CheeTah: a Lightweight Transaction Server for Plug-and-Play Internet Data Management*

Guy Pardon **Gustavo Alonso**
Information and Communication Systems Group
Institute of Information Systems, Swiss Federal Institute of Technology (ETH)
ETH Zentrum, CH-8092 Zürich, Switzerland
{pardon,alonso}@inf.ethz.ch

Abstract

The ability to maintain transactional interaction in a distributed system has proven to be a key feature in information systems. Unfortunately, as technology moves towards more distribution and decentralization, it becomes increasingly difficult to use existing transactional tools. In fact, current solutions are entirely unsuitable for what we call composite systems. Composite systems can be characterized as a collection of distributed, autonomous components, linked in an arbitrary configuration. In this paper, we describe CheeTah, a Java based set of tools for building composite components capable of interacting transactionally in arbitrary, dynamically changing configurations. We describe the technology provided, how designers would use it to build composite transactional systems, and examine in detail the performance of the resulting solution. Among the results we have achieved, the performance and the simplicity of use are of particular interest.

1 Introduction

Transactions greatly facilitate the task of dealing with failures, recovery, and concurrency control. They also allow to encapsulate operations and associate concrete semantics to them. Tools that provide transactional primitives for the design and development of information systems have a long history behind them, starting with the first TP-monitors [BN97] of almost three decades ago. Today, transactional technology is well understood and widely used.

In spite of this success, there is a growing number of applications for which existing tools are not suitable. The main problem is that current products use a centralized component for scheduling transactions [BK99]. To centralize operations in this way might be exceedingly difficult if the components reside in different organizations or if the components interact over the Internet. It may also be quite difficult in large webfarms or in clusters expanding several LANs. Unfortunately, none of the existing alternatives quite solves the problem. For instance, the TIP protocol [LEK99] provides a limited form of atomicity but no concurrency control. Similarly, persistent queues [IBM99] provide atomic asynchronous interaction but concurrency control cannot be easily enforced. In practice, we do not know of any tool or product that supports transactional interaction without a centralized monitor and without enforcing a static configuration of the components. We see this as a significant limitation in the current state-of-the-art.

At ETH Zürich, we have made this limitation one of our main research themes and have studied its theoretical aspects in great detail [ABFS97, AFPS99a, AFPS99b]. In this paper, we show how a system designer can use the set of tools we provide to build completely autonomous components that, without any centralized coordination, can interact transactionally. The components act as application servers that invoke each other's services to implement increasingly complex application logic. The components can be combined in any configuration and can be dynamically added or removed without compromising correctness. They can be used as wrappers for legacy applications or as infrastructure for transactional agents working across the Internet. They can also be used as EJB containers in the Java Business Components paradigm [EJB]. Our approach has the significant advantage of

*Part of this work has been funded by ETH Zürich within the DRAGON project(Reg-Nr 41-2642.5).

Proceedings of the 26th VLDB Conference, Cairo, Egypt, 2000.

not requiring a very large infrastructure. All a designer has to do is to instantiate and extend a number of classes. In addition, our performance results show that the technology we have developed is not only viable but also quite adequate to the task at hand.

The paper provides an example of composite systems (Section 2), describes CheeTah (section 3), and provides an extensive performance analysis (Section 4), before discussing future work (Section 5). Readers interested in additional information about CheeTah (including a longer version of this work in the form of a technical report) can consult our web pages: *www.inf.ethz.ch/personal/pardon/CheeTah.html.*

2 Motivation: state of the art and related work

2.1 Composite systems

We are interested in distributed and dynamic environments where a collection of different, autonomous information systems interact transactionally. We call such systems composite systems. Figure 1 is an example of such a composite system. The figure illustrates the hierarchy of invocation calls between different services across a variety of components (e.g., *purchase access system*, *product catalogue server*, etc.). Each component is implemented as an independent entity residing in a different location. These components invoke the services provided by other components (Figure 1.a) forming an arbitrary nested client-server hierarchy in which increasing levels of abstraction and functionality can be introduced (Figure 1.b).

In our model, each component consists of an application logic layer (a *server*) that provides access to a resource manager (usually a database). The aim is to design and implement a mechanism that allows to combine such components in any possible configuration so that transactions can be executed across the resulting system guaranteeing (transactionally) correct results even if the configuration is dynamically altered. Moreover, these application layers must remain independent of each other, that is, there should not be a centralized component controlling their behavior.

2.2 Transactions in composite systems: theoretical aspects

Composite systems pose quite challenging problems from the theoretical point of view. At a first sight (Figure 1.b), one could think that a composite system is simply another version of nested [Mos81] or multilevel transactions [Wei91]. However, there are some fundamental differences that introduce non-trivial problems when deciding on correct executions. One important new aspect is that each scheduler is now entirely independent for concurrency control and recovery purposes. Another key difference is that the structure is not regular: schedules cannot be represented as balanced trees. These problems were first addressed in [ABFS97] where an extension to nested [Mos81, BBG89] and multilevel transaction theory [BSW88, Wei91] was proposed. In [AFPS99b] three basic configurations of composite systems were studied in detail. The basic configurations considered were the *stack*, the *fork*, and the *join*. These cases can be readily identified in Figure 1.b: S_6 forms a fork with S_8, S_{10} and S_7. Similarly, S_1, S_5 and S_{11} form a stack while S_{10}, S_7 and S_{12} form a join. Finally, in [AFPS99a] a general solution to the problem of composite systems was suggested by formulating a correctness criterion for arbitrary, dynamic configurations of composite systems.

2.3 Transactions in composite systems: practical aspects

Most existing systems use a flat transaction model where resources allocated to the transaction (locks, sockets, connections, and context information) are kept until the transaction commits. This is a concern for system designers due to the overhead it introduces [Moh98]. In composite systems, the limitations of the flat model become even more acute. Note that, in theory, one could use open nested transactions [GR93] to avoid these limitations. Open nested transactions allow to release resources of each subtransaction before the global transaction commits. The only requirement is to have a compensation action for the subtransaction in case it needs to be aborted. Although open nested transactions have been discussed in the literature and their theoretical advantages are well known, there are very few examples of successful implementations. Even in those systems where *closed* nested transactions are supported (e.g., Encina [Cor95] or in the CORBA specification where they are an optional feature), there are significant practical problems that have not yet been adequately addressed. As an example, most database products do not support nested transactions. As a result, many of the advantages of nested transactions are lost by having to map them to a flat model. In practice, there are two ways to do this mapping. One is to map each subtransaction to a different local transaction. The other is to map all subtransactions of the same root to a single local transaction. If each subtransaction is converted into a separate local transaction, a transaction will deadlock itself if there are conflicts among the subtransactions. This behaviour can be observed in, e.g., Encina. If all subtransactions are mapped onto a single local transaction, concurrent subcalls are not safe because subtransactions are not isolated from each other. Moreover, this mapping has to be specified as part of the

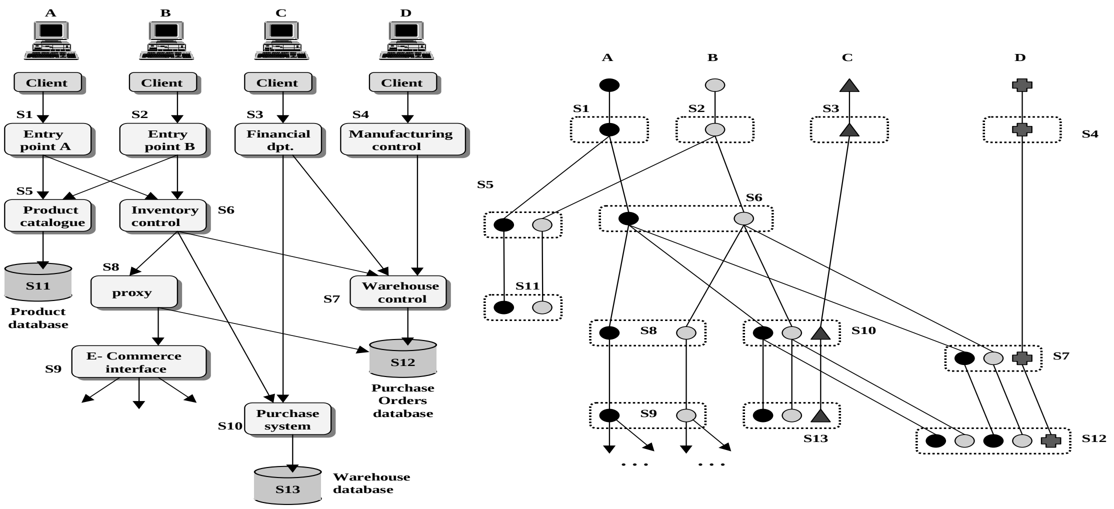

Figure 1: Invocation hierarchy (a, left-hand side) and transactional structure (b, right-hand side) in a composite system

server configuration, meaning that it can not be dynamically changed by a client. For instance, if some client wishes its subtransactions to be executed in parallel, it has no way of asking a server to treat its calls as *different* transactions in the local database. These limitations have often been used to argue that closed nested models are not feasible in practice, let alone open nested models. CheeTah provides an alternative solution to this and many other problems associated with the implementation of nested models. In fact, CheeTah proves that open nested models are feasible and can be used in practice. In CheeTah, we use an open nested model where subtransactions are *always* different local transactions, but we take care of preventing deadlocks while still releasing resources early enough to significantly improve performance.

3 The CheeTah approach to transactional interaction

3.1 System architecture

The main idea behind CheeTah is to provide a lightweight architecture where each component is in itself its own advanced mini-transaction processing monitor. To accomplish this, CheeTah has been implemented as a set of Java classes. The resulting architecture is as follows. In a composite system (Figure 1.b), each server (from S_1 to S_{13}) is an independent component performing its own scheduling and transaction management. These servers are built using Java and inheriting from the classes Cheetah provides. The interface to each server defines the services it implements. An invocation of one of these services (through RMI) results in the creation of a local transaction (child of the invoking transaction and parent of any transaction that might be triggered by invoking the services of other servers). Each transaction is a thread that can invoke SQL statements in a local database (directly connected to that server) as well as services offered by other servers. All the information required to build a global composite transaction is implicitly added by the system to each call. However, it is important to emphasize that each transaction is independently handled at each server. That is, the servers neither communicate among themselves nor rely on a centralized component to make scheduling or recovery decisions. In this way, components can be dynamically added and removed from the system without compromising correctness. All a new server needs to know is the interface and address of the servers it will invoke. Regardless of the configuration, CheeTah guarantees that transactions executed over these servers will be correct (serializable) and recoverable at a global and local level.

3.2 Scheduling and concurrency control

For notational purposes, t will denote a local transaction in a server. Each incoming RMI invocation triggers a local transaction: $start(t)$ will be the start of the transaction, $commit(t)$ the local commit, $abort(t)$ the local abort, and $globalCommit(T)/globalAbort(T)$ the notification to the server where t runs that T, the root transaction (*root* is the term we generally use for the top-level transaction) of t, has committed/aborted.

In each server, concurrency control and scheduling are based on *call level locking*. That is, checking for conflicts is done at the service level and not merely at the level of the operations used to implement those services. Internally, each server uses traditional 2PL to guarantee correctness using standard mechanisms [GR93] but these resources (including connections and internal locks) are always released upon commitment of a local transaction. Call level locks are also acquired following a 2PL discipline but they are kept until the global transaction terminates or until the server unilaterally aborts the corresponding local transaction.

A call level lock is always acquired when a service is invoked. With each server, a conflict matrix needs to be supplied by the programmer. This matrix considers the effects of the forward operation and also of the compensation of an operation. An important characteristic is that, unlike in classical multilevel models, the conflict matrix for our system needs to take *only local* effects into account: whatever is done by remote calls is handled at the remote server. This greatly simplifies the task of identifying conflicts. Informally, we say that two call level locks, l_1 and l_2, obtained on behalf of two local transactions (service invocations) t_1 and t_2, *conflict* if t_1 conflicts with t_2 or t_2 conflicts with u_1, u_1 being the compensation of t_1. For simplicity, we currently use a symmetric conflict table but this can be easily changed if necessary. There is also the possibility of defining conflicts on item level granularity (i.e., conflicts are only possible if both t_1, t_2 are on the same data item) or on service granularity (invocations of t_1, t_2 always conflict regardless what item is accessed).

Although this locking strategy provides correctness, the arbitrary configurations possible in a composite system require a more sophisticated treatment of call level locks. The key problem is that without any additional information, a server cannot distinguish between invocations that have nothing to do with each other and invocations that actually belong to the same root transaction (*siblings*). In the former case the order of execution is not relevant. In the latter case it is relevant. Not to be able to distinguish between these cases can quickly lead to inconsistencies and incorrect executions. Closed nested transactions avoid these situations by simply blocking all conflicting calls. In a composite system, if a server were to block invocations from siblings, a transaction could easily deadlock itself (which *does* happen in existing implementations of closed nested transactions). Preventing such deadlocks would require to have knowledge of the configuration, which contradicts the spirit of composite systems. To avoid such problems, an additional rule is observed at each server: if t_1 and t_2 conflict but both are children of the same root transaction, they can both be executed provided that they are not executed in parallel. This implies that $start(t_2)$ must happen after $commit(t_1)$. With this rule, the scheduler can now block conflicting invocations from other transactions and allow conflicting invocations from the same transaction to proceed.

3.3 Implementation of locking

Each incoming request to a server is mapped to a thread (since they are RMI invocations, this happens automatically). Setting the corresponding call level lock is done by the thread by creating an entry in the local lock table. If there is no conflicting lock, the thread proceeds to execute the code implementing the service. Otherwise, the thread returns with an exception (implying rollback of the local transaction). By immediately returning an exception, we force the client to be programmed in such a way so as to take into account that an invocation may not succeed the first time. On the other hand, resources are more readily available and allow ongoing transactions to terminate sooner.

To facilitate the identification of siblings, the system automatically includes the root id with each RMI call. For faster checking, the root id of a transaction is included in the lock table with the corresponding call level lock. Incoming requests are checked against the corresponding call level lock to see if they conflict and whether they are from the same root transaction.

Lock table management in CheeTah is done following Gray and Reuter [GR93], although the conflict information is more detailed (because of the call level mechanism and user-defined semantics).

3.4 Atomicity: Recovery and undo

At each server and for each service there is an undo operation provided by the designer of the service. Undo operations are local: only the local database updates are compensated. In case of abort of siblings, executing all conflicting undo transactions for the same root in the reverse order of their respective executions guarantees that all changes are undone. Any remote calls will be handled by the undo transactions on the remote servers involved. If no undo operation is provided, the invocation of the service will be treated as a closed nested transaction: resources are not released until the termination of the global transaction.

To be informed about the fate of a transaction, we use the root id (automatically propagated with the RMI call). If the local transaction is still being executed and needs to be aborted, it is undone using traditional mechanisms. If the local transaction needs to be aborted after having been committed, the undo operation is used. The call level lock guarantees that the undo operation can be applied. If the root transaction commits, then an optimized form of 2 Phase Commit is used to commit all subtransactions throughout the system (releasing the call level locks). Note that for early committed subtransactions, this termination protocol simply involves releasing call-level locks, write a log entry and cascade the decision to any remaining servers.

3.5 Implementation of Atomicity

A global transaction is committed using a cascaded variant of 2PC: each server assumes the role of coordinator for all servers it invokes. To speed up the process, different servers are contacted in parallel: each communication round in the two-phase commit protocol is implemented by one separate thread per server involved. The two-phase commit protocol uses the root identifier as the label to indicate to each server which subtransactions are to be committed. Just like all other communication in CheeTah, 2PC happens through RMI. This solves problems with firewalls, because RMI calls can automatically be tunneled through http. A negative acknowledgement (a NO vote) is implemented as a RemoteException being thrown.

In addition, and also for reasons of efficiency, it is not always feasible to wait until the root decides to abort or commit. For instance, servers could be disconnected from the rest of the system or network partitions may occur. In those cases, and given that the system is built upon independent components, each server has the right to undo local transactions on its own – as long as it has not heard from any global outcome. After the undo, all local locks and the call level lock can be released. The "right" to undo a local transaction has to be constrained, otherwise a server could undo its local transaction during the time the global commit message travels through the system. Thus, when a server receives a prepare message and agrees to it, it loses the right to perform a server-side undo.

This approach is complicated by the fact that RMI does not provide *exactly once* semantics. More precisely, the failure of a remote call does not necessarily mean that it has not been executed. It could have been executed, leaving behind a locally committed transaction (t_1) and the corresponding call levels locks set. The invoker, however, sees the call fail and may think the transaction has actually aborted. In that case, the server will eventually time out and undo the transaction locally, releasing the call level locks. This might result in incorrect executions if – on that server – later (successful) calls exist for the same root transaction. Let t_2 denote one such sibling subtransaction executed right after t_1. Locally undoing t_1 with u_1 will only be correct if the sequence t_1 t_2 u_1 is equivalent to the sequence t_1 u_1 t_2 or t_2 t_1 u_1. To avoid these and similar problems, we have taken an expeditive approach. When a server propagates a *globalCommit* operation, it adds to the message the number of invocations it has made to a given server on behalf of the root transaction to be committed. The server that receives the *globalCommit* checks this figure against its own. If they match, then the commit protocol proceeds. Otherwise, the transaction will be aborted. Since in the latter case there are discrepancies about what has been done at each node, aborting seems to be the safest option.

To keep track of all the information needed to perform these operations, CheeTah relies on logging. Each server keeps a log-table and an undo-table inside its local database. As soon as a transaction commits locally, the log-table reflects the fact that the transaction made local changes (needed after recovery). The undo-table contains the needed parameters for executing a compensation if necessary. All this data is written in the same transaction as the user's logic, thereby reducing the number of database transactions to a minimum (one, in this case). A file based log is used to keep track of the two-phase commit status of a transaction after it has been locally committed. On recovery, the system can determine the right action by inspecting the log-table in the database and combining this with the external log file information. For instance, upon restart, a transaction may appear in the database log table but not in the external log file. This is a transaction that locally committed but without a global outcome. The transaction will be compensated as part of the restart procedure, thereby ensuring consistency.

3.6 Dealing with undo operations

Undo operations play an important role in CheeTah. Thus, a key question is whether we can always undo an operation. In this regard, it is important to emphasize that we rely on the service designer to provide the undo operation. Since in CheeTah the programmer only needs to worry about the data integrity of the local server, writing undo operations is relatively straightforward. Nevertheless, if no undo is provided, the system will simply retain resources until the root commits as it is done in existing products. The advantage of CheeTah is that knowledgeable users can exploit open nested transactions to significantly increase the degree of parallelism.

From the concurrency control point of view, executing an undo poses no problem because there is a lock on the corresponding service. If an undo operation needs to be executed, it will always be serialized immediately after the operation it is supposed to undo. However, writing undo operations can be made quite complex by the underlying database system. The typical problems involve dealing with constraints and triggers. In general, as long as there are no non controllable side effects (triggers or constraints that the system – or its programmer – does not know about), CheeTah can handle these cases just like any existing system handles them. That is, by blocking concurrent updates to the same items; so-called *strict* 2PL behaviour.

From our experience working with CheeTah, the knowledge necessary to write undo operations can be compared to what a typical database designer has to know about isolation levels to ensure data consistency in the local database.

3.7 Optimizations of Logging and Locking

Any information that the undo operation may need must be persistently stored (logged). With CheeTah, doing this is quite easy. The programmer only needs to push any information needed for the undo into a *stack object*. This information can be the value of certain variables, the tables used for the undo, transaction id's, and so forth. When the transaction commits, this stack object is written into the log table. In case an abort occurs, the system restores the transaction's undo stack. The undo operation can then read this object and proceed.

When creating the log entry, we use a very important optimization. If this information were saved as an insert into a logging table, and eventually discarded by a corresponding delete, performance would be very poor. Rather, CheeTah uses a pool of undo entries in a fixed-size table (a parameter that can be changed if needed). This table is indexed on a numeric *index* entry. The server component keeps in RAM a list of available entries in the logging table, and allocates them to a transaction when needed. Storing undo data is done by *updating* the log table, rather than *inserting* into it. Without this technique, we would never have been able to reach the current performance.

3.8 A CheeTah Server

With these ideas, the structure of a CheeTah server can be summarized as shown in Figure 2. The structure depends on the particular application, the one shown here being the same one we have used for our experiments.

The server simulates a purchase point where a number of different items can be bought. The interface to the service is a method, *Buy*, that takes as argument the id of the item to be bought, *itemid*. The method is implemented as a Java program that makes calls to a local database and invokes the services of other servers (through RMI calls). This is the code that needs to be implemented by the designer. The lock table provided indicates that two invocations to the *Buy* service conflict if they have the same *itemid* (i.e., they are buying the same thing). Each server uses a local database (currently Oracle8) for storing its own data (log and undo tables) and also to act as the local application database. Access to the database takes place through a JDBC interface using connection pooling.

The server also uses the local file system to store additional information (namely, the global log used to track the progress of 2PC). Each server has conceptually (internally they are deeply intertwined) three transaction managers (*incoming TM*, *internal TM* and *outgoing TM*). The incoming TM takes care of the incoming RMI calls and uses the call level lock table to determine what to do (whether to proceed or to return an exception). It also produces the entries stored in the file system log (related to 2PC) and the undo table. Context information, root id for the transaction, overall status and any additional information related to the *composite* transaction is stored in main memory and managed by the incoming TM. Messages about termination of global transactions are rerouted to the incoming TM where they are processed as explained above. The internal TM takes care of internal consistency (access to shared variables and internal data, as well as access to the local database). It produces the entries in the database log table and it is in charge of rolling back active transactions if they are aborted. Once a transaction commits locally, the internal TM discards all the information related to this transaction. The outgoing TM is quite limited in that it only adds the root transaction data to each remote call.

4 Performance Analysis

4.1 Application Scenario and Parameters

The tests performed are based on the component shown in Figure 2. For simplicity, the server implements one single service invoked with different items as argument. The service is implemented as a Java program that performs a number of internal operations, including a short local transaction (updating the record with key *itemid*) and one service invocation *for each server on the next level*. Both the local transaction and the Java program itself have been kept as short as possible to make sure the measurements reflect the overhead caused by CheeTah and not that of the database or the JVM.

The experiments conducted were based on a total of 10 different system configurations, ranging from the simple wrapper mode (*1x1*) to complex invocation hierarchies (*3x3* or *4x2*) including well-known structures like federations (*2x5*) (see Figure 3 for an example). The goal of the experiments was to analyze the performance of CheeTah and to better understand the impact of system depth and system width on the overall

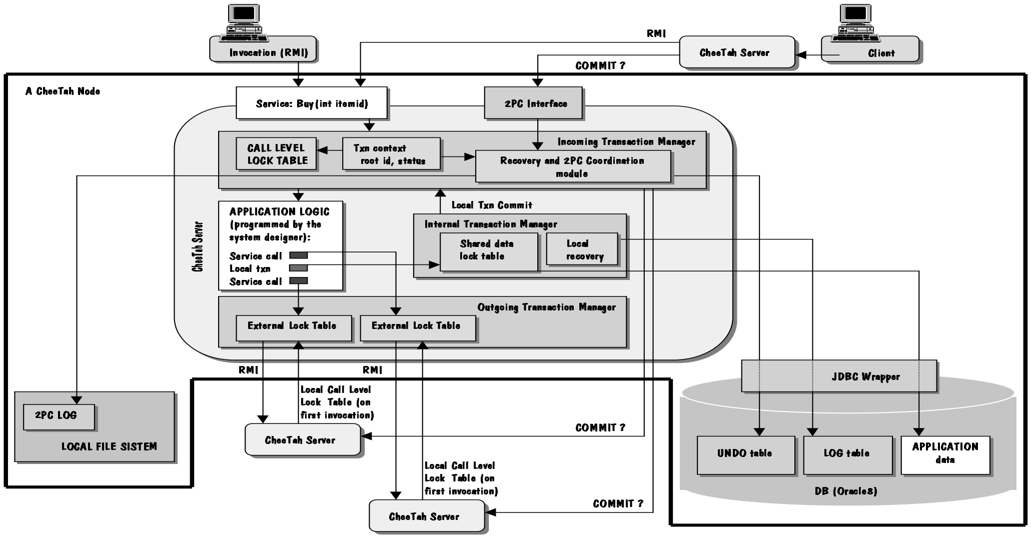

Figure 2: Component Overview

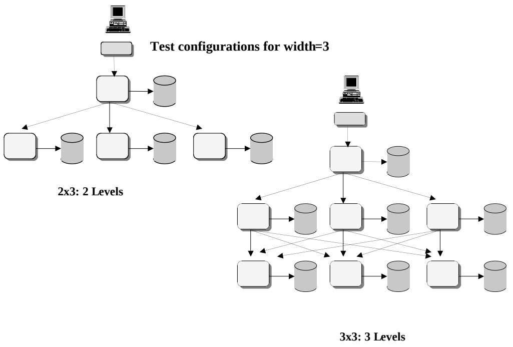

Figure 3: Examples of some configurations used for the tests

performance. For each root transaction, the depth of the system indicates the height of the transaction (how many levels until the leaves are reached). The width of the system indicates how many direct subtransactions each (sub)transaction has. Thus, for instance, in the *3x3* configuration each subtransaction has three children (the root has three children and each one of these subtransactions has another three children). To indicate how many subtransactions are executed on behalf of a given root transaction, we use C, the *cardinality*. Thus, the *3x3* configuration has a cardinality of 13 (the root, plus three children, plus three children for each child of the root). Since invocation of child subtransactions is done serially, the width of the system significantly affects the time it takes to execute a transaction.

In all but two experiments transactions conflict whenever they access the same item. Only for the configuration *4x2* we tested conflict reduction through semantics on the higher level. We considered three cases: no semantics used, half of the servers can use semantics to make conflicts disappear, and all of the servers can make conflicts disappear.

Table 1 summarizes the common settings that apply to our tests. For each one of the tested configurations, we measured throughput, response time and abort rate at the root level and also overall throughput (transactions per minute at all servers). The measurements are based on executions of 10.000 root transactions. For throughput, we measured the time the server needed per 100 transactions, yielding about 100 measurements per experiment in case of our 10.000 roots. The average throughput was obtained by averaging these 100 throughput results for each experiment. The standard deviation represents the confidence (stability) that can be attached to the results. For response times, the average of each of the 10.000 roots' individual response times is given, as well as the standard deviation which reflects the confidence interval.

It is important to emphasize that the tests were all performed on a *worst case scenario* basis. Transactions are made as complex as the system configuration, no semantic information is used to reduce conflict rates (except in two experiments where this technique was tested), and subtransactions are invoked serially. In addition, the load in the system is kept artificially high (as soon as one transaction finishes another one is submitted). The idea behind this approach is that if CheeTah can be made to work on such adverse circum-

Node CPU	Sun Ultra 5, 269MHz UltraSparc IIi CPU
Node RAM	192 MB
Node OS	Solaris 2.6
Node interconnection	100Mbps Ethernet
Java platform	Sun JDK 1.2
Java VM heapsize	16 MB
RDBMS	Oracle 8.0.3
DB Buffersize	64 MB
Database size (per node)	10K tuples
Access path (per node)	unique index on primary key (itemid)
Access mode	80-20 (80% of transactions is on 20% of records)
JDBC Connection pool size (per component)	7
Number of concurrent top-level clients (roots)	25
Root inter-arrival time (per client)	0 (worst possible load)
Total number of *root* transactions per test run	10,000

Table 1: General system parameters

stances, it will certainly work in more realistic, not so demanding environments. In practice, not all transactions will use all servers in the composite system, subtransactions can be executed in parallel, and using semantic information helps to reduce abort rates. Any of these optimizations will improve the results obtained.

4.2 Measurements and Results

Table 2 contains the results of the experiments for the configurations considered (the standard deviation for the overall throughput has been omitted for reasons of space; it is similar to that of the throughput at the root).

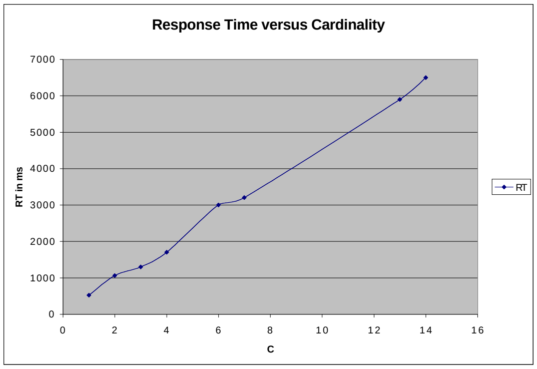

Figure 4: Response Time vs Cardinality

As expected, the throughput at the root decreases and the response time increases with system complexity (i.e., cardinality). This is easier to see in Figures 4 and 5, which show the throughput and response time as a function of the cardinality. In terms of response time, the behavior is obvious. A bigger cardinality implies more complex transactions that naturally take longer to complete. However, the linear relation observed in Figure 4 (almost matching $RT = C \cdot RT_{1x1}$) demonstrates that the increase in response time is directly related to the complexity of the transaction. Therefore, CheeTah does not add additional overhead as the system becomes more complex. This is surprising since one would expect that longer transactions would block more resources and, therefore, will add more overhead. In practice, CheeTah behaved very well. Observing each individual server, it turned out that the open nested policy allowed servers to free resources quite quickly. The policy of aborting transactions as soon as they run into a conflict also helped in that these transactions could be quickly restarted and, with high probability, succeeded the second time. By aborting them early, the overall delay introduced was minimal, even in those cases with high abort rates (*3x3* and *4x2*).

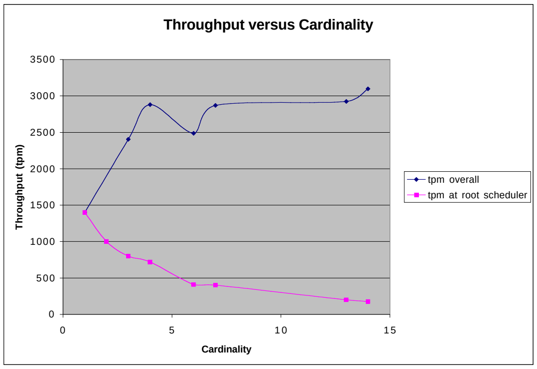

Figure 5: Throughput vs Cardinality

The throughput results are also interesting. As the lower curve in Figure 5 shows, the throughput at the root quickly decreases with cardinality. This is an artifact of the experimental setting (we maintained a fixed number of root transactions in the system at all times; the more complex the transaction, implies the longer it takes to complete and, therefore, the less transactions entering the system). In practice, we observed

Configuration (Levels x Width)	C	tpm (root) Avg	tpm Stdev	tpm (overall) Avg	Resp.Time Avg (ms)	Resp.Time Stdev (ms)	Abort Rate (%) Avg
1x1	1	1400	150	1400	520	500	0
2x1	2	1000	100	1991	1060	530	1
3x1	3	870	160	2647	1300	380	1.7
4x1	4	720	170	2872	1700	510	5.7
2x2	3	800	60	2402	1650	700	1.6
3x2	7	400	80	2872	3200	1100	7.1
4x2	14	175	25	3096	6500	1100	31
2x3	4	720	70	2882	1800	600	2.1
3x3	13	200	20	2924	5900	1000	22
2x5	6	410	20	2485	3000	800	5.7

Table 2: Results for different configurations

that, even for the highest cardinalities, the servers had enough spare capacity to run additional transactions. This is clear from the results for overall throughput. As the upper curve in Figure 5 shows, the overall throughput increases as we go from cardinality 1 to 3 and then remains stable (the bump in the curve is a result of the different configurations; the low point being cardinality 6, in the *2x5* case). Again, this proves that system complexity does not affect performance. In reality, as the system becomes more complex, there is more processing capacity. If transactions follow different paths from the root to the leaves, then the system will be able to process many more transactions. In fact, in our experiments, we only saturated the server in the *1x1* case.

These results show that CheeTah is very close to having optimal performance. For the range of configurations tested, the response time directly depends on transactional complexity and the throughput remains stable. Both parameters are unaffected by the complexity of the composite system.

For high cardinality, the number of aborted transactions is very high (31 % for the *4x2* and 22 % for the *3x3* configuration). These results, however, are due to the worst case scenario we used for tests and not to any characteristic intrinsic to CheeTah. Any system with this kind of load would have similar abort rates. In more realistic applications, these numbers will go down significantly. To test this hypothesis, we ran a set of experiments using semantic information to reduce conflict rates. The experiments are performed for the *4x2* configuration assuming all transactions accessing the same item conflict (as in Table 2), eliminating half of these conflicts and then eliminating these conflicts entirely. The results are shown in Table 3.

These results show that, once transactions follow different paths, subtransactions are executed in parallel, and the inter-arrival rate is more evenly distributed, the percentage of aborts will drop significantly. Interestingly, throughput and response times do not seem to improve by exploiting semantics. This is due to the fact that transactions that abort are aborted very quickly (because of conflicts at the higher levels in the system) and then restarted, thereby incurring in a minimal penalty in terms of response time. Similarly, and as pointed out above, since the system is far from its saturation point, abort rates have no influence on the throughput.

The results of the tests were hindered by a feature of the JDBC implementation we were using: not explicitly closing a statement after it has been executed (notably in case of exceptions) causes the server to run out of memory. The only solution (on the level of connection pooling) is to catch this error and re-open the connection. The resulting overhead is quite large. For instance, we have peak results of 2400 transactions per minute (in the 1x1 case), although the average is only 1400 tpm. The difference is largely due to this phenomenon.

4.3 Comparison with Existing Systems

The previous results show that CheeTah has close to an ideal performance behavior as the system complexity increases. To evaluate the basic overhead of CheeTah, we compared similar systems implemented in a commercial database and a commercial TP-Monitor. To obtain clear results, we compared the *1x1* case.

The comparison with the TP-Monitor provides us with a yardstick to test CheeTah against tools used in 3 tier architectures. In the *1x1* configuration, CheeTah achieved more than three times the throughput reachable with the TP-Monitor. The reason is that CheeTah performs all the transaction management inside the server and, unlike in the TP-Monitor, no context changes are necessary to access the different modules of the TP-monitor. We corroborated these results by also comparing performance in the *2x2* case. For root transactions, CheeTah again outperformed the TP-Monitor, a fact that became very clear when the overall throughput in the system is considered. Again, this is due to the fact that the TP-Monitor, like existing commercial products, uses a centralized component for

Configuration (Levels x Width)	C	tpm Avg	tpm (overall) Avg	tpm Stdev	Resp.Time Avg (ms)	Resp.Time Stdev (ms)	Abort Rate (%) Avg
4x2 (all conflict)	14	175	3096	25	6500	1100	31
4x2 (half conflict)	14	210	3300	30	6000	1200	17
4x2 (no conflicts)	14	200	3100	20	6700	1200	1.25

Table 3: Results with high-level semantics advantage

transaction management. Thus, distribution does not bring much in terms of overall performance since the centralized component is the bottleneck and it cannot be distributed. This is where CheeTah excels: each component has its own transaction manager and, therefore, the more components, the more distributed is the load on the transaction manager functionality. These results clearly speak in favor of the language framework approach followed in CheeTah.

The comparison with a commercial database gives us a way to test the performance of CheeTah against 2 tier architectures. For the test, the equivalent to a CheeTah server was implemented in a normal RMI server application that directly uses the database. The database was accessed via a pure JDBC interface and connection pooling (of the same size as in CheeTah) was used. This eliminated the typical CheeTah overhead for propagating transaction context and doing extra logging. The peak rates obtained proved that the performance of CheeTah is comparable to that of commercial databases.

5 Conclusions

We have presented CheeTah, a light weight transaction monitor implemented as a Java framework. CheeTah introduces many novel aspects, the main contributions being the composite systems structure, the use of open nested transactions, and the framework approach. We see composite systems as a key configuration in distributed environments, one that will certainly be promoted by developments like Java, component based design, and standards like Enterprise Java Beans. In such systems, the efficient implementation of transactional interaction requires open nested transactions. To our knowledge, CheeTah is the first working implementation of open nested transactions made publicly available. Finally, the framework approach to building transactional components is an entirely new paradigm. The transactional properties traditionally provided by large systems like TP-Monitors are provided by CheeTah by simply writing services that inherit from certain classes. The development effort is thus significantly reduced. Our results prove that the ideas implemented in CheeTah are feasible in practice and have excellent performance.

References

[ABFS97] G. Alonso, S. Blott, A. Fessler, and H.-J. Schek. Correctness and Parallelism of Composite Systems. In *Proceedings of the 16th ACM Symposium on Principles of Database Systems, Tucson, Arizona, USA. May 12-15.*, May 1997.

[AFPS99a] G. Alonso, A. Fessler, G. Pardon, and H.-J. Schek. Correctness in general configurations of transactional components. In *Proceedings of the ACM Symposium on Principles of Database Systems (PODS'99)*, Philadelphia, PA, May 31 - June 2 1999.

[AFPS99b] G. Alonso, A. Fessler, G. Pardon, and H.-J. Schek. Transactions in stack, fork and join composite systems. In *Int. Conference on Database Theory*, 1999.

[BBG89] C. Beeri, P.A. Bernstein, and N. Goodman. A Model for Concurrency in Nested Transaction Systems. *Journal of the ACM*, 36(2), 1989.

[BK99] K. Boucher and F. Katz. *Essential guide to object monitors.* John Wiley & Sons, 1999.

[BN97] P.A. Bernstein and E. Newcomer. *Principles of Transaction Processing for the Systems Professional.* Morgan Kaufmann, 1997.

[BSW88] C. Beeri, H.-J. Schek, and G. Weikum. Multi-level transaction management, theoretical art or practical need? In *International Conference on Extending Database Technology (EDBT'88), Lecture Notes in Computer Science, Springer-Verlag, LNCS 303, 1988*, 1988.

[Cor95] Transarc Corporation. *Writing Encina Applications.* Transarc Corporation, 1995. ENC-D5012-00.

[EJB] Enterprise javabeans technology. http://java.sun.com/products/ejb/index.html.

[GR93] J. Gray and A. Reuter. *Transaction Processing: Concepts and Techniques.* Morgan Kaufman, 1993.

[IBM99] IBM. *MQSeries.* IBM, March 1999. http://www.software.ibm.com//mqseries/.

[LEK99] J. Lyon, K. Evans, and J. Klein. Transaction Internet Protocol (TIP). Technical report, Tandem and Microsoft, February 1999.

[Moh98] C. Mohan. Transaction processing and distributed computing in the internet age. http://www-rodin.inria.fr/ mohan/abstracts.html, July 1998.

[Mos81] J.E.B. Moss. *Nested Transactions: An Approach to Reliable Distributed Computing.* PhD thesis, M.I.T. Laboratory for Computer Science, Cambridge, Massachusetts, MIT Press, 1981.

[Wei91] G. Weikum. Principles and realization strategies of multilevel transaction management. *ACM Transactions on Database Systems*, 16(1), March 1991.