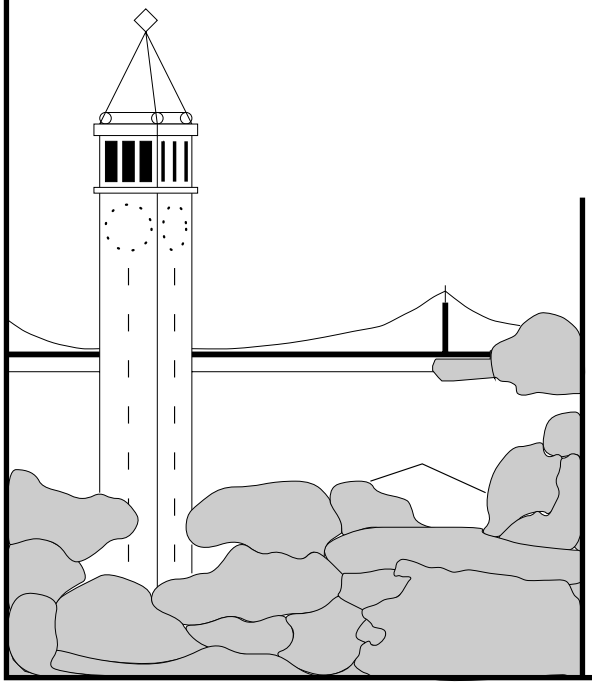
Toward System-Wide Undo for Distributed Services

Aaron Brown
EECS Computer Science Division
University of California, Berkeley



Report No. UCB//CSD-03-1298

December 2003

Computer Science Division (EECS)
University of California
Berkeley, California 94720

Toward System-Wide Undo for Distributed Services

Aaron Brown
EECS Computer Science Division
University of California, Berkeley
477 Soda Hall #1776, Berkeley, CA 94720-1776
abrown@cs.berkeley.edu

Abstract

In this report, we extend the concept of system-wide undo from self-contained services to collections of distributed, interacting services, thereby providing an undo-based recovery mechanism to the operators and administrators of distributed services. The extended undo mechanism is targeted at human operator error and other state-affecting problems like software bugs, misconfigurations, and external attack, and provides retroactive repair of past problems. We achieve the distributed extension by appealing to the concept of spheres of undo: spheres of undo surround each component of a distributed service and provide a structuring mechanism that helps identify when undo of one component can affect others. We propose two approaches for composing interacting spheres of undo: one assumes coordination and cascades undo-based recovery from the first undone sphere to all others affected; the other approach assumes independence and handles interacting spheres by compensating for previous communications that become invalid following an invocation of undo. We present criteria that can be used to decide which approach is most appropriate, and give an example using them to choose the appropriate undo approach for a distributed e-shopping service. Finally, we describe initial thoughts on how the compositions might be implemented and propose algorithms for interacting-sphere undo.

1 Introduction

System-wide undo is a recovery mechanism for Internet and corporate services that gives system operators and administrators the ability to "travel in time" and to retroactively repair problems that have corrupted or destroyed system state; it provides effective recovery for human operator error and for other state-affecting problems like software bugs and external attack [3]. Previous work on system-wide undo has been limited to self-contained services that interact only with human end-users. While the self-contained-service approach captures a large number of important services, they still represent only a portion of the application space of service systems, failing to capture important domains such as electronic commerce, where interacting services are the norm.

In this paper, we move past the limitations of self-contained services to tackle the question of how to bring undo to distributed, independent but interacting services. To make headway on the problem, we appeal to the construct of **spheres of undo**, introduced in earlier work on system-wide undo [3], as the fundamental structuring element. We use spheres of undo to build up more sophisticated undo models, composing them together in collections of coordinated and uncoordinated elements in order to model different distributed service architectures. We then develop criteria for choosing between those composition models, and investigate a detailed case study of how the different forms of distributed undo might be used to bring undo-based recovery to an Internet e-shopping service. While we have not implemented distributed undo, we have some initial thoughts and tentative implementation algorithms; we will discuss these as well.

The remainder of this report is organized as follows. We begin in Section 2 with a recap of the key concepts of self-contained system-wide undo, including spheres of undo, verbs, and paradoxes. Section 3 then

introduces our new composition approaches for modeling distributed services with spheres of undo. We then consider criteria for choosing a composition approach for a given distributed service in Section 4. Section 5 considers how the implementation of self-contained services undo might be extended to support interacting spheres of undo; the appendix (Section 9) provides further details and tentative algorithms for these extensions. Section 6 takes an example distributed web service, online shopping, and maps it into distributed spheres of undo, showing how that service can be made undoable. Finally, we wrap up with a discussion of related work in Section 7, and conclude in Section 8.

2 Recap: System-wide Undo and Spheres of Undo

The work in this paper draws heavily on the concepts and basic architecture for system-wide undo introduced in our previous work [3] [4] [5]; readers unfamiliar with that work are encouraged to consult the above citations before proceeding.

In brief, system-wide undo provides system operators and administrators with a **committable-cancelable** undo operation that provides linear, state-based undo for system-level state, coupled with command-based redo for the state belonging to end users of the service. During normal operation of a service that supports system-wide undo, periodic checkpoints are taken of the entire system state, and end-user interactions with the service are intercepted by a proxy and recorded as a linear, sequential log of **verbs**. Verbs encapsulate individual end-user requests in a context-independent manner, and also record the corresponding end-user-visible output produced by the encapsulated requests.

When an operator invokes undo, the system first **rewinds** all hard state to a previous checkpoint. The operator can then make **repairs** to that state—for example, fixing a misconfiguration made earlier. If the operator then **cancels** the undo, the system is restored to its pre-rewind state and all repairs are lost. Alternately, if the operator **commits** the undo, the original history following the rewind point is discarded. To prevent end-user work from being lost, part of the commit operation is to **replay** the log of verbs from the rewind point to the present, re-executing user interactions that were lost during rewind. Note that only end-user operations are replayed, and that the operations themselves are re-executed as opposed to simply restoring their effects; this approach ensures that the requests are reprocessed in the repaired system state created by the undo process. Of course, user requests may behave differently due to the repairs; if they behave differently enough to produce different externally-visible output, then a **paradox** is declared and managed by a compensation mechanism that explains the inconsistency to the end user. Paradox management is accomplished by way of the verbs; verbs contain predicates for detecting paradoxes in their recorded external output, for compensating for paradoxes when they occur, and for handling paradoxes that propagate through a sequence of non-commuting verbs.

Undo of a single, self-contained service can be modeled using the construct of a **sphere of undo** (**SoU**). Figure 1 gives examples of spheres of undo for desktop applications and for service systems; we are primarily concerned with the latter case here. A sphere of undo contains the service's state, acting as a bubble of state and time with its own independent timeline manipulated by undo. It encloses the set of non-transient system state that can "time travel" as a single unit, and logically extends to include the parts of the system that operate on that state. Paradoxes occur when information that has escaped the SoU boundaries is altered by a time-manipulation operation like undo. In our previous work, spheres of undo contained the entire service, with only human end users outside its boundaries; these human end users are considered the service's only **external actors**.

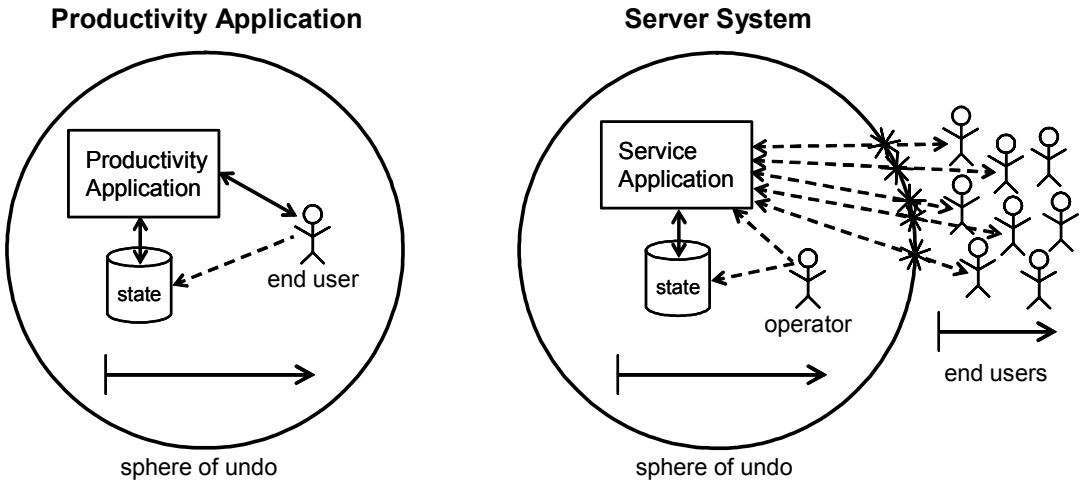


Figure 1: Spheres of Undo for productivity applications and server systems. In server systems, the undo model is complicated by the presence of end users external to the sphere of undo. While the administrator or operator can be considered to exist inside the sphere, much as the end user is in the productivity-application sphere, end users in the server case have their own timelines, independent of the timeline inside the sphere, and must therefore be kept outside of the sphere.

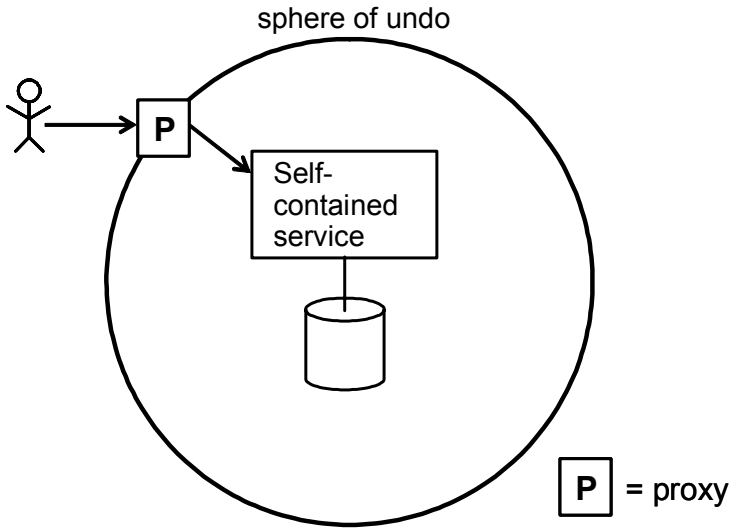


Figure 2: Self-contained single-service architecture. A self-contained service encompasses a single sphere of undo that communicates only with human end users. Requests from those end users are proxied, logged, and replayed at the sphere boundary.

3 Extending Spheres of Undo to Distributed Services

In prior work we attacked the paradox problem for one particular service architecture, the self-contained service case, where the service is entirely contained within a single sphere of undo whose external actors consist solely of human end-users. Figure 2 summarizes the self-contained service architecture, using a notation we will use throughout this report. In this architecture, all relevant hard state is centralized in the single sphere of undo and is rewound as a single unit as part of the undo process. External actors are assumed to interact with the service in a request-response model, with all requests intercepted, logically serialized, and recorded as a history of verbs. External output is limited to responses to external requests, and is intercepted and associated with the corresponding input verb; externalized output can be delivered synchronously or asynchronously. Paradoxes in

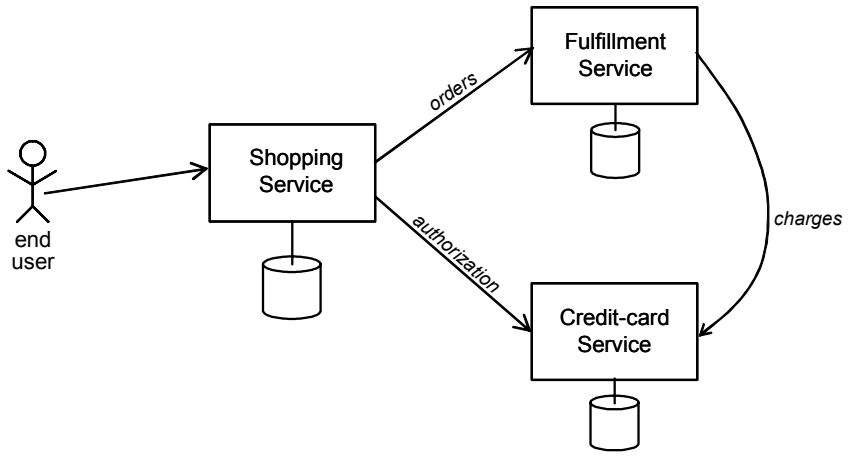


Figure 3: Hypothetical architecture for online retailer. The retailer's systems are composed of three interacting modules: a front-end web service that interacts with the user, a fulfillment service that accepts orders from the front end, and a credit card service that handles authentication for the front-end and charge requests from the fulfillment service. Each module maintains its own state database.

externalized output resulting from the undo process are detected by output predicates and managed via compensating or explanatory actions defined by the verb; there is no notion of directly altering the external actors' timelines to avoid paradoxes.

This basic architecture captures many end-user-driven services, like the e-mail and auction services discussed as case studies in our prior work [3] [5]. But what if our service looks like the one depicted in Figure 3, which shows the hypothetical high-level architecture of an electronic shopping service like an online retailer? Note that there are three separate but cooperating systems involved here—the shopping service, the fulfillment center, and the credit card processing service—with three separate sets of hard state storage and three sets of external actors. One option is to simply treat the entire conglomeration of services as a single logical service, wrap it in a single sphere of undo, and apply the self-contained service undo architecture. But this approach involves several compromises. First, it would require that all three systems be undone as a unit—there would be no way to use undo to repair, say, the shopping service, without also having to undo the fulfillment and credit-card systems. Second, we would have to treat the hard-state storage of all three systems as a logical unit, requiring synchronized temporal management of the three separate storage systems for checkpointing, rollback, and purging of expired state. Finally, we would have to treat all sets of external actors as one, funneling all requests through a single logical proxy into a single logical verb log.

To handle a multiple-system service while preserving the individual undoability of each system, we must divide it into separate but interacting spheres of undo. In our shopping example, that means one sphere for each of the three systems, as Figure 4 illustrates. Each sphere of undo is induced by the local hard state on each system. If built following the self-contained service architecture, each system could then be seen as its own undoable service. But how do we handle communication between the services? Each service is an external actor to the others: for example, the shopping service makes requests to the fulfillment service to ship orders, and likewise the fulfillment service may request credit card verification from the credit card service. Since each service is undoable and hence capable of manipulating its own timeline, we now have two options for handling paradoxes that arise between the services. First, we could simply apply the approach we used for human end users and attempt to compensate for or explain the paradoxes; we call this approach to composing multiple spheres the **uncoordinated spheres** composition. Alternately, if undo in one sphere generates a paradox in a second sphere, we could adjust the timeline in the second sphere (the one seeing the paradox) to resynchronize the

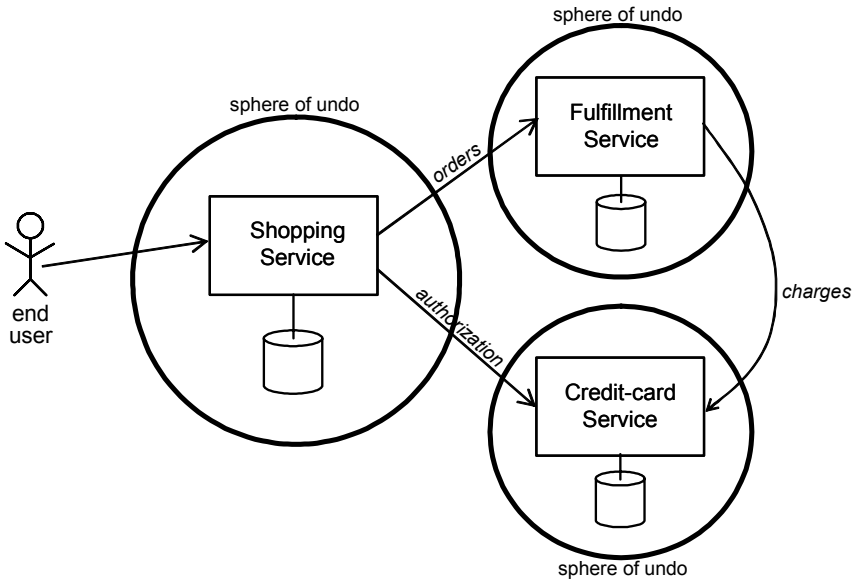


Figure 4: Spheres of undo for hypothetical online retailer architecture. Each module from Figure 3 is placed in its own sphere of undo.

timelines between the two spheres and remove the paradox entirely. In other words, we would undo the second sphere to the time at which the paradox occurred in the first sphere, then roll forward both spheres from that point, taking into account the new paradox-producing data. We call this the **coordinated spheres** composition approach.

Each of these two approaches has advantages and drawbacks. The most significant benefit of treating two or more interacting services as uncoordinated spheres is that it bounds the number of systems that are affected by the undo process. With a system of uncoordinated spheres, unilateral undo of one of those spheres is guaranteed to only affect the timeline of the undone sphere: all other spheres will continue to exist and progress forward in the present, and only the storage of the undone sphere will be rewound. While other spheres may see compensations as a result of the first sphere's undo process, they will not need to pay the cost of rolling back and re-executing their own work.

On the other hand, the uncoordinated spheres approach will only work when the interactions between the undone sphere and the system's other spheres allow for the same kind of relaxed consistency model that we required of external human end users in our self-contained single-sphere architecture. That is, all data emanating from the undone sphere must be capable of being altered after-the-fact, with those alterations either subject to meaningful and pre-defined compensations, or silently tolerated by the other spheres of undo. For example, in our earlier e-shopping example, a request from the shopping service to the order fulfillment center might be altered after undo is carried out on the shopping service; as long as the order fulfillment center provides verbs to cancel, alter, or refund the previously-submitted order, then compensation is possible and the two systems can be treated as uncoordinated.[1] A counterpoint would be a block-storage service that provides data to, say, a web server. Here, the spheres corresponding to the storage and web services respectively cannot be treated as uncoordinated, as there is no meaningful compensation at the web server level for mis-delivered data blocks from the storage service.

1. Just as with external output to end-users in the self-contained service case, there may be only a limited window during which compensation is possible (in this case, the time between ordering and shipping). Hence, undo of a given sphere will be limited to the possible compensation windows of any uncoordinated spheres with which it communicates.

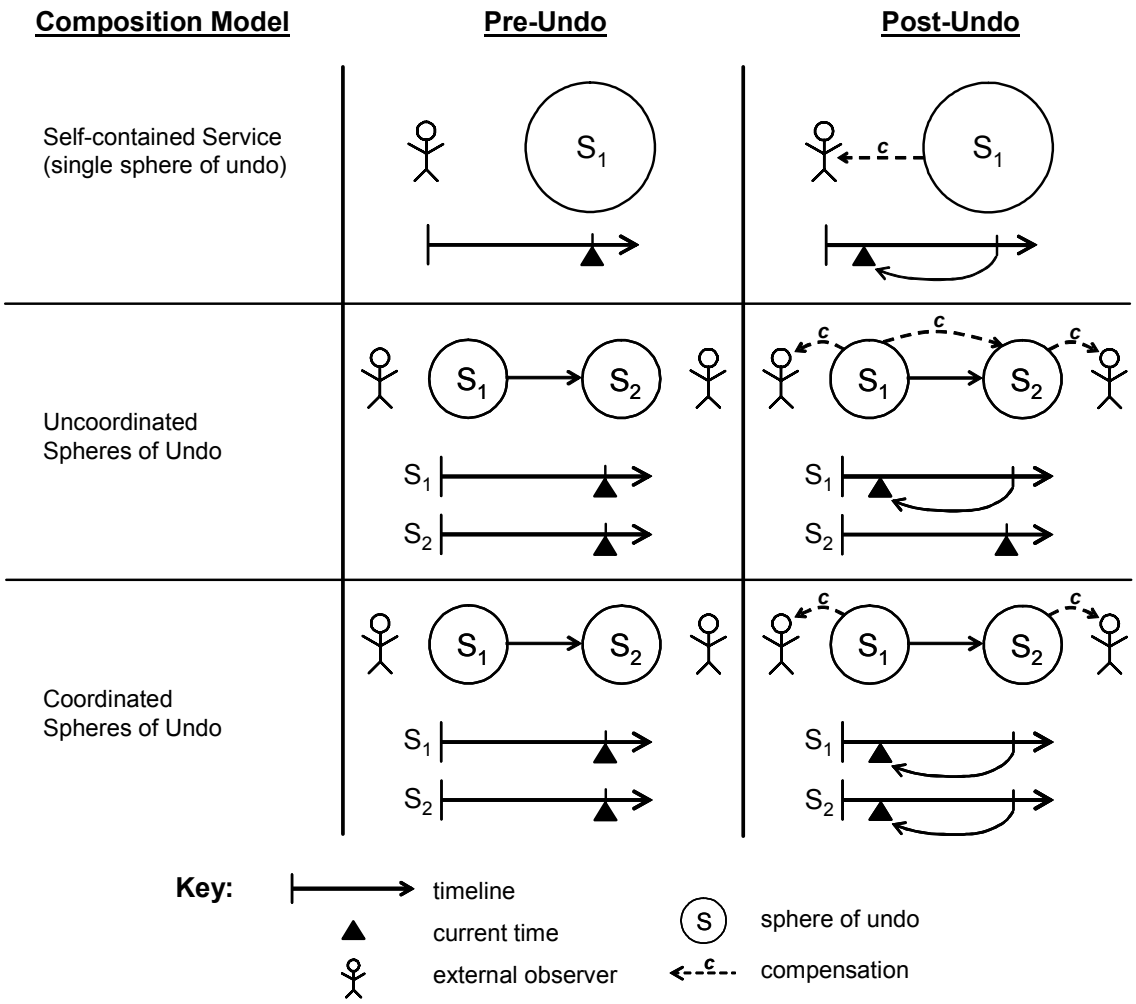


Figure 5: Summary of composition approaches for spheres of undo. For each model, the figure shows the schematic representation of the system and each sphere's timeline both before and after undo is invoked. Unless otherwise noted, S_1 is the sphere on which undo is invoked. If the undo process can cause compensations or explanatory messages to flow outside of a sphere, these are indicated with dashed arrows tagged with the letter c.

In contrast, coordinated spheres flip the advantages and drawbacks of uncoordinated spheres. Coordinated spheres do not require a weak consistency model or the existence of meaningful compensations. But on the other hand, they potentially suffer from the "domino effect" [18], where an undo invocation on one sphere can propagate to other spheres in the system, causing a cascading rollback [10]. This can increase the cost of undo, since each additional sphere that becomes involved in the undo process contributes its own overhead in terms of state rollback, re-executed work, potential compensations to its end users, and brief unavailability of the service during rewind. Because of this extra cost, the problem of accurately predicting the paradox consequences of undo takes on a new importance in the coordinated-spheres case, and is a key area for future work on undoable distributed services.

4 Mapping Service Architectures to Compositions of Interacting Spheres

Figure 5 summarizes the composition approaches we have defined for modeling interacting (uncoordinated or coordinated) spheres of undo, and compares them to the single-sphere-of-undo self-contained service model.

Next, we consider how to take a given service and derive the appropriate composition of SoU's needed to add undo functionality to that service. If we can produce the composed-SoU model for a service, then we know that we can make the service undoable by applying the implementation techniques of self-contained service undo to each individual SoU, and then by applying the techniques of Section 5, below, to add undo to the overall collection of SoU's.

4.1 Single vs. multiple spheres

The first question to answer when trying to make a service architecture undoable is: should the service be represented as a single sphere, or a collection of interacting spheres? The answer to this question is driven by the organization of the service's hard-state storage and its static patterns of external communication. There are several cases in which a single-sphere approach is warranted:

- the service is not easily decomposable into multiple components, modules, or sub-services, each with its own private collection of hard state; or
- while the service is decomposable, the service's hard state is not easily partitioned, such as when the service's hard state is a single database shared between multiple service components; or
- the service and its hard state are partitionable, but the implementation of state storage is such that rolling back a portion of the service's hard state offers little performance advantage over rolling back all or most of it; or
- the communication pattern between components of the service includes **request cycles**.

Following our principle that spheres of undo are defined in terms of the structure of system state, the first three criteria support a single-sphere approach whenever the entire service's hard state can easily be treated as a single unit, and mandate it when the service is not obviously partitionable in terms of state. The fourth criterion demands a single-sphere approach when the service has internal communication patterns that can result in deadlock; a complete discussion of request cycles and when they occur can be found in Section 9.

4.2 Decomposition into interacting spheres

When the criteria for a single-sphere model do not hold, then we can consider decomposing the service into multiple interacting spheres of undo. The granularity of decomposition depends on the partitionability of the service's state from the point of view of the service application. Physically separate state stores, such as the stores on individual machines in a cluster, or on different machines implementing different sub-services in a distributed service environment like our e-shopping example above, naturally define separate spheres of undo.

The only exception is when multiple separate state stores are logically synthesized into a virtual image of a single state store *below the level of the service applications*, in which case the sphere of undo must be defined by the totality of physical state backing that virtual image. For example, if the service sees only a single logical store provided by a RAID system or a distributed database that runs atop multiple physically-distinct state stores, the sphere of undo is defined by the logical store, not its subcomponent physical stores.

The question of coordinated versus uncoordinated spheres is more complex. Here the tradeoff is between the intrusiveness of compensations and the competing desire to bound the scope of undo-induced rollback. In general, if appropriate compensations exist, the uncoordinated-spheres approach of using compensations should be favored, as it reduces implementation complexity and minimizes the performance impact of undo to

external users. It also offers a simpler mental model to the operators invoking undo, as they can know in advance that undo operations will not cascade past the boundaries of the system on which they are invoked.

There are some cases where an uncoordinated-spheres approach is required. One such case is when one sphere of undo (SoU) interacts with an external entity that cannot tolerate rollback. This can occur when the external entity is not a SoU itself, but also when the external entity is a SoU that has independently made external requests or exposed state in ways that cannot be compensated for. If such a situation is possible, all SoU's that interact with the externalizing SoU must do so using the uncoordinated-spheres model.

Uncoordinated-spheres composition may also be required in situations where communication patterns between SoU's cross trust boundaries, as when two services run by different corporations need to interact (imagine the credit-card processing service in our e-shopping example talking to VISA's card authorization service to authorize purchases). Even if both services are undoable, there are good reasons why one provider A might not want to accept undo invocations propagated from another provider B's SoU: if B were malicious, it could produce a denial of service attack on A's service by sending phony undo requests, or could degrade A's service by causing it to frequently roll back, resulting in outages and excessive compensations to A's other users. Or A might simply want to keep full control of its service and not have to face unexpected rollbacks resulting from external services, even ones it trusts. To avoid such situations, A and B must compose their services using the uncoordinated-spheres composition model, rather than linking them.

In summary, to determine if an uncoordinated-spheres approach is feasible, each pair of interacting spheres first needs to be examined to determine if appropriate and acceptable compensations can be defined for every external interaction between those spheres. If not, then a coordinated-spheres approach is required, unless prohibited by intervening trust boundaries or the involvement of a SoU that does not always support rollback. An alternative to simply giving up on undo in these latter cases is to try to fit a hybrid model combining aspects of the coordinated- and uncoordinated-sphere compositions, with some requests handled by compensations and the remainder (for which compensations do not exist) handled by propagating the undo operation to coordinated spheres.

5 Architecture Extensions for Interacting Spheres of Undo

In this section, we discuss how the basic single-sphere undo architecture might be extended to support undo for coordinated and uncoordinated spheres of undo (SoU's). Supporting interacting SoU's is complex, and requires extensions to verbs, the verb log, and the undo algorithms themselves. As we develop those extensions, we will assume that each sphere in a composition is internally architected following the design laid out in our earlier work [3] [5].

To implement interacting spheres, we first need to extend the verb framework and log architecture of single-sphere undo to capture the notion of requests flowing between spheres. We begin by establishing some new terminology. Whereas in single-sphere undo we simply had "requests", now we are have **inbound requests** and **outbound requests**. An inbound request is one that enters a sphere of undo; it can produce a response, which flows back across the SoU boundary to the entity making the request. Conversely, an outbound request flows from an SoU to an external entity, typically another SoU; outbound requests also have responses, in this case flowing back into the SoU from the external entity. Requests can be **inter-sphere**, if they represent communication between coordinated spheres of undo, or **external**, if they represent communication between a SoU and either an external entity or an uncoordinated SoU. Figure 6 shows the different communication patterns captured by these categories of requests. Note that what we previously called "requests" are now inbound external

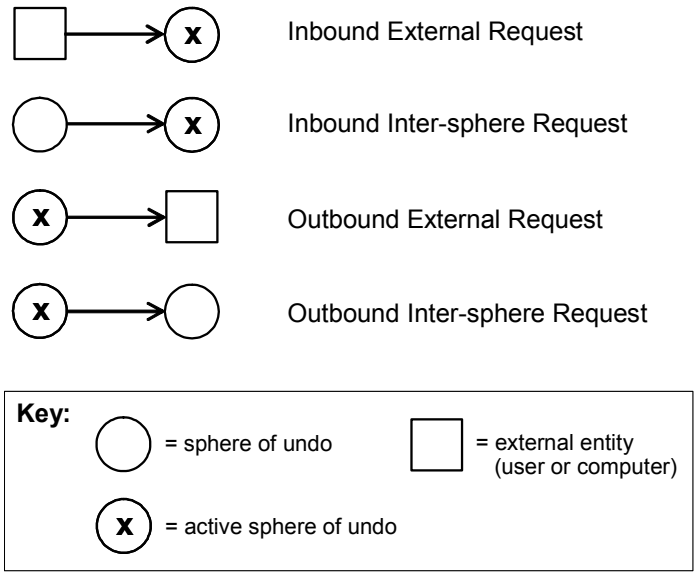


Figure 6: Communications patterns involving interacting spheres. The direction of a request (inbound vs. outbound) is designated from the point of view of the sphere marked with an **x**.

requests, and "external output" in our old terminology is now the response associated with an inbound external request. Note as well that, although we have only defined communications in terms of request/response, we can represent unidirectional communication as requests that produce no responses.

Outbound requests are new in this formulation, and can take three forms. **Synchronous outbound requests** are generated synchronously while servicing a unique inbound request. In other words, the outbound request must be completed before the original inbound request can itself complete. An example would be if a user asks a service for data that resides on another service, and the first service needs to make a synchronous request to that second service before it can complete the user's original request.

Deferred outbound requests are also causally related to a unique inbound request, but may be executed at some later time, after the inbound request completes; they do not produce responses used to service the inbound request. An example deferred outbound request from the e-shopping domain would a supplier's act of sending a package to the shipping company: while tied to an inbound request to fulfill an order, the outbound "ship" request is not processed synchronously with the order fulfillment input. We assume that deferred requests are added to their associated input verbs when those input verbs are created, even if they are to execute later in the future.

In contrast, an **asynchronous outbound request** is not initiated as part of servicing an inbound request, but is generated, for example, by a background process that detects a certain condition, such as stock quantities being low in an e-shopping service. These requests are asynchronous only in that they are not associated with a particular inbound request; however, they must be serializable with respect to the incoming stream of inbound requests.

We can represent all these new forms of requests and their associated responses by extending our model of verbs. Whereas up to now we have only had "verbs" representing inbound external requests, we will now have **input verbs** representing any sort of inbound request, and **output verbs** representing outbound requests. Input verbs are the same as the verbs we have been using all along, except that now we allow them to be generated by other SoU's as well as end users. Output verbs represent outbound external and inter-sphere requests, and

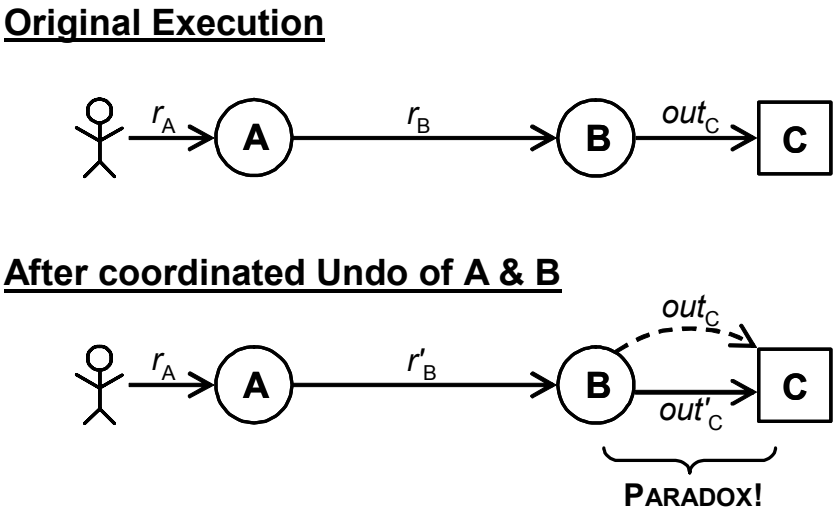


Figure 7: An external paradox case for interacting spheres. A and B are coordinated spheres, such that undoing A causes B to undo as well. If the undo of A changes the way r_B is generated, then, while B will not see a paradox, the external entity C can potentially see a paradox due to a change in the external output of B.

record both the parameters and destination of the request (in the verb tag) and the response to the request (the analog of an input verb's record of what we have called "external output"). Synchronous and deferred outbound requests map into output verbs associated with input verbs; asynchronous outbound requests map into output verbs not associated with any input verb.

We must extend our log model to capture these new output verbs. Output verbs corresponding to synchronous and deferred outbound requests can be attached to the input verbs that generate them; this requires that the input verb log records be extended with pointers to the sets of synchronous and deferred output verbs. Output verbs corresponding to asynchronous outbound requests can be stored in the history log like regular input verbs, sequenced into the input stream at the point when they occur.

Once we consider recording output verbs, we have to decide how they will behave during the undo cycle. Unlike the simple case of single-sphere undo, where we only logged inputs that do not change across a rewind/replay cycle, output verbs may be different during replay due to changes made during rewind and repair: they may have different parameters, they may result in different responses, they may not appear during replay, or they may be supplemented with new output verbs during replay. Likewise, if we are considering coordinated spheres, inter-sphere input verbs can differ during replay, as the changes in one sphere's inter-sphere output verbs can alter their target spheres' inbound inter-sphere requests. Only external input verbs can be assumed to remain consistent across undo cycles.

If any of these changes to output or input verbs result in different requests or responses flowing outside the boundary of the undoable system, then we may have paradoxes that need to be detected and handled; Figure 7 shows an example for interacting spheres. Paradoxes in the responses to inbound external requests can be handled in the same manner as for single-sphere undo, with detection predicates and handling routines attached to the corresponding input verbs. Paradoxes resulting from changes to outbound external requests are much more difficult to handle, and require both architectural changes (enhancing the verb log to support multiple **epochs**, or concurrent versions of history) and new algorithms for replay that can match up and compare

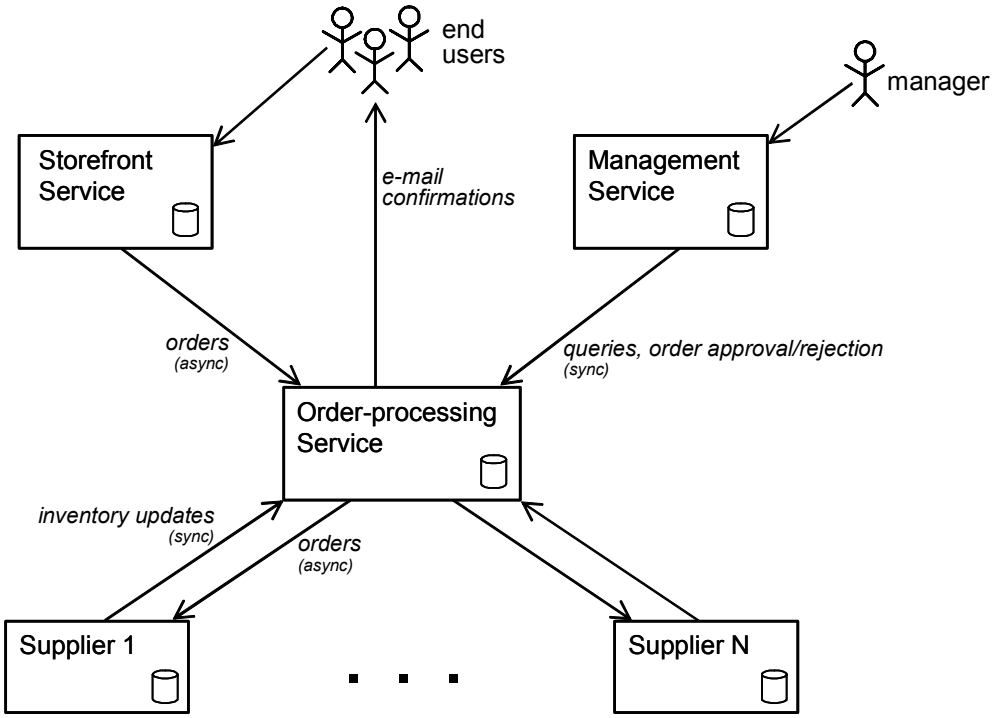


Figure 8: PetStore architecture. PetStore is comprised of three core services, the storefront, management, and order-processing services, and one or more supplier services.

output verbs from different epochs, allowing per-output-verb paradox detection and compensation analogous to the approach used with input verbs.

We will not go into detail here about how output paradoxes are handled, or how paradox detection and replay work together for undo of coordinated and uncoordinated spheres; complete details can be found in Section 9, along with proposed algorithms for rewind and replay of both coordinated and uncoordinated spheres.

6 Case Study: e-Shopping

To ground our heretofore rather abstract discussion in reality, we now turn to a detailed example of how undo might be provided for a real-world distributed service drawn from the application domain of e-commerce. Our case study will illustrate the processes of mapping a complex, non-self-contained service into multiple spheres of undo (SoU's), choosing composition models for those SoU's, and defining verbs and paradox handling mechanisms for each SoU in the overall service.

Our case study is based on PetStore, an e-shopping system developed by Sun as a blueprint for building online retailer applications [22]. PetStore is built on the J2EE Java-based enterprise application platform, but for the purposes of this section we will gloss over the J2EE-specific implementation details and focus on the PetStore system architecture.

6.1 PetStore architecture

Figure 8 schematically illustrates PetStore's architecture. The architecture consists of three core services plus one or more "supplier" services. The core services comprise:

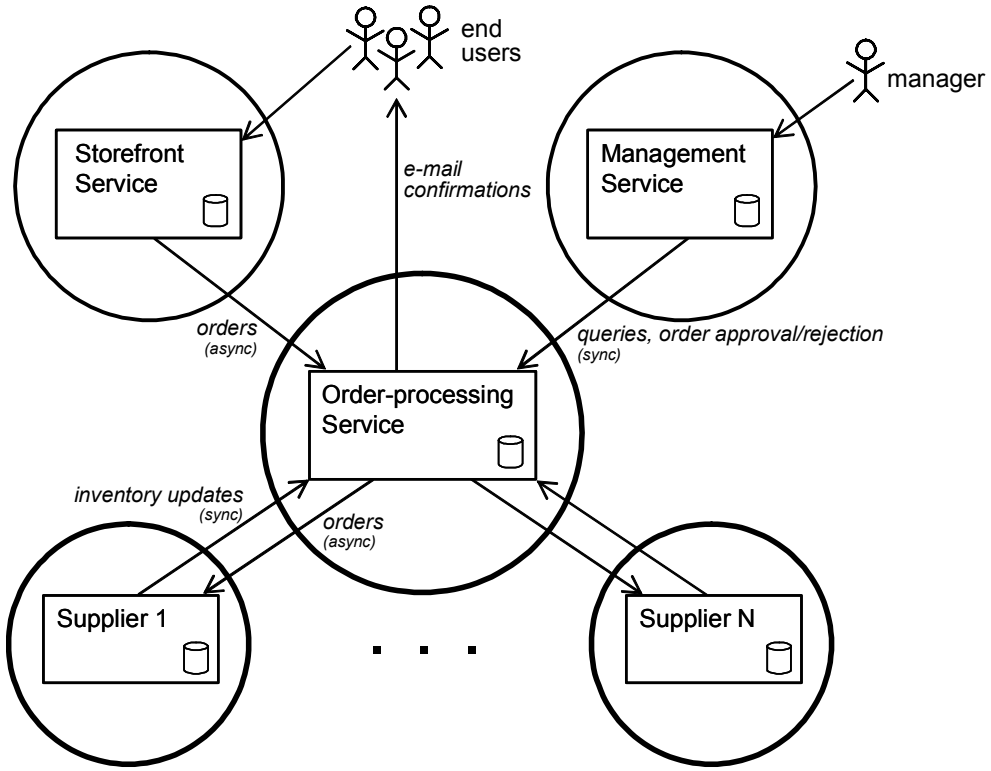


Figure 9: PetStore architecture as spheres of undo. Each subservice of the PetStore architecture naturally maps to its own sphere of undo.

- the **storefront service**: a web application that provides the online storefront functionality
- the **management service**: a web application that provides management functionality, allowing the store's manager to view order statistics and to manually approve or reject any orders totalling over $500
- the **order-processing service**: a standalone application that handles order processing, accepting orders from the storefront service, providing statistics to the management service, fulfilling orders by sending them to suppliers, and notifying end users when their orders are filled.

The core services are augmented with one or more **supplier services**, which accept orders from and supply inventory updates to the order-processing service. Each service maintains its own state repository and can be run on a separate node from all the others, if desired.

6.2 Mapping PetStore to spheres of undo

Our first task in providing undo to PetStore is to divvy up the architecture into spheres of undo. The initial partitioning is trivial: since PetStore is comprised of multiple services, each with its own independent state repository, the natural division is to assign each service to its own sphere of undo, as Figure 9 shows. The PetStore service as a whole is represented by the composition of the individual SoU's for the storefront, the management service, the order-processing service, and the suppliers. This structure provides the flexibility of allowing problems in any of the subservices to be recovered independently using undo.

Requiring a bit more thought is the choice of how the subservice SoU's are composed together, that is, which of the composition models discussed in the previous section we will use to structure the PetStore service as a whole. The choice depends partly on an assumption about the independence of the subservices: for the purposes of this discussion, we will assume that the three core services are run by one organizational entity, whereas each supplier is its own organization that may not trust the core-services organization. Given this assumption and the guidelines in Section 4.2, above, the suppliers naturally become uncoordinated SoU's. This allows them to use undo unilaterally, without forcing a rollback of the entire core service set, and likewise allows the core service set to use undo without forcing rollback of all suppliers. However, the choice to make the suppliers uncoordinated means that we will need to handle paradoxes between the core services and the suppliers; we return to this issue below.

For the core services themselves, we have the choice of composing them as either uncoordinated or coordinated spheres. Here, coordinated spheres make more sense, since we assumed above that the core services are run by the same organizational entity, and because they are reasonably tightly-coupled, such that rewinding or undoing one is likely to significantly affect the others in ways that would be difficult to compensate for. So in the end, our sphere model is that the three core spheres (storefront, management, and order-processing services) are coordinated spheres, with the suppliers treated as uncoordinated spheres.

6.3 Verbs, consistency, and compensation

Each subservice in PetStore has its own set of input and output verbs, discussed in turn below.

6.3.1 Storefront service.

The storefront service handles requests from end users, and therefore requires a set of external input verbs corresponding to the various state-altering user requests. We define our external consistency model such that paradoxes only occur if *submitted orders* are affected by undo; thus, for example, a user who has only viewed a product's information but not bought it will not be informed if that information changes later as a result of undo. This policy means that we need to define verbs for and log only those operations that affect orders: adding and removing items from a shopping cart, ordering the contents of the cart ("checkout"), and updating user information like credit card numbers and shipping addresses. We would not treat any of the cart operations as externalizing—as carts do not represent orders until checkout—but rather would treat the checkout operation as externalizing the contents of the cart. Thus, paradoxes to the end user occur only when an order is different upon checkout; they are detected by the checkout verb and could be handled by a policy such as informing the user of the paradox if the post-undo order is less expensive than the original order, and cancelling the order while informing the user that they may resubmit it if the post-undo order is more expensive.

On the output side, the storefront passes order requests to the order-processing center via an asynchronous message queue. Because we are using coordinated spheres, we will never need to detect paradoxes at the crossing between the storefront and order-processing spheres, and so we need not define compensations at that point. We can, however, encode the act of putting an order into the message queue as a synchronous inter-sphere output verb associated with the storefront's CHECKOUT input verb.

6.3.2 Management service

For the management service, we again have external requests that need to be converted to verbs and managed for paradoxes. Here, the key requests are to fetch a list of orders pending approval, and to approve or reject orders. We define our external consistency model such that we care about paradoxes in what orders have been (or need to be) rejected or approved. Thus, we need to define verbs for the requests to fetch the order list and to

approve/reject orders; output paradoxes should be detected on the input verb requesting the order list, as it externalizes the list of pending orders, and can be handled by informing the manager of the undo-induced changes using the same mechanism used to inform him or her that new orders await approval. We also need to detect paradoxes if previously-accepted approvals/rejections fail during replay; this detection is easily handled in the approve/reject verbs, and can be compensated the same way as for paradoxes in the order list. On the output side, the management service's requests to the order processing service can be encoded as synchronous inter-sphere output verbs. No compensation or paradox detection is needed across this inter-sphere boundary due to our coordinated-spheres composition.

6.3.3 Order-processing service

The situation at the order-processing service is more complicated, as it produces external output to both end users (as e-mail order confirmations) and to the uncoordinated supplier spheres (as orders). The input verbs for the order-processing service are straightforward: they include orders received from the storefront (as inter-sphere input verbs), requests for the pending order list and approve/reject requests from the management service (as inter-sphere input verbs), and inventory updates from the suppliers (as external input verbs). Output produced by the order-processing service includes synchronous responses to the management service's requests, asynchronous e-mail notifications to end users, and asynchronous order submission to suppliers. The synchronous responses to the management service require no special handling, as they are associated with an inter-sphere input verb and thus no paradoxes can occur across that communication boundary (although they may appear later when the management service externalizes information to *its own* external users).

Where the complications arise is in detecting paradoxes in the e-mail outputs to end users and in the orders sent to the suppliers. In both cases, these are asynchronous events, not outputs that arise synchronously during processing of an input verb. Further complicating matters is that each of these outputs arises through the interaction of more than one input verb. In fact, outgoing orders to suppliers can potentially depend on three or more input verbs: a submission from the storefront, an approval from the management service, and one or more inventory updates from the suppliers. The same is true of the outgoing e-mails, which are sent at the same time that their corresponding order is sent to its suppliers. So we cannot encode supplier orders or end user e-mails as synchronous output verbs attached to a specific input verb, but rather they must be mapped to asynchronous output verbs. Note that these are *external* output verbs, not inter-sphere, and thus we must include them in our paradox detection and handling.

With asynchronous external output verbs, paradoxes are handled by matching output verbs and compensating for mismatches, as described in detail in Section 9. Matching is not difficult in this service context: we simply ensure that the output verbs are tagged with the unique, time-invariant order ID assigned originally by the storefront's incoming order request (which will always be there, regardless of how many other verbs contribute to the generation of the asynchronous output verb), and match output verbs based on that order ID. Our consistency model for this external output is simple: paradoxes are generated by any undo-induced changes to existing orders, or orders that disappear as a result of undo; newly-appearing orders can be ignored. To handle paradoxes to end users, we can simply send a new e-mail message explaining the changes to their orders, and offering them the option to cancel them (using whatever facility the system already offers for cancelling existing orders). To handle paradoxes to suppliers, we send requests to the suppliers to cancel and reissue altered orders, or to cancel orders that have disappeared. If those requests fail, for example because the supplier has already shipped the order, then we have to invoke some higher-level compensation, such as writing off the loss or providing the order recipient with instructions on returning the order.[2]

6.3.4 Supplier services

Finally, we have the supplier services. These are uncoordinated spheres that can be rolled back and altered unilaterally; as such, they can produce paradoxes to their external consumers (namely, the order-processing service). They therefore need paradox detection and handling for all output that leaves their spheres. The input verbs for a supplier are the order submission requests from the order processing service, as well as requests to cancel existing orders. Order submission requests produce no external output as they are delivered asynchronously, and thus, while logged as verbs, can produce no paradoxes. Requests to cancel orders, used in the order-processing service's paradox handling, are synchronous and produce a fail/success output. Paradoxes occur with these verbs when previously-successful cancels fail after undo; such paradoxes can be detected easily within the cancel-request input verbs, and handled by informing the order-processing service, which can take those cancel-failed-requests and carry out the higher-level compensation described above.

Supplier services also must define asynchronous output verbs reflecting inventory updates to the order-processing service. These are asynchronous with respect to the input verbs because they may be triggered by external events like the arrival of a new wholesale shipment of items, or by manual inventory updates (which should also be encoded as response- and paradox-free input verbs to the supplier). Since the inventory update outputs are asynchronous, they must be treated using the same match/compensate paradox handling strategy described above. At the end of a supplier's undo cycle, the post-undo inventory-update outputs are matched to the pre-undo outputs (presumably by the unique IDs of the inventory items they specify). Any discrepancies represent paradoxes. Paradoxes can be handled in this case by simply issuing new inventory updates to the order-processing service to correct its view of the supplier's inventory.

7 Related Work

The idea of providing recovery across cooperating but disjoint systems has been well-explored in the database community, in the form of distributed transactions [13], workflow-based long-running transaction models [11] [23], and more generally in the concept of Spheres of Control [2] [7] [8]. Our spheres of undo compositions are the analogue of that work for undo-based recovery, where our basis for recovery is replaying verbs on a rewound system, rather than aborting or backing out transactions.

The connections between our work and the traditional database work are most evident at the highest level of abstraction, where our composition of spheres of undo (SoU's) are natural analogues to the composition of Spheres of Control (SoC's) in the database domain [7]. SoU's and SoC's share common goals, but in the end serve different purposes. In both cases, the goal is to provide an abstraction that contains potentially-inconsistent state from outside observation; for SoC's, those inconsistencies arise from uncommitted transactions within a sphere, whereas for SoU's they result from manipulation of the sphere's state timeline via actions like rewind. SoC's protect the external world from uncommitted data, whereas SoU's protect the world from rewound state. SoU's do this statically, as rewindability of state and the granularity of rewind are static property of the state stores making up the system; SoC's are a more dynamic construct, flowing to enclose the current transaction and all data it touches. Both kinds of spheres use the idea of compensations to define what happens when that state does escape the sphere.

While SoU's deal with rewound verbs rather than uncommitted transactions, it is possible to recast some forms of the undo problem in a transactional, SoC context. In particular, our case of interacting spheres of undo has a natural analogue in SoC's. To see this, imagine that each *external input* verb flowing into a SoU

2. While these higher-level compensations sound drastic, most retailers already have similar procedures in place to be used when order processing errors occur due to human error, bad inventory data, or system failure.

starts a transaction. These verb-transactions *never commit*, at least, not until the verb is expunged from the history due to the log filling up. Each such verb-transaction defines a new SoC containing its effects. If the verb results in the creation of output verbs, those verbs fall into the input verb's SoC. If the output verbs represent requests to uncoordinated spheres or external users, that is the equivalent of data escaping a SoC, and thus a compensation must be defined (as is required in our SoU composition framework). If the output verbs go to coordinated spheres, they propagate the original SoC along with them, and so the resulting inter-sphere input verb in the new SoU, and any output verbs it generates in turn, run within that original SoC.

Under this formulation, undo (or rewind) of a SoU can be seen as aborting all of the transactions corresponding to the rewound external input verbs. Aborting those verb-transactions causes their corresponding SoC's to abort; any data which has escaped the SoC ("committed" to the outside world) must then be compensated for, using the defined compensations. Redo can be seen as resubmitting the original history of transactions for re-execution.

Since SoC's in this formulation map onto distributed multilevel transactions [13], we could implement interacting-spheres undo in a database system given that it supported the distributed multilevel transaction model. But in doing so, we would lose some undo power compared to a direct implementation using verbs, undo managers, and spheres of undo. One major reason for this is that our undo framework presupposes that undo *will* occur, unlike the transaction model where transaction aborts are supposed to be rare. Transaction systems are typically not designed to keep every transaction open for weeks or months, which is what would be required if transactions only commit when they fall off the end of the log; keeping all those open transactions would require significant long-term state in the transaction manager. In contrast, in our undo formulation, no state beyond the log entry is required to track verbs that have completed execution.

More importantly, the transaction-based approach requires that all transactions abort and then re-execute in order to perform rewind/replay. Aborting all transactions requires compensation for all externally-visible results of those transactions, even if those transactions were to behave identically during re-execution! This means that a SoC/transaction-based undo would cause a slew of compensations to uncoordinated spheres and external users, even if the undo cycle was replayed without repairs or cancelled. In contrast, our approach of using verb-based consistency checks *during replay* defers compensation until it is actually needed—when a verb behaves differently enough during replay to warrant compensation. Verbs that behave the same during replay do not require compensation.

Furthermore, verbs offer three significant benefits over transactions for an undo system. First, they are a shadow representation of the actual application protocol traffic, and thus can be built as an add-on to an existing distributed service environment; in contrast, a transaction-oriented approach would require that the services be rewritten to communicate directly using multilevel distributed transactions. Second, verbs provide a framework for encoding an application-specific external consistency policy, making it possible to leverage the weak consistency guarantees of many network service applications to avoid compensations in cases where the external discrepancies are minor. Finally, verbs are more flexible than transactions in that they do not impose as many constraints on isolation, atomicity, and consistency, and can encode operations at the intent level, without an explicit understanding or record of the state affected by the verb's execution.

Moving beyond the database context, we find work in several other areas that is related to our composition approach for interacting spheres. One such area is a niche of the user interface community exploring time-centric user interfaces. Rekimoto's Time-Machine Computing is the best example: it is a system for time-aware applications that lets users view their desktop, file system, and application state as they exist currently or at dif-

ferent times in the past [19]. Time-Machine Computing defines a notion of **time-casting**, where different time-aware applications or computers can inform each other of their respective notions of current time, such that when the user rolls back one application (say, a calendar), then other related applications (say, an e-mail client) roll back as well. Time-casting is very similar to our idea of coordinated spheres of undo, except that it only provides a *view* into the past, not the ability to truly undo the system to the past and make retroactive changes, as in our system-wide undo model. In addition, time-casting assumes that the same user will be using all the temporally-affected applications. Taken together, these last two points imply that time-casting need not deal with external output or paradoxes, which vastly simplifies its model but makes it inappropriate for our needs.

Another area of related work lies in the fault tolerance community's investigations of recovery via distributed checkpointing and message logging, as surveyed in the excellent paper by Elnozhay et al. [10]. In this work, a model very much like our coordinated interacting spheres is assumed *a priori*—the system model for most of the checkpoint-recovery work is of a set of processes that communicate with messages, where each process logs its incoming messages and supports unilateral rollback of its internal state to earlier checkpoints. Recovery in this model is undo-like: a failed node rolls back to a checkpoint, and propagates that rollback as necessary to other nodes until the system has reached a consistent point; then, logged messages are replayed to bring the system back up to the current state. Much of the work in this area is involved in optimizing the algorithms for message logging and checkpointing to minimize the amount of global rollback and replay needed, in coordinating the replay process, and in dealing with nondeterministic events (as summarized in Lowell et al. [16]).

The key difference between the fault tolerance checkpoint/logging recovery approach and our interacting spheres of undo is in how we handle external output (output either to human end users or to uncoordinated spheres of undo that cannot accept propagated rollbacks). In the fault tolerance work, this kind of output is simply prohibited: once output has been exposed beyond the set of coordinated processes, it is "committed" and the system may not roll back beyond that point [10] [20]. In contrast, our undo-based approach explicitly allows rollback past external output points: the novelty of our work in this report is in how we manage rolled-back external output across multiple nodes of a distributed system, by tracking it via output verbs, analyzing it for inconsistency with output verb match predicates, and handling it via a framework that encodes application-specific compensations.

Rollback-recovery and message logging protocols can also be found outside of the domain of fault-tolerant computing. A particularly interesting example is Jefferson's Time Warp mechanism [14], which, while designed as an optimistic causality-protection algorithm for distributed message-passing systems, has some commonalities with our spheres of undo. Time Warp deals with nodes and messages; nodes keep a log of their arriving messages. If a node discovers that it has processed a message out of order (based on a logical timestamp), it rolls back in an undo-like operation to before the out-of-order message's timestamp, then proceeds to re-execute from that point. As part of that undo operation, the node must deal with any other nodes to which it has exposed data—just as our coordinated spheres of undo must propagate rollback to any other spheres to which they have sent output verbs. Time Warp accomplishes this with a clever **anti-message** mechanism, where as the node rolls back, it sends an anti-message for each message that it had previously sent. An anti-message annihilates its corresponding message if that message has not yet been processed, or forces a cascading rollback otherwise. Unfortunately, Time Warp, like the fault-tolerant rollback recovery protocols, does not deal with external output, requiring that any external output be deferred until all earlier messages have committed, and

denying rollback after that point. However, the anti-message approach (adapted as **anti-verbs**) would be an ideal way to optimize distributed rollback in the coordinated-spheres composition.

Finally, one might expect to find connections between our SoU composition models and yet another area of related work, multi-user undo, based on the fact that both involve managing distinct but related histories in a distributed environment. In the most expressive formulations of multi-user undo, like those seen in Timewarp [9] or Suite's Asymmetric Command model [6], each user has their own history/timeline and can manipulate it at will, with the system eventually reconciling the multiple timelines as needed.[3] However, in a multi-user undo model similar to Timewarp, all of the users' histories refer to the same common, shared piece of state (typically a document being edited). The problem of composing histories in multi-user undo is different from the compositions of spheres of undo, where each of the many histories in the system is associated with its own distinct partition of state. One could formulate multi-user undo in the SoU context by treating users as being in their own uncoordinated SoUs with their own copies of state, with the multi-user reconciliation protocol treated as an externalizing communication between SoUs; this is especially true when reconciliation or transmittal of new history commands is done via explicit data transfer commands, as in Suite [6] or Sun et al.'s real-time cooperative editing model [21]. But the converse is not true, and thus we see that our SoU compositions can model distributed undo in a broader class of systems than can be shoehorned into a multi-user undo model.

8 Summary and Conclusions

In this report, we extended our earlier work on system-wide undo to handle broader classes of service systems: we considered how undo might be designed and implemented for distributed collections of interacting services. We appealed to the construct of the sphere of undo, using spheres of undo to wrap each independent member of a distributed collection of services, and defining approaches to composing those spheres together. We saw how interacting spheres of undo can be treated as either *coordinated* or *uncoordinated*: in the former case, undo operations on one sphere that retroactively affect information exposed to other spheres result in the undo operation propagating to all the affected spheres; in the latter case, undo operations remain confined to a single sphere, with compensating actions applied to any spheres that have observed retroactively-altered information. We also defined criteria for choosing between the coordinated and uncoordinated approaches, based primarily on the possibility of compensation but also touching on issues of trust, security, and efficiency.

To demonstrate the importance of undo for distributed services, we showed how interacting spheres of undo could be used to model e-commerce shopping services as typified by Sun's J2EE PetStore; even this basic example illustrated how many interesting services could benefit from an undo system spread across a mixture of coordinated and uncoordinated spheres of undo.

While we have not implemented the architectural extensions needed to cope with interacting spheres of undo, we have laid out a plan for the changes that need to be made to verbs, the verb log, and the undo manager algorithms and implementations at each interacting sphere. More details on the implementation plan are given in Section 9.

3. There are other forms of multi-user undo where all users logically share the same global history and thus the same timeline. While these undo models (such as Abowd & Dix's model [1] or that of DistEdit [21]) sport multiple users acting on the timeline, all those users still see the same timeline and same version of state, and hence should be considered to reside within the same sphere of undo. Thus these multi-user undo models do not provide any comparative insight into our SoU composition approaches.

Finally, we saw how the problem of undo for distributed services bears strong resemblance to similar problems in many different areas in computer science, but differs fundamentally in the requirement embodied in our undo mode that we be able to rewind past externally-visible events to effect retroactive repairs of problems that have already manifested themselves to other services and to the external world. This requirement set our single-sphere undo model apart from most existing work, and it does the same here for the distributed case.

9 Appendix: Details and Implementation

This appendix contains further details on interacting spheres of undo (SoU's). We begin with a full discussion of request cycles, which, if present, can prevent a set of services from being treated as a collection of independent interacting SoU's. We then move on to discuss the details of how output paradoxes might be handled during undo of coordinated and uncoordinated spheres, and propose algorithms for rewind and replay of both coordinated and uncoordinated spheres of undo.

9.1 Request cycles

There are communication patterns between interacting services that, if divided up by interposing multiple spheres of undo, will lead to system deadlock and the inability to roll back the system at all. These communication patterns involve **request cycles**. To understand request cycles, begin by considering the overall service as a collection of interacting components or modules—the potential spheres of undo into which the service might be decomposed. Consider only the subset of communication between these modules that occurs in the form of synchronous request-response interactions (such as RPC or function call invocations); we will see below that non-request-response interactions are not an issue. For each module m, construct a directed graph as follows, where the nodes in the graph represent all the modules in the service. Draw an edge from m to another node p if any request made to m causes a request to be made *synchronously* to p as part of fulfilling the original request to m. For each such node p, draw an edge to all nodes q where one of m's requests to p requires that p make a synchronous request to q. Repeat this process until all request paths originating from m have been considered. This graph defines the potential communication patterns needed to fulfill any request to m. A simple example is illustrated in Figure 10.

A **request cycle** (hereafter **R-cycle**) exists for a service if for any module m in the service, the graph constructed as above for m contains a cycle. In Figure 10, the left graph is R-cycle-free, but the right graph has a R-cycle. In other words, if a request to some module can cause a sequence of requests that propagates back as a new request to the original module that arrives *before* the original module has finished servicing the original request, then there is an R-cycle in the system. R-cycles can cause deadlock, and can prevent multi-sphere coordinated rollback by causing infinitely-recursing rollbacks.

In more detail, R-cycles are a problem because individual spheres of undo guarantee serializability for their stream of input requests: by construction in our single-sphere undo architecture, input requests (verbs) are sequenced before execution such that they can be consistently recorded in a serial log that, when replayed, will produce an equivalent execution to the original sequence of requests. Were we to treat the modules of a service with R-cycles as separate spheres of undo, then this serializability guarantee would result in deadlock when a request propagated through a cycle and arrived back at the original sphere that generated it. To guarantee serializability, that sphere would have to defer the incoming (propagated) request until it completed the original request, but doing so would prevent the original request from completing, hence deadlock.

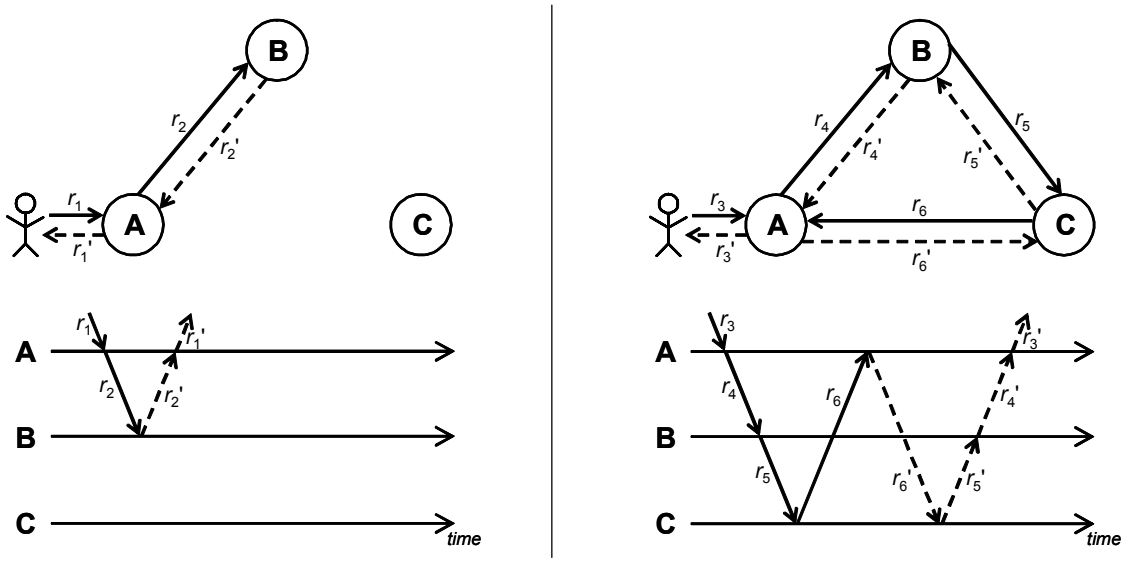


Figure 10: Examples of request-response graphs. A, B, and C are system modules. Synchronous requests are represented by solid lines, and responses by dashed lines. Below each example graph is the communication pattern associated with it. In the case depicted on the left, servicing external request r_1 to A requires that A makes a synchronous request r_2 to B. In the case depicted on the right, servicing external request r_3 to A requires that A makes a synchronous request r_4 to B, which in turn makes a further synchronous request r_5 to C, which then makes a synchronous request r_6 to A.

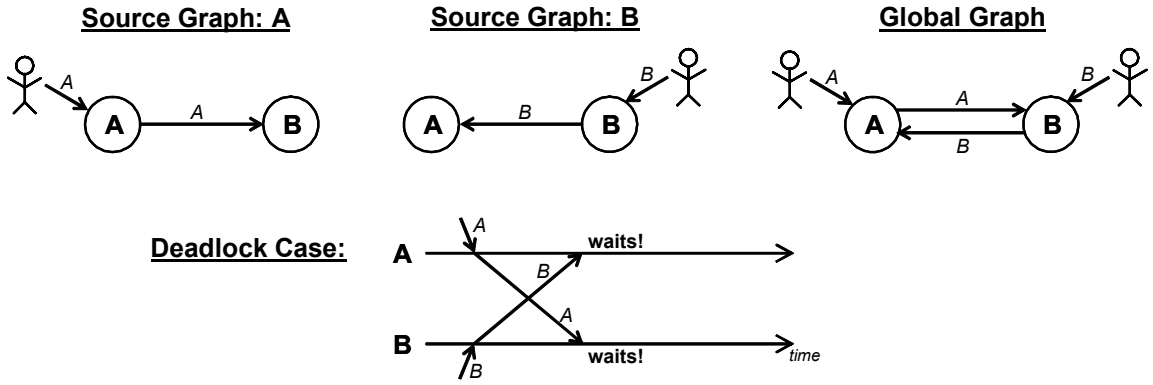


Figure 11: Example of deadlock in R-cycle-free system. The system of two nodes A and B is R-cycle-free, but deadlock can still occur. The global graph for the system, formed by annotating the edge of each source graph with the associated source and then composing all the source graphs, contains a cycle that can cause deadlock, as shown in the timeline at the bottom of the figure. Deadlock can be avoided in this case if the external input verb to A commutes with the input verb to A generated by B's request, or if the external input verb to B commutes with the input verb to B generated by A's request.

Furthermore, even if each of the per-module graphs constructed above is R-cycle free, additional criteria must be met to avoid deadlock due to interacting patterns of requests that on their own are cycle-free but produce cycles in composition. Figure 11 shows a simple example of how this can occur. Since these cycles result only from unfortunate timing of R-cycle-free request patterns, we should be able to avoid deadlock if, at points where the individual patterns converge on the same nodes, we can execute the patterns in parallel without violating serializability. More specifically, consider a global graph constructed as follows. First, extend each of the per-module graphs as constructed above so that their edges are annotated with the name of the module that sourced the requests (*i.e.*, all edges in the graph for module m are tagged with m). Then construct the global

graph by taking the union of all the extended per-module graphs. The criteria for multi-sphere decomposability over this global graph are:

- the extended per-module graphs must be free of R-cycles; and
- the global graph must be cycle-free, or
- for nodes participating in cycles where the incoming and outgoing edges in the cycle have different annotations, the input verb corresponding to the incoming edge's request must be independent of and commute with the input verb that results in the production of the outgoing edge's request.

Informally, these criteria state that deadlock can be avoided by ensuring that arriving requests in a cycle can always execute in parallel with ongoing requests. They also ensure that a globally-serializable history can be constructed, which will prove useful later on.

Thus any service with R-cycles, or any service without R-cycles but that does not meet the above criteria, cannot be decomposed into separate spheres of undo. Such services must be lumped into a single sphere, where no serializability constraints need be imposed within its boundaries. Note that non-request-response communication patterns do not suffer from the problem of R-cycles, as cycles of unidirectional communication do not result in deadlock: a message traveling through a cycle that arrives back at the original SoU can simply be deferred until the original message that triggered the cycle is processed and logged, resulting in a serializable history.

9.2 Replay and paradox handling for interacting spheres

In this section, we pick up where we left off in Section 5 and continue to develop the details of a potential approach to implementing undo for systems of interacting (coordinated and uncoordinated) spheres of undo.

9.2.1 Preserving and correlating history

At the end of Section 5, we identified the problem of output paradoxes, or undo-induced changes that affect outbound requests from a sphere of undo. We also identified two areas of our undo architecture that need to be extended to address the output paradox problem: the timeline log architecture, which needs to be enhanced to preserve multiple versions of history; and the replay algorithm, which needs to be modified to match and compare output verbs to detect paradoxes between the different versions of history.

First we consider the issue of preserving history. For self-contained services, we only preserved the original execution history, and compared it to the replay history on a verb-by-verb basis as verbs were replayed. This is insufficient for the multi-sphere case, since input verbs can change, disappear, or appear as a result of repairs that affect inter-sphere communication patterns.

Thus we must change our log architecture to allow a sphere of undo to record a new verb history during replay without discarding the history from the original execution. This can be done by building the notion of **epochs** into the log. The beginning of each undo cycle defines a new epoch; epochs are committed when the undo cycle completes, and are discarded if the undo cycle is cancelled. Because of our committable-cancellable undo model, we can get away with only two epochs, which simplifies the implementation task—at any point, we only need consider the differences between the current state of the system (the current epoch) and the most recent prior state of the system (the prior epoch, regardless of how many undo cycles have happened to produce that last state).

Epochs can be implemented most easily by maintaining two verb logs with a copy-on-write policy. The original verb log represents the prior, committed epoch. A new verb log is created when a new epoch begins. Upon commit, the changes recorded in the new log are swapped into the old log. How the new log is formed from the contents of the old log depends on whether the sphere of undo is coordinated or uncoordinated; we will discuss these policies below, in the following two sections. Note that this two-log approach is similar to the shadow-page technique used to implement transaction buffer management in some database systems [12]; in many ways, our undo cycles represent large-scale transactions over the system's history, and thus the similarity in approaches is natural.

With a dual-epoch history in place, we can now look at the other needed architectural extension: an approach to comparing the histories to detect and handle paradoxes. The specific details of how these comparisons fit into the replay process depend on whether we are dealing with coordinated or uncoordinated spheres, so we defer discussion of those details to the following two sections. But in all cases we need a basic piece of functionality: correlating two sets of output verbs, one from the original epoch and one from the replay epoch.

The algorithm we propose for this is straightforward. We require that each output verb define two routines:

- `match(v)`: compares this output verb to another to determine if they produce the same effect on the outside world, relative to the service's external consistency model;
- `compensate()`: invoked when an original-epoch output verb is no longer relevant during replay. This function must take whatever steps are necessary to compensate for or explain the discrepancy to the external target of the output verb.

These routines are directly analogous to the paradox detection-predicate and handler routines required of input verbs; the first helps to detect paradoxes resulting from changes to external output verbs, and the second handles any paradoxes that are found. Note that for deferred output verbs that have not yet executed, compensation is a no-op.

Given the `match` and `compensate` routines, the undo manager can reconcile two sets of output verbs, one from the original epoch and one from the replay epoch, by taking each replay-epoch output verb in turn and trying to match it to an original-epoch output verb. If a match is found, the replay-epoch verb is swallowed, the stored response in the original-epoch history is used to service the external output request, and the original-epoch output verb is copied to the appropriate place in the replay-epoch log. If no match is found (because the replay-epoch output verb is new or a modified version of an original-epoch verb), then the new verb is executed and inserted into the replay-epoch log. Once all replay-epoch verbs have been examined, the compensate routine is invoked on all remaining unmatched original-epoch verbs. The details of what goes into the two sets of verbs, and of how the match-and-compensate procedure fits into the overall algorithm for undo, depends on the situation; we will discuss the details for the uncoordinated and coordinated spheres cases in the following two sections.

9.2.2 Replaying uncoordinated interacting spheres

The architectural enhancements described above provide much of the framework we need to implement undo involving uncoordinated interacting SoU's. The SoU that wants to undo drives the process. Within that SoU the undo cycle proceeds as usual during the rewind and repair phases. Replay is more complex, as it is necessary to detect and compensate for paradoxes visible to the uncoordinated SoU's; in this case, paradoxes can result

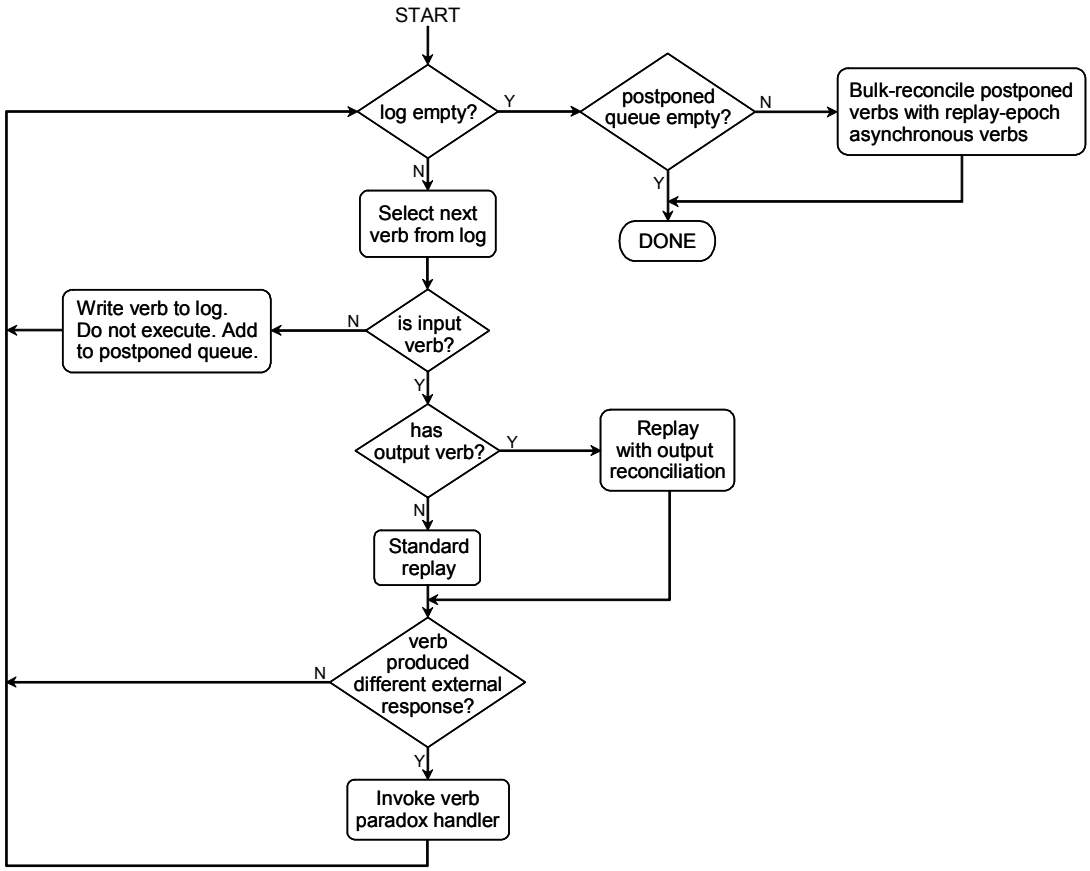


Figure 12: Flowchart for replay of uncoordinated spheres. The flowchart describes the operation of replay for an uncoordinated but interacting sphere of undo. "Standard replay" means to replay an input verb using the same approach as for a self-contained service. "Replay with output reconciliation" means to replay an input verb while trapping output verbs, reconciling them against the original history, and handling resulting paradoxes, as described in the main text. "Bulk reconcile" means to reconcile two sets of asynchronous output verbs, handling resulting paradoxes, as described in the main text.

from responses to inbound external requests, and from synchronous, deferred, and asynchronous outbound external requests.

Our algorithm for replaying an uncoordinated SoU works by sequentially re-executing every input verb in the SoU's original-epoch verb history. As a verb is executed, it is copied into the history for the replay epoch. Verb execution depends on the type of verb, as illustrated in the flowchart in Figure 12. For input verbs with no associated output verbs, replay proceeds just like in the self-contained service case. Any response produced by the input verb is trapped by the undo manager, recorded in the copy of the verb in the replay-epoch history, and compared to that of the original epoch via paradox-detecting predicates in the input verb; discrepancies invoke compensation routines in the input verb.

For input verbs with associated synchronous or deferred output verbs, replay is tricky. If we were to simply replay output verbs against their target SoU's or external entities, we may end up duplicating their effects. Here we start to see one of the limitations of the uncoordinated-spheres approach. The best we can do is try to suppress external requests that do not change as a result of the undo and repairs, substituting the recorded responses from the original-epoch history. But requests that appear for the first time during replay need to be executed in order to return results to the replaying SoU, as do requests that change as a result of the undo, so

we are forced to allow them to execute. Requests that change, or that simply disappear, require compensation to be applied to the external sphere that originally received them.

To implement this policy, we rely on the match/compensate functionality described in the previous section. In this case, we must carry out the reconciliation process on a verb-by-verb basis. As an input verb is replayed, it may generate synchronous or deferred output verbs. As each such verb is generated, it must be intercepted by the undo manager and matched against the set of output verbs stored in the original-epoch history's record of the input verb being replayed. If a match is found, the output verb's request is swallowed and the recorded response is copied to the replay-epoch history and returned to the verb's invoker. If a match is not found, the output verb's request is allowed to execute. When the input verb completes, any remaining non-matched output verbs in the original-epoch history for that input verb are identified, and their `compensate` routines are invoked.

To ensure that changes to compensations across repeated undo cycles are preserved, the compensations themselves should be recorded in the input verb; any significant discrepancies found between the compensations across epochs should result in further compensation. Thus the input verb must also define a way to record compensations. If this is done by treating compensation as extra external output (or perhaps extra deferred external output verbs, depending on the form of compensation), then the compensations can be lumped in with the output/requests stored for the current epoch, and the existing infrastructure for detecting paradoxes in output can be used to detect and compensate for paradoxes in the compensations as well.

Finally, the matter of replaying asynchronous output verbs need to be addressed. These are a challenge because they can arise non-deterministically; even if they are sequenced into the original execution history, they may not arise at the same point during replay, so we cannot necessarily do the match/compensate procedure when we first encounter such verbs in the log. One way to handle this would be to defer all of the asynchronous output verbs that arise during replay to the commit stage of the undo cycle, and carry out the match/compensate procedure at that point. This results in a bulk reconciliation, where all asynchronous output verbs from the original-epoch history are reconciled with the entire set generated during replay. Again, matching verbs are swallowed and their original-epoch versions are copied into the replay history; non-matching replay-epoch verbs are executed and logged into the replay history; and non-matching original-epoch verbs are compensated and *not* transferred into the replay-epoch history (although their compensations may be added to it). One potential downside to this approach is that deferring the execution of asynchronous verbs could adversely affect the behavior of later verbs in the history; more investigation is required to explore the impact of this technique.

A simpler solution for asynchronous verbs is to just refuse to support paradox detection and compensation for truly non-deterministic asynchronous output verbs. For some services, where asynchronous output verbs can be made deterministic, this may be a reasonable approach. Asynchronous output verbs arising as a result of background processes, or asynchronous operations like draining an overfull queue, can be made deterministic by stopping the background processes during an undo cycle, then triggering them manually and synchronously when the asynchronous verbs are encountered during replay. This approach "converts" the non-determinism (in the terminology of Lowell [16]) and makes paradox detection possible just as for a normal synchronous verb. Note that not all asynchronous output verbs can be made deterministic; we saw examples of these in the order output verbs for the e-shopping service described earlier in Section 6, where the output verbs depended on the confluence of several input verbs and not a suspendable background process.

Asynchronous output verbs may also arise through error reporting mechanisms—that is, as a way to report unusual conditions to an operator or user as they occur. Such reports are typically useful even if undo has occurred, so it may not be necessary to bother with paradox detection and handling for these cases.

To summarize, the following changes are needed (beyond the enhanced log/verb architecture described in the previous section) to support undo of uncoordinated interacting spheres:

- output verbs must be defined to represent external requests, and must include match routines;
- the undo manager must be modified to intercept and record synchronous and deferred output verbs generated during execution of an input verb;
- compensations (both for output verbs and traditional external output) must be encoded into additional external output or output verbs in the replay-epoch history;
- asynchronous output verbs must be made deterministic during replay, intentionally ignored, or postponed until replay completes and reconciled in bulk at that point.

9.2.3 Undo for coordinated interacting spheres

Implementing undo for coordinated interacting spheres is far more difficult than for the uncoordinated case. While there is less need to deal with compensations, we have all the complexity of coordinating distributed rollback. Furthermore, we have the additional challenge of dealing with *distributed* external output to non-SoU's, like end users. Were we to just adapt a distributed rollback and re-execution protocol from the checkpointing literature, we could produce a consistent replay within the coordinated SoU's of the system, but we would leave the end users of those SoU's, and of any uncoordinated SoU's, rather confused.

To see why this is a problem, consider the case illustrated earlier in Figure 7. In this case, after undoing SoU A, external entity C sees a paradox in the external output of B. A traditional distributed checkpoint/rollback algorithm will ignore the paradox, or prevent it by prohibiting the undo operation entirely. But we would prefer to permit the paradox and handle it. This means we must somehow recognize that the two different input verbs corresponding to requests to B (one during the original epoch, one during replay) are somehow related, so that we can compare their output and perform the necessary compensation. This is a difficult problem, more so when multiple spheres, long request chains, or multiplexing of requests is involved.

A simple black-box solution. One way to tackle the problem is to simply restrict our system model until the problem goes away. This means that any coordinated SoU cannot produce output to external entities (end users or non-coordinated SoU's), unless in direct response to an *external inbound request*. In other words, a SoU's only *external* output comes as responses to external requests made directly to that SoU. Or, in terms of verbs, only *external* input verbs can produce external output; no input verbs can have associated *external* output verbs; and asynchronous *external* output verbs are prohibited. Input verbs can produce inter-sphere output verbs, and asynchronous inter-sphere output verbs are allowed. In the example of Figure 7, such a restriction would disallow B from making a request to C, although C could make a request to B. We could relax the restriction slightly by allowing external output with the caveat that output produced by these verbs will not be examined for paradoxes and may be repeated during replay.

This restricted system model is equivalent to drawing a single large sphere of undo around the coordinated SoU's and treating that large sphere as a self-contained service. The inter-sphere communication patterns within that large sphere become a black box, just as we treated the inner workings of a self-contained service as

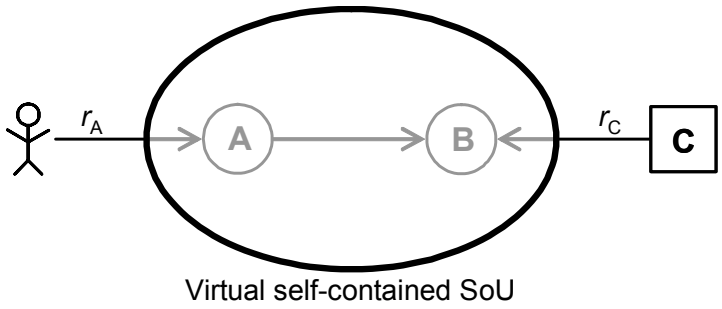


Figure 13: Restricted model for coordinated multi-sphere undo. The coordinated spheres A and B are treated as if they form a single larger self-contained SoU with respect to external output. The only output allowed from that virtual self-contained SoU is in response to external requests, like r_A and r_C.

a black box in our self-contained-service undo architecture. Figure 13 shows this approach for the example of Figure 7. Despite its restrictions, the model still holds some utility: it is a way to take a self-contained service and distribute it across multiple separately-undoable systems. In addition, since each coordinated SoU can have its own external users (as long as those external users make requests, rather than receiving them), each of the constituent subservices in this restricted model can still provide its own independent service, as long as that service follows a pull, rather than push, model.

For example, a distributed e-mail service is one that would work in this model: each e-mail store with its own users represents its own sphere of undo, and the entire e-mail network is modeled by a coordinated composition of those SoU's. Users interact with their local mail store using a pull model based on IMAP and SMTP. By invoking coordinated undo at a given e-mail store node, the administrator of that node could "unsend" e-mail erroneously delivered to any other node, or track down and recall corrupt or virus-laden e-mails that had propagated to other nodes in the system.

Figure 14 presents an algorithm for coordinated undo in the restricted system model described above, with the relaxation of allowing un-checked and potentially duplicated output from inter-sphere input verbs and asynchronous output verbs. It takes a very simple approach to rewind, assuming that the entire set of interacting SoU's rolls back together at the start; a practical implementation would want to optimize this, so that spheres are rolled back lazily, and only if their request streams are altered during replay.[4] It likewise takes a simple approach to replay, stepping through the global history of external inputs one at a time. The algorithm makes a few key assumptions about the system:

- the existence of a master process in control of the undo cycle. This could be a separate entity or a role assumed by the first sphere S_0 on which undo is invoked;
- a global notion of time that can be used to reconstruct a globally-serializable history[5] of external input verbs across all coordinated SoU's. This could be implemented by globally synchronizing clocks using a protocol like NTP [17], or by using a logical clock or virtual time approach that piggybacks the source LSN onto requests sent to other spheres of undo [14] [15]. The algorithm uses the timebase via a mapping function `time(s,lsn)` that translates a LSN for a given sphere `s` to a global time;

4. One way this could be accomplished is by adapting an anti-message protocol like that in Jefferson's Time Warp system [14], where "anti-verbs" are sent during replay only if output verbs change.
5. Note that global serializability is guaranteed by the criteria discussed in Section 4.1.

```
UndoCoordinatedSpheres(S0, lsn):
    // Performs undo for sphere of undo S0, rewinding it to LSN lsn.
    U ← Rewind(S0, lsn)
    // repairs happen here
    Replay(U)
    Commit(U)

Rewind( S0, l ):
    // Builds undo set U and performs coordinated rewind with master sphere S0
    U ← { S0 }
    W ← S0.rewindTrackback(time(S0, lsn))          // rewind S0
    while (W ≠ { })                    // W holds the set of involved but not-yet-rewound spheres
        for (s ∈ W)
            X ← S0.rewindTrackback(time(S0, lsn))
            U ← U ∪ { s }
            W ← (W ∪ X) \ s
    return U

Replay(U):
    // Carries out replay across all members of undo set U
    while ((<s, λ> ← ComputeNextVerb(U)) ≠ undef)      // λ is the LSN of the verb
        s.replayVerb(λ)

Commit(U):
    // Commits the undo cycle on each involved sphere
    for (s ∈ U)
        s.commit()

ComputeNextVerb(U):
    // Figures out which sphere has the next verb to replay, in a global ordering of input verbs
    t ← ∞
    s ← undef
    λ ← undef
    for (p ∈ U)
        m ← p.getNextExtVerb()
        if (m ≠ undef)
            t1 ← time(p, m)
            if (t1 < t)
                t ← t1
                s ← p
                λ ← m
    if (s = undef)
        return undef
    else
        return <s, λ>
```

Figure 14: Algorithm for replay of coordinated spheres. This algorithm, run by the master process in a coordinated-spheres undo cycle, defines the operation of replay for coordinated, interacting spheres.

- a way to distinguish inbound external requests from inbound inter-sphere requests. This could take the form of static knowledge of which request sources offer coordinated undo.

The algorithm also relies on several routines that should be implemented as extensions to each sphere's undo manager. These are, briefly:

- `replayVerb(LSN)`: the undo manager should replay the input verb with the specified original LSN, copying it into the replay-epoch history, recording its external output as usual, and detecting and handling paradoxes in the external output (relative to the output stored in the original-epoch history). After replaying, the sphere's virtual time should be advanced past the specified LSN.
  As with the uncoordinated sphere case, paradox compensations should be recorded in the replay-epoch history as additional external output. Note that verbs that are not explicitly replayed by this API are not transferred from the original-epoch history to the replay-epoch history; however, inter-sphere verbs that arrive during replay are recorded in the replay-epoch history.
- `commit()`: commit the undo cycle on the sphere, making the replay-epoch history permanent and resuming normal operation.
- `getNextExtVerb() -> LSN`: returns the LSN of the next not-yet-replayed external (*not* inter-sphere) input verb.
- `rewindTrackback(time) -> {spheres}`: rewinds the sphere to the specified time, and scans the rewound portion of the original-epoch history, examining each inter-sphere output verb and accumulating a list of spheres that received output from this sphere. Returns that list of spheres.

Allowing more external output: bulk matching. If we do want to allow coordinated SoU's to generate synchronous or deferred external output tied to inter-sphere input verbs, or asynchronous external output, then we need a more complicated algorithm. The algorithm must capture that external output during replay and perform a match/compensate reconciliation against the external output from the original execution, in order to detect and handle potential paradoxes.

The difficulty here is that it is non-trivial to define the sets of verbs to use in the matching algorithm. Unlike in the uncoordinated-spheres case, an input verb in the original-epoch history may not exist in the replay-epoch history (if it is an inter-sphere input verb), so it is not obvious how to match such a verb's external output requests to corresponding requests in the replay-epoch history.

If we are still willing to have a somewhat restricted system model, we can get around this complication by only allowing inter-sphere input verbs and asynchronous output verbs that generate deferred or **response-free** output requests. This lets us postpone execution of these output verbs to the end of the replay process, and reconcile the complete original- and replay-epoch output histories in bulk, without reference to how that output is generated. Essentially, we intercept and set aside all external output verbs during replay, and once replay is complete, we do the match/compare reconciliation between that set of verbs and the set of all external output verbs from the original-epoch history.

We can fit this approach into the algorithm of Figure 14, above, by making the following modifications:

- each sphere's `getNextExtVerb()` considers both external input verbs and asynchronous output verbs
- each sphere's undo manager intercepts all external output verbs (synchronous, deferred, and asynchronous) generated during replay and stores them in a queue without executing them (and returning null responses to their generators). The output verbs are, however, recorded in the history of the replay epoch.
- each sphere's `commit()` routine walks through the replayed portion of the original epoch's log, extracting external output verbs. It then performs matching, reconciliation, and compensation between that set of verbs and the queued replay-epoch verbs, updating the history records for the replay-epoch verbs based on the result of each verb's matching.

An alternate approach to the verb-based matching would be to hand the two histories of external output to an application-specific reconciler that can compare the histories and determine what needs to be done to ensure that no paradoxes exist. Writing such a reconciler is in general a difficult problem, but it affords more opportunity for intelligent, application-specific compensation (such as correcting for the difference between two similar verbs, like two versions of the same e-shopping order with different sales tax computations, rather than cancelling the first and executing the second).

Allowing more output: per-verb matching. Dropping the assumption of response-free output requests makes things even messier. Now we cannot postpone output requests, and so we must find a way to immediately correlate output requests generated during replay with their counterparts from the original execution history. One way to do this is to tie output requests to some entity that is guaranteed to exist across both execution histories—for example, the external input verb that ultimately originated the communication pattern that lead to the generation of the output request. We can do this if we make the following assumptions:

- all output verbs can be correlated to an input verb;
- requests that cross spheres can carry out-of-band information;
- an external input verb tags any generated inter-sphere output requests (synchronous output verbs) with the LSN of the external input verb and a reference to the sphere processing that input verb;
- these tags propagate unchanged through all inter-sphere communications on the out-of-band channel and are recorded along with the inter-sphere input verbs; and
- inter-sphere input verbs propagate the tags unchanged with any inter-sphere synchronous output verbs they generate.

Essentially, these assumptions let us tie the output of any inter-sphere input verb back to the original external input request that lead to it, regardless of how far back in the communication chain that request goes. An alternate, more radical approach would be to dispense with the application requests and out-of-band channels entirely, and instead handle inter-sphere communication by shipping source-tagged verbs directly between the undo managers of the interacting spheres. Both of these approaches represent a significant rethinking of the undo architecture, trading the transparency we strove for in our original self-contained-services undo architecture for tighter integration of application and undo manager. The second approach in particular would make

```
Replay(U):
    // Carries out replay across all members of undo set U
    while ((<s, λ> ← ComputeNextVerb(U)) ≠ undef)      // λ is the LSN of the verb
        for (p ∈ U)
            p.prepReplay(s, λ)          // prepare all spheres for replay of next verb
        s.replayWithMatch(λ)
        for (p ∈ U)
            p.advance(s, λ)             // complete output matching and advance time
```

Figure 15: Replay method for coordinated-spheres undo that handles response-generating output verbs. This method replaces the Replay( ) method in the algorithm of Figure 14.

the undo manager crucial to the normal-case operation of application service, but at the same time would simplify and streamline coordinated distributed undo; this is an approach that deserves further investigation.

In any case, given the ability to map an output verb back to the external input verb that ultimately originated it, we can replace the Replay method of the undo algorithm in Figure 14 to arrive at an algorithm for coordinated undo that handles response-generating output. The basic idea is to do the match/compensate reconciliation on each *external input* verb as it is executed, much like in the uncoordinated undo case, but to do the reconciliation across all spheres that participate in servicing that input verb's request.

The new replay method, shown in Figure 15, requires three new APIs for the per-sphere undo managers:

- `prepReplay(sphere,lsn)`: this is a signal that the original external input verb at LSN `lsn` in sphere `sphere` is about to be replayed (note that this sphere is likely not the same as the one executing `prepReplay`). Upon seeing this signal, the undo manager combs the original-epoch history for inter-sphere input verbs tagged with the specified `<sphere,lsn>` identifier, indicating they originally resulted from that external input verb. Any external output verbs associated with these input verbs are set aside.

- `replayWithMatch(lsn)`: the verb at LSN `lsn` is copied into the replay-epoch history and re-executed. If an external (non-inter-sphere) output verb is generated, it is trapped by the undo manager. The undo manager compares the new output verb to the set of output verbs accumulated during `prepReplay`, using the output verbs' `match` routines. If a match is found, the old verb is copied into the replay-epoch history, and its saved response is returned. Otherwise, the new output verb is copied into the replay-epoch history and executed normally.

- `advance(sphere,lsn)`: this is a signal that the original external input verb has completed replay. The undo manager takes any remaining unmatched output verbs in the set generated during `prepReplay`, and executes their `compensate` methods. The virtual time in the sphere is then advanced past the time corresponding to the specified sphere/LSN pair.

Essentially, what we are doing is using the original, ultimate external source that instigated a piece of output as a way to correlate output across the original and replay-epoch histories. As before, we match output verbs, execute any new and changed external output requests, and compensate for any missing ones, but now we do it on a verb-by-verb basis rather than in bulk at the end of the replay process.

Note that, just like in the uncoordinated case, this approach does not work for asynchronous output verbs, as they are not correlated to any input verb. At this point, a workable solution for these verbs is an open question. The bulk-reconciliation and non-determinism-conversion approaches from Section 9.2.2 should be applicable here, but may have deleterious side-effects by altering the communication patterns between spheres, causing additional paradoxes to form in the non-asynchronous requests.

Summary. We have demonstrated algorithms for providing coordinated undo in several restricted situations. If output is constrained to responses to external inputs, then we can relatively easily provide undo without requiring significant changes to the paradox detection and compensation architecture: the application developer still only need provide verbs with paradox detection predicates over their external output (responses) and corresponding paradox handlers. This model fits distributed services that logically provide a self-contained service, and is the approach that should be taken wherever possible.

We saw that we could relax this model to support response-free outbound requests to external services, at the cost of having to supply a reconciliation routine that takes two histories of output verbs and attempts to identify and handle paradoxes resulting from their differences; this is significantly more challenging than handling paradoxes at external input verbs because the understanding of user intent captured by the input verb is not available to guide paradox handling.

Finally, we saw that we could even support non-response-free outbound requests, at the cost of even more complexity by having to piggyback tags onto requests as they flow through between spheres. This approach still requires reconciliation based only on output, although at a finer granularity than the previous method; because the input verb ID is known during this reconciliation, it might be feasible to draw on that representation of intent, although the mechanism for implementing this is not obvious.

References

[1] G. D. Abowd and A. J. Dix. Giving Undo Attention. *Interacting with Computers*, 4(3):317–342, 1992.

[2] L. A. Bjork. Recovery Scenario for a DB/DC System. *Proceedings of the 1973 ACM Annual Conference*. Atlanta, GA, 1973, 142–146.

[3] A. B. Brown. A Recovery-Oriented Approach to Dependable Services: Repairing Past Errors with System-wide Undo. *Ph.D. Dissertation, EECS Computer Science Division, University of California, Berkeley.* December 2003.

[4] A. B. Brown and D. A. Patterson. Rewind, Repair, Replay: Three R's to Dependability. *Proceedings of the 10th ACM SIGOPS European Workshop*. St. Emilion, France, September 2002.

[5] A. B. Brown and D. A. Patterson. Undo for Operators: Building an Undoable E-mail Store. *Proceedings of the 2003 USENIX Annual Technical Conference*. San Antonio, TX, 2003.

[6] R. Choudhary and P. Dewan. A General Multi-User Undo/Redo Model. *Proceedings of the Fourth European Conference on Computer-Supported Cooperative Work (ECSCW '95)*. Stockholm, Sweden, September 1995, 231–246.

[7] C. T. Davies. Data Processing Spheres of Control. *IBM Systems Journal*, 17(2):179–198, 1978.

[8] C. T. Davies. Recovery Semantics for a DB/DC System. *Proceedings of the 1973 ACM Annual Conference*. Atlanta, GA, 1973, 136–141.

[9] W. K. Edwards and E. D. Mynatt. Timewarp: Techniques for Autonomous Collaboration. *Proceedings of the ACM Conference on Human Factors in Computing Systems (CHI)*. Atlanta, GA, March 1997.

[10] E. N. Elnozahy, L. Alvisi, Y. M. Wang, and D. B. Johnson. A Survey of Rollback-Recovery Protocols in Message-passing Systems. *ACM Computing Surveys*, 34(3):375–408, 2002.

[11] H. Garcia-Molina and K. Salem. Sagas. *Proceedings of the 1987 ACM SIGMOD International Conference on Management of Data*, San Francisco, CA, 1987, 249–259.

[12] J. Gray, P. McJones et al. The Recovery Manager of the System R Database Manager. *ACM Computing Surveys*, 13(2):223–242, 1981.

[13] J. Gray and A. Reuter. *Transaction Processing: Concepts and Techniques*. San Francisco: Morgan Kaufmann, 1993.

[14] D. R. Jefferson. Virtual Time. *ACM Transactions on Programming Languages and Systems*, 7(3):404–425, 1985.

[15] L. Lamport. Time, clocks, and the ordering of events in a distributed system. *Communications of the ACM*, 21(7):558-565, 1978.

[16] D. E. Lowell, S. Chandra, and P. Chen. Exploring Failure Transparency and the Limits of Generic Recovery. *Proceedings of the 4th Symposium on Operating System Design and Implementation (OSDI 2000)*. San Diego, CA, October 2000.

[17] D. L. Mills. Internet time synchronization: the Network Time protocol. *IEEE Transactions on Communications*, 39(10):1482–1493, 1991.

[18] B. Randell. System Structure for Software Fault-Tolerance. *IEEE Transactions on Software Engineering*. 1(2):220–272, 1975.

[19] J. Rekimoto. Time-Machine Computing: A Time-centric Approach for the Information Environment. *Proceedings of the 12th ACM Symposium on User Interface Software and Technology (UIST '99)*, 1999.

[20] R. E. Strom and S. Yemini. Optimistic Recovery in Distributed Systems. *ACM Transactions on Computer Systems*, 3(3):204–226, August 1985.

[21] C. Sun, X. Jia *et al*. Achieving Convergence, Causality Preservation, and Intention Preservation in Real-Time Cooperative Editing Systems. *ACM Transactions on Computer-Human Interaction*, 5(1):63–108, 1998.

[22] Sun Microsystems. *Java BluePrints: Java Pet Store Demo 1.3.1*. http://java.sun.com/blueprints/code/jps131/docs/index.html, 2003.

[23] H. Wächter and A. Reuter. The ConTract Model. In *Database Transaction Models for Advanced Applications*, A. Elmagarmid (Ed.). San Francisco: Morgan Kaufmann, 1991, 219–263.