Persistent Middle Tier Components without Logging

David Lomet
Microsoft Research
Redmond, WA
lomet@microsoft.com

Abstract

Enterprise applications need to be highly available and scalable. In the past, this has required "stateless" applications, which essentially require the application to manage its state explicitly by storing it in transactional resource managers. Despite "stateful" applications being more natural and hence easier to write and get correct, having the system manage this state automatically has been considered too difficult and too costly. The Phoenix/App system showed how to manage state in stateful applications transparently, by logging interactions between components, guaranteeing "exactly once" execution of the application. By introducing some minor restrictions on Phoenix/App components, no logging need be done for middle tier components, thus making it easy to provide both availability and scalability. Because there is no logging, the performance of failure free application executions is excellent.

1. Introduction

1.1. Robust Applications

Robust applications are those that, when deployed, are capable of providing enterprise systems with highly available and highly scalable service. Such applications must be able to survive system crashes, permit flexible re-deployment on other computers as the system changes and particularly, as it grows. The semantic requirement for robust applications is that, despite all this dynamic activity, including system crashes, it provides "exactly once" execution semantics. Such an application may start execution on one computer, that system crash, be redeployed on another, etc., and to the application client, it looks like a seamless execution in which the application executed exactly once without crashing or moving, etc.

Application developers naturally tend to let the business logic of the application dictate how the program is structured. This natural programming style has, in the past, interfered with enterprise system requirements for high availability and scalability. An application written in this "natural" way may retain control state necessary for correct and successful execution across transaction boundaries. Such an application is characterized as a "stateful" application. The problem with stateful applications is the risk of losing state when the system on which they execute crashes. This creates a "semantic mess" that may require human intervention to repair or restart the application, resulting in long service outages.

The classic transaction processing response to this problem [6, 7, 13] is to require that an application be stateless, where stateless means "no meaningful information is retained across transactions". But stateless applications force a rather unnatural "string of beads" style of programming 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. Note here that "state" is not so much avoided as it is made the responsibility of the application program to manage it in a transactional way. Potential performance and scalability problems related to the message and logging cost of two phase commit (2PC) may also be encountered.

To sum up the situation: the system builder is faced with a dilemma, having to choose between:

- fast and easy development, resulting in applications that are more likely to be correct applications, implemented in a natural stateful programming style, but which fail to provide availability and scalability; and

- the high availability and scalability of the stateless programming model which adds to development time and where correctness is more difficult to achieve because of the intrusion of additional concerns of explicit state management.

1.2. New Departures

There have been recent efforts to escape from the stateful/stateless dilemma. We mention two here, one of which is our own, and upon which this paper makes a significant further advance. Both efforts require some constraints in the programming model, but both nonetheless have stateful aspects to them.

1.2.1. e-Transactions. The e-transaction approach [10, 11, 12] focuses primarily on reducing state management while putting restrictions on how applications are structured and deployed. Indeed, their model has sometimes also been called "stateless", though we would argue that it is not. With e-transactions, a middle tier application program has a lifetime of one client service request. That is, its state lives between the client request and its reply to the client. Middle-tier application submits a single transaction to one backend server. The server must support "testable" state for the transaction, i.e. make it possible to determine the outcome of transactional requests (see, e.g. [13]. This requirement is not really new. Indeed, it is frequently the purpose of using queues with databases in the transaction processing [6]. The deployment of elements in the e-transaction model is illustrated in Figure 1.

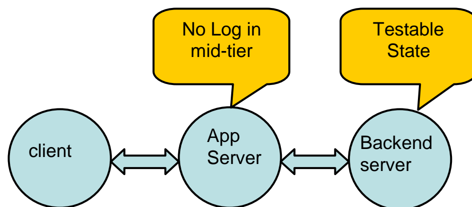

Figure 1: The e-transaction system model. A single client has a single request/reply interaction with a middle-tier app server that does no logging. The app server interacts with the backend server in a single transaction.

A client interacts with a component supported by a middle-tier application server. That component typically provides business logic, submits a transaction, and then returns to the client. Note that the mid-tier application has state outside of the backend transaction, potentially both before and after the transaction. Hence, the application can respond to transactional errors, etc. Further, and crucially, the middle-tier component can be made robust without middle-tier logging. The client re-issues the request and the re-created middle-tier component re-connects with the backend database to retrieve the transaction outcome, complete execution, and return its result to the client.

1.2.2. Phoenix/App. In Phoenix/App [2, 3, 4], components can be stateful. Components declared as persistent (Pcom's) survive system crashes. Components declared as transactional (Tcom's) must have testable transaction state, as in transaction processing and e-transactions. Other component types have other requirements. We focus on Pcom's here because persistence provides availability and scalability in Phoenix/App. Pcom's are not constrained as in the e-transaction model. They may serve multiple calls from multiple clients, send messages to other Pcom's or Tcom's etc., while providing exactly-once semantics. A system including Pcom's and Tcom's is shown in Figure 2.

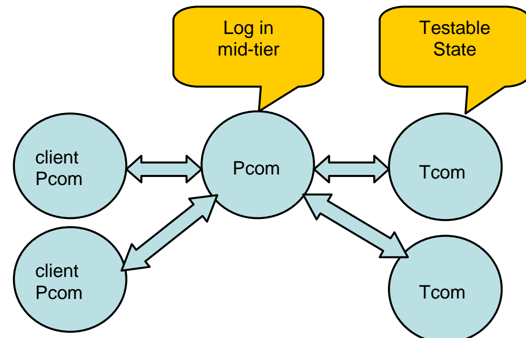

Figure 2: The Phoenix/App system model. Multiple clients have multiple message interactions with a mid-tier Pcom that does logging. The mid-tier Pcom interacts with multiple backend servers in multiple transactions.

In order for Phoenix/App to ensure that Pcom's persist across system crashes, it logs the interactions of each Pcom so that the Pcom can be deterministically replayed [8, 9], using the log to capture non-deterministic events and their potentially non-deterministic arrival order. A Pcom log also permits it to be recovered independently of other components. The logging is what permits it to satisfy the requirements of what are called "interaction contracts" [1, 5], which each Phoenix/App component must do. These contracts require components to guarantee that their state and messages will survive system crashes and provide exactly once executions. It is this logging

that permits a Pcom to engage in relatively unconstrained activity, with other Pcom's and Tcom's while maintaining persistence across crashes.

In [2], we introduced an optimization called the "multi-call optimization" to reduce the logging required for Pcom's. This depended on the observation that the components that were called were required to guarantee that they would support exactly-once execution by ensuring that duplicate calls were detected and that the result message was persistent and would be returned whenever duplicate requests were submitted. Since called components were responsible for making their result message persistent, the caller could depend up on that persistence to avoid its having to force log the result message. The log was finally forced when this component itself returned, thus ensuring that it could re-create its state subsequently.

1.3. Stateful Component Persistence without Logging

We introduce a **new component type that avoids logging** while providing a persistent, stateful model of components [5]. This component exploits the logging done already by other components, as identified in our multi-call optimization, and also exploited in e-transactions. We call these **logless components** and subsequently define them in detail and discuss how they would work in practice. Importantly, these logless components can be called multiple times, and interact with a number of backend systems involving a number of transactions, while retaining persistent state. Thus they support a natural stateful programming model in which the application programmer can focus on application logic, not issues system issues.

Logless components can be easily redeployed across an enterprise system since there is no log that needs to be shipped to enable this. In addition to their clear advantages for availability and scalability, logless components have an important performance advantage during normal execution. Because no logging is required of them, they save the execution path for storing information in the log buffer, and the delays associated with log forces.

We introduce logless components and their advantages in section 2. Section 3 describes the limitations that we need to place on persistent components to enable them to be replayed without logging. In section 4, we describe some of the engineering implications of logless components and how one might effectively deploy them. We provide an example application that illustrates the capabilities of logless components in section 5. We end with a brief discussion of further issues in section 6.

2. Logless Components

2.1. A New Phoenix Component Type

The multi-call optimization [2] showed that potentially many transactions can be executed via several calls to backend servers without the need for any intervening log force. This led us explore ways to provide components that do not require any logging, but which nonetheless are persistent and stateful. We found that by circumscribing what a Pcom could do, it was possible to define what we shall call "logless" components or LLcom's that are stateful, like Pcom's, but that do not require a log. Importantly, LLcom's will not require more logging or more log forces from the components with which they interact. For these components, the LLcom can be treated as if it were a Pcom, though these components may be required to keep persistent messages for a longer period. They may, at their option, choose to force their logs somewhat more often to provide for faster recovery.

We want to understand under what circumstances components can avoid doing logging. Clearly, we need to have some component(s) do some logging. Exactly where does that occur? For which among persistent, transactional, read-only, functional, subordinate, or external component is it possible to interact without an LLcom needing to log the interaction? How do we enforce "exactly once" execution? We will answer these questions further on. But first, we want to discuss the advantages of logless components.

2.2. Advantages of Logless Components

If a component does not need to log its interactions, then we do not need to move the log in order to move the component. This permits LLcom's to trivially be failed over to some other server, hence enhancing ***availability***. This also means that there is absolutely no work involved in reclaiming resources from these components. There is no stable state that needs to be maintained, as presumably, its entire state can be recovered without any log. Hence ***scalability*** is easily achieved, and components can be redeployed as servers and resources become available.

Availability and scalability are possible with Pcom's as now defined, but 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 interrupted state. But the cost of maintaining the log, forcing the log, and shipping the log are all avoided with logless components.

2.3. From Persistent to Logless Components

We need to specify precisely how persistent components can avoid logging. If this could be done in general, we would have over-engineered Phoenix/App. But that is not the case. Fully general Pcom's need to log interactions. However, with carefully crafted restrictions imposed on Pcom's, we can turn them into LLcom's. We describe this in the next two sections.

3. Making Replay Possible

3.1. Idempotent Interactions

The multi-call optimization [2] shows us that forced logging after each interaction with a Tcom or Pcom is, in fact, not required. With this optimization, a Pcom can fail after some number of interactions without logging, and yet be recovered. What is on the log is the Pcom's initiation or last checkpoint, followed (perhaps) by some sequence of interactions that were successfully logged. But one or more interactions subsequent to those on the log may have occurred and not have been logged. How do we deal with these?

We note that interactions between components rely upon both components having logged the interaction or made it durable in some other way. Thus, we require these other components to enforce the requirements that they would have had they been interacting with a Pcom. That is: (i) eliminate duplicates so as to enforce "exactly once" execution semantics, and (ii) to return on request the result message that they have promised to maintain about the interaction. These requirements led, in our Phoenix context, to the multi-call optimization for components that we access for the first time. However, when accessed a second time, the earlier "contractual" requirement for the persistence of result messages is terminated.

The guarantee provided by a "live" committed interaction contract (CIC) or transaction interaction contract (TIC) ensures that the interaction is idempotent, i.e. it can be re-played multiple times while only producing a state change exactly once, and always returning the same result message. It is idempotence that describes this "reliable" interaction replay.

A direct consequence of our observation that what we need is idempotence is that we can handle functional interactions (i.e. interactions with functional components [2]) trivially. Functional components have no effect on state and always provide the same response to the same input.

3.2. Non-Idempotent Interactions

Idempotence permits interactions to be replayed without logging. Conversely, any non-idempotent interaction precludes it. Logging enables us to replay non-idempotent interactions by using the logged results. This turns their non-determinism into deterministic replay. But without logging, this is not possible.

Thus, we cannot permit LLcom's to interact with or read external (and hence non-deterministic) system state or affect external state outside of Tcom interactions. While a read does not change state, there is no guarantee that replaying the interaction will produce the same result. Thus reads, which are possible with Pcom's, cannot be permitted for LLcom's. For interactions that change external state, the situation is worse. We can have multiple executions, hence violating exactly-once execution, as well as having different results returned. Hence, such interactions cannot be permitted. Even with Pcom's that do log interactions, such interactions cannot be guaranteed to be idempotent should a failure occur *during* the interaction, i.e. prior to the Pcom logging the interaction.

3.3. Deterministic Interaction Order

The contracts for each interaction guarantees idempotence for the interaction [1, 5]. But this is only one source of nondeterminism for which logging was exploited. Logging in Pcom's is also used to remove the nondeterminism resulting from the order of interactions.

Consider a long-lived object, implemented as a Pcom. It will normally become active via a call to one of its methods from some, perhaps unknown, other component. We usually know neither which method will be invoked nor the identity of the invoking component. Both these aspects are captured in Pcom's via logging.

With an LLcom, of course, we have no log to rely upon. Hence we need to insist that all interactions arise via a deterministic replay of idempotent interactions. An easy case to see is when an LLcom represents the execution of a single request to a single method. The method may call out to other components (using idempotent interactions), and if it is single threaded, the order of execution is completely determined by the code that it executes and the results of the idempotent interactions encountered during execution.

It is tempting to simply end the discussion here. A single procedure, whose execution is single threaded, and whose interactions are all request/reply, surely can be realized as an LLcom (though there is one more issue to deal with, which we discuss in the next section). But, in fact, we can permit more than this.

What is required is that the LLcom execution deterministically identify the next interaction as to the kind of interaction (send or receive) and which component the interaction is with. If it is a message send, then clearly this will be true as it is the component's deterministic execution that leads to the message send. It is the receive style interactions whose determinism needs to be scrutinized.

For a receive message that is part of a request/reply setting, it is clear that the reply (a message receive) is from the recipient of the request message, and the reply message is awaited at a deterministic point in the component execution. For other receives, we must be more guarded. Clearly, truly nondeterministic receives must be excluded. But some receives are not ruled out. For example, an LLcom might receive a series of requests from a given client component in what might be called a conversation. In such a case, we would know that the initial client would be the next sender of a message for which we are waiting. During replay, when execution reaches the state where the message is due, we could re-request the message from the client.

3.4. Functional Initiation

The entire history of an LLcom must be replayable without logging. That is what we described above. But another source of nondeterminism is the way that logical components are named and mapped to the underlying physical resources. That nondeterminism needs to be removed without our requiring that this information be logged, which is what is done in Phoenix/App [2, 3].

Thus, we require that an LLcom have what we call a "functional" initiation or creation. This means that from what is in the creation message, the entire information about the identity of the LLcom must be derivable. This requirement permits a re-execution of this creation message send 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 will respond to the re-instantiated LLcom exactly the same way, regardless of where the LLcom executes.

3.5. Pragmatics of Short Lifetimes

An LLcom has no log. Hence, it is meaningless to talk about checkpointing its state so as to shorten its recovery time. Whenever an LLcom crashes or is de-allocated to free up resources to enable scalability or other system management goals, it can be re-created only via complete replay of its entire execution history, starting from its initiation message. Obviously it is desirable to perform this replay quickly to achieve high availability and minimize system overhead. This argues for keeping the lifetime of LLcom's short.

It is important to understand what "short" means in this context. An LLcom relies upon other components for logging its interactions. So the replay time is governed by (1) LLcom execution time, and (2) the time required for other components to respond to its replayed interactions. The time involved in (2) will usually be much shorter than the original execution time, since other components will usually not need to re-execute requests during replay. Rather, they will normally simply look up the messages sent to them by the LLcom and generate replies based on information that they have retained in a table. So lifetime will usually be a function of LLcom execution path plus the number of interactions times the replay time for each interaction, not the original time required to execute code to reproduce the result of the original interaction.

Keeping LLcom lifetime short, while important, is a pragmatic consideration. Long lived LLcom's will work correctly, subject to the other considerations addressed here. It is their practical value that diminishes as their lifetime increases.

A system that implements LLcom's may want to be able to determine lifetime in some syntactic way prior to deploying the components in a live system. An easy way to do this is to preclude loops and perhaps to impose a limit on the number of calls that an LLcom can make or can receive. This permits us to know the execution path of the LLcom and the number of interactions in its lifetime. Perhaps less restrictive conditions could also be satisfactory.

One important point to stress is that at the end of an LLcom lifetime (indeed this is true for Pcom's as well), it goes "stateless". At that point, there is no state that needs to persist. For example, if an LLcom's lifetime is bracketed by a method call/return, then once its caller logs the return message, there is no longer any state that needs to be recovered. At that point, replay of the LLcom is no longer necessary. An LLcom can be deallocated at this point.

4. Detecting Failures and Recovering

4.1. The Role of the Initiator Component

The initiator component, i.e. the component making the initiating call, must also initiate recovery for the LLcom. Unlike Pcom's, where the infrastructure hosting the Pcom supports a log and a recovery manager that handles recovery for local components, with LLcom's there is no log. So even were the infrastructure to have a recovery manager, it could not recover the LLcom. To provide LLcom recovery, the initiation call has to be replayed.

Because the initiator Pcom must replay the initiation call for LLcom recovery to happen, it must also be able to detect an LLcom's failure. Detecting failure requires that the initiator expect a message from its initiated LLcom and fail to receive it after some timeout period. While this expected message can be a "ping", it is clearly more useful if it is a reply to a Pcom request.

Because of these constraints, LLcom system interactions and configurations are more restricted than for Pcom's. The initiator needs execution control to return to it in some way from every LLcom that it initiates. If can send multiple messages to an LLcom that it initiates, but it must always reach a state in which a message is expected from the LLcom. It is the failure of such a message to arrive that triggers the initiator to begin recovery via replay of the initiating message.

A component can initiate more than one LLcom. And an LLcom can initiate other LLcom's. But the initial LLcom in the system must be initiated by a Pcom. These requirements ensure that LLcom's are recoverable. The Pcom initiating the first LLcom is independently recoverable via logging. Other LLcom's are recoverable either directly by the Pcom or by an LLcom that is recoverable by the Pcom. The originating Pcom makes this "recursion" well founded. Thus a Pcom can initiate a "tree of LLcom's" and successfully recover them.

LLcom's must be terminated as well. One way to do this is to impose responsibility on the initiating component. It must always await a message from the LLcom whenever the LLcom is active, so that it can provide recovery. This also means that it can terminate the LLcom via a final message. However, this may not be essential. LLcom's that are inactive for a sufficiently long time might simply be deallocated as they can be re-instantiated via replay if they are needed again.

Logging need not be forced when a Pcom interacts with an LLcom that it initiates. The initiating call is an example of the multi-call optimization. Further, there is no need to force log subsequent interactions. What is on the stable log in this case is useful solely to optimize Pcom recovery. In all cases, the LLcom is guaranteed to be restartable from its initiating call, and subsequent replay of interactions with it is entirely deterministic. So Pcom replay, as long as it includes replay of an LLcom's initiating call, need not even log LLcom interactions, except to optimize its own replay.

Should the LLcom be alive during Pcom replay and only have retained information about its last call from the Pcom, it needs to self-destruct so that its complete replay is possible. When the LLcom no longer exists at a middle tier site, the site can respond to the initiating Pcom that the message failed to be delivered because the target does not exist. At this point, the initiating Pcom can recover the LLcom by replaying messages to it starting at its initiating call.

A Pcom must be prepared to replay an LLcom from its initiating call forward. This makes Pcom checkpointing during the LLcom lifetime more complicated than were it interacting with another Pcom. Any checkpoint must be sure to include the LLcom initiating call together with the Pcom messages to the LLcom before the checkpoint, so that the LLcom can be recovered.

4.2. Minimizing Dependencies

LLcom's, when failures occur, may have longer outages than Pcom's. Pcom's can be recovered based on local logging. In Phoenix/App, a local monitor, much like a monitor used for database recovery, is deployed which looks for Pcom's that need recovery. When one is detected, the monitor brings up a recovery process, points it at the Pcom's log, and initiates its execution. In contrast, LLcom's require the replay of the initiating call, which will usually not be local. Further, this call will only be replayed after the initiator has timed out waiting for a message from the LLcom. This suggests that the LLcom may be less responsive when failures occur.

Consider a committed interaction contract (CIC) [1, 3], used between a pair of Pcom's when they interact. The sender of a message is responsible for guaranteeing message and state persistence. In particular, the sender may be required to re-transmit the message upon being asked by the receiver, as part of receiver recovery. Consider now an LLcom as the sender with Pcom as receiver. Were both to fail, e.g., as the result of a power failure or some site wide failure, and then initiate recovery, the receiver would not be able to recover until the LLcom is recovered. And this might result in a much larger delay. The initiator of the

LLcom would first need to re-send the initiating message, and then the LLcom would need to replay from its start up to the point where the message was sent.

The delays described above can be avoided if, instead of using a CIC for the interaction, an ICIC (immediately committed interaction) were used [5]. In that case, any message needed for the recovery of a component interacting with an LLcom is known to be logged at that component, and would not need to be provided by a sending LLcom. Thus, instead of the two step release of the responsibility for the message in a CIC, the responsibility of the sender to resend the message is released all at once in an ICIC.

Using an ICIC, e.g. as part of a request/reply interaction in which the LLcom is the requestor, need not lead to additional forced logging, though it may add more data to the log. The sender, whether under a CIC or ICIC, periodically re-sends a message until it receives an ACK that the message has arrived. Should the message not have arrived (or not been logged), the message is not part of the receiver's recovered state-hence is not needed during recovery. If it has arrived, that is indicated by its being on the log, and so the sender need not provide the message during receiver recovery. Thus, the receiver can recover independently of sender recovery.

Request/reply interactions avoid the need for a message sender (in this case, the replier with its reply message) from needing to continually resend the message. The replier knows that the sender is expecting a reply. It need only send the reply once, knowing that the requestor will ask again for the reply (via resending the request). Thus, having an LLcom as an initiator of a request/reply interaction does not result in needless retransmissions of the reply message in the case where the LLcom has failed. This is important as the LLcom outage may be longer than would be the case for a locally recovered Pcom.

4.3. LLcom's to Aid Recovery

An LLcom not logging does not mean that it can't be helpful in providing recovery for other components. For example, the volatile tables that are constructed for Pcom's (and which in that case are "backed up" via logging so as to be persistent) [3] might also be maintained for LLcom's. Thus, a message sending LLcom might be expected to retain the message in a message table so that it can be re-sent should another component need it for recovery. The persistence of the message (and of the volatile table) would be guaranteed by the recovery of the LLcom. In the case where the LLcom does not fail, but the receiver of the message does, the LLcom is in a position to be as helpful as any Pcom in hastening receiver recovery.

While LLcom's and Pcom's have strong similarities, we have already seen that LLcom replay limits LLcom interactions to idempotent ones. The difference in how they are recovered requires us to impose restrictions on how LLcom's are deployed in a multi-tier application. We next present an architectural proposal for how one might proceed in realizing a system that exploits LLcom's.

4.4. A Simple Architectural Proposal

One easy way to structure a system involving LLcom's is to require LLcom interactions to be of the request/reply form. This ensures that requestors await replies. Failure of an LLcom reply to arrive in a timely fashion triggers LLcom recovery. LLcom requestor failure does not trigger repeated retransmissions of the reply. The reply is simply held until the recovered LLcom re-requests it.

Restricting LLcom's to request/reply interactions still leaves these components with great flexibility. An LLcom has state that might live across multiple interactions that it might make with other (backend) components, where it is the requestor for the interaction. Further, it is capable of living across multiple transactions, e.g. that it is involved with via request/reply interactions with Tcom's. Additionally, such an LLcom can engage in a request/reply "conversation" with its initiating Pcom or other LLcom. While LLcom's have restrictions on their deployment when compared with Pcom's, they still retain a large degree of flexibility.

Many multi-tier systems should be amenable to support by LLcom's. LLcom's are not restricted to the three tier systems or the single transaction execution of the e-transaction model. Rather, there can be multiple layers of LLcom's between front end client, realized as a Pcom, and backend databases, realized as Tcom's.

One such deployment is illustrated in Figure 3. The Pcom initiator of the first of potentially many layers of LLcom will log the LLcom reply-- which effectively summarizes the impact of all the LLcom's calls and activity into the one message. This message, when logged, permits this Pcom to be replayed from the log. Should the LLcom call not return, it can be replayed in an idempotent way. Hence recovery for the initiator Pcom is provided. This is all done without the LLcom's doing any logging whatsoever.

LLcom can also play the role of a Pcom in interaction with a web service [14]. No middle tier logging is needed. A Pcom client logs interactions which it may make to one or more LLcom's in the

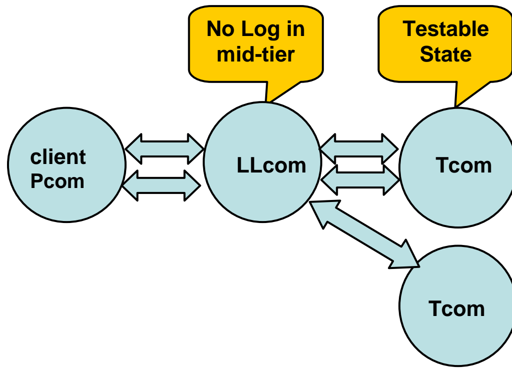

Figure 3: The Phoenix/App system model with logless component in middle tier. A client can have multiple message interactions with a mid-tier LLcom that does not log. The LLcom interacts with multiple Tcom's in multiple transactions.

middle tier. These LLcom's would then interact with web services using the web service interaction contract (WSIC). The web service provides testable state. The logging at client Pcom and at the web services is sufficient for exactly once execution.

5. Example

To illustrate an application that can be supported by logless components, consider an online book buyer application. To be more specific, consider that the application is for ordering textbooks from wholesalers. The book buyer has a preference to use SupplierA as the supplier for textbooks, either because of a better price or because SupplierA has some special deal with the book buyer. So it is SupplierA that is first asked to satisfy an order. Only if SupplierA cannot provide books for the order is SupplierB used.

Consider now Figure 4. Our client wants to order 50 books. So the book buyer application requests the 50 textbooks from SupplierA. SupplierA only has 35 of the books in stock and returns to the book buyer confirming an order for 35. Our book buyer proceeds to order the remaining 15 books from SupplierB.

The book buyer application is truly a stateful application. The fact that SupplierA supplied 35 books is state that needs to be carried over for a correct interaction with SupplierB. Previous approaches require that this application either log its interactions, or be implemented in two steps as part of a stateless application.

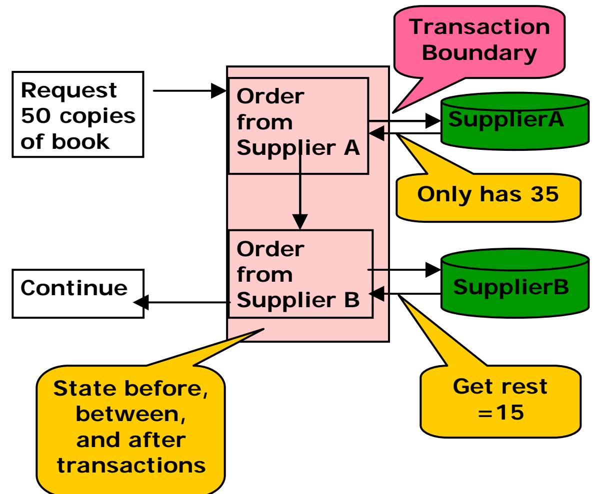

Figure 4: An online book buyer application that can be realized as a logless component. This component needs to remember that SupplierA provided 35 books so that it can correctly order the remaining 15 books from SupplierB.

However, our application (in the pink box) can be realized as a logless component... The client, requesting the 50 books, needs to make that request persistent, either by logging it explicitly, or by logging the user input so that replay can re-generate the request. Each supplier needs to present an idempotent interface where repeating the exact order will yield the same result. Thus all nondeterminism has been captured at either the client (who captures the request) or the back end suppliers (who capture the order that was submitted and provide idempotence). Hence the middle tier application does not require any logging in order to ensure persistence across system failures.

If the system crashes (or merely times out because of slow servers or network), the client can re-submit his request anywhere in the system, without concern about log shipping (to send the state) or worries about double ordering at the suppliers. Any duplicate request to a back end supplier is guaranteed, because of idempotence, to not trigger multiple executions, and further to return the exact same result as the original request.

It is even possible for two book buyer components to be active simultaneously, so long as the client can distinguish them and their reply message. There is never more than a single order placed at each supplier. And the orders are always the same number of books.

6. Discussion

6.1. Idempotent Read Interactions

Normally, reads are not repeatable in an idempotent way. That is, if we request information from some data source at one time, and then ask again later, there is no guarantee that the result returned will be the same. Thus, were reads permitted for LLcom's, during recovery we would need to re-execute the interaction. Since the result is not guaranteed to be the same, deterministic replay to the same state would not be possible.

However, were we to perform a versioned read of data based, e.g., on its timestamp, where the time requested is a deterministic function of the parameters of the initiating call to the LLcom, such a read would be idempotent. We would read data as of some time, and use the same time during redo recovery. So this restricted form of read is permissible for LLcom's.

6.2. Server Affinity

It is frequently useful for a client to return to the same server as the one that handled a prior request. Holding an LLcom lifetime to the period between initial request and the reply to that request, as done in the e-transaction model, makes LLcom replay much like the replay of Pcom's. That is, a Pcom interaction (request/reply) is redone only if it is the last interaction of the Pcom. However, if this requirement were relaxed to permit multiple request/reply interactions with an LLcom by its initiator, replay remains possible.

Permitting multiple request/reply interactions from initiator Pcom to LLcom directly supports server affinity. So long as there is no failover or other redeployment, the LLcom remains instantiated at the host system where it originated. The initiating Pcom can re-access the LLcom with additional requests, and the state continues to exist and to play a role in keeping system execution efficient. Recovery, should one of the later Pcom requests to the LLcom not return in a timely way, now requires rebuilding the LLcom, starting with the initiating request.

6.3. Garbage Collection

Middle-tier garbage collection of LLcom state does not pose a problem. Indeed, we do not need to know when an LLcom goes stateless. Clients can walk away from their "sessions" and none of this represents a large problem. Why? Because we can always replay LLcom's from their initiating call to re-create them. So a very simple time-out based reclamation method will work fine.

Garbage collection of testable state "items" maintained by backend servers remains a potential problem. However, it is a problem for Pcom and e-transaction systems as well. Contract release still needs to be carefully considered. Essentially, for LLcom's, backend contracts cannot be released until the initiator Pcom logs the final (terminating) LLcom reply. The e-transaction papers [9, 10, 11] suggest doing the garbage collection based on time-outs. The web services paper [14] suggests that, at least in some cases, never doing garbage collection might also be ok.

6.4. Architectural Advantages

Using request/reply, as with keeping LLcom lifetimes short, is a pragmatic way of dealing with and providing robust components in a distributed system. That is, these two characteristics are not essential. One could imagine a system that provided more flexibility, at the cost of more complexity, and perhaps longer delays when failures occur. So request/reply is a pragmatic choice, and surely a good starting point for functionality.

Even in a pragmatically constrained deployment, LLcom's go beyond the multi-call optimization by entirely eliminating logging. LLcom deployment goes beyond e-transactions in permitting client conversations and multiple backend transactions. Stateful LLcom's can retain state between calls in a conversation from a client, and between calls to servers, whether transactional or not. Indeed, when interacting with a web service [14], an LLcom might not even know whether no, one, or several transactions were executed for a single request/reply interaction.

LLcom's simultaneously achieve the goals of offering a natural stateful programming model, high normal case performance, and high availability and scalability, making them an excellent vehicle for structuring and deploying enterprise applications.

7. References

[1] 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.

[2] R. Barga, S. Chen, and D. Lomet, "Improving Logging and Recovery Performance in Phoenix/App", *ICDE Conference,* Boston, 2004, pp. 486–497.

[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 and D. Lomet, "Phoenix Project: Fault Tolerant Applications", *SIGMOD Record* 31(2) 2002, pp. 94–100.

[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", *ACM 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, W. Blau, W. Graetsch, F. Herrmann, and W. Oberle, "Fault Tolerance under UNIX", *ACM TOCS*, 7(1), 1989, pp. 1-24.

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

[10] S. Frølund and R. Guerraoui, "A Pragmatic Implementation of e-Transactions", *IEEE Symposium. on Reliable Distributed Systems*, Nürnberg, 2000, pp. 186–195.

[11] S. Frølund and R. Guerraoui, "e-Transactions: End-to-end Reliability for Three-tier Architectures", *IEEE Trans. on Software Eng.* 28(4), 2002, pp. 378–395.

[12] S. Frølund and R. Guerraoui, "Implementing e-transactions with Asynchronous Replication", *IEEE Trans. on Parallel and Dist. Syst*ems, 12(2), 2001, pp. 133–146.

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

[14] D. Lomet, "Robust Web Services via Interaction Contracts", *TES Workshop,* Toronto, 2004, pp. 1–14.