Dependability, Abstraction, and Programming

David Lomet

Microsoft Research, Redmond, WA
lomet@microsoft.com

Abstract. In this paper, we look at what is required to produce programs that are dependable. Dependability requires more than just high availability. Rather, a program needs to be "right" as well, solving the problem for which it was designed. This requires a program development infrastructure that can, by means of appropriate abstractions, permit the programmer to focus on his problem, and not be distracted by "systems issues" that arise when high availability is required. We discuss the attributes of good abstractions. We then illustrate this in the programming of dependable systems. Our "abstraction" is a *transparently persistent stateful programming model* for use in the web enterprise setting where exactly-once execution is required. Work on this abstraction is reviewed. The new technical meat of the paper is in (1) describing how to reduce the performance cost of using the abstraction; (2) extending the flexibility of using this abstraction; (3) and showing how to exploit it to achieve dependability.

Keywords: dependability, abstraction, application persistence, availability, scalability, programming model, enterprise applications

1 Introduction

Systems need to be dependable. But what exactly does that mean? Brian Randell and co-authors [1] defines dependability as *the system property that integrates such attributes as availability, safety, security, survivability, maintainability.* This is more than just high availability (the fraction of the time that the system is operational), with which it is frequently confused. Rather, it encompasses systems that do what they are supposed to (correctness), can be modified to keep them up to date (maintenance), cover all the expected cases, handle exceptions as well as normal cases, protecting data appropriately, etc. It is no accident that dependable systems have been in short supply as achieving them is very difficult.

Tony Hoare argues [14] that *the price of reliability is utter simplicity- and this is a price that major software manufacturers find too high to afford.* Perhaps my view is colored by being employed by a software vendor, but I do not believe the difficulty stems from want of trying. Rather, it is extraordinarily difficult to make systems simple, without drastic reductions in functionality and the very aspects desired, such as availability. The key point is that unless it is easy, natural, and simple, programming for dependability may well compromise it.

This difficulty has led to a very unsatisfactory state. Jim Gray [12] characterized the situation for availability as: (1) *everyone has a serious problem*; (2) *the BEST people publish their stats*; (3) *the others HIDE their stats.* This is not unexpected. It represents a very basic human reluctance to own up to difficulties, especially when doing so makes companies look worse than their competitors.

This is not simply an abstract problem. It has business impact. Dave Patterson [21] has characterized this as *Service outages are frequent* with *65% of IT managers reporting that their websites were unavailable over a 6-month period.* Further, *outage costs are high with social effects like negative press and loss of customers who click over to a competitor.*

To summarize, dependability is essential and we do not currently achieve it consistently. We all too frequently deploy undependable systems, which result in real and on-going economic costs.

The rest of this paper focuses on abstraction as the key to realizing dependable programs. We first discuss some existing techniques employed to make systems scalable (increasing the hardware supporting the application permits the application to serve a corresponding increase in system load, e.g. number of users served) and highly available (no prolonged down times, high probability of being operational), important aspects of dependability. We argue that these techniques compromise other aspects of dependability, specifically correctness, maintainability, and simplicity. We next discuss the characteristics of good abstractions, and illustrate this with existing examples that have survived the test of time.

The technical meat of this paper is a "new" abstraction, *transparently persistent stateful programming,* which we argue will enable dependable applications to be written more easily. We elaborate on this abstraction, showing how it can be made to work and provide availability and scalability. We then show that unexpected capabilities can be added to this abstraction that increase deployment flexibility and performance, hence increasing its practical utility.

2 The Current Situation

The classical tool in our bag of dependability technology, in the context of enterprise programming where exactly-once execution is required, is transactions. All early enterprise systems used transactions, usually within the context of a TP monitor. So we start by taking a look at how transactions have been used and the implications and problems of using them.

2.1 Using Transactions

Database systems use transactions very successfully to enhance their dependability. TP monitors also use transactions to provide high availability for applications. However, transaction use with applications has some implications for the application and for the context in which the application operates.

System context: TP monitors [8, 13] were developed in the context of "closed systems", where an organization controlled the entire application from user terminal

to mainframe computer. This provided a high level of trust among the elements of the system. In particular, it permitted an application with transactional parts to participate in a transaction hosted by a database or transactional queue. Such transactions were distributed, requiring two phase commit, but because of the level of trust, this was acceptable.

Programming model: The key to using transactions for application dependability is to have meaningful application execution state exist ONLY within a transaction. When the transaction commits, the essential state is extracted and committed to a database or queue [7]. This approach is also used in workflows [24]. An application is broken into a series of steps where each step executes a transaction that reads input "state" from a transactional queue, executes business logic, perhaps invoking a backend relational database, and writes output "state" to another transactional queue, followed by the commit of this distributed transaction. These applications have been called "stateless" (now a term that is a bit overloaded) in that no required execution state exists in the application outside of a transactional step.

Web applications: The web setting, because of the trust issue and the uncertain reliability or latency of communication, frequently precludes the use of two phase commit. Thus, as shown in Figure 1, most web application code lives outside any transaction boundary. The key to making such systems available and scalable is again "stateless" applications, the idea being that since there is no essential program execution state that needs to be preserved in the executing application, it can fail without any consequence for availability (though clearly with some performance impact). The term stateless application continues to mean that application execution state outside of transactions is not essential for application availability, but the techniques used for managing what is the state of the application are frequently quite different, involving a mix of posting information to backend databases, including it in cookies that are returned to users, or replicating it elsewhere in the middle tier.

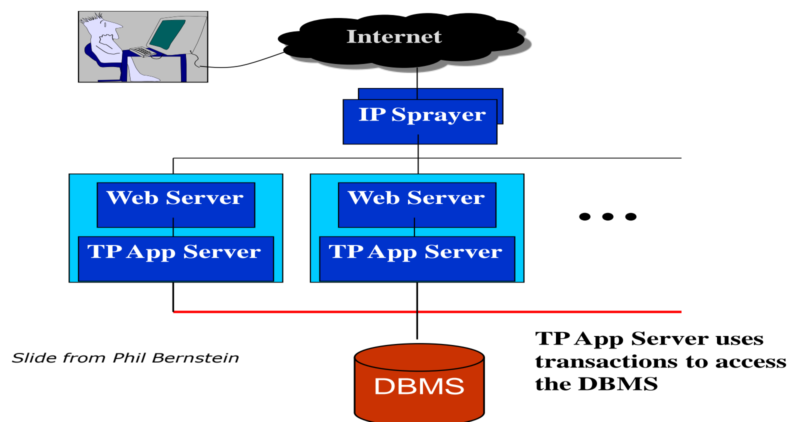

Figure 1: A Typical e-commerce system deployment.

Scalability: Stateless applications can enhance web application scalability as well. Because there is no execution state retained, an application can move to other application servers easily simply by re-directing a request (perhaps with the appropriate cookie) to another application server. This permits new application servers to be easily added to the running system, hence providing scalability.

The above discussion seems to suggest that any debate about the stateless application approach is pretty much over. In the next subsection, we discuss why we do not think that is the case.

2.2 Problems with Stateless Applications

Stateless applications, and the distributed transactions that are sometimes used to provide durability (and hence availability) are indeed a way to provide highly available applications. But the approach is not without difficulties.

Two phase commit: In a web setting, trust is frequently not present because the level of control necessary for it isn't present. Web host systems frequently have little control over the sea of users running web browsers or other potentially unreliable or even malicious software. Thus they are reluctant to be hostage to the commits of users, or even of other systems that may be involved in an overall business transaction. This has been called the "site autonomy" issue, and it is hard to envision distributed transactions being used with any frequency across organizations.

Latency is also a very big issue in web systems. It is important for performance for transactions to commit promptly. Large latencies introduce corresponding delays in the release of locks, which interferes with concurrency in these systems.

State management: Stateless frameworks force the application programmer to deal explicitly with state. Sometimes this is via cookie management, sometimes via storing information in a transactional queue, sometimes storing state in a database system. The application programmer needs to identify the state that is needed, and manage it explicitly. Then the programmer needs to organize his program into a sequence of steps ("string of beads") where the "state" before and after the execution of a bead is materialized and persisted in some way. Thus, the program needs to be organized to facilitate state management. Because the programmer needs to manage state within his program, the concerns of availability and scalability get mingled with the business logic, making programs more difficult to write and maintain, and sometimes with worse performance as well.

Error handling: With stateless programs, there is no meaningful program execution state outside of the transactions. The assumption is that any execution state that exists can be readily discarded without serious impact on the application. But this poses a problem when handling errors that occur within a transaction that might prevent a transaction from committing.

Since state outside of a transaction might disappear, it is very difficult to guarantee that code will be present that can deal with errors reported by a transaction. In classic transaction processing systems, after a certain number of repeated tries, an error message would eventually be posted to a queue that is serviced MANUALLY. It then becomes people logic, not program logic, which deals with the remaining errors.

3 Abstractions and Applications

Transactions are a great abstraction, but not in the context of reliable, scalable, and highly available applications, as described in section 2.2. We need to take a fresh look at what abstraction to use to make applications robust. To guide us in this process, we first consider the characteristics of good abstractions, so that we have some basis for making judgments about any new proposed abstraction.

3.1 Characteristics of a Good Abstraction

Good abstractions are hard to come by. This can readily be seen by simply observing the small number of abstractions that have made it into wide use—procedures, typed data and objects in programming languages, files, processes, threads in operating systems, and in databases, transactions and the relational model. There are surely other abstractions, but the list is not long. So why are good abstractions so few? It is because there are difficult and sometimes conflicting requirements that good abstractions must satisfy.

Clean and simple semantics: A good abstraction must be simple and easy to understand. But this is surely not sufficient. "Do what I mean" is simple but it is not, of course, implementable.

Good performance and robustness: If performance is bad, then either programmers will not use it at all, or will greatly limit their use of it. If the implementations are not robust as well as performant, it will be impossible to interest enterprise application programmers in using them. A system must perform well and provide reliable service as a basic property.

Good problem match: An abstraction must "match" in some sense the problem for which it is being used. Without this match, the abstraction is irrelevant. The relational model has a tremendous relevance for business data processing, but is surely less relevant for, e.g. managing audio files.

Delegation: When the characteristics above are present, the application programmer can not only exploit the abstraction for his application, but can frequently **delegate** to the system supporting the abstraction some "systems" aspect of his problem. This means that the programmer need no longer dedicate intellectual cycles to this part, permitting him to focus on the application business problem.

Historical examples of abstractions resulting in successful delegation include:

1. **Transactions:** a programmer using transactions delegates concurrency control and recovery to the system. This permits him to construct a program that is to execute within a transaction as if it were the only code executing.
2. **Relational model:** a programmer using the relational model delegates physical database design and query processing/optimization to the system, permitting him to work on a "data independent" conceptual view. This permits him to focus on business relevant issues, not storage/performance relevant issues.

3.2 A New Abstraction for Enterprise Applications

We want to propose a new abstraction to help an application programmer deal with the complex difficulties introduced by the stateless programming model, with its application program requirement to explicitly manage state. Our new abstraction is, in truth, an old abstraction, that has been used by application programmers for years. We call it the stateful programming model, and when applied to enterprise internet applications, it is the *transparently persistent stateful programming* model.

This seems hardly to be an abstraction at all. Application programmers, in simple contexts, naturally tend to write stateful applications. Such programs are easier to write and to understand, as the application programmer can focus on the requirements of the business, rather than the intricacies of managing state so that it will persist even if systems crash, messages get lost, etc. Because of this focus on business requirements, such applications are easier to read and maintain, and are more likely to be correct as the program is not cluttered with state management code. This aspect of the program has been delegated to the system infrastructure.

Figure 2 shows a stateful application in a web context. Note that there is important application state between the interactions with suppliers A and B. It is application state that captures the fact that there remain 15 copies of a book that still need to be ordered after supplier A has only been able to commit to providing 35 copies. This information is captured in the *execution state* of the program, without the programmer needing to take any measures to otherwise "manifest" this state. Thus, the application programmer has, in a sense, delegated state management to the system. And this permits the programmer to focus on what the business requires.

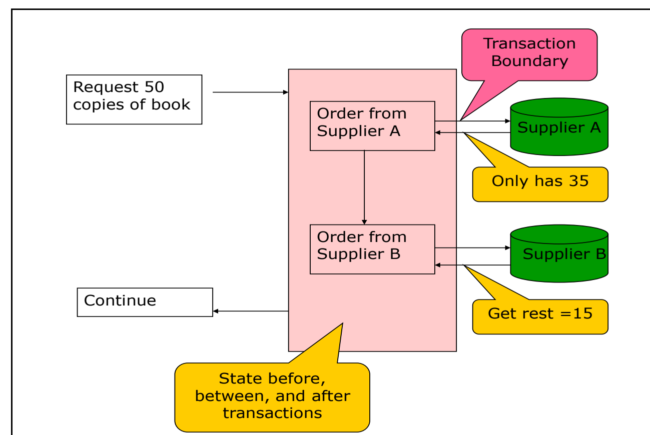

Figure 2: A stateful application in the web context.

3.3 It's been Tried and Failed

We, of course, want stateful applications to work in a web context, and support mission critical functions. That means such applications must provide "exactly once" execution semantics, i.e. as if, despite a wide range of possible failures, the program

has executed without failures and exactly once. We further want our application to be highly available and scalable. To accomplish these things transparently requires that our supporting infrastructure handle state management and state persistence.

The problem for stateful applications, classically, has been that they fail to provide the properties we enumerated above. That is, the application cannot be a made transparently scalable and available with exactly once execution semantics. Consider Figure 2 again. If the system crashes after the application's transaction with Supplier A, the fact that it has arranged for 35 books from Supplier A and needs another 15 books from Supplier B is difficult to reconstruct. The user does not even know yet what the order number used for Supplier A was. So not only is application state lost when a crash happens, which is bad enough, but the state of backend resource managers may now reflect some of the changes induced by committed transactions of the stateful application. Thus, we cannot simply restart the application and let it execute all over again. That strategy works for stateless applications, but not here.

4 Phoenix/App: Transparent Persistence

4.1 How Transparent Stateful Applications Might Work

As indicated in section 2, the stateless application model forces the application programmer to manage state explicitly. Usually the infrastructure supports this by wrapping application steps automatically with a transaction. But transactions are an expensive way to persist state and require explicit state management. So why not use database style recovery for applications: use redo logging with occasional checkpointing of state, all done transparently by the enterprise application support infrastructure. This is an old technology [9, 10], but our Phoenix project [2, 3, 4, 15] applied it to enterprise applications. It was previously applied in the context of scientific applications, not to support exactly once execution, but rather to limit the work lost should systems crash while the application is executing.

The way this works is that the redo log captures the non-deterministic events that a software component is exposed to. To recover the component, it is replayed, and the events on the log are fed to it whenever it reaches a point that in its initial execution experienced a non-deterministic event. Frequently such events are things like receiving a request for service (e.g. an rpc), or perhaps an application read of the state of some other part of the system (e.g. the system clock, etc.).

Unlike scientific applications, with enterprise applications, we need to provide exactly once execution. This requires "pessimistic logging". Logging is pessimistic when there is enough logging (and log forces) so that no non-deterministic event is lost from the log that is needed to provide the exactly once property. Optimistic logging permits later state to be lost, as with scientific applications. Pessimistic logging can require a log force whenever state is revealed to other parts of system (we refer to this as committing state). We focus on improving the performance of

pessimistic logging. Many log forces can be eliminated– and our Phoenix project has published some *slick* methods [1].

The result is that Phoenix manages an application's state by capturing its execution state on the redo log. Once the state is successfully captured, (1) it can be used to re-instantiate the application for availability should its system crash; and (2) it can be shipped to other systems for scalability and/or manageability, i.e. the log is shipped to another part of the system. The application programmer need merely focus on writing a correct program to solve his business problem and then to declaratively characterize the attributes he wants of his software components.

4.2 Transparent Robustness

We want to provide robust, dependable applications. If we are to do this, we need a "natural" programming model. Thus, the Phoenix goal has been to provide these enterprise attributes transparently using the stateful programming abstraction. If we can do this, then the application program need not include program logic for robustness. That has been delegated to Phoenix. So how do we do this?

Phoenix is implemented using the Microsoft .Net infrastructure [19]. ".Net Remoting" supports the development of component software for web applications. Among its facilities is the notion of a context such that messages (e.g. rpc's) that cross a boundary between two contexts are intercepted by .Net and made available to the Phoenix infrastructure code. This is illustrated in Figure 3. The application program "sees" the standard .Net interfaces for communication between components. When .Net gives Phoenix the intercepted message, Phoenix logs it in order to provide exactly once execution and application availability, scalability, and manageability. These requirements are captured in an interaction contract (CIC) which is shown in Figure 3 and described below.

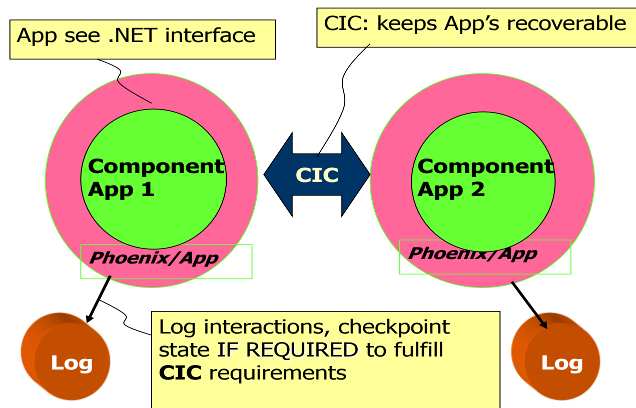

Figure 3: The Phoenix .NET infrastructure for transparent persistence.

4.3 Component Types and Interaction Contracts

Phoenix classifies application software components into three types [4] (subsequently, more types were introduced to support various kinds of optimizations). These are:

1. **Persistent** (PCOM): PCOM's represent a part of the application whose persistence, scalability, availability, etc. is being guaranteed by Phoenix. An application executing at an app server, executing business logic and interacting with a backend database system might be represented as a PCOM. A PCOM exploits the transparently persistent stateful programming abstraction provided by Phoenix. Phoenix logs interactions and checkpoints state as required to provide this persistence.
2. **Transactional** (TCOM): Some parts of a web application will execute transactions, e.g. SQL DB sessions. We call the components executing transactions TCOM's. When a PCOM interacts with a TCOM, it must be aware that transactions can either commit or abort. If committed, the post-state of the transaction must be durably remembered, while if aborted, the pre-state of the transaction must be restored.
3. **External** (XCOM): Applications are rarely self-contained. For example, an application frequently is executed in response to end user input or from an input from an autonomous site not controlled by Phoenix. We call these components XCOM's. We need do nothing for the persistence of XCOM's (indeed there is frequently nothing we can do). But we need to promptly capture via forced logging, the interactions that a PCOM has with an XCOM to minimize the window in which, e.g. a user might need to re-enter data as a result of a system failure.

Figure 4 illustrates the deployment of Phoenix components in support of a web application. We characterize the end user as an XCOM, and a SQL Server database (actually a session with it) as a TCOM, while the mid-tier components execute application logic in a context that ensures its exactly once execution.

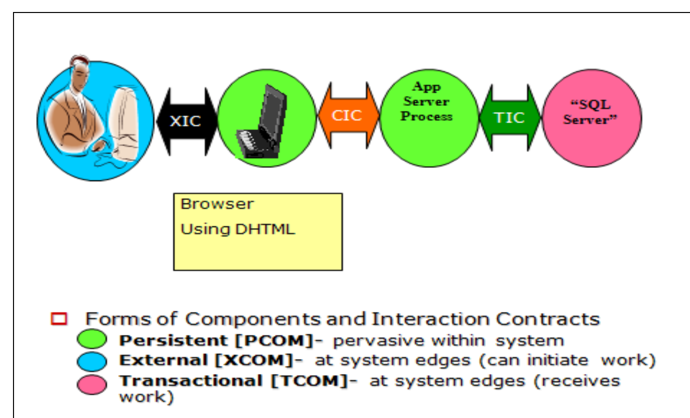

Figure 4: System schematic of deployed Phoenix components.

Depending upon what component type a PCOM interacts with, Phoenix needs to provide different functionality for exactly once execution. This is captured in what we call interaction contracts [5, 6, 22], of which there is one for each "flavor" of component. Where these interaction contracts are used is also shown also in Figure 4.

1. **Committed interaction contract** (**CIC**): A CIC interaction is between a pair of PCOM's. It must guarantee causality, i.e. that if the receiver PCOM remembers a message, then so must its sender PCOM, regardless of possible failures. We frequently enforce a stronger guarantee, i.e., that the sender guarantees that its state is persistent at the point when it sends a message.
2. **Transactional interaction contract** (**TIC**): A TIC interaction is between a PCOM and a TCOM. The PCOM may or may not remember that it has interacted with a TCOM should the transaction abort. But if the transaction at the TCOM has committed, the PCOM must remember the interactions it had with the TCOM. Transactions permit Phoenix to log less aggressively during the TIC, but require that state persistence be guaranteed at the point of commit.
3. **External interaction contract** (**XIC**): When a PCOM interacts with software not controlled by Phoenix we enforce an external interaction contract that does prompt log forces to reduce the window when failures may be unmasked to the duration of the interaction. Thus, usually failures are not exposed, e.g. to end users. Further, should software external to Phoenix provide messages to PCOM's, or receive messages from PCOM's, prompt logging minimizes the risk of a non-exactly-once execution.

5 Extending Phoenix

Our Phoenix/App prototype is quite robust. (Indeed, it can be tough to "kill" a Phoenix supported application since the system so promptly brings it back to life.) Nonetheless, there are some limitations that we would like to eliminate or at least ease. In particular, for scalability, we need to move the state of PCOMS from one app server (A) to another (B). This requires that we ship the log of PCOM events from A to B. We might do this regularly by mirrored logging, or periodically by copying the log from place to place. Alternatively, we might do our logging at a backend server where the log might be very widely available to app servers in the middle tier.

While transparent to the application, this incurs costs of the sort that stateless programs incur when managing state. That is, the application must post the state explicitly in a place, e.g. a transactional database, where it can be read by other app servers. It would be nice, for example, if we did not need to have a logging facility in the middle tier. And it would be even nicer if we did not have to always incur the cost of shipping application state (from the log) across the middle tier, especially since the crash frequency of web applications is low. So, *can we have persistent stateful applications without logging?* The answer is "No", but we can remove this requirement from parts of the system, especially parts of the middle tier, while still providing the customary Phoenix robustness attributes, and in particular permitting an application to be re-instantiated anywhere in the middle tier.

5.1 e-transactions

A PCOM in the middle tier has great flexibility. It can serve multiple client PCOM's, interact with multiple backend TCOMS, read XCOM generated external state, etc. Logging is required at a PCOM to capture the nondeterminism in the order of clients' requests, and to capture the result of reads outside the Phoenix system boundary. The key to "eliminating" logging is to restrict what such PCOMS can do.

There is precursor work called e-transactions [11] that removed the need for logging in the middle tier. A deployment for e-transactions is illustrated in Figure 5. What is it about this deployment that makes middle tier logging unnecessary?

1. All interactions are request/reply and have unique identifiers.
2. Only a single client is served. This means that all mid-tier nondeterminism via servicing client calls can be captured at the client, on the client's log.
3. The single backend server provides idempotence via testable state. Calls from the middle tier to this server can be repeated without compromising exactly once execution.

In this very simple scenario, the client tags its request with a request ID and logs it. The mid-tier, after doing whatever processing is required, forwards a request to the server using the same request ID. If there is a failure anywhere in the system, the request can be replayed based on the client log. The backend server's idempotence ensures that the request is executed at most once. The request is replayed until the client receives a reply to the request. This guarantees that the request is executed at least once. The combination provides exactly once execution.

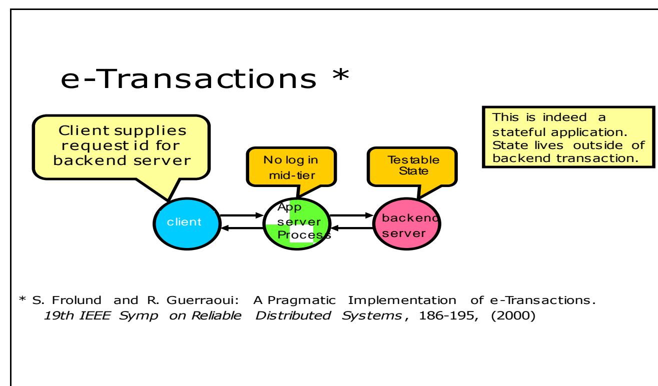

Figure 5: e-transactions in which the middle tier does no logging, yet a stateful middle tier application component is recoverable.

5.2 "Logless" Components

Within the Phoenix framework, we can define a "logless" component type (LLCOM) that can provide application state persistence while doing no logging [16], and can be exploited more flexibly than e-transactions. 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.

For LLCOM's to provide persistence without logging, we, as with e-transactions, restrict all interactions to be request/reply. 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. That nondeterminism must be removed without having to log it. Thus, an LLCOM must have what we call functional initiation, i.e., its creation message fully determines the identity of the LLCOM. This permits a resend of the creation message to produce a component that is logically indistinguishable from earlier incarnations. The initiating component (client) 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: A middle tier PCOM's 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. So, as with e-transactions, we restrict an LLCOM to serving only a single client PCOM (i.e., we make it *session-oriented*) and rely on this client PCOM to capture the sequence of method calls.[1] Since our PCOM client has its own log, it can recover itself. Once the client PCOM recovers, it also recovers the sequence of calls to any LLCOM with which it interacts, and can, via replay, recover the LLCOM. This single client limitation results in an LLCOM that can play the role, e.g., of a J2EE session bean [23]. Only now, the "session bean" has persistence across system crashes.

Determinism for Calls from an LLCOM: An LLCOM's execution must, based on its early execution, identify the next interaction type (send or receive) and with which component. A next message that is a send is identified and re-created via the prior deterministic component replay that leads to the message send. To make receive interactions deterministic requires more. However, a receive message that is a reply to a request is from the recipient of the request message, and the reply is awaited at a deterministic point in the execution. Like e-transactions, what LLCOM's need is idempotence from the backend servers for LLCOM requests. This is usually required in any case. Replay of the LLCOM will retrace the execution path to the first called backend server. 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.

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

Figure 6 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. However, unlike e-transactions, our LLCOM can interact with multiple backend servers.

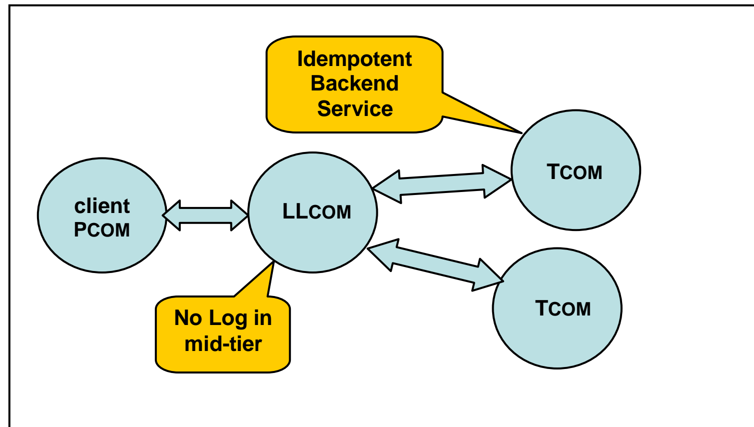

Figure 6: Logless component in a web deployment.

6 Middle Tier Reads

Requiring idempotence for servers executing updates has a long history in the context of TP monitors and queued transactions [8, 13, 24]. It is required to provide exactly-once execution in the presence of failures. We do not usually require idempotence for interactions that merely read data without changing it. However, an LLCOM that cannot do such reads is fairly restricted. So, can we permit non-idempotent reads? Superficially, it looks like the answer is "no", but we have identified two situations that we explore next where this is possible [17].

6.1 Exploratory Procedures

Generalized Idempotent Request (GIR): 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.

"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, e.g., by a TP monitor. That 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. However, the use of request identifiers to detect duplicate requests enables support for a **generalized idempotence** property. A server supporting generalized idempotent requests (**GIR**'s) permits a resend of a message with a given request identifier to have other arguments in the message that are different. This is exactly what we need for 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 "○" 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. A GIR request ensures that a state change at the backend occurs exactly once, with the first successful execution prevailing for both resulting backend state change and reply.

E-proc: We require every exploratory read to be within an exploratory procedure (**E-proc**). E-proc's always end their execution with a GIR request to the same server, using the same request identifier on all execution paths. Thus it is impossible to exit from the exploratory procedure in any way other than with the execution of a GIR request to the same server using the same request identifier. (More flexibility is possible, e.g. see the next section, but describing and enforcing the required constraints is more difficult.) Since a GIR request can have arguments other than its request ID that differ on replay, and it is guaranteed to return the result of its first successful execution, exploratory reads within the E-proc can determine on first successful E-proc execution, what the GIR request does.

Restricting exploratory reads to E-proc's confines their impact to the arguments for the GIR request. We permit exploratory reads to update local variables, as these have local scope only, but we do not permit non-local variables to be updated from within an E-proc. The GIR influences subsequent LLcom execution via the result that it returns and by how it sets its output parameters. Hence, whenever the E-proc is replayed, its impact on LLcom execution following its return will be dependent only on the first successful execution of its GIR request.

The net effect is that E-proc replay is "faithless", not faithful. But the faithlessness is confined to the internals of the E-proc. At its return, the E-proc provides idempotence. Thus, the LLcom replay is faithful overall, in terms of its effects on backend services and on the LLcom client. An LLcom with an E-Proc for a rental car application is shown in Figure 7. A customer wants a convertible when the deal is good, but otherwise a sedan. The exploratory read permits this choice, with execution

guaranteed to be idempotent for the rental request, even when, during replay, the customer's choice within the E-proc is different.

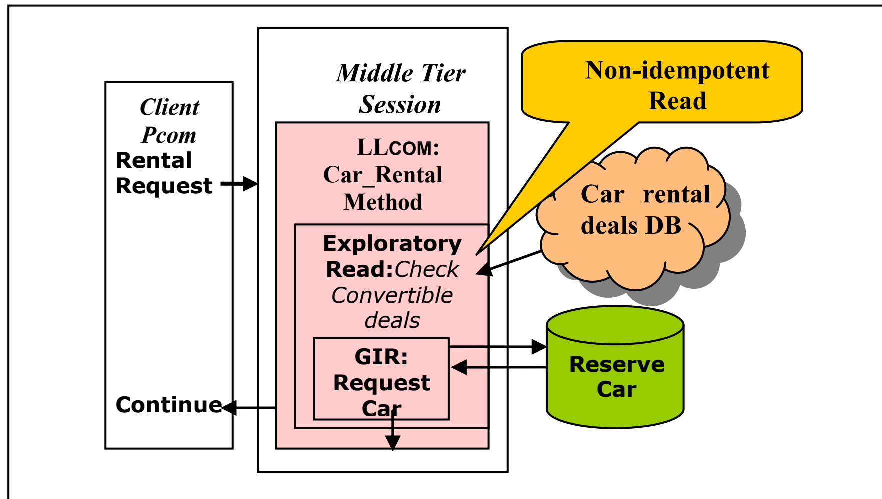

Figure 7: A GIR request is made to a car rental company, based on the result of an exploratory read about the deals offered by the company for a convertible.

Handling Aborts: Traditional stateless applications make it difficult to handle transaction aborts within an application because there is no *persistent* application code that can respond to the transaction failure. TP monitors usually automatically retry the failed transaction a number of times, which is fine for transient failures, e.g. system crashes, deadlocks, etc.. Hard failures, e.g., application bugs, cannot be dealt with in this manner. For those failures, the TP system usually enqueues an error message on a queue for manual intervention.

A PCOM stateful application can handle transaction failures because it has persistent execution state outside of transactions. It can inspect an error message (outside of a transaction), and provide code to deal with the failure. We want this capability for LLCOMS. However, aborts are not traditionally idempotent. Rather, a backend server "forgets" the aborted transaction, erasing its effects from its state.

Like a read, an abort outcome for a GIR leaves backend state unchanged. By embedding a GIR commit request within an E-proc, we are in a position to respond programmatically to responses reporting that the transaction has aborted. We can then include in the E-proc a loop that repeatedly retries the GIR request until that request commits. Further, we can use additional exploratory reads, and prior error messages, to change the arguments (but not the request identifier) of the GIR request in an effort to commit the transaction. Thus, an E-proc lets us respond programmatically and make multiple attempts to commit the request.

However, we cannot leave the GIR with an abort outcome. The problem here is now the non-idempotence of aborts. Should an LLCOM executing the E-proc abandon its effort to commit the transaction and then fail, upon replay of the E-proc, its GIR

request might commit the transaction. Subsequent state of the LLCOM will then diverge from its initial pre-failure execution.

One way out of this "conflicted" situation is to provide a parameter to the GIR that permits the calling LLCOM to request an idempotent abort. Normally, we expect that several executions of the GIR request to 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 could, for example, try other backend services. Idempotent aborts do require the cooperation of backend servers and are not now normally provided. This is unlike GIRs, where the usual implementation of idempotence already relies on request IDs.

6.2 Wrap-up Procedures

Exploiting the Client PCOM Log: We cannot use an E-proc to select which back end server to call. For example, we cannot use an E-proc to explore prices for rental cars and chose the web site of the rental company offering the best rates. To choose a backend server requires this choice to persist across system crashes. So the information from non-idempotent reads that determines our decision needs to be stable, encountered during LLCOM replay, and used to persistently make the same choice. An LLCOM has no log. Further, the "log" at the backend server, which we exploit in E-proc's, requires that we know which server to invoke, which is not very useful here.

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 since there is only a single client PCOM.

Consider a travel site LLCOM with a read-only method that checks rental car prices and returns that information to our PCOM client. The client PCOM, after interaction with the user, proceeds to make a subsequent call indicating his choice of car and intention to rent. Again, the read-only method is NOT idempotent. Upon replay, the car prices returned might be very different. Despite the lack of idempotence, we can permit read-only LLCOM methods. When the car rates are returned to the client PCOM, the client forces this information to its log prior to initiating the LLCOM call requesting the car. The LLCOM uses the client supplied information to select the rental web site. Replay is deterministic because the client PCOM has made information from the non-idempotent read(s) stable. The client can even avoid repeating the read-only calls to the LLCOM during its replay as it already has the answer it needs on its log. Thus, the first successful read execution "wins", and here we exploit client PCOM logging to guarantee this. Figure 8 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.

Unlike dealing with an idempotent middle tier component, where log forcing is not required (though it can be 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. Thus, we need to identify LLCOM methods that need forced logging.

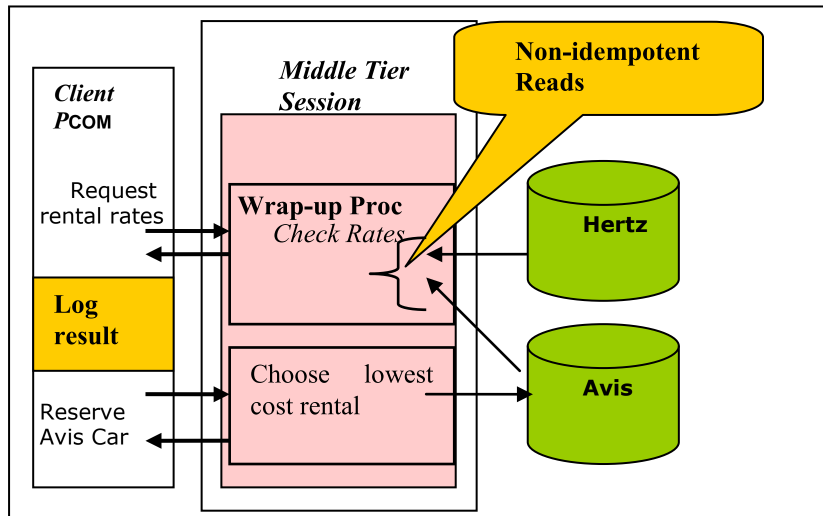

Figure 8: A request is made to a car rental company, based on the cheapest rates we have read. Logging for this choice is done at the client.

WU-Proc: We can exploit client logging more generally, however we must preclude the LLCOM from making state changes that depend upon unlogged non-idempotent reads. The requirement is that activity preceding wrap-up reads be idempotent. Hence, the wrap-up reads may be preceded by invocations of E-proc's within a **wrap-up procedure** (**WU-proc**). So long as this wrap-up reading of external state does not result in updates to LLCOM state outside of the WU-proc activation, it will have no further impact, except via subsequent client requests—and hence after the client has logged the LLCOM reply. Thus, with WU-proc's, the client must be aware that it is calling a WU-proc and force log the WU-proc reply. Subsequent client requests become replayable because of the logging at the client PCOM.

This is reminiscent of middle tier components returning a cookie to the client. The client then returns the cookie to the web service so that the web service can restore the appropriate middle tier state for the client. However, here there is 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, an end user might see the results of middle tier comparison shopping and make a choice that determines the web site the middle tier visits to consummate a purchase. 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. And, of course, the client log needs to be forced before the "reserve" call is made to guarantee deterministic replay.

7 Achieving Availability and Scalability

In this section we provide some perspective on how one might put together an infrastructure that supports multi-tier stateful web applications to achieve dependability using the concepts we have described. This involves not just our logless components, but the surrounding system elements that enable them. We start with these surrounding components and then discuss how to exploit logless middle tier components to provide scalability and availability.

7.1 Enabling Logless Components

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 [5] 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). One downloads DHTML from the middle tier to IE that permits IE to act as a PCOM in our multi-level enterprise application. For our purposes, this DHTML needs to include the functional create call that instantiates the logless middle tier session oriented component, the client logging of user input, and the logging of results returned from the middle tier. The DHTML also specifies how to recover a crashed middle tier LLCOM. Thus, an enterprise infrastructure in the middle tier can exercise control over the client system by providing the client functionality that it desires or requires. This includes the kind of forced logging that is required for wrap-up reads.

The Backend Server: In order to provide persistence in the presence of system failures, idempotence is required for component interactions. Logging enables replay of the interactions to be avoided most of the time. But it does not cover the case where the failure occurs during the interaction. When application replay once again reaches the interaction, it is uncertain 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. Idempotence is required for the fault tolerance techniques of which we are aware. In TP systems, it is often a transactional queue that captures the request and/or reply to enable idempotence.

In many, if not most, services that provide idempotence, detection of duplicate requests relies on a unique identifier for the request message. The remaining arguments of the message are usually ignored, except in the actual execution 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 request identifier only. This style of implementation for idempotence already supports our generalized idempotent requests, though here we require the backend service to provide idempotence in exactly this way when checking for duplicates.

Thus there is frequently nothing new that a backend web service need do to become a GIR request server. The hard part has already been done. Thus, the web service

interaction contract that we proposed in [15] can be readily transformed into a generalized form, usually without having to change the implementation. This contract is a unilateral contract by the web service, requiring nothing from the application. However, in its generalized form, it can support exploratory reads by logless persistent session-oriented components.

7.2 Exploiting Logless Components

Scalability: Because there is no log for an LLCOM, we can re-instantiate an LLCOM at any middle tier site simply by directing its functional initiation call to the site. Should an LLCOM take 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 elsewhere. Note, of course, that the original middle tier 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.

Availability: To a large extent, the persistent component at the client can choose the level of robustness for the middle-tier application. Replication is used for high availability in CORBA [18, 20]. 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 when and how it manages LLCOM instances, a client can achieve varying degrees of availability at varying costs.

1. It may have no standby for its mid-tier session, but rather recover the existing session in place or create and failover to another instantiation should a failure occur.
2. It might create a "warm standby" at another site that has not received any calls following its initiation call. The client would be expected to replay the operational calls, but recovery time would be shortened by avoiding standby creation during recovery.

3. It might create a standby that it keeps current by continuously feeding it the same calls as 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 application servers need not know anything about replication. They simply host the logless session components.

8 Summary

It has long been recognized that dependability is an essential 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.

We have argued that *transparently persistent stateful programming* can be highly effective in providing dependability by permitting an application programmer to delegate to application infrastructure the tasks of scalability and availability. This delegation is essential as it is what enables the programmer to focus almost exclusively on the business logic of the application. And it is this focus that results in applications that are simpler, more maintainable, and more likely to be correct.

Our Phoenix/App prototype at Microsoft Research demonstrated the feasibility of the general approach, which uses automatic redo logging to capture the non-deterministic events that permit applications to be replayed to restore their state. This technology was subsequently extended to enable persistent session-oriented middle tier components that themselves do not need to log, greatly simplifying scalability and availability.

In this paper, we have shown how to extend these persistent session-oriented components to enable them to read and respond to system state without requiring that such reads be idempotent. It requires a stylized form of programming, frequently called a "programming model" that includes E-proc's and WU-proc's. However, the resulting model is only mildly restrictive and the payoff is substantial.

The result is application state persistence with exactly once execution semantics despite system crashes that 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 middle tier log is required (i) performance for normal execution of middle tier applications is excellent and (ii) components can be deployed and redeployed trivially to provide 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. A. Avižienis, J-C Laprie, B. Randell, "Fundamental Concepts of Computer System Dependability", *IARP/IEEE-RAS Workshop*, Seoul, 2001
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 and D. Lomet, "Phoenix Project: Fault Tolerant Applications", *SIGMOD Record* 31(2) 2002, pp. 94–100.
4. R. Barga, D. Lomet, S. Paparizos, H. Yu, and S. Chandrasekaran, "Persistent Applications via Automatic Recovery", *IDEAS Conference*, Hong Kong, 2003, pp. 258–267.
5. 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.
6. R. Barga, D. Lomet, and G. Weikum, "Recovery Guarantees for Multi-tier Applications", *ICDE Conference,* San Jose, 2002, pp. 543–554.
7. P. Bernstein, M. Hsu, and B. Mann, "Implementing Recoverable Requests Using Queues", *SIGMOD Conference,* Atlantic City, 1990, pp. 112–122.
8. P. A. Bernstein and E. Newcomer, *Principles of Transaction Processing*, Morgan Kaufmann, 1996.
9. A. Borg, J. Baumbach, S. Glazer, A message system supporting fault tolerance, *Symposium on Operating Systems Principles*, 1983, pp.90-99.
10. 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)
11. S. Frølund and R. Guerraoui, "A Pragmatic Implementation of e-Transactions", *IEEE Symposium on Reliable Distributed Systems*, Nürnberg, 2000, pp. 186–195.
12. J. Gray: Keynote address "Internet Reliability", *2nd HDCC Workshop*, Santa Cruz, CA. May 2001.
13. J. Gray and A. Reuter, *Transaction Processing: Concepts and Techniques*, Morgan Kaufmann, 1993.
14. H. A. R. Hoare: as quoted in B. Randell: Turing Memorial Lecture Facing Up to Faults. *Comput. J.* 43(2): 95-106 (2000)
15. D. Lomet, "Robust Web Services via Interaction Contracts", *TES Workshop,* Toronto, 2004, pp. 1–14.
16. D. Lomet, "Persistent Middle Tier Components without Logging", *IDEAS Conference*, Montreal, 2005, pp 37-46.
17. D. Lomet, "Faithless Replay" for Persistent Logless Mid-Tier Components. Microsoft Research Technical Report MSR-TR-2008-50, April, 2008.
18. P. Narasimhan, L. Moser, P. M. Melliar-Smith, Lessons Learned in Building a Fault-Tolerant CORBA System. *DSN 2002,* pp 39-44.
19. MSDN Library: Persistence Overview. http://msdn.microsoft.com/workshop/author/persistence/overview.asp.
20. OMG: CORBA 2000. Fault Tolerant CORBA Spec V1.0. http://cgi.omg.org/cgi-bin/doc?ptc/00-04-04.
21. D. Patterson: Recovery Oriented Computing (talk). http://roc.cs.berkeley,edu Sept. 2001
22. G. Shegalov, G. Weikum: Formal Verification of Web Service Interaction Contracts. *IEEE SCC* (2) 2008: 525-528
23. Sun 2001. Enterprise Java Beans Specification, Vers. 2.0, http://java.sun.com/products/ejb/docs.html
24. G. Weikum and G. Vossen. *Transactional Information Systems.* Morgan Kaufmann, 2002.