Issues in the development of transactional Web applications

by R. D. Johnson
D. Reimer

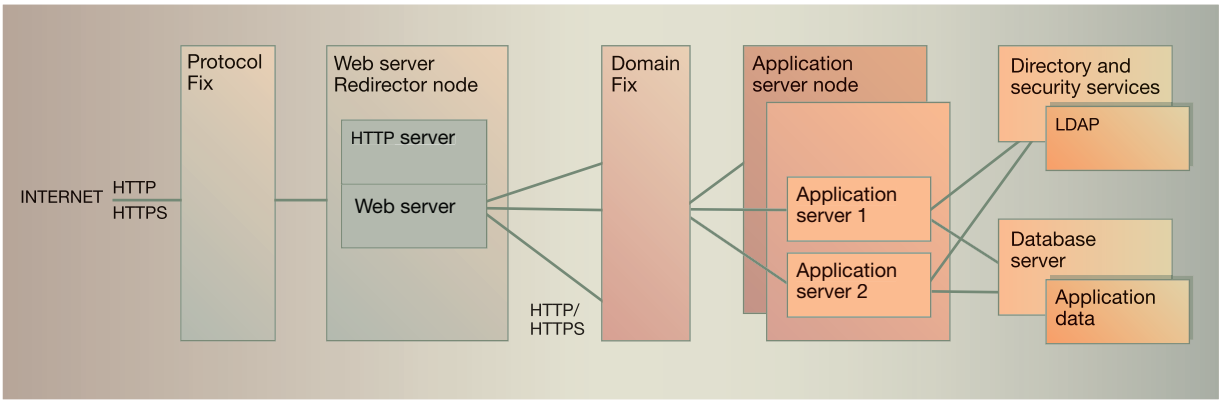
The establishment of the Web application as a successful application model may be attributed to its adherence to open standards, to the powerful J2EE™ application framework for the execution of business and process logic, and to its integration with heterogeneous external systems, transactional processes, and Web technologies. In addition, effective tool support has enabled rapid development and deployment, powerful processes composed from components, and access to a steadily increasing range of function. Many Web applications are designed for high performance, scalability, and highly reliable utilization and are capable of extending modes of usage and growing throughput. Successful application development in these environments involves bridging the gap between exercising available programming specifications and the proper design, coding, and life-cycle management of the application. This paper addresses issues, transactional and nontransactional, in application development that are important for the successful deployment and execution of enterprise Web applications in production environments. These issues are based on experience with deployments of Web applications in various customer environments and are not new, but are either fundamental to previous transactional systems or have appeared in other classes of applications. Understanding these issues is essential both to building effective tooling to support the runtimes and to the evolution and definition of the runtimes themselves.

The advent of the Internet revolution has given rise to a new form of enterprise applications known as enterprise Web applications. The main feature that differentiates enterprise Web applications from their predecessors is the ability to seamlessly integrate multiple disparate systems. However, at their core, enterprise Web applications differ very little from past incarnations of transactional processing systems. Over the past several years, as more and more businesses have automated and integrated business processes, the result has been that the number of individuals developing enterprise applications has grown dramatically. In the past, developers building enterprise applications formed a relatively small community with specialized skills to develop high-throughput, reliable transactional systems. As the number of developers building enterprise applications has grown, more and more general software developers have moved into the development of enterprise applications. In this context it is useful to examine the issues facing the platforms on which enterprise Web applications are built.

At their most basic level, enterprise Web applications are used for transaction processing. Therefore, it stands to reason that the well-known issues from the rich history of transaction processing apply equally well to enterprise Web applications as to past transactional processing systems. The main issues that must be dealt with in the development of trans-

0018-8670/04/$5.00 © 2004 IBM

Figure 1 Example of a topology for a Web application



actional systems include: resource management and utilization, concurrency and parallelism, failure management, persistence of data, and configuration management. Since enterprise Web applications do more than just transaction processing, other issues must be examined in addition to the characteristic transaction processing (TP) issues. These nontransaction processing (non-TP) issues include memory leaks, the efficiency at which data flows through the layers of the application and underlying frameworks, and overly generalized application components and frameworks. An understanding of both the characteristic transaction processing issues and the nontransaction processing issues is not only necessary to develop and deploy successful enterprise Web applications, but is also necessary in looking to the future of the platforms that support these applications. Similar to the spirit of the article "Hints for Computer System Design,"[1] which illustrated the issues in the development of operating systems, this paper shows the issues in the development of transactional Web applications.

The next section gives a brief introduction to the topology of the environment in which transactional Web applications execute. The intent of this section is not a tutorial on how these applications can be configured; rather it is to describe the distributed nature and the complexity of the environment in which these applications run. We then discuss the TP issues in the third section. The fourth section discusses the non-TP issues, and the last section concludes the paper.

Topology of Web applications

Figure 1 shows an example of a topology for a Web application, taken from the WebSphere* 5.0 handbook.[2] This topology is not meant to be representative, but rather just a simple example to illustrate the concepts in the layout of Web applications.[3] Real topologies are typically much more complex when the goal of the deployment is integration of disparate sources of data or functionality. In this topology, a Web server node sits between the application servers and the Internet. The Web server services static content (such as HyperText Markup Language, or HTML, pages). Any requests that require dynamic content, that is, processing by servlets, JavaServer Pages** (JSP**), Enterprise JavaBeans (EJB**), back-end data, and so forth are sent to the application server node.

There can be multiple application servers (multiple Java Virtual Machines each running an application server) on a single machine or multiple hardware platforms running application servers. When a single hardware platform (application server node) runs multiple application servers, the scaling of application servers is referred to as vertical scaling. When multiple physical application server nodes exist in the picture, the scaling is referred to as horizontal scaling. Multiple application servers can run the same individual units of work in the Web application. For example, in one scenario, two application server nodes can exist, each running one application server. On one node, the application server may be executing the presentation layer of the application, and on another node, the application server may be execut-

ing the business logic layer. The business logic application server will be mostly interacting with the database server.

In this topology, the Web server sits between two firewalls, increasing the levels of protection. On one side, the Web server links to the outside Internet using the protocol firewall, and on the other side, the application server nodes are protected still further by the domain firewall. The directory services back end (Lightweight Directory Access Protocol, or LDAP) typically provides other security services for users of the Web application—password, user name, authentication, authorization, and so forth. Finally, the database servers provide persistent storage for retrieval and storage of user data related to the business transactions.

Characteristic issues in TP application design

In this section, we discuss transactional processing issues.

Transactional servers are different from clients. The vast majority of application programmers begin their careers writing applications for clients. Beginning in this way is only natural, because the vast majority of humans begin their experience with computers by using client applications. Those programmers who are experienced with using shared computing resources rarely write transactional applications. As a result, application programmers approach their work with the point of view of client-side, interactive applications on machines where CPU and other resource utilization is not a problem, where concurrency is minimal (since eventually everything is bounded by user-interaction rates), using a particular set of application libraries (e.g., windowing toolkits based on graphical user interfaces, or GUIs, and event-based programming paradigms), and, when using shared resources, with the assumptions of time-sharing systems.

None of these habits, libraries, and patterns serve the programmer writing server-side applications well, for reasons which we will expand upon in this and the following subsections. First, and most important, are issues of resources, scheduling, and concurrency. On a transactional server, unlike on a client, a unit of work has the following properties:

- It almost invariably executes with only its initial input from the user.
- Its output is not complete (and often not even transmitted) until the unit of work is completed.
- Unlike on a time-shared system, every unit of work that actually starts executing must be completed, unless it is aborted, in which case it never is completed.
- The concurrency model that is almost always presented to the programmer is of fully serialized execution semantics,[4] in which no other unit of work is also executing while the unit of work is executing. To the extent that this model is violated, the programmer must exercise greater and greater discipline and knowledge of the underpinnings of the system.
- A transactional server, the context in which the "unit of work" executes, is intended to run *forever*. In contrast, client-side programs operate for the most part under the assumption of at least infrequent (and often frequent) restart; most programs are not designed to run forever, and if they are run in that manner, will eventually display various anomalies. As a researcher once remarked, "Exit(2) is a wonderful garbage collector."[5]
- As implied by the term "unit of work," there is a notion of requests and responses, as well as a notion of the work required on receipt of a request, to construct the desired response. Transactional servers are systems designed to process units of work in a smooth and steady manner.

These constraints yield some surprising (from a client platform point of view) rules that must be followed in order to achieve best results. These rules also provide the following insights as to where tooling and platform support can be enhanced to most effectively help the developer:

- Many client-side and native libraries are simply not usable, because either they are not designed to be re-entrant or they depend on the use of multiple threads, which is ill-advised in a transactional environment.
- As described later, applications and infrastructure must adhere to the two-phase resource-acquisition discipline.[6]
- The developers of such applications must be able to compute (or at least understand) the "budget" of CPU, memory, I/O, and locking required for each transaction.
- Concurrency should be avoided when at all possible. Multithreaded code is difficult to reason about, difficult to code, and difficult to debug. In addition, typical problems with multithreaded code are difficult to diagnose and remedy. Given the difficulty in writing correct multithreaded compo-

nents, it is best if threads are not created dynamically within such applications.
- The legitimate uses of multithreading that remain are almost all for the sake of creating requests to multiple back-end systems in order to achieve overlap of the latencies involved.
- Errors and failures which, on a client, would warrant retries, time-outs, or other measures to recover, almost invariably should *not* be retried on a server. There is an important exception to this rule: when performing operations on some back-end system such as a database, a retry is warranted if the first operation fails because the connection handle might be "stale," but otherwise, no such retry should ever be attempted.

Resource management and utilization. It is not an overstatement to say that the distinctive property of transaction-processing systems that sets them apart from other systems is their unique approach to resource management. There are at least three aspects to resource management from a TP perspective that we feel are absolutely critical and that programmers routinely get wrong. As in the rest of this paper, the rules and provisos we enumerate are in fact described elsewhere, for example, in Reference 7 and other classic texts.

The first important proviso of resource management is that important shared resources, such as database connections, threads, and network connections to remote servers, must be pooled so that they are not created, used, and deleted during each transaction. Although it is obvious to pool database connections, we routinely find both customer code and application code libraries (again, designed for running on clients where this is not a problem) that do not pool or reuse threads. Since Java** threads (or, for that matter, threads in C or C++) have a native memory component, not pooling them, but instead creating, using, and destroying them, causes churn in the native memory allocator, which, under even the least stressful of situations, will eventually fragment native memory, resulting in a progressive increase in the size of the native heap. We have observed the same problems with database access libraries that were written in C, and we conjecture that this is a problem in general for Java server applications accessing native code.

The second important proviso of resource management is that, generally, application programmers must consider the true resource consumption of their code. In client applications, to a first approximation, programmers care about the response time of their programs and about the overall size of their programs. Assumptions are made that most applications will run alone on a client machine, and it is not terribly important whether their code consumes use of the entire machine while it is processing or merely a tiny fraction; what matters, again, is response time. In stark contrast, on a transaction processing system, especially a busy system, it is assumed that at all times, all critical shared resources (memory, CPU, disk I/O) are fully utilized, and hence, the requirements for each transaction must be understood at some level of detail at least.

Simply put, for each transaction, the application programmer should be able to summarize the CPU budget (number of CPU seconds), the number of database operations, their complexity, their read and write requirements on the database, the scope and duration of locks held, and so forth. This summarization is actually critically important because, for instance, if even a small number of CPU-intensive units of work enter the system, they can monopolize the CPU, rendering the entire system unresponsive for all users.

The third important proviso of resource management, a critically important property of transactions, is that they should adhere to the two-phase resource-management discipline.[6] This proviso applies to both traditionally shared resources, such as database connections, as well as to the CPU, the network, and other resources that are not normally thought of as shared. The following two anecdotes from the real world illustrate this point.

In the first anecdote, a customer followed a common design pattern, which is to encapsulate the logic of database access in various functions, as for instance, a function that would take a (retail) stock-keeping unit (SKU) and return its descriptive information, looking it up in a database. Each such function call would take the SKU as a parameter and would acquire a database connection, run the query, release the connection, and return the result. When used in an environment where hundreds of SKUs had to be mapped to descriptive information, hundreds of queries would be run, but, worse, connections would be acquired and released hundreds of times during the same transaction. This activity is not a problem as long as the number of threads desiring connections is fewer than the number of available connections. However, if the number of threads ever exceeds the number of connections, then the *first*

thread will be able to perform its *last* query only after the *last* thread has been able to perform its *next-to-last* query. When the number of threads is greater than the number of connections, the rate of progress made on an individual thread is governed by its ability to acquire a connection, not by any property of the work being done. This rate is a catastrophic degradation in response time because we would expect that the time the first thread would take to perform its last query should be independent of how many other threads are waiting to use connections.

In the second anecdote, another customer code had the same coding pattern, but sometimes a function (which was running a query and, hence, had a connection to the database) would invoke a different function, which would itself request a (different) connection, running other queries. The application would deadlock because of an insufficient number of (pooled) connections, but there was also the possibility of deadlocking the two connections against each other. (Indeed, there were even more than two connections in some situations.)

A common problem here is a lack of understanding as to which resources are being used by the transaction, where, and in what number. In the first case, the application repeatedly allocated and then released the same resource. Under any sort of serious load, the response time of the application would degrade catastrophically. This problem is essentially (in a technically precise sense) a version of thrashing derived from operating systems[8]—the catastrophic degradation of response time under even a modest overcommitment of resources. In the second case, the application, not understanding its resource needs, actually allocated more than one of the desired resource, risking the possibility of:

- Resource exhaustion and resource-related deadlock: If the application already had a connection and needed a connection when no other connections were free, it could end up waiting for a connection, and if all other threads were doing the same, they would all wait forever.
- Classic database deadlock: Because of the same thread performing operations on two different connections that, if they were run on one connection, would be perfectly serializable (by definition, since they were run on one connection) but run on two connections, were in fact nonserializable.

As described above, classic CPU thrashing is a version of this problem that is subtly related to the use of general-purpose time-sharing operating system principles, rather than special-purpose transaction-processing operating system principles. In the first example, we posited a resource (database connections) of which the application needed only one, but for a large number of operations. The application acquired, used, and released that resource repeatedly. Consider any transaction that requires a nontrivial amount of CPU processing. If the application is not able to finish any particular run of CPU-intensive instructions within a scheduling quantum, then it will also be subject to the same problem as in our first example above: the last thread waiting for the CPU will finish its next-to-last unit of CPU-intensive work immediately before the first thread can begin its last unit of CPU-intensive work. In short, the CPU itself has become over-committed, and threads spend their time cycling through the queue, waiting to acquire the CPU.

The solution to this sort of problem is invariably the same and is referred to as the two-phase resource-acquisition discipline: After having acquired a unit of a particular resource, do not release it until such time as the resource will no longer be required for the remainder of the transaction. Likewise, having acquired a resource, store it in such a manner that any other subprogram of the transaction that requires that resource can immediately discover the already allocated unit in a well-known place.

Concurrency and parallelism. On client machines, there is often no real exploitable parallelism. Aside from periodic events (like timers), and background tasks (like MP3 players), the applications themselves are designed to respond to user input and to wait until such input is provided. Although client GUI applications are written in a multithreaded fashion, in fact, the threading is merely a mechanism for expressing concurrency. As Ousterhout[9] has observed, they could have been written using events, and there continues to be a significant debate as to which style of program structure is "better" either from the standpoint of programmer understanding or efficiency. Regardless of which model is chosen for structuring the GUI application, though, by and large the actual code is not truly re-entrant under parallel execution conditions. Because there really is, at any moment, at most one unit of work to execute, application programmers make implicit assumptions about the non-interruptibility of their code; that is, adjacent instructions will not be separated by a context switch. They do not intend to make these assumptions, but without an effective way of testing their systems under

truly concurrent load, it is well-nigh impossible to find latent examples of such bad practice in their code.

In a server, in stark contrast to a client, one must assume that there will always be many independent units of work to execute, and that, hence, any two instructions will eventually be separated by a context switch. This assumption causes two different but related problems:

1. As described above, the intricacies of concurrency make it difficult to understand the behavior of code that manipulates a concurrently shared state. As a result, it is more or less a rule that programmers should never explicitly manage shared state; that instead, they should use external data managers to do this, so that they can write their own code in the model of fully serializable execution.
2. A specific problem occurs when programmers must write code that manipulates shared state in the context of pooling resources. In our experience with commercial transactional applications, the task of writing custom-written database connection pools and custom-written thread pools is an error-prone task. In addition to it being error-prone, the types of errors that result are extremely difficult to diagnose and remedy. Given the difficulty and risk of implementing these structures, we would strongly recommend that programmers avoid attempting the task whenever possible.

A second class of issues is related to concurrency, surrounding contended objects, locking, and hot locks. Transactional applications often manipulate shared state in databases and, as we have described, sometimes do so in server memory. When this shared state is very frequently accessed, and the locks used to enforce mutual exclusion are held for long periods, there is contention among many threads for access to the lock-protected state. The literature on transaction processing is rich in methods for converting algorithms using such shared contended variables into algorithms that do not require such locking.[4]

We will mention only one such classic problem: the "serial number problem." Consider a shopping Web site which, when a browser visits the site for the first time, generates a cookie that is sent back to the browser to identify the browser session. The generation of the cookie could be done by a number of schemes, but a standard one is to use a counter to count upwards, giving each consecutively visiting browser a consecutively numbered cookie. In essence, the browsers are given "serially numbered cookies." If other operations need to occur after the cookie is generated and before the response can be sent back to the browser, the read/write lock on the counter must be held until those operations are complete (and hence will not fail). This can result in contention for the serial-number-counter lock. There are standard solutions to this problem to be found in the literature,[7] depending on how much the consecutive-serial-number criterion can be relaxed.

In a high-concurrency system, lock duration is a critical factor that can affect the throughput of a system. The duration for which locks are held obviously is affected by the number of threads contending for the CPU, the disks, and so forth. It is difficult to predict the interactions among the many kinds of shared variables in an application, and due care must be taken to carefully understand that shared state and the behavior of transactions that will be accessing it. Even a small percentage of misbehaving transactions can dramatically worsen performance and stability by holding locks for too long.

Failure management. When client applications experience failures, usually the following three actions can be taken:

- Automatically retry
- Communicate with the user to determine corrective action
- Exit, crash, or otherwise cancel or stop

Because transactional applications are always run in a mode where user interaction is impossible, and because the equivalent of crashing is to return an error page, applications will often attempt to retry operations, for instance, closing and re-opening connections, and rerunning database queries. There are many reasons why this coding pattern can be problematic. First and most important, it is difficult to know whether failures are persistent or transient or whether retries will or will not aggravate some more complex situation that is still in the process of falling apart. Second, although retrying some operations may be safe, other operations may be impossible to retry because of the failure of semantics required of the transaction as a whole. Thus, for instance, if two-phase commit semantics is expected across two databases, and the connection to one of them fails halfway through the transaction, the only valid course of action may be to fail completely.

In general, it is important for application programmers to consider simply aborting their transactions and allowing some higher-level management to deal with retrying. Even when retries are necessary, it is important to attempt to unwind whatever computation and side effects have occurred to some outermost or earliest point in the code of the transaction and then retry from there, so that any side effects are only committed in the course of a transaction that does not experience any failures.

The second area in which applications tend to inadequately deal with failure is in the handling of exceptions and exceptional conditions.[10] Although it is important to properly treat exceptions in client-side applications, the importance of the following cannot be overemphasized:

- Properly and faithfully recording exceptional condition information in log files *before* attempting any possible remedial action
- Never catching one exception and raising a new one, without either wrapping the original exception in the new one or logging the original exception
- Never employing high-level services, such as databases and publish/subscribe or messaging transports, to log or deal with exceptions

It is common for applications to "swallow" exceptions (never logging them), invoking possibly exception-raising code between the catching of an exception and logging it, or to log exceptions to databases or messaging systems. Such practices in production environments make problem determination in failure situations dramatically more difficult because the relevant exception information will be lost if the database or messaging system is the subsystem that is actually malfunctioning. Another important aspect of dealing with failure is to minimize the number of "moving parts," or number of components or interactions involved, and the complexity of the configurations being used. Many applications are divided into multiple servers, for instance, a "business logic" server and a "presentation logic" server, ostensibly to "separate concerns." Such a design can be a problem because independent failures in the various address spaces, all of which are involved in a single transaction, will significantly increase the chance that transactions will hang or fail in mysterious ways. In general, the failure management discipline described here is known as "fail fast."[11]

Persistent data management. Database-backed applications, unlike client-side GUI applications, have always needed to deal with the fact that the data on which they operate are of two sorts: data that are persistent, stored in database tables, and data that are not persistent, perhaps because such data are derived from persistent data, from user input, or are in a memory-only pseudo-conversational state. Hiding the details of what data are persisted and how the data are persisted makes it simpler to write distributed transactional applications, but these hidden details come with a cost. Although here persistence is described in terms of a database application, applications that use other persistent store techniques suffer from the same issues. In addition, the same issues that apply to the management of persistent data apply to data that are communicated between processes via serialization.

The delineation between persistent and nonpersistent data has significant implications for application design as follows:

- The cost of *transferring* data to and from persistent storage can be prohibitive.
- The cost of *converting* data to and from the persistent *form* can be prohibitive.
- Data legitimately pointed to by legitimately persistent data, but which is not itself legitimately persistent, can inadvertently end up being persisted. A common example is a container such as a hash table containing various translation data. The legitimately persistent data may point to the translation data, but the translation data should not itself be persisted because it is derived information.
- Data (as a subcase of the previous item) that were never even meant to be persistent, but somehow have been made persistent because of programmer error. For example, items such as a database connection or a socket should never be persisted, and if these items are persisted, it is most likely the result of programmer error.
- Applications that depend on the identity of certain objects can be confused when those objects are persisted, then brought back into memory, perhaps while their original versions are still in memory. For example, persisting a singleton will cause problems if the singleton is reactivated while the original singleton is still a live object.

Because database application designers must be aware of the entity-relationship structure of their persistent data at a detailed level, they have been, for the most part, only prey to the first two problems—

that of transport and conversion. It is difficult to imagine a database application that unintentionally duplicated records, unless it was literally a significant programmer error.

In general, high-level frameworks and development platforms, including J2EE** (Java 2 Platform, Enterprise Edition),[12] shield the application programmer from the required level of detail, in at least the following two important places:

- Container-managed persistence frees application programmers from worry about the details of exactly how their data will be converted to persistent form, but at the cost of persisting more information than they intended.
- Automatic HTTP session persistence, combined with a tendency (by application programmers) to store far too much information in the HTTP session object, can result in the persistence of both arbitrarily large data objects and even objects that are illegitimate to persist (e.g., database connections).

In our experience, we have seen the following three problems arise when hiding the details of persistence:

- Being unaware of the actual sizes of objects serialized by Java Object Serialization[13] and, hence, of the extent of CPU, disk, and memory usage caused by such serialization
- Using session state as a convenient "bag" in which to store all sorts of transient objects, which end up being persisted, again with catastrophic consequences for performance
- Objects sometimes not even safely persistable (or worse, their reactivation may cause errors or even crashes), for example, Java objects with native memory or network components

Even after such errors are eliminated, the use of container-managed persistence means that programmers are not aware of the actual database operations that will occur on behalf of their application; often the rate and size of updates can be surprising. In addition, what might have been a simple join in the database may turn into nesting that causes hundreds of database operations. Although this discussion was framed around container-managed persistence, this issue is even more problematic for HTTP session persistence where there is no structure around the data being persisted.

This litany of issues is not new, and, at some level, it is only to be expected. Higher levels of abstraction such as hiding the details of persistence come with a cost of which developers must be aware. Although the details of persistence may be able to be abstracted and hidden from developers for small, relatively simple applications, this abstraction becomes problematic as the complexity and scaling of the application increases. In these larger applications, it is critical that some small core of the application programmers understand what data will be persisted, the size of that data, and the read and update operations that will occur on that data in the course of each of the transactions making up the application.

Configuration: Deployment and update. Deployment is an important role in the life cycle of enterprise Web applications. Deployment consists of installing or updating the configuration of a Web application. The configuration of a Web application is stored in a complex data store, which in Gray and Reuter[7] is referred to as the "repository." In a sense, we can view the administration of a Web application system—a distributed system—as the consistent creation, update, and, in general, management of this configuration data as it resides in the various nodes of the distributed system.

Different runtime platforms typically have different mechanisms for storing and propagating this information. Some versions store the information in a central database from which some subset is cached locally in "administration servers" on each node. Live update of the configuration data is possible, as is putting into effect some amount of the changed parameters. In some versions, this information is stored on each node separately, using a shared-nothing approach, with lazy file replication as a means of moving information from centralized master servers to (possibly distributed) slaves. Most configuration changes require address space restarts, but some configuration changes (e.g., updates to JSP files) will occur much more rapidly (controlled by a timeout).

The key point to understand in deployment is that neither of these methods ensures an "atomic" update across a distributed system, and that neither really ensures any sort of true atomicity for single-node updates.

In neither model above is a complex update going to be reflected in the configuration information atomically, and in neither model is there any assurance that operations in flight might not see an inconsistent configuration state. Moreover, in neither

model is there the ability to roll back or undo configuration operations. As elsewhere in this paper, these considerations are not new. It has been the case in the database world that many sorts of database configuration change simply do not take effect except for new connections or when the database server is restarted. Database administrators accept these as facts of life.

Even among the sorts of configuration change that database administrators can effect on-line, almost all are known to be unwise to attempt in any sort of production environment. For instance, changing the contents of stored procedures (which sometimes requires recompilation) or altering data-definition language (tables, views, and so forth) are operations that database administrators well understand are simply not done during production hours.

In the Web-application world, though, we have seen applications that have:

- Updated configuration parameters during high-load periods
- Updated program files, for example, JSP, during high-load periods
- Performed other sorts of configuration in an interactive manner, relying on operator verification that the appropriate changes were made

These activities are all significant causes of both instability and of paradoxical, almost-impossible-to-reproduce errors, and as such, are almost impossible to trace. It is an important rule that:

- All configuration operations on Web application server systems, to whatever extent possible, occur during nonload times, and as much as possible with systems removed from service, for example, in service windows.
- In order to minimize such service windows, it is critical that any configuration operations are effected using scripts, rather than by manual operator interaction, and that, to whatever extent possible, a new, modified version of an application is deployed, rather than modifying an already installed version.

Let us put these restrictions into a theoretical framework: the distributed configuration of a TP system is in fact a complex distributed database. The administration application that manages that data enforces various consistency constraints, but the following are either extremely difficult or impossible to do:

- Ensure that the running application reflects, at every moment, the configuration in the database.
- Allow any arbitrary series of updates to the configuration to be atomically reflected in the running application configuration.
- Allow arbitrary rollback of changes, again automatically reflected in the running application configuration.

In fact, there is a way to achieve this level of atomicity: to always effect changes by (automatically) building a new version of the application, and hot-swapping in the application. It may be impossible to hot-swap the persistent application data, but certainly for the entirety of the running code, it should be possible.

Such a method of ensuring atomicity is onerous. Not only is it difficult to do full configurations, but the extra hardware required is almost certainly unattainable. A less onerous method, with fewer and less iron-clad assurances, is to always use automated scripts to both apply and undo upgrades. This method of applying changes can be combined with backups to achieve, in most circumstances, the desired atomicity.

In either of the two methods just described, users of a system who are in the process of completing multistage (pseudo-conversational) transactions may notice that an upgrade has occurred and may even see erroneous behavior. Hence, it is important that in both cases a service window is used to ensure that no application traffic is being served during the upgrade period.

Thus from purely theoretical considerations, we can see that these restrictions on administration methods and policies are forced by the requirements of atomicity, consistency, and undoability (the ability to back out) of administration operations.

Characteristically non-TP-related issues in TP application design

In addition to examining the characteristic issues in TP application design, it is worthwhile to look at the most serious non-TP-related issues that affect the development of TP applications. Considering the history of transaction processing, it is not surprising that many issues from normal programming arise in TP application design, development, and debugging.

What is interesting is not the issues themselves, but their appearance in a special class of runtimes, where there is little tooling support for their resolution.

Consider the following classic set of issues:

- Native memory leaks: In C or C++ applications, memory may be allocated and never freed. Likewise, native memory allocators may allocate and then reclaim memory in such a manner as to fragment the free memory into chunks so small as to be unusable.
- Garbage-collected memory leaks: In the Java language or other garbage-collected languages, memory may be allocated, made reachable from some long-lived object, and then never disconnected, for example, a hash table containing all the HTTP session objects ever created, where session objects never expire (at which point they would be removed). Equally, the expiration time may be set far in the future.
- Recomputation of temporary values: Repeated computation of some time-consuming expression (perhaps requiring database access) or merely requiring significant resources.
- Inadequate reuse of complex intermediate values or buffers: Caching and reuse of buffers, formatters, and various other sorts of objects within a transaction and across transactions.
- Overly general and factorized frameworks: The prevalence of object-oriented design methodologies has resulted in the tendency to add unnecessary and inappropriate layers of interfaces to application designs. These layers simply add bloat and path length.

These issues are endemic to all sorts of programming, and no particular area has a monopoly. It is best if these problems are avoided during application development, especially in the transaction-processing world because of the difficulty in discovering these sorts of anomalies in high-stress, high-concurrency, high-throughput environments, and especially in production environments. Several tools that perform very well in the context of client-side applications or single-user applications are typically difficult to use on transactional servers because of the performance cost they impose, the difficulty in starting and stopping transactional servers in production environments, and the distributed nature of these transactional systems.

Conclusion

Enterprise Web applications are significantly different from client-side applications. The transactional nature of these applications typically requires very skilled application programmers to develop them. The issues related to the complexity of these applications are not new and have been studied in depth in earlier transactional systems.

However, what is new is the development challenges that these applications impose on application programmers who are highly skilled in client-side application development and typically do not have the expertise in issues such as concurrency management, resource management, failure management in highly distributed environments, persistent data management, and configuration management. Although frameworks such as J2EE allow the application programmer not to worry about the details of such issues, completely neglecting these issues during development can be catastrophic to application performance, stability, and correctness, particularly in high-stress situations. In this paper, we have discussed the above-mentioned issues related to transaction processing. In addition, we also discussed issues that are common to TP and non-TP applications. However, these issues can have more adverse effects in TP systems under high-load (high-stress) situations.

Acknowledgments

We would like to thank Chet Murthy for his insights on transactional systems and Web application development. We would also like to thank Harini Srinivasan, Gary Sevitsky, Nick Mitchell, David Coleman, Todd Mummert, Edith Schonberg, and Kavitha Srinivas for providing valuable feedback and discussion.

Cited references

1. B. Lampson, "Hints for Computer System Design," *ACM Operating Systems Review* **15**, No. 5, 33–48 (October 1983).
2. C. Sadtler, J. Heyward, A. Iwamoto, N. Jakusz, L. B. Laursen, W. Lee, I. Mauny, S. Rabbi, and A. Sanchez, *IBM WebSphere Application Server V5.0 System Management and Configuration*, WebSphere Handbook Series, IBM Redbooks (April 2003).
3. J. Adams, S. Koushik, G. Vasudeva, and G. Galambos, *Patterns for e-business*, IBM Press (MC Press), Lewisville, TX (2001).

4. N. Goodman, P. A. Bernstein, and V. Hadzilacos, *Concurrency Control and Recovery in Database Systems*, Addison-Wesley Publishing Co., Reading, MA (1987).
5. A. Siegel, private communication (1988).
6. J. T. Robinson, P. A. Franaszek, and A. Thomasian, "Concurrency Control for High Contention Environments," *ACM Transactions on Database Systems* **17**, No. 2, 304–345 (June 1992).
7. J. Gray and A. Reuter, *Transaction Processing: Concepts and Techniques*, Morgan-Kaufmann, San Francisco (1993).
8. G. Gagne, A. Silberschatz, and P. B. Galvin, *Operating Systems Concepts (Sixth Edition)*, John Wiley & Sons Inc., New York (October 2003).
9. J. K. Ousterhout, "Why Threads Are a Bad Idea (for Most Purposes)," *Invited Talk: Usenix Conference* (1996).
10. D. Reimer and H. Srinivasan, "Analyzing Exception Usage in Large Java Applications," *ECOOP 2003 Workshop on Exception Handling in Object-Oriented Programs*, Darmstadt, Germany (July 2003).
11. J. Gray, "Why Do Computers Stop and What Can Be Done about It?" *Symposium on Reliability in Distributed Software and Database Systems* (1986).
12. *Java 2 Platform, Enterprise Edition Specification 1.4*, Sun Microsystems, Inc. (2003), http://java.sun.com/j2ee/1.4/docs.
13. *Java Object Serialization Specification 1.4.4*, Sun Microsystems, Inc. (2002), http://java.sun.com/j2se/1.4.1/docs/guide/serialization.

Accepted for publication December 16, 2003.

Robert D. Johnson *IBM Research Division, Thomas J. Watson Research Center, 19 Skyline Drive, Hawthorne, New York 10532 (robertdj@us.ibm.com).* Dr. Johnson is a research staff member and senior manager in the Systems and Software Department at the Watson Research Center. He received his B.S. degree in physical chemistry from the University of Michigan in 1976, an M.S. degree in physical chemistry from Oregon State University in 1980, and a Ph.D. in physical chemistry in protein dynamics from the University of California at Davis in 1983. He subsequently joined IBM, where he has worked on disk drive technology, characterization of carbon nanotubes, the Sash Web applications framework, the pre-J2EE WOM application server framework, and tools for e-business application performance optimization. Dr. Johnson has received IBM Outstanding Technical Achievement Awards for his work in the areas of Web application server optimization and high-performance head/disk interface characterization, and in carbon-60 and metallofullerene technology. He is a member of the IEEE and the ACM.

Darrell Reimer *IBM Research Division, Thomas J. Watson Research Center, 19 Skyline Drive, Hawthorne, New York 10532 (dreimer@us.ibm.com).* Mr. Reimer is a senior software engineer in the Systems and Software Department at the Watson Research Center. He received his B.S. degree in computer engineering from the University of Manitoba in 1996. He is a technical leader of the SABER code validation project and has significant experience in performance assurance in a large number of J2EE deployments. His research interests include end-to-end program understanding and performance analysis of large Web application deployments and the development of next generation tools and frameworks.