"Faithless Replay" for Persistent Logless Mid-Tier Components

David Lomet

Microsoft Research
Redmond, WA
lomet@microsoft.com

Abstract. *A goal for enterprise applications has been to provide "exactly once" execution regardless of system failures. This has classically required "stateless" applications that manage their states explicitly via transactional resource managers. Support for "stateful" applications requires the system to do more to manage state, which has been considered too difficult and costly. The Phoenix/App system manages application state transparently by logging interactions between components to guarantee "exactly once" execution. But logging for availability and scalability requires that the log be accessible by multiple processors. For middle tier session-oriented components, interactions need not be logged in order for them to be recoverable. Because there is no logging, performance of failure-free execution is excellent and no log need be moved to relocate the application on a different processor. Functionality has been restricted however to only idempotent interactions with backend servers. Here we show how such logless components can execute non-idempotent reads and yet be recoverable, by exploiting "faithless replay" where the original execution path might not be re-executed. The result is highly capable recoverable session-oriented components that are easily redeployed within an enterprise application system that requires high availability and scalability.*

1. Introduction

1.1 Robust Applications Dilemma

To be robust, enterprise applications need to survive system crashes and provide exactly once execution semantics, i.e. results that are equivalent to a single failure free execution of the program. In addition, applications must be re-deployable on other computers as the system changes and grows to provide availability and scalability. An application may start execution on one computer, that system crash, be redeployed on another, etc., and to the application client, it should look like a single seamless failure-free execution.

Business logic requires that many enterprise applications, e.g. an online e-commerce web site, have state. A "stateful" application program has control state across transaction boundaries and incurs the risk of losing state should the system on which it executes crash. In the past, "stateful" applications have compromised robustness, making it impossible to guarantee exactly once execution. Crashes created a "semantic mess" that could require human intervention to repair the state and result in long outages.

Classic transaction processing [6,7,11,19] required that application programs be "stateless", which means "no meaningful control state" is retained across transactions. This stateless model forces an unnatural "string of beads" programming style where an application must, within a transaction, first read its state from, e.g., a transactional queue, execute its logic, and then commit the step by writing its state back to a transactional queue for the next step. "State" is not avoided. Rather, it is managed in a transactional way. That has a performance impact due to the logging and message costs of transactions. It also impacts scalability, as redeploying an application requires access to its stored state. Finally, such applications can require two phase commit, which greatly restricts their adoption in an internet setting where sites usually insist on local autonomy.

An application programmer thus has faced a dilemma, having to choose between:

- ***stateful application programs*** which are easier to write and where the system manages state, but which fail to provide an exactly once execution guarantee; and
- ***stateless application programs*** which require explicit state management, require more log forces, and can compromise site autonomy, but which can guarantee exactly once execution.

1.2 Phoenix/App Stateful Persistence

Phoenix/App [1,2,3] provides persistent components (Pcom's) that can be stateful while surviving system crashes and transactional components (Tcom's) that have testable transaction state, as in transaction processing. Other component types have other requirements. We focus on Pcom's because persistence guarantees exactly once execution and can facilitate availability and scalability.

Phoenix/App transparently logs the interactions of each Pcom to enable its deterministic replay, without this being visible in the application program. The log captures nondeterministic events (messages between components)

and their potentially non-deterministic arrival order. We assume that the Pcom execution is piece-wise deterministic, i.e. it is deterministic between the logged nondeterministic events. This ensures that Pcom's can be replayed after system crashes and be recovered independently of other components. This logging satisfies the requirements of "interaction contracts" [4, 5], which define obligations for each Phoenix/App component. These contracts require components to guarantee that their state and messages will survive system crashes and provide exactly once executions.

Logging enables a Pcom to engage in relatively unconstrained activity with other Pcom's and Tcom's while being recoverable via replay. A system including Pcom's and Tcom's is illustrated in Figure 1.

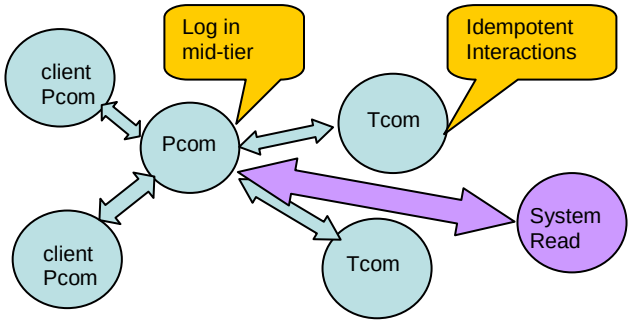

Fig. 1. The Phoenix/App system model: Multiple *clients* have multiple message interactions with a mid-tier *Pcom* that does logging. The mid-tier *Pcom* interacts with multiple backend *Tcom*'s in multiple transactions**.** A Pcom can also read data from external read-only sources because it will log the results returned before any subsequent write "exposes" its state.

1.3 Eliminating the Middle Tier Log

We introduced **logless components** (LLcom's) in [13], a new **session-oriented** component type that **avoids logging** while being persistent and stateful. An LLcom exploits the logging done already by other components. It can be called multiple times and interact with a number of backend systems involving a number of transactions, while retaining *persistent* state. LLcom's can be easily redeployed across an enterprise system since no log needs to be shipped. In addition to their availability and scalability advantages, logless components perform better during normal execution because no logging is required, indeed no interception of messages is required.

LLcoms provide persistence by constraining its activities such that their nondeterministic events (messages, results) are captured by other components within the multi-tier system. These captured events enable an LLcom to be faithfully replayed (re-executed) so as to reconstruct its complete state, including its control state.

1.4 Non-idempotent Reads- a New Capability

To provide persistence without logging, LLcom's must be restricted in what they can do. As defined in [11], all interactions initiated by the middle tier at the back end must be idempotent. That is, when the interactions are re-executed, the backend recognizes duplicate requests, ensures that the requests are executed only once at the backend, and that the same result is returned for the resubmitted request.

The contribution of this paper is in showing how logless components can read system state without the need for these reads to be idempotent. This is very important as

- pure reads are rarely idempotent and there is a serious burden to make them so. Backend servers do not customarily provide read idempotence.
- reads facilitate examining important information, e.g. descriptions and prices of items, in deciding how execution should proceed, e.g. which brand of goods to purchase.

How can non-idempotent reads be permitted when only logged information at both client and backend can impact the persistent state of an LLcom? Reads are permitted only in a context in which the execution path of the LLcom can be forced back to the path that was followed during the original execution. We introduce two mechanisms for enabling this.

1. We generalize the notion of idempotence that is provided at back end services. This enables read results to vary without changing the back end state.
2. We introduce "wrap up" reads that do not impact middle tier state in the current method call, but can return results to the client that impact subsequent execution of both client and middle tier.

In these cases, the read is followed by logging that captures the logical impact of the first execution, including the read results seen by that first execution. A read, when replayed, can produce different results. We call this ***faithless replay***. This is not unconstrained in its "faithlessness". Rather, it is what might be called "locally" faithless, while being "globally" faithful. It is the first read whose effects are captured on some log (client or backend) that will determine subsequent state and execution of the middle tier component.

With these mechanisms, it becomes possible for logless session components to do a non-idempotent read that permits it to make choices as to future action of two forms.

1. choosing which of a number of actions to take a an already chosen web site. This can be done with no additional logging.

2. choosing the web site at which to perform actions. This will require additional logging, though not in the middle tier, and we will show two ways in which this can be accomplished.

Logless components already impose rules on what an application programmer can do within the program. This is usually called a "programming model". We no longer have the kind of natural stateful programming permitted when persistent components are logging. However, the system infrastructure continues to manage state under the more limited logless component requirements. In particular, it is not necessary to separately identify the portions of the state that need to persist and explicitly write them to a transactional resource manager. So, while care is required in structuring the program, much of the naturalness of writing a stateful program remains.

The benefits of logless components in terms of availability and scalability are fully preserved. Without a middle tier log, components can be freely redeployed to achieve either or both of these benefits. And, because there is no log in the middle tier, the performance of these components during normal execution is comparable to a program written without the robustness attributes needed for enterprise applications.

1.5 Paper Organization

We describe logless components and their advantages in section 2. Section 3 describes what we call generalized idempotent requests (GIR's), one of the techniques permitting non-idempotent reads. We illustrate the use of a GIR in enabling choices to be made about what is done at a given backend service. In section 4, we attack the problem of how to enable a logless component to choose to which backend service to send requests. This expands the capabilities permitted of a mid-tier logless component to encompass great flexibility, but does require some additional logging. We illustrate this with an example as well. Related work is briefly discussed in section 5. We end with a discussion of further issues and a brief summary in section 6.

2 Logless Components

2.1. A New Persistent Component

Here we describe the initial constraints that we imposed on persistent components to avoid logging [13]. Of course, if logging could always be avoided, we would have over-engineered Phoenix/App. But that is not the case as fully general Pcom's need to log interactions.

LLcom's do not usually require additional logging or log forces from the components with which they interact. Indeed, we can sometimes advantageously reduce some logging. To other components, the LLcom can usually be treated as if it were a Pcom, though these components may be required to keep messages stable for longer periods.

Availability and scalability are possible with Pcom's that log, but this requires shipping logs. Being logless eliminates even this step. As when a log is involved, a component will need to be replayed in order to re-create the crash interrupted state of a component. But the cost of maintaining the log, forcing the log, and shipping the log are all avoided with logless components.

2.2 Making Replay Possible

To provide application replay, one must remove the nondeterminism that, during replay, would produce a different execution path and a different set of outcomes. Here we describe how this non-determinism is eliminated, through a combination of restrictions on capability and exploitation of logging that is being done elsewhere.

For LLcom's to provide persistence without logging, we restrict all interactions to be of the request/reply variety. This satisfies a very large class of applications, and indeed is consistent with the way that enterprise applications are typically structured. We need more than this, however, and we describe this below.

Functional Initiation

One source of nondeterminism is how components are named and mapped to the underlying physical resources. That nondeterminism must be removed without having to log it, as done in Phoenix/App.

LLcom must have what we call a ***functional*** initiation or creation, i.e., from what is in its creation message, the entire information about the identity of the LLcom must be derivable. This requirement permits a resend of this creation message to produce a component that is logically indistinguishable from any earlier incarnation. The initiating component can, in fact, create an LLcom multiple times such that all instances are logically identical. Indeed, the initiator component might, during replay, create the LLcom in a different part of the system, e.g. a different application server. The interactions of the LLcom, regardless of where it is instantiated, are all treated in exactly the same way. During replay, any Tcom or Pcom whose interaction was part of the execution history of the LLcom responds to the re-instantiated LLcom in exactly the same way, regardless of where the LLcom executes.

Determinism for Calls to an LLcom

Pcom logging removes the nondeterminism resulting from the order of calls to it. Its methods might be invoked in a nondeterministic order from an undetermined set of client components. We usually know neither the next method invoked nor the identity of the invoking component. Both these aspects are captured in Pcom's via logging.

LLcoms have no log upon which to rely. However, if we restrict an LLcom to serving only a single client Pcom (i.e., we make it ***session-oriented***) we can exploit the fact that the client Pcom is *already* capturing this sequence of method calls. Thus, we require that an LLcom be initiated from a client that is a Pcom and only serve calls from this initiator.[1] Since a Pcom has its own log, it is capable of being recovered based on its local log. When the client Pcom recovers, it also recovers the sequence of calls to an LLcom with which it interacts. This single client limitation results in an LLcom that can play the role, e.g., of a J2EE session bean.

Determinism for Calls from an LLcom:

An LLcom's execution must, based on its prior execution, be able to identify the next interaction as to the kind of interaction (send or receive) and with which component. If it is a message send, then clearly this is accomplished via replay as it is the component's deterministic execution that leads to the message send. It is the receive interactions for which determinism needs to be provided.

Truly nondeterministic receives are excluded. However, for a receive message that is a reply to a request, the message is from the recipient of the request message, and the reply is awaited at a deterministic point in the component execution. The multi-call optimization [1] showed us that forced logging after each interaction with a Tcom or Pcom is, in fact, not required. A Pcom can fail after some number of interactions, without having logged their replies. This means that one or more interactions subsequent to those on the log may have occurred and not have been logged. How do we deal with these?

A committed interaction contract (CIC) or transaction interaction contract (TIC) [4,5] requires that both components of an interaction must make an interaction durable in some way. In interactions between Pcom's and between a Pcom and a Tcom, at least initially, only a message sender must meet the guarantee. However, we require of components called by an LLcom the same guarantees for their reply messages, i.e., that they: (i) eliminate duplicates so as to enforce "exactly once" execution semantics, and (ii) to return on request their reply message. Thus a request/reply can be re-played multiple times while only producing a state change exactly once, and always returning the same result message. It is idempotence of the calls by LLcom's to "back end" servers that describes this "reliable" interaction replay.

So what LLcom's need is idempotence from the backend servers for the LLcom requests. Replay of the LLcom will retrace the execution path to the first called backend server invoked. That server is required, via idempotence to only execute the request once, and to return the same reply message. That same reply message permits the LLcom to continue deterministic replay to subsequent interactions, etc.

Figure 2 illustrates a deployment of an LLcom in the middle tier. Note that there is only a single client, which is a Pcom, interacting with the LLcom. The client's log will thus enable the calls to the LLcom to be replayed. Further, the only actions permitted of the LLcom at the backend are idempotent requests. Reading of other state is not permitted as it can compromise LLcom deterministic replay.

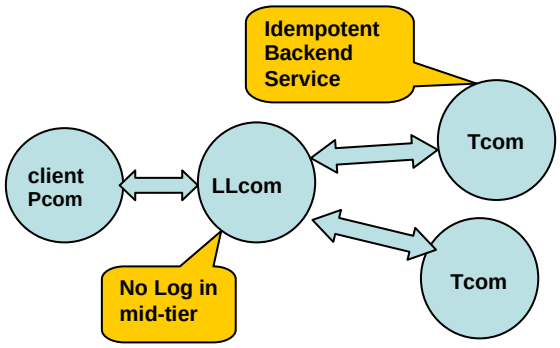

Fig. 2. The Phoenix/App system model with logless component in middle tier is illustrated. A single client can have multiple message interactions with a mid-tier LLcom that does not log. The LLcom interacts with multiple Tcom's in multiple transactions, all of which provide idempotence. Note that this does not permit non-idempotent reads.

3 Generalized Idempotent Requests

In [11], LLcom's are permitted only idempotent calls to backend servers. An LLcom could not read system state outside of an idempotent interaction. Idempotence imposes a strong requirement on the backend server. It must guarantee, e.g., via logging or transactional queues, that it will remember requests that have been successfully executed so as to be able to detect duplicate requests by not re-executing them, and it must return the same output results. Non-idempotent reads are precluded because their results during replay can differ from the initial execution, leading to divergent execution paths.

We can do better. We can permit non-idempotent reads if we can (i) guarantee exactly once execution at backend

[1] A bit more generality is possible, though it is both harder to define and more difficult to enforce.

servers that return the same result despite receiving different input argument values, and (ii) force the execution path of the logless component back to the path of its initial execution, hence "wiping out" the effects of the non-idempotent reads on subsequent LLcom state.

3.1 Exploratory Reads

Exploratory reads are reads that precede an invocation of a request to a backend server. These reads permit the middle tier application to specify, e.g., items included in an order, the pickup time for a rental car, the departure time for a subsequent flight, etc. That is, an exploratory read permits an LLcom to exploit information read outside of idempotent interactions in its dealings with a given backend service. In each case, the read is followed by the sending of an "idempotent" request to a backend service. We need to ensure that the backend server is "idempotent" (note now the quotes) even if exploratory reads are different on replay than during the original execution. Further, we need to prevent exploratory reads from having a further impact, so that the execution path of the LLcom is returned to the path of its original execution.

3.2 Generalized Idempotent Requests (GIR's)

We begin with backend servers. Usually, in the transaction processing community, and in transactional components (Tcom's), the responses of backend servers are considered to be idempotent. Thus, if a duplicate request is received, it will not be executed. Rather, it will be recognized as a duplicate, and a reply message identical to that of the first execution will be returned. "Idempotence" is typically achieved not by remembering an entire request message, but rather by remembering a unique request identifier which is a distinguished argument, perhaps implicit and generated by, e.g., a TP framework. This technique will surely provide idempotence. Should an identical message arrive at a server, it will detect it as a duplicate and return the correct reply.

Requiring request identifiers to detect duplicate requests enables support for what we refer to as a **generalized idempotence** property. Thus, a server supporting generalized idempotent requests (**GIR**'s) permits each resend of a message with the same request identifier to have other arguments of the message that are different. This is exactly what we need in order to permit exploratory reads to have an impact on backend requests.

Idempotent requests (IR's) satisfy the property:

$$IR(ID_X, A_X) = IR(ID_X, A_X) \circ IR(ID_X, A_X)$$

where ID_X denotes the request identifier, A_X denotes the other arguments of the request, and we use "$\circ$" to denote composition, in this case multiple executions. Generalized idempotent requests (GIR's), however, satisfy a stronger property:

$$GIR(ID_X, A_1) = GIR(ID_X, A_X) \circ GIR(ID_X, A_1)$$

where A_1 represents the values of the other arguments on the first successful execution of the request. Thus, it is the effect produced and the result returned by the first successful execution of a GIR that determines the effect of subsequent executions, even when the other arguments A_X are different from the original arguments.

A GIR request ensures that a state change at the backend occurs exactly once, with the first successful execution prevailing. That it also returns the reply produced by this first execution makes it possible to both exploit the reply to control the subsequent course of calling LLcom execution and perhaps to limit the impact of the exploratory reads.

3.3 Exploratory Procedures

To complete our picture, we need to describe how we deal with the non-repeatability of reads at the LLcom. This requires that we limit the propagation of the effects of these reads in some way. There are several ways this might be done. We describe one way here, introducing what we call ***exploratory procedures***.

We require that whenever there is to be an exploratory read, it must be done within an exploratory procedure (**E-proc**). An E-proc always ends its execution with a GIR request to the same server, using the same request identifier on all execution paths. That is, it is impossible to exit from the exploratory procedure in any way other than the execution of a GIR request to a given server using the same request identifier. Because the request is a GIR request, it can have arguments other than its request identifier that differ from call replay to call replay, and it is always guaranteed to return the same result as its first successful execution. So the exploratory reading within the E-proc can have an impact on the GIR request, determining on its first execution, what the GIR request will do.

The reason for restricting exploratory reads to E-proc's is to limit their impact to only the arguments for the GIR request. We are helped in this by the fact that variables local to a procedure have procedure activation scope, and hence disappear when the procedure exits. Hence we can permit our exploratory reads to update local variables of the E-proc. However, we do not permit non-local variables to be updated from within an E-proc, prior to its GIR request.

Of course, we want our GIR request to be able to influence subsequent LLcom execution. Therefore, we permit the E-proc to return results, and set output parameters, based on the results of the GIR request. We know that the GIR request, should it be replayed, produces the same output as its initial execution. Hence, regardless of

the number of times the E-proc is replayed, its impact on LLcom execution following its return will be dependent only on the first successful execution of its contained GIR request. This is illustrated in Figure 3.

The net effect is that when E-proc's are replayed, their replay is "faithless", not faithful. But the faithlessness is confined to the internals of the E-proc. From the viewpoint of the LLcom calling the E-proc, the replay is faithful, and replay of the E-proc is, in fact, idempotent. Because the E-proc is idempotent, the execution at the level of the call to the E-proc is faithful. Hence the overall execution of the LLcom, in terms of its effects on backend services and on its client is also idempotent, which is exactly what is desired.

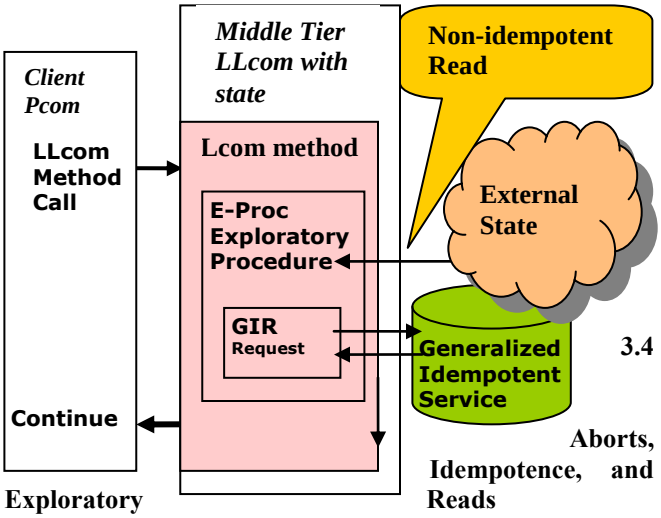

Fig. 3. Client Pcom calls a method of a stateful middle tier LLcom. The LLcom calls an E-PROC exploratory procedure that reads external state. At the end of the E-proc, a GIR-request. The reply from the backend service determines the result of the E-proc, which can then be used to change the state of the LLcom.

A difficulty with traditional stateless application technology is that it is difficult to deal with transaction aborts within the application. Because all code in a stateless application executes within a transaction, should the transaction fail, there is no persistent code executing that can deal with the transaction failure. TP monitors usually cope with this by automatically retrying the failed application a certain number of times. This is a successful approach for transient failures, e.g. system crashes or network problems, and difficulties such as deadlocks, which may not be an issue on subsequent executions. But for hard failures, due, e.g., to incorrect applications, the problem cannot be dealt with in this manner. For those failures, the TP system usually enqueues an error message on an administrator queue for subsequent manual intervention.

Stateful applications of the sort provided by Phoenix/App enable stateful applications to deal programmatically with transaction failures. Stateful Pcom's have persistent execution state outside of a failing transaction. When the transaction fails, the Pcom can inspect the error message (outside of the transaction), and provide program logic to deal with the failure. We would like to support this capability while using logless components.

However, aborts are not traditionally idempotent. Indeed, it is usually undesirable for them to be idempotent. Rather, a backend server "forgets" the aborted transaction, wiping its effects out of the state it maintains. This is fine for the backend, but it makes it difficult to provide deterministic replay for LLcom's.

What we propose here is to treat transaction abort like an exploratory read. Like a read, abort leaves the state unchanged when it is the outcome of a GIR request. By embedding a GIR request within an E-proc, we are in a position to respond programmatically to failure responses that indicate that the request was aborted. We can then include in the E-proc a loop that repeatedly re-executes the GIR request until that request commits. Further, we can use additional exploratory reads, and prior error messages, to change the arguments other than the request identifier of the GIR request in or effort to commit the GIR.

Including GIR requests, which might abort, within an E-proc thus lets us respond programmatically and make multiple attempts to commit the request. But this does not permit us to abandon the request. The problem here is that aborts remain non-idempotent. Thus, should the LLcom executing the E-proc abandon its effort to commit the transaction and then fail, then upon replay of the E-proc, its GIR request might this second time actually commit. Subsequent state of the LLcom will then diverge from its initial execution prior to the failure.

One way out of this "conflicted" situation is to provide an added parameter to the GIR request. This parameter permits the calling LLcom to request an idempotent abort result. Normally, we expect that the first several attempts at executing the GIR request, should these attempts be aborted, would be non-idempotent, permitting the request to be retried, perhaps with different values to arguments. But faced with repeated failure, requesting an idempotent abort permits the LLcom execution to proceed past this request. Once the GIR abort is stable, subsequent logic in the LLcom (but outside the E-proc where the GIR request was invoked) could try other backend services. Note, of course, that this does require the cooperation of the backend service, and is a facility not now normally provided. This is unlike the GIR functionality, where the usual implementation of idempotence relies on request IDs already.

3.5 Example Application

Consider an online travel service application. In our application, a client sets up a session with the service. The first method called authenticates the client using an idempotent request to the customer database. The session method loads customer preferences derived from the customer database into session variables and returns to the client. The client next invokes the trip planner routine, indicating the desired itinerary and the dates of the trip.

After the flight has been planned, the session proceeds to reserving the rental car. It is the method call involving the rental car that we focus on here, because it will exploit exploratory reads. The rental car company is asked about car availability and special deals. The customer preference, previously downloaded from the customer database, is to rent a convertible should one be available and the price be within 20% of the price of a midsized car. Let us say that in this case. After this exploratory read, the reservation is requested, via a GIR request, to the rental company. The rental car company commits the convertible request in its database and sends a reply. We show the session-oriented component rental method schematically in Figure 4. Figure 4 is simply our Figure 3, now with application specific names labeling the system elements.

The critical moment is after the convertible request for has been sent to the car rental company when the middle tier crashes. Then we must replay the session component. When we replay the rental method and check once again for convertibles, because this read is not idempotent, the result returned may indicate that good convertible deals are not currently offered. So the GIR "Request Car" request sent to the car rental company now asks for a midsize car. Nonetheless, because the first execution of the GIR request reserved a convertible, that reservation remains in effect. Further, what is returned to the middle tier session component is information pertaining to the convertible reservation, which is then returned also to the client.

Finally, when the client has also reserved a hotel, a final "print itinerary" method call can be invoked that prints the information that has been saved in the session component's state about the details of the trip. These details reflect the first successful execution, i.e. the convertible reservation.

We have illustrated, in Figure 4, how a session-oriented component can exploit exploratory reads and GIR requests to respond to state read by an exploratory read, use that state in conjunction with information previously acquired in a registration step, and make the first execution be the only execution that counts. Persistence of the component is assured via replay, and does not require any logging in the middle tier. This is a very simple example, but it does illustrate some of the flexibility provided by our session-oriented logless components.

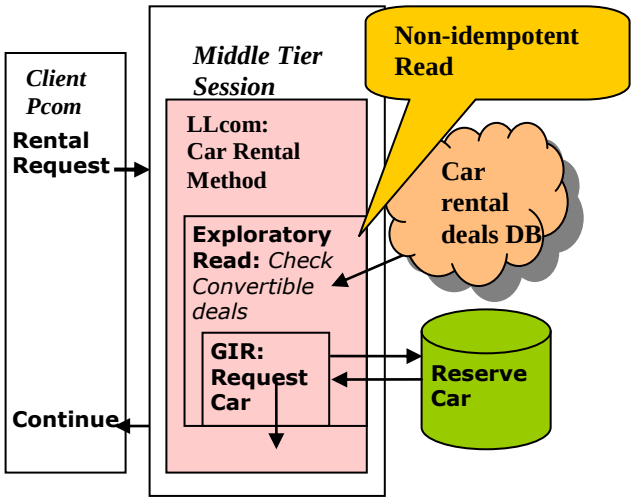

Fig. 4. A request is made to a car rental company, based on the result of reading about the deals offered by the company for a convertible

4 Reads to Choose a Backend Service

4.1 Limitations

To understand why we need more functionality than provided via GIR's and E-proc's, we must first understand their limits. In particular, we cannot use E-proc reads to select which back end service to call. So, for example, we cannot use an LLcom to explore prices, say for rental cars, and based on the price, chose the web site of the rental company offering the best rates on the kind of car we want.

The hard fact is that to choose a backend service requires the choice be stable, i.e., persist across system crashes. So the information that we have examined via our non-idempotent reads, enough for us to make this decision, needs to be recorded on a log somewhere, where it can be encountered during replay and used to persistently make the same choice each time the LLcom is replayed. The LLcom itself has no log. Further, the "log" at the backend service, which we exploit in E-proc's, requires that we know which service to invoke, which is not very useful here. We need another technique.

4.2 Wrap-up Reads

With E-proc's, backend logging provides idempotence and E-proc scope forces execution back to its original "trajectory". Here we use client Pcom logging for a similar purpose. This permits added flexibility as logging at the

client Pcom does not involve making any additional "persistent" choice (itself needing to be logged) for where the log is. We begin with a simple case, and generalize it subsequently.

Read-only LLcom Methods

Consider an LLcom read-only method, perhaps a method call that checks airline flights and prices and returns that information to our Pcom client. The client, after interaction with the user, proceeds to make a subsequent call indicating his choice of flights and intention to purchase tickets. The read-only method is, of course, NOT idempotent. Upon replay, the flights returned might be very different.

Despite the lack of idempotence, it is possible to permit such read-only LLcom methods. When the flight information is returned to the client Pcom, the client must ensure that it is logged (and forced) prior to its initiating a subsequent non-read-only call to the LLcom requesting a flight. The LLcom can now use the client supplied information to select some specific airline web site. This replay is deterministic because the client has force logged information derived from the non-idempotent read(s).

The client can avoid repeating the call to the LLcom during its replay as it already has the answer it needs on its local log. We could also replay the method call without any complication since it is read-only. Client replay is effective so long as the logged result of the original invocation is used during replay and we ignore the result of the replayed call. This permits the first successful read execution to "win", and here we exploit client Pcom logging to guarantee this.

Unlike when the client Pcom deals with an idempotent middle tier component, where log forcing is not required (though logging is useful for efficiency of client recovery), we now need the client Pcom to force the log prior to revealing state via a subsequent interaction, since the read-only method execution results in a non-deterministic event that is not captured in any other way.

Non-read-only LLcom Methods

We can exploit client logging in a general setting. The essential thing is precluding the LLcom from making state changes that depend upon non-idempotent reads. For LLcom methods which do some updates, e.g., via GIR requests, we can permit reading as the last activity (a "wrap-up" activity), after all backend requests and just prior to returning to the client. (A read-only method is a special case where there are no prior requests.) So long as this read activity does not change the state of the LLcom, it will have no further impact on LLcom state, except via subsequent client requests—after the client has logged the LLcom reply.

Subsequent client requests become replayable because of the logging at the client Pcom. However, as with exploratory reads, we need to ensure that wrap-up reads do not have any impact on subsequent LLcom state except via the information that is logged, in this case by the client Pcom. A technique similar to our E-proc's will work here as well as for the exploratory reads. We call these procedures WU-proc's for an obvious reason.

A WU-proc can read external state (without idempotence) and freely update its local variables. When it returns, it returns to a part of the LLcom method that immediately issues the return for the method call. The only thing the LLcom can do here is to include the results from the WU-proc in what it returns to its caller (the client P-com). It cannot alter the state of the LLcom.

Having the client Pcom do the logging is reminiscent of middle tier components sending cookies to the client. The client then returns to cookie to the web service so that the web service can restore the appropriate middle tier state for the client. However, here there is much more flexibility. The client need not return the reply that it was sent (the "cookie"). Rather, it can do further local execution, perhaps interacting with an end user, before deciding what to do. For example, it can let the end user see the results of the middle tier comparison shopping, have the end user make a choice that determines the web site the middle tier will visit to consummate a purchase, and convey this choice back to the LLcom for it to follow up and complete the transaction.

4.2 Example Application

Figure 5 shows how client logging, in this case with a read-only method, enables us to choose the rental company at which to make a reservation. Logging of the rate information at the client comes before we choose "Hertz" or "Avis". Having logged, we can now select the car rental web site at which to place our reservation. Note that the client log needs to be forced before the "reserve" call is made in order for deterministic replay to be guaranteed.

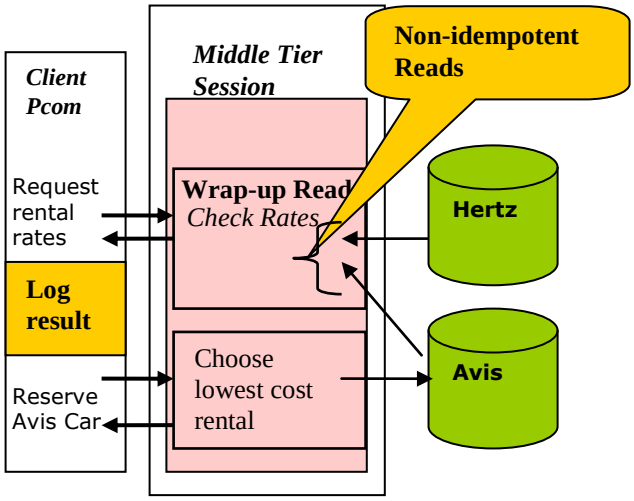

Fig. 5. A request is made to a car rental company, based on choosing the cheapest rates among those we have read. Logging for this choice is done at the client.

4.3 A Decision Service

Depending upon the specifics of the application, it may be more convenient to have the extra logging needed for the persistent decision on which backend server to visit next not be logged at the client, but rather to log it elsewhere. We introduce what we call a decision service for this purpose.

A decision service requires no new system infrastructure beyond what we have already described. Rather, it is a particular service, perhaps implemented by the middle tier provider, which accepts GIR requests. A request to it will be embedded within an E-proc, just like the backend services that we dealt with in section 3. The decision service may be located as part of the middle tier. Its function is very simple. It will return in its reply message whatever is sent to it in the GIR request. Hence this is a trivial service to implement.

The responsibility of the decision service is to make the GIR requests that it receives stable. Whenever it receives a GIR request with a given request id, it will return the argument given to it in the first successful execution of the GIR request with that same request id. The expected implementation of the decision service is for it to simply make this argument stable and return it as its reply on the first call with its request id and on any subsequent calls with the same request id. For flexibility, the argument accompanying the request id might be a variable length character string, essentially permitting essentially any information to be stored.

Obviously, one can make almost anything stable in this manner. However, the idea is to record only enough information so that an impending decision as to which backend server(s) to visit next can be replayed correctly. Thus, we would expect that typically very little information would need to be recorded at this service. Indeed, we would expect that many middle tier LLcom's would not need to invoke such a service at all. But it is an option if there is no other convenient alternative.

5. Systems Issues

We have discussed in [11] (i) how logless components, like ordinary Pcom's, can maintain volatile information that assists other components in their recovery; and (ii) how garbage collection of the information required in supporting idempotence might be performed or, alternatively, avoided as a problem. In this section we describe how other software components of a multi-tier system can be realized, and strategies that a client component might use to provide the level of availability and scalability desired.

5.1 The Client Pcom

We require that clients support persistent components (Pcom's) in order for logless middle tier components be enabled. So how is this functionality provided? We could require that this client software be installed on every machine that intends to play the role of client for our enterprise applications. However, that is unnecessary.

The EOS system [4] demonstrates a technique for "provisioning" a web browser with Pcom client functionality, while not requiring any modification to the web browser, in this case Internet Explorer (IE) [15]. The idea is to download from the middle tier, instructions to the web browser that permits it to act as a Pcom in our multi-level enterprise application. We single out IE as our browser because it supports the ability to make information stable by means of dynamic HTML (DHTML) instructions. In EOS, the logging needed by the client Pcom was implemented via writes to IE's XML store.

The DHTML is downloaded from the web site that is initially contacted as the service point for the application. The DHTML returned, in our deployment, would include the functional create call that instantiates the logless middle tier session oriented component. This DHTML also specifies client logging of user input and the logging of results returned from the middle tier.

Finally, the DHTML would also specify how to recover a crashed middle tier LLcom. If the middle tier session component fails (or simply times out), our client replays the functional create call and the subsequent method calls until it reaches the state at which the time-out occurred. At that point, execution continues as if the re-created component is the original. Various failover options during recovery are described in sections 5.3 and 5.4 below.

Our main point is that the enterprise application, in the middle tier, can exercise control over the client system by providing the client functionality that it desires or requires.

5.2 The Backend Server

In order for applications to provide persistence in the presence of system failures, idempotence is required for their interactions. Logging enables replaying the interactions to be avoided most of the time. But it does not cover the case where the failure occurs during the interaction. Here it is uncertain, when the application once again reaches the point of interaction, whether the effect intended on the other component (in our case, the backend server) happened or not. Idempotence permits the recovering application to safely resubmit its request. Thus idempotence is required for the fault tolerance techniques of which we are aware. In TP systems, it is often a transactional queue that that has captured the request and/or reply enables idempotence.

In many, if not most, services that provide idempotence, detection of duplicate requests relies upon a subset of arguments that act as a unique identifier for the request message. The remaining arguments of the message are usually ignored, except in the actual execution of the functionality of the request. When the request is completed, a reply message is generated, again identified and tied to the incoming request by means of the identifier arguments only. This style of implementation for idempotence already supports our generalized idempotent requests. The difference here is that we require the backend service to provide idempotence in exactly this way, i.e., we require that the service ignore the remaining arguments when doing duplicate checking.

The main point here is that there is frequently nothing new that web services need do to become GIR request servers. They have already done the hard part. Thus, the web service interaction contract that we proposed in [12] can be readily transformed into a generalized form, usually without having to change the web service implementation.[2] This contract is a unilateral contract by the web service, requiring nothing from the application. However, in its generalized form, it now supports exploratory reads by logless persistent session-oriented components.

5.3 Redeploying Middle Tier Components

Logless component technology is a foundation for systems that can provide high availability and scalability. The fact that there is no log in the middle tier for an LLcom means that we can re-instantiate the component by replaying its functional initiation call, directing this call to any convenient place in the middle tier. Should a request to an LLcom time out, i.e. the LLcom takes too long to respond to a client request, the client can choose to treat the LLcom and the server upon which it runs as "down". At this point, the client will want to redeploy the component on another server. Note, of course, that the original server might not have crashed, but simply be temporarily slow in responding.

One great advantage of LLcom's is that whether a prior instance has crashed or not, one can initiate a second instance of it safely. This avoids a difficult problem in many other settings, i.e. ensuring that only a single instance is active at a time. The client simply recovers the LLcom at a second server site by replaying its calls from its local log. The new instance repeats the execution of the first instance until it has advanced past the spot where the original instance either crashed or was seriously slowed. From that point on, the secondary instance truly becomes the primary.

So what happens if the original LLcom is, in fact, not dead, and it resumes execution? In this case, it will eventually return a result to the client Pcom. The client Pcom should always be in a position to terminate one or the other of the multiple LLcom instances. An LLcom might support a self-destruct method call that puts it out of its misery. In this case, the client Pcom might invoke that method. Alternatively, the middle tier infrastructure might simply time-out an idle LLcom at some point, which requires nothing from the client.

This failover approach is much simpler than is typically required for stateful system elements. It avoids shipping state information; and it avoids any requirement to ensure that a primary is down before a secondary can take over.

5.4 Replication for High Availability

To a very large extent, the persistent component at the client can choose the level of robustness for the middle-tier application. Because there is no requirement to "kill" a primary before a secondary takes over, it is possible to use LLcom's in a number of replication scenarios. Clients can issue multiple initiation calls, thus replicating a mid-tier LLcom. By controlling how it subsequently calls the replicas, it can control the progress of each replica separately. A client may pursue a number of strategies.

- It may have no standby for its mid-tier session, but rather create and failover to one should a failure occur.
- It might create a "warm standby" that has been created at another site and that has not received any calls following the initiation call. The client would be expected to replay these calls. But recovery time would be shortened.
- It might create a standby that it keeps current by continuously feeding it the same calls as sent to what it might consider the primary. Indeed, it can let the two mid-tier logless components race to be the first to execute. If one of them fails, the other can simply continue seamlessly in its execution.

Replicas do not directly communicate with each other, and there is no explicit passing of state between them. Rather, they run independently, except for their interactions with the client Pcom and their visits to the same collection of back end servers—where the first one to execute a service call determines how subsequent execution proceeds for all replicas. No replica needs to know about the existence of any other replica, and indeed, the middle tier need not know anything about replication. It simply hosts the logless session components.

[2] Application programmers would, of course need to verify that their web service implementations satisfied this generalized form of idempotence and hence the generalized form of web service interaction contract.

6 Discussion

It has long been recognized that robustness is an important attribute for enterprise applications. Software developers have known that database transactions are not enough, by themselves, to provide the level of guarantees that these applications require. Techniques for providing persistence in the presence of failures have been around for a long time, as have efforts to improve their performance. Here we briefly discuss some of the prior approaches.

6.1 Transaction Processing

Our goal of exactly once execution is shared by the transaction processing community. This community has a 35 year history, beginning with IBM's CICS system [7,11]. CICS was the first instance of what became known more generically as TP monitors. The terminology has shifted a bit, and we now frequently speak and write about application servers. These servers come in many flavors within the common middleware environments, e.g. Sun's EJB [17], and Microsoft's COM+ [15], the latest being web services [18].

Common to application servers and TP monitors is the notion of a stateless application. Read-only components are more or less naturally stateless. But the application server approach has been to make the updating components stateless as well, meaning that they take input state from transactional servers, execute, and put their output state back into transactional servers all within a single transaction. Stateless applications can be freely moved and redeployed on other middle tier servers since it is a transactional server that is the persistent home for application state.

The above mode of operation imposes (i) a strict programming model on the application, (ii) a performance burden to preserve state, and (iii) a frequent need to use two phase commit for distributed transactions.

6.2 Distributed Systems

There is a substantial community in distributed systems that have been interested in making distributed applications fault tolerant. The early work of Borg [8] involved replicating the components of an application. This usually involves an atomic messaging protocol to faithfully preserve messages and their arrival orders at each replica, combined with periodic sending of checkpoints. This was its way of implementing what were subsequently crystallized as recovery guarantees [5]. This work enabled exactly once execution of applications. However, the overhead of providing this guarantee had a significant impact on normal execution of the application. Much of the distributed systems work since has pursued a less ambitious goal than exactly once execution. Rather, its intent was to avoid having to completely re-execute a long running application should a failure occur. This work explored optimistic logging and distributed checkpoints, both of which permitted some loss of work, but operated with much lower overhead during normal execution [9].

6.3 Recent Work

The fault tolerant applications area continues to be an area of active interest. The Borg approach has been applied in the context of Corba in the Eternal system [14] in which atomic multicast is used to distribute messages to replicas. The replicas can be of the cold, warm, or hot variety. It is the responsibility of the messaging layer to prevent duplicate executions and provide exactly once execution. Another way of saying this is that the messaging layer is responsible for making the software components idempotent.

While the Borg and Eternal approaches used replication, the recovery guarantees work [5] tells us that it is the persistence guarantees that capture the requirement. Persistence can be provided, of course, using replication, but also by the database technique of logging and recovery. The logging approach was pursued in both Phoenix [1,2,3] and EOS [4]. There may be more potential for optimization within the logging approach, reducing normal execution cost at the possible expense of longer "time to repair". Running a "hot" server permits log forces to be shared by multiple components, a la database group commits. Further, by distinguishing component types, it is possible to reduce, sometimes dramatically, the number of log forces [1].

Finally, it is possible, by suitable restrictions, to remove the need for logging in parts of a system, while preserving recoverability. This idea evolved from the e-transaction work [10] where a simple session oriented middle tier component that executed a single request for a client could do so without logging its activity. That was expanded into the notion of logless component [13] in which multiple requests to idempotent backend servers could be made "persistent" without logging.

6.4 The Current Work

We have shown how to provide persistent session-oriented components in the middle tier that can read and respond to system state without requiring that such reads be idempotent. This is done without requiring logging in the middle tier. It does, however, require a stylized form of programming, frequently called a "programming model". However, the payoff is substantial.

What is provided is full persistence with exactly once execution semantics, in which system crashes can occur at arbitrary times, including when execution is active within

the component or when the component is awaiting a reply from a request. Because no log is required (i) performance of normal execution of middle tier application is excellent and (ii) components can be deployed and redeployed trivially to provide high availability and scalability. Aside from the stylistic requirements of exploratory and wrap-up reads, persistence is transparent in that the application programmer need not know about the logging being done elsewhere in the system to provide middle tier persistence.

References

1. R. Barga, S. Chen, and D. Lomet, "Improving Logging and Recovery Performance in Phoenix/App", *ICDE Conference*, Boston, 2004, pp. 486–497.
2. R. Barga and D. Lomet, "Phoenix Project: Fault Tolerant Applications", *SIGMOD Record* 31(2) 2002, pp. 94–100.
3. R. Barga, D. Lomet, S. Paparizos, H. Yu, and S. Chandrasekaran, "Persistent Applications via Automatic Recovery", *IDEAS Conference*, Hong Kong, 2003, pp. 258–267.
4. R. Barga, D. Lomet, G. Shegalov, and G. Weikum, "Recovery Guarantees for Internet Applications", *ACM Trans. on Internet Technology* 4(3), 2004, pp. 289–328.
5. R. Barga, D. Lomet, and G. Weikum, "Recovery Guarantees for Multi-tier Applications", *ICDE Conference*, San Jose, 2002, pp. 543–554.
6. P. Bernstein, M. Hsu, and B. Mann, "Implementing Recoverable Requests Using Queues", *SIGMOD Conference*, Atlantic City, 1990, pp. 112–122.
7. P. A. Bernstein and E. Newcomer, *Principles of Transaction Processing*, Morgan Kaufmann, 1996.
8. A. Borg, J. Baumbach, S. Glazer, A message system supporting fault tolerance, *Symposium on Operating Systems Principles*, 1983, pp.90-99.
9. E. N. Elnozahy, Lorenzo Alvisi, Yi-Min Wang, David B. Johnson, A survey of rollback-recovery protocols in message-passing systems. *ACM Comput. Surv*. 34(3): 375-408 (2002)
10. S. Frølund and R. Guerraoui, "A Pragmatic Implementation of e-Transactions", *IEEE Symposium. on Reliable Distributed Systems*, Nürnberg, 2000, pp. 186–195.
11. J. Gray and A. Reuter, *Transaction Processing: Concepts and Techniques*, Morgan Kaufmann, 1993.
12. D. Lomet, "Robust Web Services via Interaction Contracts", *TES Workshop*, Toronto, 2004, pp. 1–14.
13. D. Lomet, "Persistent Middle Tier Components without Logging", *IDEAS Conference*, Montreal, 2005, pp 37-46.
14. P. Narasimhan, L. Moser, P. M. Melliar-Smith, Lessons Learned in Building a Fault-Tolerant CORBA System. *DSN 2002*, pp 39-44.
15. MSDN Library: Persistence Overview. http://msdn.microsoft.com/workshop/author/persistence/overview.asp.
16. OMG: CORBA 2000. Fault Tolerant CORBA Spec V1.0. http://cgi.omg.org/cgi-bin/doc?ptc/00-04-04.
17. Sun 2001. Enterprise Java Beans Specification, Vers. 2.0, http://java.sun.com/products/ejb/docs.html
18. Web Services Interoperability Organization, http://www.ws-i.org/about/Default.aspx
19. G. Weikum and G. Vossen. *Transactional Information Systems*. Morgan Kaufmann, 2002.