selection and payment) are handled by a different site. We can map this example to the generic **iTX** example above: Action A_2 (plan entry) occurs on the airline's site, while all subsequent interactions, i.e., actions A_3 (previous itinerary check) to A_{11} (itinerary verify), are handled by the outsourced provider. In this scenario, the user's interaction in the ticket-purchase process spans two sites, with separate session ids on each site.

Clearly, in this case, the notion of a Web session, limited as it is to a single site, is not sufficient to characterize this interaction. Thus, we introduce the notion of a *user session*, in which we relax the Web session requirement that a session takes place on a single site, and allow a user session to span multiple sites. Here, we can define the notion of a user session clearly by considering a "user-centric" point of view: A *user session* is a span of time in which a single user interacts with one or more Web sites during which there exists at least one valid active Web session (i.e., a Web session that has not timed out at the Web site).

3.3 Properties of iTXs

iTXs possess a number of interesting properties across several dimensions. Two specific properties are of interest in this work: 1) differing Web site and user expectations of the interaction and 2) the potential for multiple objectives. We consider these properties in light of the differences between **iTXs** and extant transaction models, as well as the impact each property has on the design of a recovery protocol. In the ensuing discussion, and throughout the remainder of the paper, we refer to both the single and multiple-site **iTX** cases as the single-site case without loss of generality, except in instances where the cases differ.

3.3.1 Web Site and User Expectations of an *iTX* Differ

One essential intuition needed to understand **iTXs** is that the Web site and the user have different expectations. Let us first consider what the Web site expects. Since the HTTP protocol is stateless [11], *the Web site treats each user action as an independent task*. Here, a user's action may *enable other actions*, e.g., a user must first select a flight (action A_8 in our example **iTX**) before he can select a seat (action A_9), but the site *does not enforce a specific workflow*. For example, the site does not force the user to select a seat after selecting a flight, but he cannot select a seat without first selecting a flight.

From the perspective of the user, however, each component action is part of a sequence aimed at reaching a specific goal or set of goals (much like operations in a transaction). In other words, the user views his actions as a logically related sequence of actions, rather than as a set of independent actions. For example, in the airline reservation **iTX**, a user who has selected a flight is very likely to next select a seat on the flight.

The above discussion reveals our first **iTX** property:

Property 1. *In an **iTX**, the Web site and user have differing expectations with respect to a user's interaction on the site.*

Given these differing expectations, one natural question that arises is: "How does a site provide the user with the feeling that his actions are semantically related, while treating each of his actions as independent?" This is accomplished by maintaining a *state* for each user, which represents the cumulative effect of the user's actions on the site. After each component action A_i of an **iTX**, the user's state S_i is either 1) a *root state* S_0 in the case of a newly-started **iTX**, or 2) a modification of the state S_{i-1} resulting from the previous action A_{i-1}. Thus, an interaction can be captured by a sequence of states, corresponding to the actions in an **iTX**. Using this notion, we can think of an **iTX** as *evolving*, with each action resulting in a new state built on the state resulting from the previous action (if any) and any new information resulting from the current action.

In virtually all Web sites, the notion of a user state is implemented in one of two ways: 1) state information is maintained in a *session object* on the site, and referenced with a *session id* (which is passed back and forth between the user and the site in each HTTP request and response), or 2) all state information is passed back and forth between the user and the site in the HTTP request and response using mechanisms such as HTTP headers or hidden fields.

Intuitively, we can think of a *user state* on a Web site as referring to a "snapshot" of the information that results from a user's interaction with a Web site. Because of the stateless nature of the HTTP protocol [11], this interaction occurs on an action-by-action basis. Corresponding to each such action-response cycle, information is exchanged between the user and the site—the *state* captures a snapshot of this exchange.

Essentially, the state comprises four types of information:

1. an HTTP request, i.e., the request the user has sent to the site,
2. an HTTP response, i.e., the actual text sent to the user in response to a request,
3. identifying information and results of the operations the site has executed on behalf of the user, such as would be found in cookies, and
4. a validity parameter, which denotes whether or not the state is valid.

Thus, we define a user's *state* on the Web site as follows:

Definition 2. *In an **iTX**, a user's state changes from action to action. After each component action A_i in an iTX, the user state S_i is a 4-tuple $\langle K_i, Q_i, P_i, L_i \rangle$, where*

- *K_i is the set of cookies valid for the user after action A_i.*
- *Q_i is the user's HTTP request corresponding to action A_i.*
- *P_i is the site's HTTP response to Q_i.*
- *L_i is the result of a function $f(K_i, Q_i, P_i)$ denoting the validity of the user state.*

In our airline site example in Section 3.1, the user submits the details of his travel plan (action A_2), and the site gives back a set of itineraries to the user. After receiving the itineraries, the user's (valid) state is composed of K_2, his cookie information (which includes his login information), Q_2, the HTTP request for the itineraries (containing his travel plans), and P_2, the site's HTTP response to his query (the page containing a set of itineraries).

The validity of a state, L_i, requires further elaboration. A user's state S_i is based on the information available in the Web site's database at the time of action A_i. The database state may change after A_i, in which case we may not want to use the information in S_i for recovery purposes. Consider our airline **iTX** example. Suppose that our user's **iTX** fails after itinerary selection (action A_8) but before seat selection (A_9) and that, before he reconnects, the site sells the last seat on the user's preferred flight. In this case, the user's state containing the originally offered itineraries is invalid since it is inconsistent with the database at the Web site, and should not be served to the user when he reconnects.

A common mechanism for invalidation is the use of "time-to-live" periods; here, a user response sent out from the origin server would include a maximum age value, after which the response would be considered invalid. This can be achieved through the use of the *Expires* header in the HTTP Protocol [11]. Consider, for example, our airline request example. Suppose that a user has reached step A_9, i.e., he has requested a particular seat on a specific flight. In this situation, the airline site may send back a response confirming the seat selection with an *Expires* header with a date value 30 minutes later than the generation time of the response. This *Expires* header would indicate that the site will consider the seat reservation valid for 30 minutes, after which (if the user has not completed the purchase process by submitting payment information) the seat reservation would be returned to the pool of available seats and offered to other customers.

Having considered the difference in the Web site and user expectations of an **iTX**, and having introduced the idea of a user state, we now consider how **iTX** Property 1 compares with existing transaction models. Virtually all transaction models share a common notion of a transaction as a sequence of operations over which *ACID* semantics are imposed: *atomicity*, *consistency*, *isolation*, and *durability*. We consider these in the context of an **iTX**. Here, consistency and durability are clearly must-haves. Returning to our airline reservation example, a site that sells the same seat to two people (violating consistency) or "loses" a user's reservation (violating durability) would not attract or retain many customers. These characteristics are typically provided by the site's database(s).

Clearly, failure atomicity must also be supported on the site. For example, returning to the airline site **iTX**, if a user selects a seat on a flight (action A_9), but departs before actually purchasing the seat, the site must eventually return that seat to inventory. This is typically accomplished through a combination of Web site application code and database operations. For example, a site may, on the timeout of a user's session (at which point the site assumes that the user has departed), initiate actions to "undo" partially completed purchases by returning unpurchased items (e.g., seats in our airline example) to available inventory in the site's database.

In the context of a Web site environment, the isolation property, in which a transaction is prevented from seeing the partial results of other transactions, is virtually impossible to support. Returning to our airline example, consider a scenario in which a user U_1 has selected a seat on a flight, but has not yet actually purchased the seat. Further suppose that user U_2 requests a view of the available seats on the same flight immediately after U_1 selected his seat. Here, U_2 will see that the seat U_1 chose is "taken", *even though U_1 has not actually purchased the seat*. Clearly, partial results are visible here. Supporting isolation in this environment would require locking key portions of the database (e.g., available itineraries in the airline example), which would severely restrict the number of users a site can support simultaneously.

Given the above discussion on the support of ACID semantics in **iTXs**, a comparison with support of ACID semantics in extended (long) transaction models (e.g., [12]) would seem to be in order. Readers will note that extended transaction models do allow the violation of the isolation property at the root level. However, at the component transaction level, extended transaction models mandate that execution be isolated while **iTXs**, in contrast, do not require isolation even at the component action level. The reason for this is subtle but important: A component transaction in an extended transaction may itself be a multiaction *transaction*, whereas the *component action in an* **iTX** *is always a single-action* task, where the single action is a HTTP request submitted by the user, causes the execution of a script that may result in multiple (atomic) updates on databases *without necessarily enforcing isolation across this set of updates*.

Given the discussion above, we are interested in how Property 1 impacts the design of a recovery protocol. Virtually all extant transaction models utilize *backward recovery* and/or *forward recovery* mechanisms. In backward recovery, the recovery protocol undoes the effects of a partially completed transaction. In forward recovery, the recovery protocol performs or (in the case of transaction restart) reperforms the operations in a transaction.

Neither backward recovery nor forward recovery are truly appropriate for **iTX** *recovery*. The reader will recall from the discussion in Section 1 that the Web site application and database do not require recovery; rather, we are interested in *user recovery*, which need not involve the Web site systems. In this environment, backward recovery is not possible—virtually no Web site exposes an interface that will allow the automatic invocation of compensating operations (i.e., without a user choosing the operation through an HTML page) for user actions. Nor is forward recovery possible—the user's future actions are unknown and the results of past actions may be invalid. In this latter

case, the automatic resubmission of a user's previous HTTP requests might result in errors. What is really needed here is the ability to provide the user with a *recent and valid interaction state*, with the goal of minimizing the amount of work that must be redone. Based on this discussion, we can define the following feature requirements for an **iTX** recovery protocol.

Feature 1. *An **iTX** recovery protocol should have the ability to:*

- *Store user states for use in the case of connection failure.*
- *Return a recent and valid state to the user upon reconnection after failure.*

3.3.2 An *iTX* may have Multiple Objectives

A single **iTX** may reveal several user objectives. Recall the example **iTX** in Section 3.1. Here, the user reveals several separate objectives:

1. canceling his previously planned itineraries,
2. verifying that his frequent flier mileage balance is correct, and
3. purchasing a new ticket.

Each of these objectives can be represented by a (possibly overlapping) subsequence of component actions called a **sub-iTX**.

Property 2. *An **iTX** may be composed of several potentially overlapping **sub-iTXs**, where each **sub-iTX** represents a separate user objective.*

We define a **sub-iTX** as follows.

Definition 3. *Consider an **iTX** $I = \langle A_1 \ldots A_n \rangle$. A **sub-iTX** B of an **iTX** I consists of a sequence of component actions $\langle A_j, A_k, \ldots, A_m \rangle$, meeting the following conditions:*

1. *Each component action of B is a component action in I.*
2. *For any two component actions A_a and A_b in B, if A_a precedes A_b in B, then A_a must also precede A_b in I.*
3. *B semantically represents a single user objective.*

For instance, the **sub-iTX** representing the itinerary-canceling objective consists of the subsequence $\langle A_1, A_3, A_5 \rangle$ (login, previous itinerary check, itinerary cancel), while the **sub-iTX** for mileage account-checking consists of $\langle A_1, A_7 \rangle$ (login, mileage recheck), and the **sub-iTX** for the ticket-purchase objective consists of the subsequence

$$\langle A_1, A_2, A_8, A_9, A_{10}, A_{11} \rangle$$

(login, plan entry, itinerary selection, seat selection, ticket payment, itinerary verify). **Sub-iTXs** can be arbitrarily interleaved within a single iTX. Depending on the possible interactions among objectives, **sub-iTXs** may or may not overlap by sharing a common prefix sequence of component actions. In this example, all **sub-iTXs** in the iTX share a common prefix: the login action (A_1). Note that only one of these **sub-iTXs** can be *active* at any given time. For example, at component action A_5 (mileage-checking), the user is clearly active in the mileage-check **sub-iTX**, and not the ticket-purchase **sub-iTX**.

Given the above discussion, the following question arises: Relative to which objective should the user interaction recover? Our proposed solution is to postulate that the user should be returned to the objective (i.e., the **sub-iTX**) that was active at the time of failure, i.e., the **sub-iTX** of his most recent action on the site. This places an interesting requirement on a recovery protocol: *In order to return the user to a useful point in the* **iTX**, *the user's* **sub-iTX** *of interest at the time of failure must be discerned, and all other* **sub-iTXs** *filtered out from consideration in the recovery protocol*. For example, in the airline **iTX**, recovery from a failure at action A_9 (seat selection) should not place the user at the mileage-checking action A_4 (mileage check). Similarly, failure at component action A_4 (mileage-check) should not recover the user to the travel-plan entry action A_2 (plan entry). This reveals the following required feature in our recovery protocol.

Feature 2. *The recovery protocol must be able to distinguish among multiple **sub-iTXs** in an **iTX**.*

We will show, in Section 4, how to distinguish **sub-iTXs** in an **iTX**, which makes it possible to design a recovery protocol supporting Feature 2.

4 Developing a Graphical iTX Model

In an **iTX**, data generated in one user action may be used by the site in further interactions with the user, i.e., in responses to later actions by the same user. Consider, for instance, the airline site example in Section 3.1. Here, clearly, our user cannot select a seat on a flight (action A_9) without first selecting a flight (action A_8), which in turn cannot occur if the user has not yet entered his travel plan (action A_2). This implies a form of dependency among actions in an **iTX**, i.e., action A_8 (itinerary selection) is dependent on parameters that are somehow passed to it from action A_2 (plan entry). Since these dependencies are based on parameter-passing, we refer to them as *parametric dependencies* (PDs). Such parametric dependencies can be used to distinguish **sub-iTXs** within an **iTX**.

The first issue that arises is how such PDs can be recognized. In fact, in light of the stateless nature of the HTTP protocol (i.e., the fact that the Web site observes two parametrically dependent actions as independent, rather than related, as noted in Property 1), such recognition would appear to be impossible. However, it turns out that PDs can indeed be recognized in the context of modern *dynamic Web sites*. A *dynamic* Web site is one in which the content sent to a user is generated on the fly by running a *script*. In contrast, in a *static site*, static files are served directly to the user from the file system, i.e., no process runs to generate the page. Without a process, there can be no parameters passed, therefore, there are no PDs in static sites.

Several site building technologies, such as Java Server Pages (JSP) [15], and Active Server Pages (ASP) [7], support dynamic page generation. These have become popular because they dramatically increase the potential for interactivity with a user—a dynamic site composed of few scripts can potentially generate a vast number of different pages based on information retrieved from an underlying database, subject to various input parameters.

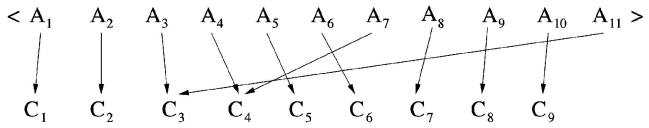


Fig. 1. Action-Script mapping.

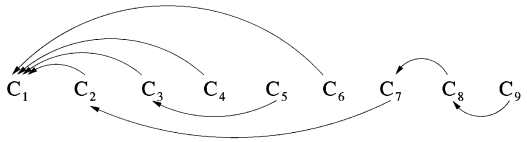


Fig. 2. Script dependencies.

When a user interacts with a dynamic site, each component action A_i corresponds to an execution instance of some script C_i. Note that multiple component actions in the course of a single iTX may map to the same script (e.g., the user may check his frequent flier mileage more than once during a single visit to an airline Web site). Consider the airline reservation system example. Here, when the user submits a request with his travel details (e.g., component action A_2), the site runs a program using the user's travel details as the input parameters of the program. This program accesses the airline database, which returns a set of suitable itineraries. The program takes the itineraries returned from the database, adds an HTML presentation layer, and serves the itineraries back to the user as the site's response.

In fact, we can map each action A_i in an **iTX** to a script C_i, i.e., the logic in script C_i embodies the semantic action A_i. For example, an action that involves submitting login information on a Web site might map to a particular script `login.jsp`.

In our airline reservation example, we can map the actions in **iTX** I to scripts in the airline site as depicted in Fig. 1.

Here, component action A_1 (login) maps to script the script C_1 (`login.jsp`), while component action A_2 (plan entry) maps to script C_2 (`submit_travel_plan.jsp`). Similarly, component action A_3 (previous itinerary check) maps to script C_3 (`show_current_reservations.jsp`), component action A_4 (mileage check) maps to script C_4 (`check_ff_mileage.jsp`), and component action A_5 (itinerary cancel) maps to script C_5 (`cancel_itinerary.jsp`). Component action A_6 (request mileage update) maps to script C_6 (`request_mileage_update.jsp`). Component action A_7 (mileage recheck) maps to script C_4 (`check_ff_mileage.jsp`), component action A_8 (itinerary selection) maps to script C_7 (`select_itinerary.jsp`), and component action A_9 (seat selection) maps to script C_8 (`select_seat.jsp`). Finally, component action A_{10} (ticket payment) maps to script C_9 (`enter_payment_info.jsp`) and component action A_{11} (itinerary verify) maps to script C_3 (`show_current_reservations.jsp`).

In sites built with these dynamic page generation technologies, it is possible to distinguish the **sub-iTXs** *in an* **iTX** *by examining the parameters passed between scripts on a Web site.* Each component action is carried out by dynamic scripts that run on an application server, such as Microsoft's Internet Information Server (IIS) [8] and BEA Systems' WebLogic Server [20]. Since HTTP is inherently a connectionless protocol [11], each of these dynamic scripts runs independently. However, quite often, a script may be dependent on information obtained from, or generated in, another script. Consider, for example, the scenario in which our traveler has selected his preferred itinerary on the airline site (component action A_8), and is then presented with a seat selection. The script that generates the itinerary choice list executes independently of the seat-selection script, but the seat-selection script must somehow know which flights are in the user's selected itinerary. Clearly, some method of passing information from one script to another is required. There are four ways to achieve this:

1. hidden fields in forms,
2. cookies,
3. session variables, and
4. explicit parameter passing through the HTTP GET/POST call.

Since the number of ways scripts can pass information is limited and decided at site design time, the site designer can statically determine the dependency list among scripts using commonly-used site analyzer/designer tools, e.g., Visual Interdev [9]. Parametric dependencies across sites (which lead to multiple-site **iTXs**) are also well-defined at design time and can easily be added manually to the set of parametric dependencies on receiving sites.

We can define the notion of a script dependency as follows:

Definition 4. *A script C_i is parametrically dependent on another script C_j, denoted $C_i \rightarrow C_j$, when data produced during the execution of C_j becomes input parameters referenced during the execution of C_i. In the case of multiple-site parametric dependencies, C_i and C_j reside on different Web sites.*

Note that the symbol "$\rightarrow$" denotes the flow of dependency (i.e., the "depends-on" relationship); parameters flow between scripts in the opposite direction. The reader should not confuse the notion of parametric dependency with the familiar notion of functional dependencies from the database literature.

We now return to our airline reservation system example (described in Section 3.1) to clarify the notion of parametric dependencies among scripts. Script dependency analysis in this example would produce dependencies among the scripts as shown graphically in Fig. 2.

Here, script C_2 is parametrically dependent on C_1, i.e., $C_2 \rightarrow C_1$ (C_2 relies on a set of parameters generated in C_1). Similarly, $C_3 \rightarrow C_1$ (though C_3 may require a different set of parameters from C_1 than C_2), $C_4 \rightarrow C_1$, $C_5 \rightarrow C_3$, $C_6 \rightarrow C_1$, $C_7 \rightarrow C_2$, $C_8 \rightarrow C_7$, and $C_9 \rightarrow C_8$. Script C_1 is not parametrically dependent on any other script, i.e., it draws data from no other script on the site. Note that a particular dependency $C_j \rightarrow C_i$ may span site boundaries—in this type of scenario, C_j receives data it needs from C_i.

Using the action-script mapping (as shown in Fig. 1), we can derive parametric dependencies between component actions in an **iTX**. For instance, in our airline example, action A_2 (plan entry) maps to script C_2, and action A_8

TABLE 1
Sub-iTX Dependency Sequences

Sub-iTX ID	Action Dependency Sequence	Script Dependency Sequence
sub-iTX$_1$	$A_{10} \rightarrow A_9 \rightarrow A_8 \rightarrow A_2 \rightarrow A_1$	$C_9 \rightarrow C_8 \rightarrow C_7 \rightarrow C_2 \rightarrow C_1$
sub-iTX$_2$	$A_6 \rightarrow A_1$	$C_5 \rightarrow C_1$
sub-iTX$_3$	$A_4 \rightarrow A_1$	$C_4 \rightarrow C_1$
sub-iTX$_4$	$A_7 \rightarrow A_1$	$C_4 \rightarrow C_1$
sub-iTX$_5$	$A_5 \rightarrow A_3 \rightarrow A_1$	$C_5 \rightarrow C_3 \rightarrow C_1$
sub-iTX$_6$	$A_{11} \rightarrow A_1$	$C_3 \rightarrow C_1$

(itinerary selection) maps to script C_7. Since script $C_7 \rightarrow C_2$, we can derive a *component action dependency*. Formally, we define the notion of component action dependency as follows:

Definition 5. *A component action A_i is parametrically dependent on another component action A_j if the script corresponding to action A_i (i.e., C_i) is parametrically dependent on a script C_j corresponding to action A_j. We denote this dependency in a manner similar to script dependencies: $A_j \rightarrow A_i$.*

Based on Definition 5, we can provide a precise definition of a **sub-iTX**.

Definition 6. *A **sub-iTX** B is a sequence of parametrically dependent component actions $\langle A_1, A_2, \ldots, A_n \rangle$ that represents a single user objective, where 1) for any adjacent sequential pair of actions $\langle A_i, A_j \rangle$, $A_j \rightarrow A_i$, and 2) the first component action A_1 in the **sub-iTX** has no parametric dependency on any other component action.*

Using these component action dependencies, we can derive **sub-iTXs**, i.e., dependent component action sequences, within an **iTX**. **sub-iTX**s derived for the airline example are shown in Table 1.

From the above discussion, it is clear that **iTXs** are amenable to a graph-based representation, modeling the component actions of an **iTX** as vertices and dependencies between component actions as edges. Accordingly, we model an **iTX** as a *Dependent Component Action Graph* (DCAG), which is defined as follows:

Definition 7. *Corresponding to an **iTX** I, there exists a DCAG $G_I = (V, E)$, where G_I is a directed, acyclic graph, a vertex $v \in V$ is a component action of a particular **iTX** I, and an edge $e = (v, u) \in E$ represents the dependency of v on u, i.e., $v \rightarrow u$ (where v is dependent on parameters generated in u).*

Returning to our running airline site example, the DCAG of the dependencies for our user's **iTX**, as developed in Section 4, is depicted graphically in Fig. 3.

Here, we can clearly see the six **sub-iTXs** discovered in the course of the discussion of the airline site example in Section 4: The path from A_{10} (ticket payment) to A_1 (login) represents **sub-iTX**$_1$; the path from A_6 (request mileage update) to A_1 (login) represents **sub-iTX**$_2$; the path from A_4 (mileage check) to A_1 (login) represents **sub-iTX**$_3$; the path from A_7 (mileage re-check) to A_1 (login) represents **sub-iTX**$_4$; the path from A_5 (itinerary cancel) to A_1 login represents **sub-iTX**$_5$; and the path from A_{11} (itinerary verify) to A_1 (login) represents **sub-iTX**$_6$.

The DCAG is of great importance in recovering users' **iTXs**; as the reader will see in Section 5, it is a crucial part of our recovery strategy. Intuitively, each path in a DCAG represents a *failover path*. Here, if a user's **iTX** fails at some action A_r, and there exists an edge in the DCAG (A_r, A_q), then, assuming that there exists a valid state for the user for A_q, we can recover the user to A_q (if A_q is valid), from which he can again attempt action A_r. If A_q is invalid, and there exists an edge (A_q, A_p), we traverse the DCAG from A_q to A_p, and consider the validity of A_p, and so on, until we encounter a valid state in the DCAG, or until we reach a node from which there are no outgoing edges (in which case, recovering the user state is infeasible). Essentially, then, the DCAG represents a *failover map* for the recovery protocol. Henceforth, we will use the terms DCAG and failover map interchangeably.

We now consider the notion of DCAGs in the context of our airline site example (Section 3.1). Suppose that the **iTX** failed at component action A_{10} (ticket payment). Then, since we know that the user was involved in **sub-iTX**$_1$ at the time of failure, the recovery protocol can follow the directed edge from A_{10} (ticket payment) to A_9 (seat selection), the most recent component action in the same **sub-iTX**. Then, if

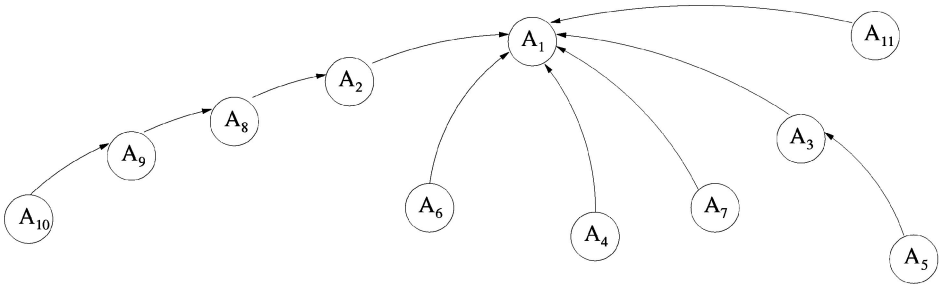


Fig. 3. Example of dependent component action graph.

the state S_9 associated with A_9 (seat selection) is valid, the recovery protocol will recover the user to state S_9. If S_9 is not valid, the protocol will traverse the DCAG from A_9 (seat selection) to A_8 (itinerary selection), checking S_8 for validity, and so on, through **sub-iTX**$_1$ until it finds a valid state, or reaches the beginning of the **sub-iTX** without finding a valid state.

4.1 A Cautionary Note

The goal of our recovery protocol is to handle **iTX** failures. In other words, if an **iTX** fails, our proposed mechanism will allow a user to "recover" much of the work performed in the failed **iTX**. However, our protocol should not be taken as the cure-all recovery scheme for anything a user might do on a Web site. For instance, *there may exist dependencies among separate* **iTXs** *initiated by a user*. In these cases, the knowledge of the semantic relationship between a set of separate **iTXs** does not reside in the **iTXs** themselves; rather, it exists in the user's view of his interaction with the Internet. In these cases, our protocol can recover each of these **iTXs** independently. But, as our protocol will not be privy to the semantic dependencies across these different **iTXs**, certain explicit actions may be required on the part of the user in order to take care of the residual effects of these dependencies. The following example clarifies this notion.

Consider the case of a user who is interested in planning a vacation, including not only airfare to his destination of choice, but also a hotel room. Here, three potential interaction cases exist, each of which maps to a different set of **iTXs** and **sub-iTXs**. We describe each in turn, specifically considering the type of **iTX** in each case.

In the first case, the user might make a hotel reservation on one Web site (perhaps a hotel's Web site), and purchase his airline ticket on another (perhaps the airline's site). Clearly, here, the user's vacation planning does not map to a single **iTX**. Rather, since the hotel plans and airline ticket were purchased on separate sites, and no data was passed between the sites, the hotel and airline site interactions map to separate **iTXs**. In this scenario, a user who has reserved a hotel and then finds that he cannot purchase airline tickets for the dates he prefers, must explicitly cancel his hotel reservation since there is no way for the two sites to condition the placement of the hotel reservation on the success of the airline ticket purchase.

In the second case, the user might place his hotel reservation and purchase his ticket from the same Web site (perhaps an online travel site), but *in separate transactions*. Here, he would complete the hotel reservation, then begin searching for airline tickets for his preferred dates. In this scenario, the site does not pass any data between the hotel reservation and airline ticket purchase transactions; thus, they are separate **sub-iTXs** within the same **iTX**. As in the first case, a user who made a hotel reservation and then failed to find matching airline tickets, would need to explicitly cancel the hotel reservation.

In the third and final case, the user might be able to bundle the hotel reservation and airline ticket purchase *in the same transaction* on the same site. Here, when the user searches for hotel reservations, the site prompts him for airline tickets,[1] and asks for a "commit" point (at which both his hotel and airline plans would be booked) only once. Here, the user's vacation plans map to a single **iTX**.

1. Up-selling, i.e., bundling airline and hotel plans in the same transaction, occurs on many travel sites, such as www.expedia.com.

5 Recovering from Failure in an iTX

Based on the features described above in Section 3.3, we now describe how we recover user interactions with the Web. Intuitively, our proposed recovery protocol has the following steps:

a. log a user state for each component action of a user's **iTX** in an *action log*,
b. generate and continually update a *failover map* (i.e., a DCAG) of the user's **iTX**, showing the various **sub-iTXs** within it, and
c. upon failure of a user's **iTX**, consult the log and the failover map to recover the user.

Steps a and b occur concurrently with the user's interaction with the site, while step c occurs upon failure. Here, four issues of importance arise:

1. the location of the action logs, failover maps, and recovery logic,
2. how the action logs and failover maps are generated,
3. choosing the action from which recovery should be undertaken, and
4. upon failure of an **iTX**, how the recovery protocol can use the user's log and failover map to recover his **iTX**.

We discuss each of these issues in turn. In addition, we include a brief description of our design and implementation efforts, which are currently underway, as well as our plans for testing our protocols once implementation is complete, as shown in Appendix F.

5.1 Location of the Logs, Failover Maps, and Recovery Logic

Recovery functionality must reside somewhere on the (inclusive) path between the Web site and the user's browser. Since the only input to our protocol is the captured user states, the recovery functionality can be resident on any device that observes the state of every component action of the user, i.e., at any of several points on this path, in both wireless and wired architectures, where state can be observed and logs can be maintained and processed.

Wireless devices connect to the Internet through a software interface called a *gateway*, e.g., the Nokia WAP Server [6]. This architecture is depicted graphically in Fig. 4. As readers may not be familiar with the particular role of the gateway in this architecture, we provide a brief overview here. The functionality of the gateway is to serve as an interface between the binary wireless transmission protocol and the TCP/IP transmission protocol used on the Internet. (In addition, the gateway also handles numerous other tasks, such as content filtering, billing, and security.) Since wired-to-wireless or wireless-to-wired translation must occur for each HTTP request and response, all interaction between a user and the Internet will pass through a gateway. Gateways typically serve a large geographic area and must be explicitly specified by the user (i.e., there is no notion of transparent hand-off between

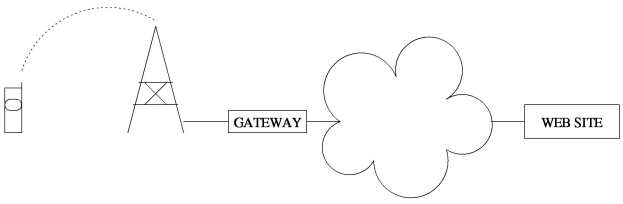


Fig. 4. Mobile unit to Web site connectivity.

gateways, as exists in cellular networks)—users typically dial in to a gateway in much the same way that a modem dials in to an ISP.

As noted above, the recovery functionality can reside at different points in the architecture depicted in Fig. 4, e.g., on the user's browser, on the gateway, on devices in the network cloud, or on the Web site.

The wireless device is a nonoptimal choice since it has limited battery, memory, and processing power. We might consider using *persistent cookies* on the wireless device to store a user's recovery information locally. In this approach, we would store the user's session id, as well as enough other information to recover the user to a useful interaction state, in a persistent cookie on the client's machine; that is, after each action, all this information would be written to persistent memory in a new cookie. The first problem with this approach arises from the limits imposed on persistent cookies in the standard: persistent cookie files are limited in both the space allotted to a single cookie (4KB) and the number of cookies that a particular domain can set for a given user (20 per domain) [10]. Storing recovery information at the client in this format is likely to require multiple cookies since recovery information must be stored for each click. Given this, and the fact that many sites use several of their "allotted" cookies to store user-identification and profile information, it is likely that a recovery protocol utilizing persistent cookies may in some cases run out of "cookie space," resulting in the inability to store recovery information, and potentially preventing recovery due to the lack of needed data.

The various network nodes (in the Internet cloud depicted in Fig. 4), such as routers, cannot serve this purpose since user requests are not guaranteed to traverse the same path throughout a user's session with a site.

The Web site is also a poor choice of location for the data structures and algorithms of our recovery protocol. First, and foremost, the servers on the Web site are already overburdened with functionality; adding additional work would decrease the scalability as well as performance of an already-overloaded component of the site architecture. Storing each user's interaction information on the site's persistent storage (since, in spite of the perceived abundance of memory, transient memory space on Web and application servers is typically a hotly-contended resource) raises a serious performance issue on the site—interaction information would need to be written to disk *for each user click*, introducing an enormous I/O overhead burden. In addition, if a user's interaction spans multiple sites, as is the case in some systems, recovery information would be spread across multiple autonomous servers, leaving open the question of how a recovery protocol would handle such distributed recovery data.

Transaction Processing (TP) monitors, which are typically used as recovery mechanisms within a Web site infrastructure, might seem to be applicable to this problem at first glance. However, on disconnection, the user loses his session state, including his session id. Once this occurs, there is no way for the site to associate the user with his session on return—the user's IP address will not necessarily work since the site may associate several users coming to the site through the same proxy as having the same IP address. TP monitors, as part of the Web site infrastructure, are subject to this restriction and, thus, cannot be used to solve this problem.

The gateway, however, seems to be a natural choice for the location of the recovery functionality since all Web site interaction for a particular user during a particular session funnels through the same gateway. A recovery mechanism residing at the gateway can monitor outgoing (from the user) requests, as well as incoming responses, allowing the gateway to capture states. Since, as noted above, gateways must be explicitly set in the user's wireless device (and connected to, in much the same way a modem user dials into an ISP), users are unlikely to reconnect to different gateways after failure—so, the gateway that captures a user's state(s) will also be able to recover the user upon reconnection. Thus, we have chosen to describe the workings of our protocol in the context of a gateway-resident recovery system for the remainder of this paper.[2]

5.2 Generating Action Logs and Failover Maps

We now turn to the second aspect of our recovery protocol, namely, the storage of user states and the construction of a DCAG or failover map corresponding to a specific **iTX**. In principle, this is quite simple—the gateway, by observing incoming and outgoing traffic, traps and stores user state information in a log. Simultaneously, by using the log, it continually updates the failover maps of the **iTXs** currently being observed. We now 1) present a set of data structures for storing user states (i.e., design the log), 2) describe algorithms for trapping state information from user HTTP requests and responses, and 3) describe algorithms that generate failover maps using information from the log, as well as other information.

5.2.1 The Action Log Data Structure

We provide a relational representation for the log below for ease of discourse; clearly, many other data structure representations are possible (e.g., simple text logs). We store each action as a tuple in an *ACTION_TABLE* relation, where each tuple contains the following nine attributes:

- **ROW_ID**: the unique identifier for each component action.
- **USER_ID**: a unique identifier for the user (in a wireless scenario, the user identifier can be derived from the device identifier of the wireless device, which uniquely identifies it).

2. We provide a brief discussion of the security implications of this choice in Appendix G.

- **SCRIPT_ID**: the unique identifier of the dynamic script specified with the request and associated with the component action (part of the HTTP header).
- **REQ_TIME**: the time at which the gateway receives the user's HTTP request (i.e., a time stamp),
- **RES_TIME**: the time at which the response is generated in the origin server (i.e., a time stamp).
- **HTTP_RESPONSE**: the HTTP response received in response to the request.
- **COOKIE**: the cookie information that has been sent by the Web server along with the HTTP response.
- **TIME_TO_LIVE**: The duration for which the HTTP response is valid; if this parameter is not specified, time-to-live is assumed to be infinite, i.e., the HTTP response never becomes invalid. This parameter, as noted in Section 3.3.1, may be passed through the HTTP *Expires* header.
- **NODE_PTR**: Pointer to the corresponding graph node in the failover map.

5.2.2 Trapping Action Log Data from HTTP Requests and Responses

Next, we consider how action log data is extracted from an HTTP request at the gateway. As each component action passes through the gateway, the gateway traps this information using procedures PROCESS_REQUEST and PROCESS_RESPONSE, shown in Algorithm 1 and Algorithm 2 (both given in Appendix A, with complexity given in Appendix E), respectively, stores it in the ACTION_TABLE, and updates the failover map.

In order to make clear how the gateway processes requests, consider our running example of the ticket-purchasing **sub-iTX**, i.e., **sub-iTX**$_1$. Suppose our user has logged in (action A_1) and submitted his travel plan (action A_2). As a result of submitting his travel plan, the site has offered the user a set of possible itineraries. Upon selecting his itinerary, the user submits an HTTP request for his preferred flight (action A_8). Here, the gateway calls the PROCESS_REQUEST function, which adds a new row to the ACTION_TABLE containing our user's *user_id*, the *script_name* requested (i.e., `select_itinerary.jsp`), and a time stamp for the request.

To clarify the workings of response processing on the gateway, we return to the airline-ticket example begun in our description of PROCESS_REQUEST in the previous paragraph. Here, our user has logged in to the site, submitted his travel plan, and has requested a particular flight (i.e., his **sub-iTX** consists of $\langle A_1, A_2, A_8 \rangle$). In return, the site sends him an HTTP response offering him a choice of available seats on the flight. Upon receiving this HTTP response on behalf of the user, the gateway extracts several necessary items of information: the time at which the response was generated on the origin server, the age of the response, the script list (which contains script dependencies for the recovery protocol—how the script dependency information is obtained is discussed in Section 5.2.3), any cookie(s) in the response, the user id, and the TTL of the response. The gateway then creates a new node in the user's failover map for the current action and updates the row of the ACTION_TABLE that contains the request that prompted the current HTTP response with the information from the HTTP response.

Having discussed the processing of both HTTP requests and HTTP responses, we now consider how failover maps are updated with new user actions.

5.2.3 Generate Failover Maps

The failover map, i.e., the DCAG, corresponding to a particular user's **iTX** is dynamically created as he clicks on the Web site. Before we delve into the specifics of failover map generation, we first consider the information required to generate a failover map.

As the reader will recall from the discussion in Section 4, generating a DCAG requires script dependency information in order to determine which component actions are dependent on one another. In order to create a DCAG for a user's **iTX** on the gateway, the gateway must obtain this script dependency information, which is server-resident. It turns out that this is fairly easily achieved—the site can pass this information to the gateway by adding a *ScriptList* directive to the HTTP header (this can be achieved simply by writing the attribute into the HTTP header in the output of the script on the site) in the HTTP response. Since script dependencies are static and predefined, no computation is required at runtime on the Web site to determine the dependencies.

The *ScriptList* directive consists of a list *ScriptList* where *ScriptList = ScriptName[, ScriptList]* and *ScriptName is a script name, including the full URL path*. Here, HTTP response *is dependent on each script* listed in the *ScriptList*. For instance, in our airline example, suppose that there exist two scripts on the site, `login.jsp` (corresponding to A_1, the login action), and `submit_travel_plan.jsp` (corresponding to A_2, the travel plan entry action). As noted in Section 4, $C_2 \rightarrow C_1$, i.e., `submit_travel_plan.jsp` script is dependent on `login.jsp` page. The header of the HTTP response for `submit_travel_plan.jsp` should contain following line: *Script-header: http://www.airlinereservation.com/login.jsp* (assuming the example site domain is *airlinereservation.com*).[3] We now move on to describe our algorithm for actually generating a failover map.

For each component action of a user's **iTX**, a new node is created in the DCAG using the UPDATE_DYNAMIC_ACTION_GRAPH function in Algorithm 3 (given in Appendix B, with complexity given in Appendix E).

Once again, we return to the airline ticket purchase example to clarify the workings of this algorithm. Here, a new node in the graph is created for the new action. Then, the ACTION_TABLE is searched for the row representing the user's most recent action upon which the current action is dependent (according to the *ScriptList* dependency information), and the new node is set to point to the node on which it is dependent. Here, the gateway finds that action A_8 (itinerary selection) is dependent on action A_2 (plan entry) (since `select_itinerary.jsp` is dependent on `submit_travel_plan.jsp`). Accordingly, an edge

3. Note that adding this information to the HTTP header does no harm. Applications look for the header directives they require; extra header directives are simply ignored. The *ScriptList* directive would likely have wider application beyond that described in this paper; thus, it might be worthwhile to propose it as an extension to HTTP.

(A_8, A_2) is added to the failover map, indicating this dependency. In this manner, the gateway will create a DCAG for each user's **iTX**.

The first step in the recovery protocol following the failure of an **iTX** is identifying the user's most recently completed action. We discuss this next.

5.3 Identifying the User's "Most Recently Completed Action"

One question we must answer in order to discuss our recovery protocols is: *What was the "most recently completed action," i.e., the action with respect to which valid states are computed and recovery is undertaken?* We define this "most recently completed action" as follows: If failure occurs before a request for an action is sent (and, hence, logged) by the gateway, the "most recently completed action" is the one immediately prior to this request. If it happens after the request is sent by the gateway, then this action is the "most recently completed action" and it is with respect to this action that recovery is undertaken. This implies that recovery can be undertaken only after there is a response to this pending action. As with the wired HTTP calls, we assume that timeouts are associated with requests for actions and, so, when sites eventually respond or timeouts occur, recovery from disconnections can be undertaken.

5.4 Using the Logs and Failover Maps to Recover User iTXs?

We just discussed how to identify the "most recently completed action." This identification is done as the first step in recovery following the failure of an **iTX**. With this action identified, we can recover a valid state for the user during recovery by following these steps:

1. **Determine the active sub-iTX**: When an **iTX** fails, the user may have already initiated multiple **sub-iTXs** in his **iTX**. The goal of our recovery protocol is to return the user to a state from which he can continue. Here, we assume that the active **sub-iTX** at the time of failure is the **sub-iTX** in which the user was involved in the most recently completed action. (Alternatively, the user may be offered a selection of **sub-iTXs** from which recovery can begin. This is a trivial modification since both mechanisms identify a **sub-iTX** from which recovery should proceed.)
2. **Check validity of the saved states within the active sub-iTX**: We wish to ensure that the response served back to a user is a valid one. Specifically, the recovery protocol should, in this step, choose *the most recent valid state for which all preceding states in the* **sub-iTX** *are valid*. To see why all preceding states must be valid, we consider the following example, based on our running airline reservation system example. Consider a situation in which a user making a reservation over a mobile phone selects a flight at time t_1. Suppose the site assigns a time-to-live value of 10 minutes to this response (which offers seats on the selected flight), such that the response has a valid period of $t_1 + 10$. Now suppose that the user receives a phone call, forcing him to put the reservation connection on hold to take the call, which lasts for nine minutes. Upon returning to the reservation system, the user selects a seat, at time t_2, where $t_2 = t_1 + 9$. In response, the site sends a confirmation of the selected seat, which has a time-to-live of 10 minutes. Now, suppose that, two minutes later (i.e., at time $t_1 + 11$), as he is filling out credit card payment information, the user's connection fails. Here, the system has two choices for recovery, the seat-selection page (i.e., the response from the itinerary-selection action) and the payment page (i.e., the response from the seat-selection action). However, the failure time $t_1 + 11$ is greater than the valid time of the response $t_1 + 10$ from the itinerary selection action, indicating that the response from the itinerary selection action is no longer valid. Since the seat-confirmation/payment page is dependent on the validity of the previous page, *even though it may itself be valid, it cannot be served because it is dependent on an invalid response*.
3. **Rebuild the user's state for recovery**: A user's recovery state consists of all the state information (e.g., cookies) from the component actions on which the recovery action is dependent. We can find these component actions by traversing the DCAG from the recovery state to all the nodes reachable from it and gathering the cookie information from the corresponding rows in the ACTION_TABLE.
4. **Replay the recovery state**: Once the appropriate recovery state is chosen and rebuilt, the corresponding HTTP response from the ACTION_TABLE is sent to the user.

The recovery protocol is presented as the RECOVER_USER function in Algorithm 4 (given in Appendix C, with complexity given in Appendix E).

Once again, we return to the airline example, in which our user has logged in (action A_1), submitted his travel plan (action A_2), and selected an itinerary (action A_8). He now wished to select his seat. Suppose, that at this point, the user's connection drops. Upon his reconnection, the gateway will call the RECOVER_USER function. The recovery protocol will identify the user's most recent HTTP request in the ACTION_TABLE (i.e., the row with the maximum timestamp for the user). This row contains the user's most recent action (in our example, this is action A_8, selecting an itinerary). Assuming this action, as well as those corresponding to his previous actions (i.e., A_1 (login) and A_2 (plan entry)), to have a valid state, the gateway will replay the corresponding HTTP response, placing the user at the point from which he can select a seat on his preferred flight. If however, the timeout period for his itinerary selection action has elapsed, but those for the login and travel plan submission actions have not, then the recovery protocol will recover the user only to the HTTP response for action A_2 (plan entry), from which he will again be able to select an itinerary.

To reduce the space requirements of the ACTION_TABLE, periodically the system runs a garbage collection procedure to remove user states that have expired from the ACTION_TABLE and the DCAG. This can be accomplished with the

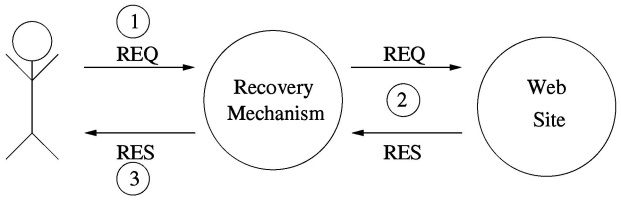


Fig. 5. Failure cases.

GARBAGE_COLLECTION function shown in Algorithm 5 (given in Appendix D).

6 Expected Behavior in the Case of Failure

We will consider connection failure in the context of our online airline reservation example from Section 3. Suppose that our user has entered all his ticket-payment information, e.g., credit card information, and has selected the "confirm purchase" button, which submits his payment information to the airline site. In between submitting his request and receiving the site's response, his connection fails. (This is perhaps the most interesting example from a user's point of view, since pressing the "confirm purchase" button is a "point of no return" from the user's point of view.) At this point, we can ask the following question: Has the user purchased the ticket? That is, is his purchase action completed? We consider this question in light of the possible failure cases, described below.

There are three general failure cases, depicted graphically in Fig. 5, possible in the scenario described in this paper:

1. The user's connection fails after he has submitted a request containing his payment information, but before that request reaches the gateway and is logged. In this case, the user's request containing his payment information is never logged at the gateway. Here, the recovery protocol aborts the action in progress and performs recovery assuming that the most recently completed action was seat selection.
2. Once the gateway logs a request from a client, two outcomes are possible: either a) the request will reach the server and be processed; or b) the server fails after receiving the request, but before processing the response. In case (2a), the gateway will receive the response and process it normally.

   Case (2b) is more challenging, in that there is no good way for the gateway to "sense" that the server has failed. The tricky problem here is that there is no way for the gateway to distinguish between the "server slow" condition (in which the server will eventually respond) and "server failed" condition (in which the server will not respond), while it is waiting for a response. Here, we are interested in the period before the HTTP request times out. Once the timeout period has elapsed, the gateway will consider the request to have failed, whether the server has actually failed or not. Note that system behavior in the timeout case is the same for users of both wired and wireless connections—in both these scenarios, reaching timeout results in a failed request. Thus, in case (2b), even if a disconnection occurs between the client and the gateway *before* the site's response is logged at the gateway, either the response will arrive at the gateway before timeout, at which point it will be logged, or the request will time out, at which point the request will be considered to have failed. Should recovery be attempted before the response is received or the timeout period has not yet fully elapsed, the gateway will simply wait for the response or timeout, which is essentially the same behavior the end user would observe had a connection failure not occurred. If the gateway receives a response, the gateway would simply forward it to the client. If, alternatively, the request times out, the gateway will forward a timeout error to the client.
3. The user's connection fails after the gateway has received the response with the purchase confirmation, but before the user receives the response. In this case, both the request containing the user's payment information, as well as the response containing the site's confirmation of payment, have been logged at the gateway. Here, the action is in fact complete, and so this action is identified as the "most recently completed action." (Note that in the case of the ticket purchase action, the response never becomes invalid, and, so, the recovery system replays the purchase confirmation (sent by the site) to the user (and logged at the gateway).)

7 Conclusion

With the expansion of Web sites to include business functions, a user interfaces with e-businesses through an interactive and multistep process, which is often time-consuming. At the same time, mobile Internet access is becoming more common. In mobile environments, a loss of connection, or any other system failure, can result in the loss of work accomplished prior to the disruption. This work must then be repeated upon subsequent reconnection—often at significant cost in time and computation for both the Web site and the mobile user. In some environments, such as wireless scenarios, this "disconnection-reconnection-repeat work" cycle may cause Web clients to incur substantial monetary as well as resource (such as battery power) costs as well. In this paper, we proposed a protocol for "recovering" a user to an appropriate recent interaction state after such a failure. The objective was to minimize work that needs to be redone upon restart after failure.

Whereas classical database recovery focuses on recovering the system, i.e., all transactions, our work considers the problem of recovering a particular user interaction with the system. This challenging recovery problem encompasses several interesting subproblems:

a. modeling user interaction in a way that is useful for recovery,
b. characterizing a user's "recovery state,"

c. determining the state to which a user should be recovered, and
d. defining a recovery mechanism.

To address these, we began with two foundation notions: a) **iTX**s, which are sequences of user actions with one or more Web sites, and b) user-site interaction states, which represent the information exchanged between the user and a Web site during the course of a single user action on the site.

We presented the properties of **iTXs** as well as the features required of an **iTX** recovery scheme. We then developed a graph model of an **iTX** for use as a *failover map* in our recovery protocol. For the user-site interaction model, we developed the notion of a *user recovery state* based on the observable information in HTTP requests and responses. Our recovery protocol is based on these models. Our recovery protocol logs user recovery states and continually updates users' failover maps. Upon failure, the recovery mechanism, located at the gateway—for reasons discussed in the paper, consults the log and the failover map to find the appropriate recovery state in terms of state validity and user objective, and recovers the user to this state.

APPENDIX

Appendices A through G are available as supplemental materials in the Digital Library at http://computer.org/tmc/archives.htm.

ACKNOWLEDGMENTS

An early version of this work appeared in the Proceedings of the Second International Workshop on Technologies for E-Services (TES 2001).

REFERENCES

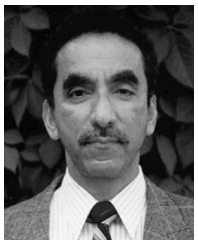
[1] R. Agrawal and D.J. DeWitt, "Integrated Concurrency Control and Recovery Mechanisms: Design and Performance Evaluation," *ACM Trans. Database Systems,* vol. 10, no. 4, pp. 529-564, 1985.
[2] G. Alonso, D. Agrawal, A. El Abbadi, M. Kamath, R. Gunthor, and C. Mohan, "Advanced Transaction Models in Workflow Contexts," *Proc. Int'l Conf. Data Eng.,* pp. 574-581, 1996.
[3] R. Barga and D. Lomet, "Measuring and Optimizing a System for Persistent Database Sessions," *Proc. Int'l Conf. Data Eng.,* pp. 21-30, Apr. 2001.
[4] P.K. Chrysanthis and K. Ramamritham, "Acta: A Framework for Specifying and Reasoning about Transaction Structure and Behavior," *Proc. 1990 ACM SIGMOD Conf.,* 1990.
[5] T.H. Cormen, C.E. Leiserson, and R.L. Rivest, *Introduction to Algorithms.* The MIT Press, 1998.
[6] Nokia Corp, Nokia Wap Server, http://www.nokia.com/wap/development.html, 2001.
[7] Microsoft Corporation, Active Server Pages Tutorial, http://msdn.microsoft.com/workshop/server/asp/asptutorial.asp, 2001.
[8] Microsoft Corporation, Internet Information Server, http://www.microsoft.com/windows2000/server/evaluation/features/Web.asp, 2001.
[9] Microsoft Corporation, Visual Interdev, http://msdn.microsoft.com/vinterdev/, 2001.
[10] Netscape Communications Support Documentation, Persistent Client State Http Cookies, http://home.netscape.com/newsref/std/cookie_spec.html, 2001.
[11] R. Fielding, J. Gettys, H. Frystyk, L. Masinter, P. Leach, and T. Berners-Lee, "Hypertext Transfer Protocol—http 1.1," Available via FTP: ftp://ftp.isi.edu/in-notes/rfc2616.txt, 1999.
[12] H. Garcia-Molina and K. Salem, "Sagas," *Proc. 1987 ACM SIGMOD Conf.,* 1987.
[13] D. Georgakopoulos, M.F. Hornick, and A.P. Sheth, "An Overview of Workflow Management: From Process Modeling to Workflow Automation Infrastructure," *Distributed and Parallel Databases,* vol. 3, no. 2, pp. 119-153, 1995.
[14] T. Harder and A. Reuter, "Principles of Transaction-Oriented Database Recovery," *ACM Computing Surveys,* vol. 15, no. 4, pp. 287-317, 1983.
[15] Sun Microsystems, Java Server Pages, http://java.sun.com/products/jsp/, 2001.
[16] C. Mohan, D. Haderle, B. Lindsay, H. Pirahesh, and P. Schwartz, "Aries: A Transaction Recovery Method Supporting Fine-Granularity Locking and Partial Rollbacks Using Write-Ahead Logging," *ACM Trans. Database Systems,* vol. 17, no. 1, pp. 94-162, 1992.
[17] N. Neves and W.K. Fuchs, "Adaptive Recovery for Mobile Environments," *Comm. ACM,* vol. 40, no. 1, pp. 68-74, 1997.
[18] C. Pedregal-Martin and K. Ramamritham, "Guaranteeing Recoverability in Electronic Commerce," *Proc. Workshop Advanced Issues in Electronic Commerce and Web-Based Information Systems,* 2001.
[19] A.P. Sheth and M. Rusinkiewicz, "On Transactional Workflows," *Data Eng. Bull.,* vol. 16, no. 2, pp. 37-40, 1993.
[20] BEA Systems, Weblogic Server, http://www.bea.com/products/Weblogic/server/index.shtml, 2001.
[21] J.D. Tygar, "Atomicity Versus Anonymity: Distributed Transactions for Electronic Commerce," *Proc. 24th Int'l Conf. Very Large Data Bases,* pp. 1-12, 1998.
[22] U. Varshney and R. Vetter, "Emerging Mobile and Wireless Networks," *Comm. ACM,* vol. 43, no. 6, pp. 73-81, 2000.
[23] G.D. Walborn and P.K. Chrysanthis, "Pro-Motion: Management of Mobile Transactions," *Proc. Symp. Applied Computing,* pp. 101-108, 1997.

Debra VanderMeer received the BS degree from the Georgetown University and the MS degree in management information systems from the University of Arizona. She is a doctoral candidate in computer science at the Georgia Institute of Technology. Her research interests include performance and scalability improvement in e-commerce systems, with particular emphasis on data-intensive systems, as well as the design of mobile data access systems. She is also the director of technical services for Chutney Technologies, a software company that develops solutions to improve the scalability and performance of enterprise Web applications. She has considerable experience in the design and development of commercial software systems. Prior to commencing her doctoral studies, she worked for Tandem Computers on the development of public key infrastructure software. She has published articles in well-known computer science journals, such as the *ACM Transactions on Database Systems*, *IEEE Transactions on Knowledge and Data Engineering*, and the *VLDB Journal*. She is a student member of the IEEE and the IEEE Computer Society.

Anindya Datta is an associate professor at the Georgia Institute of Technology and founder of the iXL Center for Electronic Commerce. Previousy, he was an assistant professor at the University of Arizona, after finishing his doctoral studies at the University of Maryland, College Park. Dr. Datta's undergraduate education was completed at the Indian Institute of Technology, Kharagpur. His primary research interests lie in studying technologies that have the potential to significantly impact the automated processing of organizational information. Examples of such technologies include electronic commerce, data warehousing/OLAP, and workflow systems. He has published more than 50 papers in prestigious refereed journals such as the *ACM Transactions on Database Systems*, *IEEE Transactions on Knowledge and Data Engineering*, *INFORMS Journal of Computing*, and the *VLDB Journal*, and in reputed conferences such as ACM SIGMOD, and VLDB. He has also chaired as well as served on the program committees of reputed international conferences and workshops. Dr. Datta is also the CEO and founder of a venture-backed Internet infrastructure startup called Chutney Technologies, and is regarded as an industry authority on Web acceleration. Chutney's flagship product, the PreLoader, is part of an emerging category of solutions called Web Application Optimization, which allows enterprise applications to scale to support significantly higher user loads and improves application performance substantially. Chutney is defining and leading the Web Application Optimization space, which analysts claim is poised to significantly impact Web and application server scalability. As one of the creators of this software space, Dr. Datta is frequently invited to speak at academic and industry meetings. Prior to founding Chutney, Dr. Datta has had extensive experience leading teams in the development and implementation of large-scale database systems, including a large commercial project for the USDA. He has served as a consultant for AT&T, US West, IBM, and the Israeli government in the fields of data warehousing, data mining, and e-commerce. A substantial contributor to several innovations, Dr. Datta holds numerous patents for a variety of data management and Internet technologies. One of his most recent contributions was in developing technologies similar to those incorporated by IBM in the DB2 product for the AS/400 platform. He has also worked on broadcast technologies for mobile users and the access security of subscription-based broadcast information services. He is a member of the IEEE Computer Society.

Kaushik Dutta received the BS degree in electrical engineering from the Jadavpur University, India, and the MS degree in computer science from the Indian Statistical Institute. He is a doctoral candidate in the Dupree College of Management at the Georgia Institute of Technology. His research interests include performance and scalability improvement in e-commerce systems, with particular emphasis on data-intensive systems, as well as the design of mobile data access systems. He is also the chief architect for Chutney Technologies, a software company that develops solutions to improve the scalability and performance of enterprise Web applications. Dr. Dutta has considerable experience in the design and development of commercial software systems. Prior to commencing his doctoral studies, he worked for Intarka Inc., a startup funded by NEA venture, on development of e-commerce products. He has published articles in well-known computer science and information systems journals, such as *VLDB Journal*, *Management Science*, and *IEEE Internet Computing*. He is a student member of the IEEE Computer Society.

Krithi Ramamritham received the PhD degree in computer science from the University of Utah and then joined the University of Massachusetts. He is currently at the Indian Institute of Technology, Bombay, as the Vijay and Sita Vashee Chair Professor in the Department of Computer Science and Engineering. He was a Science and Engineering Research Council (UK) visiting fellow at the University of Newcastle upon Tyne, UK, and has held visiting positions at the Technical University of Vienna, Austria, and at the Indian Institute of Technology, Madras. Dr. Ramamritham's interests span the areas of real-time systems and data-based systems. He is applying concepts from these areas to solve problems in mobile computing, e-commerce, intelligent internet, and the Web. Prof. Ramamritham is a fellow of the IEEE, the IEEE Computer Society, and the ACM. His conference chairing duties include the Real-Time Systems Symposium—as program chair in 1994 and as general chair in 1995, the Conference on Data Engineering—as a vice-chair in 1995 and 2001 and as a program chair in 2003, and the Conference on Management of Data—as program chair in 2000. He has also served on numerous program committees of conferences and workshops. His editorial board contributions include *IEEE Transactions on Knowledge and Data Engineering*, *IEEE Transactions on Parallel and Distributed Systems*, *IEEE Internet Computing*, the *Real-Time Systems Journal*, the *WWW Journal*, the *VLDB Journal*, and ACM SIGMOD's *Digital Review*. He has coauthored two IEEE tutorial texts on real-time systems, a text on advances in database transaction processing, and a text on scheduling in real-time systems.



Shamkant B. Navathe received the PhD degree from the University of Michigan. He is currently a professor at the College of Computing, Georgia Institute of Technology in Atlanta. He is known for his work on database modeling, mapping and design, distributed database design, design tools, engineering data management, and database integration. His most recent work involves biological human genome applications, data warehousing and mining, document retrieval and text mining, electronic commerce applications, information security, and intermittently synchronized and mobile databases. He has been an associate editor of *IEEE Transactions on Knowledge and Data Engineering* (1994-1998) and *ACM Computing Surveys* (1986-1996), and is on the editorial boards of *Information Systems* (Pergammon), *Distributed and Parallel Databases*, and the *WWW Journal* (Kluwer), *Data and Knowledge Engineering* (Elsevier), and *Information Technology and Management* (Chapman and Hall). He was an elected member of the VLDB foundation (1992-1998), the program chairman of the 1985 ACM-SIGMOD Conference, and a general chair of the 1996 VLDB Conference in Bombay, India. He is a coauthor of the leading text, *Fundamentals of Database Systems*, with Ramez Elmasri (Addison Wesley, edition 4, 2004), and has also coauthored *Conceptual Database Design: An ER Approach* with Carlo Batini and Stefano Ceri (1992). Prior to his current position, Dr. Navathe was with the Database Systems Research and Development Center at the University of Florida in Gainesville (1979-1990). He has worked with IBM and Siemens in their research divisions and has been a consultant to various companies, including Honeywell, Nixdorf, CCA, ADR, Digital, MCC, Equifax, and Harris corporations, and has written more than 125 refereed publications. He is a member of the IEEE and the IEEE Computer Society.

▷ **For more information on this or any computing topic, please visit our Digital Library at** http://computer.org/publications/dlib.