User Action Recovery in Internet SAGAs (iSAGAs)

Kaushik Dutta[1], Debra VanderMeer[1], Anindya Datta[1], and Krithi Ramamritham[2]

[1] Georgia Institute of Technology
Atlanta, GA
{kaushik, deb, adatta}@chutneytech.com
[2] University of Massachusetts-Amherst and IIT-Bombay
krithi@cs.umass.edu

Abstract. With the expansion of Web sites to include business functions, a user's typical interaction with a Web site is a multi-step process. The loss of connection, or other system failure, can result in the loss of work accumulated before the disruption. This work must then be repeated – often at significant cost in time and computation. In some environments, such as wireless scenarios, these costs can include battery power as well.

This paper presents a protocol for recovering from such scenarios. We first describe Internet-based interactions using the familiar concepts of transactions and recovery. Using these abstractions, we then describe a protocol for recovering users to valid states in Internet interactions.

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 restarting the sequence, often at significant expense to both the user and the Web site.

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.

If his wireless connection drops (as occurs frequently in wireless systems [17]) during the purchase step, he must reconnect to the Internet, re-login to the airline site and resubmit all the pertinent information in order to reach the state where he can again attempt to purchase a ticket. Restarting this sequence costs the user time and battery power, both valuable commodities. Resources are also wasted on the server, affecting performance, since (potentially) large amounts of work must be redone.

The main problem in this scenario is that the user must restart the sequence of actions from the beginning, including expensive steps, such as processor-intensive login and authentication steps, I/O intensive database lookups, as well as consume valuable connections to the database. Significant amounts of time and effort are wasted in redoing previously completed work, and, in the case of a wireless environment, there are additional costs in battery resources and airtime.

The purpose of this paper is twofold: (a) to propose ways for users to "recover" from interaction failures by minimizing the amount of work that must be redone upon reconnection; and (b) to demonstrate that this user "recovery" problem differs significantly from the classical recovery problem in database systems. Here, rather than redoing the entire sequence of steps completed before failure, a user would, upon restart, be placed at a much later stage in his sequence of actions (ideally the last action before failure). This problem encompasses several interesting subproblems:

How can we characterize a sequence of user actions with a site? A user interacts with a Web site through a sequence of actions aimed at achieving a specific goal. Upon preliminary considerations, this appears to resemble a long transaction (such as a SAGA [8]). Thus, we model user interaction with a Web site using the long transaction metaphor, and refer to a user's action sequence as an *Internet SAGA (***iSAGA***)*. However, there are significant differences between long database transactions and **iSAGAs**, due to the differences between the highly constrained context of a database system and the loosely constrained context of the Internet. These differences require a rethinking of the basic properties of long transactions in the context of the Internet. We describe these differences in detail in Section 3.

Given the **iSAGA** *characterization, what is a good protocol for allowing users to "recover"?* Given a means of describing user interaction with a Web site as a long transaction, we need a protocol for actually recovering a user to a previous state after a disruption. The main question is: How can we describe user states in the context of Internet long transactions? We present such a method in Section 4.

To which state should a user be recovered? Given a set of user states produced by a user's **iSAGA**, how can we decide which state to use for recovery? Ideally, we would simply choose the user's last state before failure. However, this is not always possible. For instance, the data in the underlying database may have changed during the time between failure and reconnection. Thus, user states may become invalid.

Another complexity that arises is the potential for multiple objectives in an **iSAGA**. Here, the sequence of user actions as seen by the server may not correspond a linear sequence of actions toward a single objective, but rather may consist of multiple interleaved objectives. Consider, for instance, the airline ticket purchase example. Here, before purchasing a ticket, our user might check his frequent flier mileage. Effectively, in the same interaction session, the user is performing two tasks: (a) mileage checking, and (b) ticket purchase. The mileage-checking action is not directly related to the ticket-buying action. Thus, our user might express, in the same **iSAGA**, more than one objective. We present a detailed discussion of these complexities in Section 5.

Given that a user might express multiple objectives, which objective should be considered by the recovery protocol? Given that our user may have more than one objective, which sequence of actions should we consider when recovering from a failure? We expand our recovery protocol in Section 6 to encompass this possibility.

To summarize, the objectives of this paper are as follows:

1. To model an **iSAGA**, i.e., a user's interaction with a Web site.
2. To demonstrate that **iSAGAs** give rise to issues not relevant in the context of database transactions.
3. To describe how to determine valid states in an **iSAGA**.
4. To find ways of extracting different user objectives and the corresponding sequences of user actions from a single **iSAGA**.
5. To propose a user action recovery technique to handle failure.

The remainder of this paper is organized as follows. Section 2 describes related work. Section 3 presents our model of Internet interactions using a transaction metaphor, and distinguishes **iSAGAs** from SAGAs. Section 4 describes a simple recovery protocol that illustrates the basic ideas of recovering a user's state. Section 5 expands the notion of an Internet transaction to model multiple user goals in a single transaction. Section 6 expands the recovery protocol to include recovery in the context of multiple user goals. 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 [10] describe the basic tenets of transaction processing and compare the performance of transaction processing protocols. In [12], the authors describe a comprehensive recovery protocol for database systems.

The metaphor of transaction processing has since been extended to other application areas. The ACTA framework [3] describes a formal language for extending the idea of a transaction to other areas. For example, [2] describes workflow processing using transactions, while [18] utilizes transactions to model mobile interactions. The notion of transactions has even been applied to electronic commerce, as in [15] and [16].

The works cited above all have one idea in common: the activity modeled as a transaction is a self-contained activity, with a single goal. This paper describes a user's interaction with a Web site using the metaphor of a transaction, while relaxing the restriction that the transaction model an interaction with a single goal. For this, we draw on the idea of a long transactions [8] and [3].

We also draw on work in recovery. Database recovery is described clearly in [12]. In [13], the authors describe a protocol for mobile recovery. Here, the authors consider basic transaction recovery; our work differs from this in that we consider the validity of stored user states, as well as multi-objective transactions.

3 Internet SAGAs (iSAGAs)

The ideas presented in this work revolve around the notion of an **iSAGA**. Thus, we first define formally what an **iSAGA** is. We then discuss how users interact with Web sites using the notion of an **iSAGA**. Finally, we discuss the ways in which **iSAGAs** differ from SAGAs.

3.1 What is an iSAGA?

Definition 1. *An* **iSAGA** *I is a sequence of user actions $\langle A_1, A_2, \ldots, A_n\rangle$ (called* component actions*) on a particular Web site to achieve at least one, possibly multiple, user objectives.*

In order to make clear these notions, as well as others in this paper, we now describe an example scenario. We utilize this scenario as a running example through the remainder of this paper to illustrate various properties of **iSAGAs** as well as our proposed recovery protocol.

Example of an iSAGA Consider an online airline ticket purchase scenario. Here, our example user logs on to the airline site with his frequent flier number, enters his desired travel plan, and checks his frequent flier mileage. He then returns to the reservation page, and re-enters his travel plan, chooses among the available itineraries, chooses a seat, and submits credit card information to complete the purchase. This **iSAGA** I consists of a sequence of several component actions: the login component action A_1, the plan-entry action A_2, the mileage-checking action A_3, the return-to-reservations action A_4, the plan-reentry action A_5, the itinerary-choosing action A_6, the seat-choosing action A_7, and the ticket-payment action A_8. More formally, we denote the **iSAGA** I as the sequence $\langle A_1, A_2, A_3, A_4, A_5, A_6, A_7, A_8\rangle$, as shown below:

I: A_1 (login);
A_2 (travel plan entry);
A_3 (mileage check);
A_4 (return to reservation form);
A_5 (repeat travel plan entry);
A_6 (itinerary choice);
A_7 (seat selection);
A_8 (ticket payment);

It is worthwhile to note the interleaving of different objectives in the example above. Here, the user reveals two separate objectives: (a) checking his frequent flier mileage, and (b) purchasing a ticket. Each of these objectives can be represented by a (possibly overlapping) sequence of component actions called a **sub-iSAGA**.

Definition 2. *Consider an* **iSAGA** $I = \langle A_1 \dots A_n \rangle$. *A* **sub-iSAGA** B *of an* **iSAGA** I *consists of a sequence of component actions* $\langle A_j, A_k, \dots, A_m \rangle$, *meeting the following conditions: (a) each component action of* B *is a component action in* I; *(b) 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; *and (c)* B *semantically represents a single user objective.*

For instance, the **sub-iSAGA** representing the mileage-checking objective consists of the sequence $\langle A_1, A_3 \rangle$, while the **sub-iSAGA** for the ticket-purchase objective consists of the sequence $\langle A_1, A_2, A_4, A_5, A_6, A_7, A_8 \rangle$. Clearly, **sub-iSAGAs** can be arbitrarily interleaved within a single **iSAGA**.

3.2 Web Site and User Views of an iSAGA

The essential intuition needed to understand **iSAGAs** is that the user and the Web site perceive user actions differently. Let us first consider how the site perceives user actions. Effectively, the Web site observes a sequence of HTTP requests coming from the user. Since the HTTP protocol is stateless[7], *the Web site sees each user action as an independent task, rather than as part of a larger transaction*. For instance, in the **iSAGA** of Section 3.1, the Web site has no knowledge that the seat-choosing action A_7 is logically followed by the ticket-payment action A_8. Rather, it observes A_7 and A_8 as independent transactions. Consequently, both before and after each component action of an **iSAGA**, the state of the Web site components (e.g., the associated database systems) are all consistent.

From the perspective of the user, however, each component action is part of a sequence (i.e., a transaction) aimed at reaching a specific goal. To achieve his goal, the user must complete the **iSAGA** entirely; partial executions provide no benefit. Consider the airline example in Section 3.1. After completing the sequence of component actions $\langle A_1 \dots A_8 \rangle$, the ticket-buying **iSAGA**, is complete.

iSAGAs differ significantly from long database transactions (which, generically, we will henceforth refer to as SAGAs) in several ways: (a) evolutionary growth; (b) view updatability and validity; and (c) potential for multiple objectives. We discuss each in turn.

3.3 Evolutionary Growth

An **iSAGA** is created through user interaction, one component action at a time, and lack the notion of a predefined sequence and well-defined end. Since each of the component actions of an **iSAGA** is an individual transaction from the perspective of Web sites the user interacts with, the Web site must treat each user action as if it were the last in the user's **iSAGA**. The user's **iSAGA** ends if the user *explicitly* disconnects, e.g., by closing the browser, or *implicitly* disconnects, e.g., if his connection fails.

Even if the user's **iSAGA** ends unexpectedly, the database states of the Web sites he interacts with, as well as his own user state, remain consistent with (i.e., they will continue to reflect) the updates, if any, made by the actions completed up to the point of failure. Since the iSAGA has terminated, any needed compensatory actions should be implicit in the Web site, i.e., compensation must be (and in practice, is) built into the site's business logic where required. That is, *no explicit compensating transaction is required from the user*, i.e., the user need take no action to ensure the web site state is consistent after failure recovery. We illustrate this via an example.

Consider the airline ticket purchase **iSAGA** in Section 3.1. Suppose that our user has completed component action A_7 (seat-choice), and that his connection drops while he is in the process of action A_8 (payment-submission). Here, the database at the Web site has allotted a seat for the user on his preferred flight, but the corresponding ticket has not been purchased. Moreover, since the user's connection, and therefore his session on the Web site, has been lost, he *cannot* purchase the ticket for the seat he has reserved, since the site (most likely) persisted his session with a cookie, which disappeared on disconnection. Had the site used a disk-based cookie, to persist his session, he would likely be able to re-establish his session upon reconnection. However, most sites refrain from this practice, since it would require writing the cookie to disk with every user click, adding I/O and storage overhead on the user's browser.

The airline site has a mechanism to handle such scenarios; here, the user's seat selection is simply a reservation for the seat selected, a reservation that expires after some site-specific amount of time if the ticket corresponding to the reservation is not purchased. The important point to understand here is that Web sites (and their associated database systems) are designed to handle such scenarios, and *no explicit action is required in the interaction between the user and the site to handle such "failures"*. Note that such a design also makes practical sense: An airline reservation site can not rely on a customer to explicitly cancel/undo a reservation he/she made, but chose not to purchase.

The implication of this is subtle, but extremely important: Any failure will be handled by the recovery mechanisms on the database and application servers, rather than requiring explicit compensating action(s) from the user interaction. In other words, a failure does not result in inconsistency in an **iSAGA**, from both the Web site's view as well as the user's. Rather, a failure results in the potential need to redo work (were the user to reconnect to the site), wasting resources. Thus, in this work *we need not worry about database or application server transaction recovery*; rather, we *focus on recovering a user's state*, consistent with the current state of the database at the time of recovery, mitigating the need to restart the sequence of actions in the **iSAGA** from the beginning.

3.4 View Updatability and Validity

Another way in which **iSAGAs** differ from SAGAs is that views in **iSAGAs** can be *updated* at any point in the **iSAGA**, while SAGAs operate in a more restricted view update context. These updates can cause an **iSAGAs** view to become *invalid*. We first describe how views can be updated, and then discuss the consequent invalidation effect. By the term "view", we refer to the classical notion of a tranasaction's view of a system state, i.e., *"the objects and the state of the objects visible to a transaction at a point in time."* [4]

View Updatability In a SAGA, as in traditional database transactions, a transaction's view is *protected* by some type of concurrency control mechanism (such as a locking protocol) to prevent two transactions from writing over one another's updates. Whenever a view is given to a transaction, it is the database system's responsibility to lock (or otherwise control read/write access to) the appropriate portion of the database. In this way, database systems guarantee that the view given to a transaction remains valid until the end of the transaction. Such protection, however comes at the cost of increased response time. In typical long database transactions, such as CAD design, the relatively low user load (perhaps tens to a few hundred users), and the lack of a requirement for real-time response times make these penalties tolerable.

In the Internet environment, locking (or otherwise controlling access to) part of the Web site is infeasible - doing so would severely impact the performance of the transactions of other users on the site. Here, the site database may be accessed to serve many thousands of users simultaneously, and these users expect real-time responsiveness from the site. In this context, concurrency control mechanisms would place an unbearable burden on the site. Thus, an **iSAGA**'s view is *unprotected*, i.e., the view is open for updates.

For example, return to our airline **iSAGA**. In response to action A_2 (the travel-plan entry action), the airline site offers several itineraries to a user, based on his input preferences. It is not possible to place concurrency control restrictions on those itineraries (which have been offered as a user view on the database) because those same itineraries may be of interest to other users on the site, and controlling access would limit the number of users who can access the information.

Given system loads and response time expectations, concurrency control across component actions of **iSAGAs** is simply not a valid solution. Moreover, since each component action is viewed as an independent HTTP request by the Web site, it is not feasible to control access to particular data items.

View Validity One consequence of view updatability is that **iSAGA** views may become *invalid*. However, database are not the only cause of view invalidity; a view can also become invalid when it no longer has value to the user. Clearly, view invalidity can occur in one of two ways, either from the user's perspective or the site's perspective. We discuss each in turn.

User-based invalidation A view becomes invalid from the user's perspective when the user no longer needs the data in the view. Consider again the airline **iSAGA** described in Section 3.1. Suppose that our user's **iSAGA** fails during action A_6 (itinerary-selection). Further suppose that our user then (possibly frustrated) called the airline's toll-free number to purchase his ticket. The next time he connects to the Internet (perhaps several hours later), in a system equipped to recover after a failure, he might be offered itineraries matching the travel plans he entered before losing his connection. Clearly, the user's state at disconnection is no longer useful. Thus, we must consider mechanisms for determining the value of user's state at the time of recovery.

System-based invalidation A view becomes invalid from the site's perspective when the data in a view provided to a user has changed. Let us return to the airline **iSAGA** example. Suppose that our user's **iSAGA** fails in the middle of action A_6 (choosing the itinerary) and that, while he reconnects, the site sells out of seats at the user's preferred fare on his preferred itinerary. 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.

Both types of validity described above must be considered in helping a user recover from failure. In simple cases, invalidating user-recovery states can be achieved by limiting the temporal validity period of a state through the use of a "time-to-live" (TTL) mechanism. Interestingly, this works in both cases, i.e., for both user-based as well as system-based invalidation.

In the case of user-based invalidation, it is fairly safe to assume that the value of recovery to the user decreases as the time between failure and reconnection increases. For instance, consider the airline **iSAGA**. If the user connection fails after seat selection action A_7, i.e., during the payment phase A_8, and that the user turns to the airline's telephone reservation system to purchase his ticket. The next time our user visits the airline site, a large amount of time is likely to have passed (in comparison to the amount of time that would have elapsed, had the user reconnected immediately), sufficient time for the site to assume that the user is no longer interested in recovery. Thus, a site can set a TTL on user recovery states based on the average time to reconnect among users utilizing the recovery feature. For instance, suppose Web logs show that 90% of recovered users reconnect within five minutes after a failure. Here, a TTL on user states could confidently be set at five minutes – users reconnecting after failure would have a high likelihood of recovery.

In the case of system-based invalidation, a TTL value for a view can be set based on the minimum of average times to update a specific data items in a view. For instance, if the fares in the database are updated every hour, the site could confidently set a TTL value for database views of 10 minutes. While this does not guarantee absolute correctness, sites are typically designed to handle transaction failures that occur when the underlying data values in a user's view have changed. If absolute correctness is required, event-driven invalidation, described next, can be used.

In the more general case, invalidation may be event driven. For example, consider the airline reservation system, and suppose that the fare offered to a user is dependent on the fare classes available for a given flight on a particular day (thus, a user would expect to see some variation in fares over time). Here, if a fare in the database is changed, the views using that fare data become invalid, and should no longer be used to serve user requests. This invalidation can be achieved using active database mechanisms, such as triggers.

3.5 Potential for Multiple Objectives

In database transactions, which represent specific goals, all the actions in a transaction work toward the same goal. Consider the example **iSAGA** in Section 3.1. Here, our user reveals two separate objectives: the mileage-checking objective and the ticket-buying objective, each represented by its own **sub-iSAGA**. Note that only one of these **sub-iSAGAs** can be active at any given time. For example, at component action A_3 (mileage-checking), the user is clearly active in the mileage-check **sub-iSAGA**, and not the ticket-purchase **sub-iSAGA**.

In a multi-objective **iSAGA**, given that the user may be hopping arbitrarily between different objectives, the possibility of failure in any one of the **sub-iSAGAs** of an **iSAGA** gives rise to an important question: to which objective should the user be recovered? Our solution is to postulate that the user should be returned to the objective (i.e., the **sub-iSAGA**) that was active at the time of failure. This places an interesting requirement on a recovery protocol: *in order to return the user to a useful point in the* **iSAGA***, the user's* **sub-iSAGA** *of interest at the time of failure must be discerned, and all other* **sub-iSAGAs** *filtered out from consideration in the recovery protocol.* For example, in the airline **iSAGA**, recovery from a failure at action A_7 (seat selection) should not place the user at the mileage-checking action A_3. Similarly, failure at component action A_3 (mileage-checking) should not recover the user to the travel-plan entry action A_2.

Clearly, other schemes for choosing the **sub-iSAGA** for recovery are possible, e.g., offering the user a choice among all **sub-iSAGAs** in his **SAGA**, or choosing the **sub-iSAGA** with the largest number of actions. Note, however, that the scheme for choosing the **sub-iSAGA** to which a user should be recovered is independent of the recovery protocol, i.e., once the **sub-iSAGA** for recovery has been chosen, the protocol for recovering the user (in Section 6) remains the same.

Having described **iSAGAs** and how they differ from database SAGAs, we move on to describe a simple recovery technique for **iSAGAs**.

4 A Simple Recovery Technique

Recovering a user after a failure in an **iSAGA** requires (a) the ability to describe what a user has done, i.e, to model and capture a user state, and (b)the ability to bring the user back to a useful state upon reconnection after failure. We first provide a formal description of a user state, and then present our protocol.

4.1 What is a user state?

Intuitively, a user state comprises all the information the Web site knows about a user at a given time. More formally,

Definition 3. *In an* **iSAGA***, a user's state changes from action to action. After each component action* A_i *in an* **iSAGA***, the user state* S_i *is a 3-tuple* $\langle K_i, Q_i, P_i \rangle$*, 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*, and*
- P_i *is the site's HTTP response to* Q_i*.*

After every component action the user arrives at a specific state, and the subsequent component action is done over this new state.

In our airline site example in Section 3.1, the user submits the details of his travel, and the site gives back a set of itineraries to the user. After receiving the itineraries, the user's state is composed of his cookie information (which includes his login information), the HTTP request for the itineraries (containing his travel plans) and the site's HTTP response to his query (the page containing a set of itineraries). The itineraries given to the user are dependent on the state of the airline database at the time the query is processed. Thus, the user's state is dependent on the view of the database the site used to generate its response to him, i.e., the projection of the database that contains the itineraries related to user's travel details. Here, the user has a view V_i on the database. After execution of a component action A_i, the user's state S_i will be a function of the view V_i.

Given that a user's state is dependent on the state of the database and, (as noted in Section 3.4), might lose its value to the user over time, we associate a validity period with each user state. In simple cases the validity can be a TTL period. More complex cases might utilize a flag associated with a boolean expression, where this expression represents predicates over the underlying database.

For example, consider the seat-selection action in our airline site example. Suppose only a single seat is available on a particular flight, and that the Web site has offered that seat to two users. Further suppose that, in the process of selecting the open seat, one user's connection is lost. While he is reconnecting, the second user succeeds in reserving the final seat on the flight. When the first user reconnects, the Web site attempts to recover him to a recent valid state. However, as now there is no seat available in the flight the previous response as given to the user would be invalid. If a short TTL period (less than the average time to reconnect after failure) had been associated with the seat-selection user state, this state would likely have been invalid at the time of recovery, and the user recovered to a prior valid state in his **iSAGA**.

4.2 Recovery Protocol

Our proposed recovery protocol has two phases: (a) a *preparatory* phase, in which users' states are saved as they interact with the Internet; and (b) a *recovery* phase, in which the saved states are used to bring the user back to a valid recent state.

Preparatory Phase In the preparatory phase, subsequent to each component action, a state is saved in a state log. In other words, upon the completion of a component action A_i, a state log records the user state S_i corresponding to A_i. The location where these saved states are stored is dependent on the system architecture, since wired and wireless scenarios present different challenges and opportunities for recovering users.

In a wireless scenario, for example, many client devices usually connect to the Internet via a single system called a gateway, which serves to bridge the protocol gap between wireless and wired systems (in a manner similar to dialing via modem to an ISP). Our recovery protocol can be deployed at a wireless gateway by enabling the gateway to store the users' states locally. Upon the need to recover, the gateway can check the recovering user's state set for the most recent valid state. If such a state exists for the user, the gateway can bring the user back to that state by replaying the response in the valid state to the user. In a wired network, the state-caching function of the gateway in the wireless scenario described above might be served by the site itself, or perhaps a proxy server or even an ISP.

Recovery Phase The recovery phase ensues when an **iSAGA** fails. More formally, we can recover a valid state for the user on restart by following these steps.

1. **Check log for saved states:** The recovery system first checks the log for saved states for the restarting **iSAGA**.
2. **Check validity of the saved states:** For any saved states retrieved from the log, check the states' validity and remove any invalid states from consideration.
3. **Replay the most recent valid state:** Resend the response in the most recent valid state to the user.

Let us see how this recovery scheme works in our example **iSAGA**, described in Section 3.1. Suppose that our user was disconnected after action A_5, i.e., after receiving a choice of itineraries corresponding to his preferred origin and destination. The recovery system will recover the user differently, depending on when he restarts. If he reconnects within a very short time, say δt, and the validity time t of the user's state containing the itineraries is more than δt (assuming use of TTL based invalidation), then we can recover the user from the itinerary page. However, if the user reconnects after t, then the user cannot be recovered at that page, and would need to be recovered to a prior valid state.

The above scheme of recovery will work well if the component actions are related. However, unlike database transactions, this is not always the case. In an **iSAGA**, a user may be diverted to various unrelated actions while working toward his main goal. Consider the airline reservation **iSAGA**. Here, while purchasing an airline ticket, our user might check his frequent flier mileage (action A_3) in the middle of the **iSAGA**. If his **iSAGA** fails, say after action A_7 (seat-selection), the recovery system need not consider the mileage check, since it does not affect the ticket-buying process. In this type of scenario, the system should consider a subset of the user's sequence of the user's actions, rather than all actions, during recovery. In order to handle this aspect of **iSAGAs**, we modify our simple recovery technique. However, before embarking on a description of this more complete recovery technique, we first describe a means of identify different active **sub-iSAGAs** within an **iSAGA** using the notion of *dependency*.

5 Dependency of iSAGA actions

A user's **iSAGA** consists of a sequence of actions $\langle A_{1,} A_2, ..., A_n \rangle$, not all of which are necessarily related to the same goal. The question of interest here is as follows: given a sequence of actions, $\langle A_{1,} A_2, ..., A_n \rangle$, which might contain different **sub-iSAGAs**, how we can extract the different **sub-iSAGAs** within it?

Extracting this information is possible due to the widespread use of dynamic page generation technologies. In sites using this type of technology, the response given to the user is a not a file residing in the web server, but rather the output generated by the execution of a program (called a *script*), based on parameters in the user request. There are several technologies, such as JSP [11], ASP [5], PHP [9], and Perl [14], that support dynamic page generation. These technologies have become popular because they dramatically increase the potential for interactivity with a user – a dynamic site composed of few scripts can potentially generate an infinite number of different pages based on information retrieved from an underlying database.

When a user interacts with a dynamic site, each component action A_i corresponds to running some script C_i. Consider the airline reservation system example. Here, when the user submits a request with his travel details (e.g., component action A_5), the site runs a program using the user's travel details as input parameter 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 sites built with these dynamic page generation technologies, it is possible to determine the **sub-iSAGAs** *in an* **iSAGA**. Each component action is carried out by dynamic scripts that run on an application server. Since HTTP is inherently a connectionless protocol [7], each of these dynamic scripts runs independently. However, quite often, a script may be dependent on information obtained or generated in another script. Consider, for example, the scenario where our traveler has selected his preferred itinerary on the airline site (component action A_6), 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 several ways to achieve this: (a) hidden fields in forms, (b) cookies, (c) session variables, and (d) explicit parameter passing through the HTTP get/post call.

Since there is a clear data dependency inherent in such parameter passing techniques, we can derive dependent sequences within an **iSAGA**. In particular, if $\langle A^{'}_{1,} A^{'}_{2}, ..., A^{'}_{n} \rangle$ indicates an **iSAGA**, then to achieve a particular goal, we will require running of scripts $\langle C^{'}_{1,} C^{'}_{2}, ..., C^{'}_{n} \rangle$ in sequence. These scripts will pass information from one script to the next using one of the methods described above. For example, $C^{'}_{i}$ will pass information to $C^{'}_{i+1}$. In this case $C^{'}_{i+1}$ can be said to be dependent on $C^{'}_{i}$ (symbolically $C^{'}_{i+1} \rightarrow C^{'}_{i}$). Since the the number of ways scripts can pass information is limited, and decided at site design time, the site designer can statically describe the dependency list among scripts. Many commonly-used site analyzer/designer tools, e.g., Visual Interdev [6] can automatically determine the dependencies among scripts.

Let us assume that a user has completed a sequence of actions $\langle A_{1,} A_{2}, ..., A_{n} \rangle$. Corresponding to this sequence we have a sequence of scripts that has been executed for these actions, $C = \langle C_{1,} C_{2}, ..., C_{n} \rangle$. From this sequence of scripts we will be able to find a sequence $C' = \langle C'_{1}, C'_{2} ..., C'_{m} \rangle$ such that $C'_{i} \in C$ and $C'_{i+1} \rightarrow C'_{i}, \forall C'_{i} \in C'$. From C we can identify all such C'.

Note that several variations on this theme are possible. For instance, a C_i such that $C_i \in C'$ and $C_i \in C''$, this means same script may be part of two **sub-iSAGAs**. We may also have a script C_i occurring twice in a sequence C'. Formally, there may exist a scripts C_i and C_j such that $C_i \in C^{'}$ and $C_j \in C^{'}$ and $C_i = C_j$.

After identifying all dependent sequences of scripts, we can map these scripts to corresponding component actions in the original **iSAGA** $A = \langle A_{1,} A_{2}, ..., A_{n} \rangle$. This will result in a set of action sequences $A^{'} \subset A$, $A^{''} \subset A$, and so forth. Each such action sequence will correspond to a **sub-iSAGA** for a user. Formally, we define the notion of dependency, including invalidation requirements, as follows.

Definition 4. *A component action A_i is dependent on another component actions A_j if the script corresponding to action A_i (i.e., C_i) is dependent on a script C_j corresponding to action A_j.*

Axiom 1 *Let $\mathcal{S}_i$ be the state of the user after execution of action A_i. A_i is dependent on A_j and $\mathcal{S}_j$ is invalid indicates that $\mathcal{S}_i$ is also invalid. In other words, the validity period for $\mathcal{S}_i$ should be less than validity period of $\mathcal{S}_j$ if A_i is dependent on A_j.*

6 Complete Recovery Technique

Given the above discussion of multiple user objectives and dependency, we revisit our simple recovery technique. We first describe our more complete recovery technique intuitively. We then formally describe the recovery technique.

6.1 Intuition

When a component action in an **iSAGA** fails, the user may have already initiated multiple **sub-iSAGAs** in his **iSAGA**. The goal of our recovery protocol is to return the user to a state from which he can continue. As mentioned in Section 3.5, in the context of a multiple-objective **iSAGA**, the user will be recovered to the active **sub-iSAGA**. Consider the airline **iSAGA**, in which there are two clear **sub-iSAGAs**, the mileage-checking and the ticket-buying **sub-iSAGAs**. Suppose the **iSAGA** fails before the purchase action A_8 is complete. Clearly, here, the proposed recovery technique should bring the user to a state from which he can continue the ticket buying **sub-iSAGA**, and should not consider the user's mileage checking **sub-iSAGA**. Rather, it would make sense to bring the user either to the itinerary page or the page after login.

The recovery protocol is a two-step process. First, we identify the active **sub-iSAGA**, the one corresponding to the user's most recent completed action. Second, we will recover the user to the last valid recoverable state in the active **sub-iSAGA**. The preparatory phase, in which user states are collected and stored, is the same as in the simple recovery technique discussed in Section 4.

6.2 Recovery Technique

Consider a scenario in which a user has completed actions $\langle A_1, A_2, A_3, ..., A_n \rangle$, and component action A_{n+1} failed. We are interested in recovering the user to a recent valid state.

1. **Identify the active sub-iSAGA:** Let $A^{'} = \langle A^{'}_1, A^{'}_2, ..., A^{'}_m \rangle$ indicate a single **sub-iSAGA** such that $A^{'}_m = A_n$, following the procedure described in Section 4. If there exist two action sequences $A^{'}$ and $A^{''}$ which satisfy this condition, then we will consider the preceding action A_{n-1} to determine which **sub-iSAGA** was active before the failure occurred.
   Generally, we identify a sequence of actions $A^{'} = \langle A^{'}_1, A^{'}_2, ..., A^{'}_m \rangle$ indicating a single **sub-iSAGA** such that $A'_m = A_n, A'_{m-1} = A_{n-1}, ..., A'_{m-k} = A_{n-k,} A'_{m-k-1} \neq A_{n-k-1}$, where k is the maximum for all such k found among all **sub-iSAGAs** $A^i \in A$. Any tie will be resolved in step 2.
2. **Identify the most recent valid state:** We identify the action $A^{'}_i \in A^{'}$ for which there exists a valid state S_i such that i is maximum. In the case of a tie in step 1, we consider all such A_i's and choose the one with the latest timestamp corresponding to state S_i. We resolve the tie by considering the valid state which has occurred most recently.

7 Conclusion

In this paper, we consider the problem of recovering from failure during Internet transactions. We model user interactions with Web sites using the metaphor of a transaction, and describe a simple protocol for recovering from failure. We then expand our simple protocol to handle the complexity introduced by allowing multiple user goals in a single **iSAGA**.

References

1. R. Agrawal and D.J. DeWitt. Integrated concurrency control and recovery mechanisms: Design and performance evaluation. *ACM Transactions on Database Systems*, 10(4):529–564, 1985.
2. G. Alonso, D. Agrawal, A. El Abbadi, M. Kamath, R. Gunthor, and C. Mohan. Advanced transaction models in workflow contexts. In *Proceedings of the International Conference on Data Engineering*, pages 574–581, 1996.
3. P.K. Chrysanthis and K. Ramamritham. Acta: A framework for specifying and reasoning about transaction structure and behavior. In *Proceedings of the 1990 ACM SIGMOD Conference*, 1990.
4. P.K. Chrysanthis and K. Ramamritham. Synthesis of extended transaction models using acta. *ACM Transactions on Database Systems*, 19(3):450–491, 1994.
5. Microsoft Corporation. Active server pages tutorial. http://msdn.microsoft.com/workshop/server/asp/asptutorial.asp, 2001.
6. Microsoft Corporation. Visual interdev. msdn.microsoft.com/vinterdev/, 2001.
7. 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.
8. H. Garcia-Molina and K. Salem. Sagas. In *Proceedings of the 1987 ACM SIGMOD Conference*, 1987.
9. The PHP Group. What is php? http://www.php.net/, 2001.
10. T. Harder and A. Reuter. Principles of transaction-oriented database recovery. *ACM Computing Surveys*, 15(4):287–317, 1983.
11. Sun Microsystems. Java server pages. http://java.sun.com/products/jsp/, 2001.
12. 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 Transactions on Database Systems*, 17(1):94–162, 1992.
13. N. Neves and W.K. Fuchs. Adaptive recovery for mobile environments. *Communications of the ACM*, 40(1):68–74, 1997.
14. Inc. O'Reilly & Associates. What is perl? http://www.perl.com/pub, 2001.
15. C. Pedregal-Martin and K. Ramamritham. Guaranteeing recoverability in electronic commerce. In *Proceedings of the Workshop on Advanced Issues in Electronic Commerce and Web-Based Information Systems*, 2001.
16. J.D. Tygar. Atomicity versus anonymity: Distributed transactions for electronic commerce. In *VLDB'98, Proceedings of 24rd International Conference on Very Large Data Bases, August 24-27, 1998, New York City*, pages 1–12, 1998.
17. U. Varshney and R. Vetter. Emerging mobile and wireless networks. *Communications of the ACM*, 43(6):73–81, 2000.
18. G.D. Walborn and P.K. Chrysanthis. Pro-motion: Management of mobile transactions. In *Proceedings of the Symposium on Applied Computing*, pages 101–108, 1997.