Building Extensible and High Performance Distributed Transaction Service[*]

Xin ZHANG zhangxin@otcaix.iscas.ac.cn
Chang XU changxu@otcaix.iscas.ac.cn
Beihong JIN jbh@otcaix.iscas.ac.cn
Technology Center of Software Engineering, *Institute of Software*
The Chinese Academy of Sciences, *Beijing* 100080, *China*

Abstract

Distributed transaction service is one of the most important services in a J2EE application server, and its efficiency has a big impact on the latter's overall performance. This paper presents a distributed transaction service design which is both extensible and portable in a J2EE application server, and strategies to optimize its performance. Experimental data indicate much improved performance of the transaction service without sacrifice of transaction integrity.

1. Introduction

An application server generally offers a complete environment for constructing, deploying and running large-scale distributed applications. Its importance and potential have been widely recognized among researchers and developers. Java-based J2EE[1] application server is built upon a series of specifications of various services required in distributed computing environments. The distributed transaction service following the JTS[2] and JTA[3] specifications is a core service and is usually provided through a distributed transaction manager.

Distributed transaction manager needs to interact with application server and a set of distributed heterogeneous resource managers to accomplish distributed transaction service cooperatively. An improperly designed transaction manager is prone to be tight-coupled with one specific application server therefore difficult to be extended and reused. This paper presents the design and implementation of a distributed transaction manager, ISTX, which implements JTS service and JTA interface in a J2EE application server.

The design presents in this paper uses a variety of techniques such as decoupling the distributed transaction interface from the implementation to support various resource managers, offering the possibility of choosing different atomic commit protocols in different situations and environments, optimizing the commit protocol procedure, and so on, to make the transaction service extensible. It takes only some minor modifications of interfaces to adapt ISTX to different application servers. Further, ISTX can be extended to be a distributed transaction framework for a number of extended transaction models.

The efficiency of distributed transaction service has a big impact over the performance and throughput of J2EE application servers. While J2EE specifications bring convenience for developing a distributed transaction manager, but relatively rigid constraints may reduce the flexibility of the software and limit transaction-processing capacity. Based on our experience in distributed transaction processing, the bottle-necks generally appear in two occasions: opening connections to resource

[*]This work was supported by the National Grand Fundamental Research 973 Program of China under Grant No.2002CB312005; the National High-Tech Research & Development 863 Program of China under Grant Nos.2001AA113010, 2003AA115440.

managers and doing the blocked synchronization operations in *2PC* (Two Phase Commit) protocol procedure. Therefore, corresponding performance tuning strategies are introduced into ISTX to resolve the above performance bottle-necks: *connection pooling* and *asynchronous operation dispatching*. Performance comparisons validate our observations and approaches: connection pooling greatly reduces the average time of getting a connection to resource managers from 10 milliseconds to about zero millisecond and asynchronous operation dispatching also improves the transaction throughput by over 100 percent. These optimization measures aid to realize the high-performance distributed transaction service as expected.

The rest of this paper is organized as following: section 2 presents our design essentials, which is proved to have better extensibility; section 3 describes the optimization strategies and implementation; section 4 gives performance tests and results; related researches are discussed in section 5, and we conclude the paper in the last section.

2. Extensible transaction service

In J2EE framework, there are four parts involved in a distributed transaction application: *application*, *application server*, *transaction manager* and *resource manager*. A J2EE compliant *application server* provides a running environment for distributed applications, and it offers distributed transaction service via a *transaction manager*. A *resource manager* provides the ability for applications to access the resources. In order to guarantee the consistency among resources and the correct implementation of transaction, we need a mechanism to coordinate these four parts, which is defined in JTS/JTA specification in details. JTA specifies local Java interfaces between a transaction manager and the parties involved in a distributed transaction system [3]. JTS specifies the implementation of a transaction manager which supports the JTA specification at the high-level and implements the Java mapping of the OMG Object Transaction Service (OTS) 1.1 Specification at the low-level [2]. Figure 1 shows the interactions in a distributed transaction.

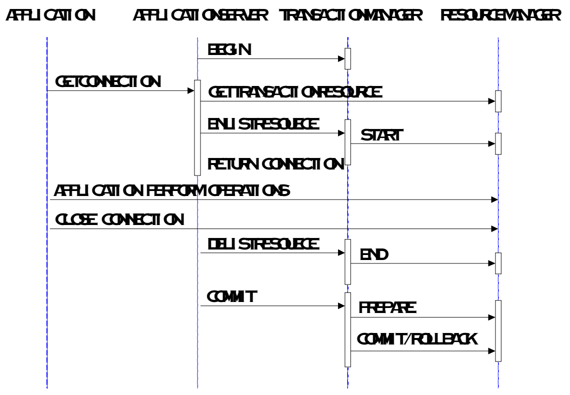
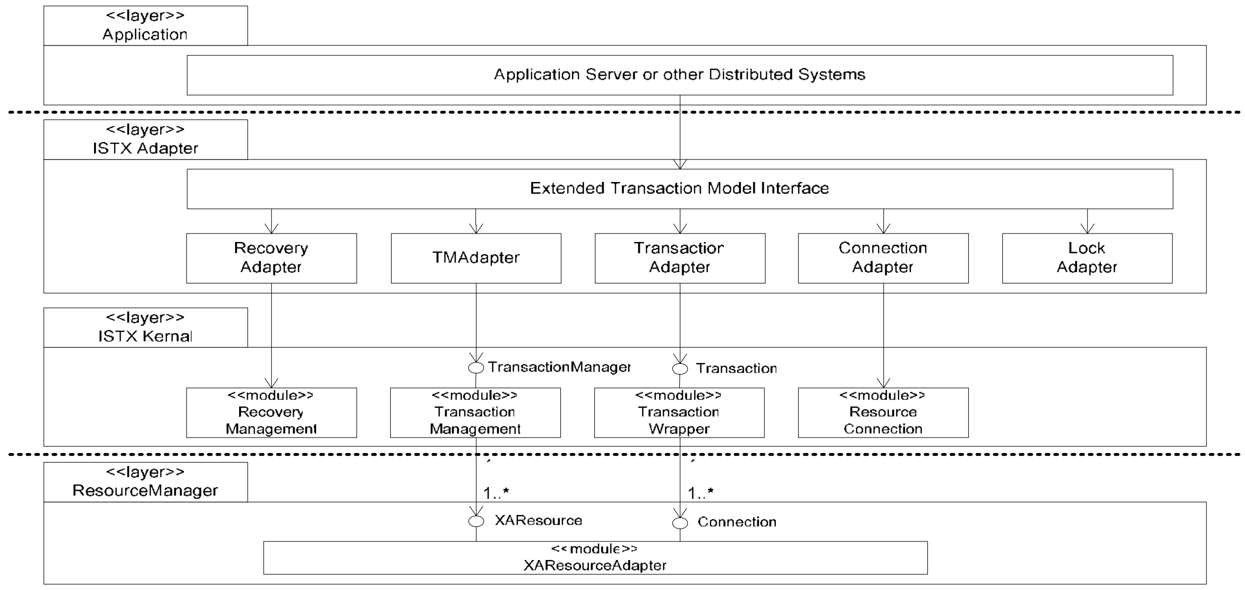


Figure 1. **Interactions in a distributed transaction**

The architecture of ISTX and ISTX's relations with application server and resource manager are shown in Figure 2. We divide ISTX kernel into four main modules: transaction management, transaction wrapper, recovery management and resource connection module. The transaction management module is responsible for the life cycle managing and transaction scheduling. Transaction wrapper module implements the commit protocol and supplies *Transaction* interface to application server. Recovery management module provides recovery during failure via log files. And resource connection module answers for resource connection and connection pooling management. The Adapter layer is used to support other extended transaction models, which we will describe in details later.

In order to improve the extensibility, we bring the following schemes into our design.

Wrapping the implementation of transaction object: JTA specification defines *Transaction* interface for transaction manager and application server. While keeping in mind to leverage the existing implementations of transaction objects, we utilize *Adapter* in ISTX. That is, we separate the object TransactionWrapper and TransactionImpl in TransactionWrapper module, in which the former emphasizes on *Transaction* interface wrapper and light-weight front-end implementation, and the latter is responsible for the implementation of commit protocol.



<<layer>> Application
Application Server or other Distributed Systems
<<layer>> ISTX Adapter
Extended Transaction Model Interface
Recovery Adapter
TMAdapter
Transaction Adapter
Connection Adapter
Lock Adapter
<<layer>> ISTX Kernal
TransactionManager
Transaction
<<module>> Recovery Management
<<module>> Transaction Management
<<module>> Transaction Wrapper
<<module>> Resource Connection
<<layer>> ResourceManager
1..*
1..*
XAResource
Connection
<<module>> XAResourceAdapter

Figure 2. **The architectural overview of ISTX**

By decoupling the interface and implementation of transaction object, we can transform an existing transaction object implementation into a transaction wrapper with standard *Transaction* interface, and then we can replace the inner transaction object implementation easily. And similarly, we also separate the implementation from the interface of connection pool module, so that we can replace it with another existing pool implementation.

Encapsulating replaceable commit protocol: Commit protocol defines the relationship and interactions between the participants involved in a distributed transaction, while 2PC protocol is used in most cases. However, 2PC protocol is relatively complicated and not suitable for all scenarios. For instance, if only one resource involved or just "read" the operations which have been imposed on resources, some simpler commit protocols, such as 1PC protocol, seems to be more appropriate. It will be hard to optimize or replace the commit protocol if it is tightly coupled with the transaction manager. Hence in TransactionWrapper module, we encapsulate a commit protocol module which implements a number of commit protocols. As a result, we can optimize the protocol and also replace its implementation conveniently.

Extending to support other transaction models: Building an extended transaction facility from scratch is rather tough. Extending an existing transaction system which limits the work to a relatively modest scope seems to be more effective. ISTX is intended to be a platform upon which extended transaction development may take place. Extended transaction semantics is provided by extended transaction model interface including transaction demarcation interface and other extend transaction utilities. Adapter layer wraps the basic implementation of ISTX. We need to add some new adapter facilities for extended transaction model, which is also shown in Figure 2.

Let's take *Nested Transaction Model*[4, 5] for example. It is an extended transaction model put forward by E. Moss in 1985, which allows a new transaction to begin inside an existing transaction. Nested transaction model differs much from ordinary flat transaction model. For instance, in flat transaction model when a thread has been involved into a transaction, it will not be allowed to start a new transaction while it is opposite in nested transaction model. And they also differ in the vote collection procedure in 2PC. So we use *Adapter* to encapsulate the basic implementation for which we desire, such as the *TMAdapter*, which supports transaction reentering, and the *TransactionAdapter*, which does some minor modifications on the basic 2PC implementation. Moreover, some new adapter facilities may have to be added. In nested transaction model, relationship of inheriting locks between parent transaction and sub-transactions are much more complicated. Therefore, *LockAdapter* is introduced to the adapter layer. All these adapters define a set of extended transaction model interfaces. After encapsulating and adding adapters correspondingly, ISTX is able to give a full support to nested transaction model.

3. Strategies and implementation of optimization

Performance is one of the most significant factors of distributed transaction service. In ISTX, many efforts have been made in order to improve the performance, among which *Connection Pooling* and *Asynchronous Operation Dispatching* have been proved to be effective.

3.1. Connection pooling

Opening connections to resource managers is a costly operation. If requests for opening connections arise frequently, the performance will be greatly decreased. This situation often appears in Web environments. Connection pooling mechanism can considerably improve the performance[6].

In fact, database that supports JDBC2.0 specification commonly supplies connection pooling mechanism inside the database itself. However, it has some limitations. First, the quantity of pooled connection managed by database is always very limited. Moreover, in a Web environment, the most popular scenario for application servers, applications often hold connections for a long time. When the quantity limit is reached, other application has to create a new connection rather than reuse a pooled connection. Second, the pooled connections inside the database are almost impossible to configure or control by user applications. So we encapsulate relatively simple connection pool offered by databases into our configurable one that will benefit users much. As we can see from section 4.1, our mechanism is proved to have greatly improved the performance, besides better meeting our requirement.

Connection allocation: When requesting for opening connection arises, ISTX will check for free connections in the pool. When found, the connection will be given to the application. If there is no free connection, ISTX will create a new one if the limit has not been reached, otherwise, the application will have to wait. During the period of time of waiting, if other application releases a connection or garbage collection thread deletes some invalid connections, a background waiting thread will be wakeup to assign application with connection. Especially, when there is more than one application in the waiting mode, the waiting thread will choose one application from the waiting list according to the priority of waiting application and time it has waited. That is to say, when free connections are available, ISTX is using a FCFS (First come, first serve) mode. While no connection is available, event trigger mechanism is put into use.

Connection reuse: We create connection pool to conserve connections that has been built but not currently used. Connections will be returned to the pool rather than closed after using, for later reuse.

Each connection acquired from the pool has been registered with an event listener. Once the application finished using the connection, a recycle event will be triggered. The event handler first inspects that whether the connection ought to be delisted from the transaction manager, and secondly check whether the connection is still valid, then put it back to the pool if valid, or directly delete it otherwise.

Garbage collection: If connections that held by applications are forgotten to be closed, or have been involved in deadlocks, these connections will be held for a long time and cannot be released, and consequently become *dead connections*, which greatly decrease the performance. So we put *Garbage Collection* into use. We setup a recycle thread, which inspects connections in the pool periodically, recycles connections that has been held but not been accessed for a relatively long time, and deletes invalid connections. The recycle thread is also in charge of recycling the valid connections. If the idle time of a free connection in the pool is beyond a threshold, this connection will also be recycled.

3.2. Asynchronous operation dispatching

We bring *asynchronous operation dispatching*[7] into use in ISTX. In this way the server needn't block itself to wait for the completion of operations, but just hand the ready operations to a background work thread, and return back immediately to handle the next operation. Later the 0server will gather all results of these operations at some critical points.

2PC's execution is the most expensive while involving various heterogeneous and distributed resource managers. According to our experiments, *asynchronous operation dispatching* can greatly improve the concurrency inside transaction by executing operations on resource managers simultaneously.

In the procedure of transaction 2PC protocol, there are many operations requiring execution on all resource managers, such as preparing, committing and rolling back. JTA specifies ISTX should access resource managers through *XAResource* interface which, unfortunately, only allows synchronous operations.

But we notice that the transaction 2PC protocol doesn't appear an eager appetite for some intermediate results. Let's take the vote procedure for example. The first early vote from the fastest responded resource manager isn't used immediately until ISTX has gathered all the needed votes that are enough for the next decision on the current transaction.

To increase the concurrency inside every transaction, we try to implement *asynchronous operation dispatching*. Because *XAResource* interface provides no aid for asynchronous operations, we have to rely on our own design. And what we want to implement is *asynchronous operation dispatching* inside transaction that doesn't influence JTA interface.

In *asynchronous operation dispatching* model, ISTX never waits for the completion of each operation on resource managers, but just issues the operation and passes it to a background work thread. ISTX continues to do so until the time to gather all execution results to make the next decision.

Asynchronous operation dispatching also need multi-threading support. Since frequent creation of new threads leads to extra costs that may reduce the advantage from optimization, ISTX builds a thread pool as the necessary technical support. For the sake of better software reusability, we apply *Rendezvous* pattern[8, 9] here. Figure 3 illustrates the implementation of *asynchronous operation dispatching* in ISTX. It comprises the following classes:

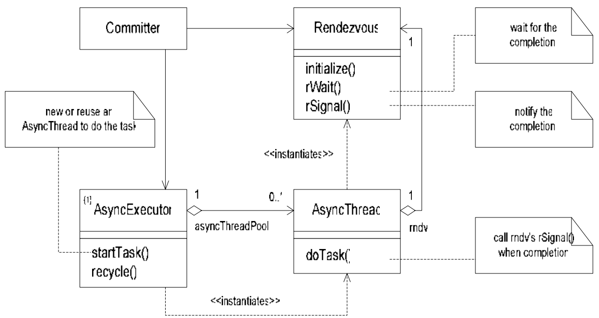


Figure 3. **Asynchronous operation**

Committer: *Committer* is responsible for the implementation of the transaction 2PC protocol. It passes each operation on resource managers to *AsyncExecutor* and obtains a corresponding *Rendezvous* object at the same time. Later it can check the operation result through *rWait* method of this *Rendezvous* object.

AsyncExecutor: *AsyncExecutor* accepts each operation from *Committer*, and creates or reuses an *AsyncThread* that actually executes the operation. After that, *AsyncExecutor* returns a corresponding *Rendezvous* object to *Committer* for synchronization control.

AsyncThread: *AsyncThread* executes each operation passed from *AsyncExecutor*. Once its task completed, *AsyncThread* invokes *rSignal* method of its corresponding *Rendezvous* object to inform *Committer* of the completion.

Rendezvous: *Rendezvous* object is used for synchronization control of operation dispatching.

Asynchronous operation dispatching separates the completion time of an operation from the result-checking time of this operation, which reduces the coupling between the executor (*AsyncExecutor*) and the invoker (*Committer*). The executor, upon completing the operation, saves the completion notification into the corresponding *Rendezvous* object from which the invoker can acquire required information.

Asynchronous operation dispatching has some influence on the implementation of the transaction 2PC protocol, but it is entirely transparent to client programming. *Asynchronous operation dispatching* also has an impact on the complexity of the software, but it enhances the concurrency inside the transaction at the same time.

3.3. Other optimizations

We adopt many other optimization measures such as *synchronous multi-threading, timeout object management optimization*, *asynchronous logging*, and so on.

Synchronous multi-threading is a widely used approach in enhancing software performance. By starting multiple threads to handle transaction requests from various clients, we are able to enhance ISTX's concurrency, and thus improve the performance of processing distributed transactions. Moreover, it is transparent to clients.

We also utilize *Balanced Branch Tree* to improve timeout object management, which demonstrates better efficiency than a traditional *Queue* structure. When a new timeout object is added, the time complexity with a Queue data structure is O(n), which in the former is O($\log_2$n).

Logs of 2PC ought to be recorded to persistent storage, typically, harddisk. However, operations on harddisk are relatively slow. So we utilize *asynchronous logging* to record some *lazy* records, which doesn't necessarily wait for the end of the logging operation, to increase the concurrency.

4. Performance tests

We have made two groups of tests aiming at our two main optimizations, which we present in detail as follows.

4.1. Performance tests for Connection Pooling

We adopt JBoss 2.4.4, an excellent and well-known open-source J2EE application server, as the platform, and Oracle9i as the database in the test for *Connection Pooling*. Our test suite comprises three kinds of tests. The first uses *DriverManager* interface implemented by Oracle9i, which is defined in JDBC1.0 specification. It doesn't offer methods of getting transactional connection, and thus cannot supply recovery function when system crashes. The second uses *DataSource* interface implemented by Oracle9i, which is defined in JDBC2.0 specification. It utilizes connection pooling mechanism offered by database. The third test uses *DataSource* interface based on our *Connection Pooling* implementation. We try to get a connection to database for 1000 times using the above three methods, and get three groups of data respectively, as shown in Table 1.

Table 1. **Comparison of the three ways of opening a connection (unit milliseconds)**

Program No.	Average Time	Min. Time	Max. Time	Standard Deviation
1	46	31	9156	288.23
2	10	0	1172	36.93
3	0	0	16	0.887

The average time of opening connection is the most important factor for estimating performance. As shown in Table 1, the average time of opening a connection in the first test program, which operates without any *Connection Pooling* optimization, is 46 milliseconds. And it costs 10 milliseconds to open a connection using database connection pooling in average. And our optimization reduces the average time from 10 milliseconds to about zero millisecond (The results are round to millisecond.). The standard deviation for the result of test program 1 is 288.23, indicating that the results fluctuate more acutely. And the standard deviation of test program 3 is 0.887.

The magnitude of max time in opening a connection is seconds when using interfaces offered by database vendor, while it is 10 milliseconds when using ISTX. We owe such a performance boost to the connection pre-building. When ISTX starts up, it will build one connection pool for each involved data source and pre-build a certain number of cached connections to this data source in the pool, which will be used for a faster acquisition of connections.

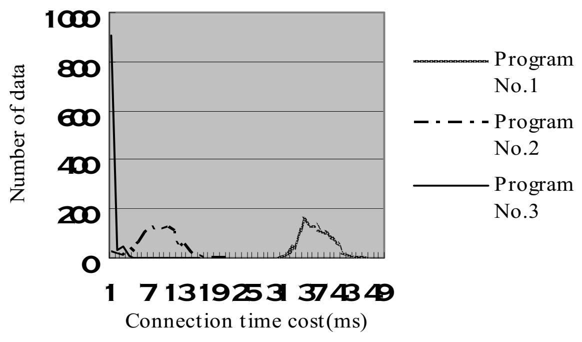


Figure 4. **Statistics distribution**

Figure 4 shows the distribution of the statistics. The x-axis denotes the time cost to get a connection to database (millisecond, in integer). The y-axis represents how many times that a given one connection time cost appears during the 1000 times circle, i.e., if it cost 50 milliseconds to obtain a connection, and the 50 milliseconds appears 10 times during the 1000 times' test. Then we draw a point at (50, 10). Some points, such as the points which represent the max time in test program 1 and 2, are so far from the average time that we didn't place them in the figure. As we can see from the figure, these curves approximate to be normal distribution. And the results obtained by using our optimization measures are much better than those obtained by using other two measures.

4.2. Performance tests for asynchronous operation dispatching

TPC-C[10] is the performance test standard for evaluating distributed transaction managers in industry. We follow its basic principles and do some minor adjustments to better meet our testing requirements. We choose two J2EE application servers in company with ISTX for the performance test. They are JBoss 2.4.4 and WebFrame 2.0, and the latter is a research prototype developed by the Technology Center of Software Engineering, Institute of Software, Chinese Academy of Sciences.

We find that a*synchronous operation dispatching* doesn't produce a significant performance promotion under the local area network condition. Considering the speed of accessing local databases is too high, we try to simulate a wide area network environment by increasing accessing time. In the simulated wide network environment, we find that the quantity of the resource managers becomes an important factor that affects ISTX's performance much. The test program will perform 900 transactions. The quantity of resource managers involved in transactions can be adjusted from 1 to 5 at most. We try to compare ISTX's performances under different conditions. Here are the test results.

Table 2. **Throughput of JBoss + ISTX (No. of transactions processed / process time (seconds))**

Opt.	1 DBs	2 DBs	3 DBs	4 DBs	5 DBs
Off	204.73	70.16	46.29	34.98	27.70
On	203.07	132.22	123.97	118.92	111.55

Table 3. **Throughput of WebFrame + ISTX (No. of transactions processed / process time (seconds))**

Opt.	1 DBs	2 DBs	3 DBs	4 DBs	5 DBs
off	183.64	66.31	45.12	34.10	27.45
on	186.84	123.61	118.69	114.74	109.44

Experiment data indicate that *asynchonous operation dispatching* produces a considerably good performance promotion in the simulated wide area network environment, and the effect will be even greater as the number of resource managers involved in transactions grows. The following is improved performance percentage calculated from Table 4 and Table 5.

Table 4. **Improved Performance Percentage**

App. Server	1 db.	2 db.s	3 db.s	4 db.s	5 db.s
JBoss	N/A	88.5%	167.8%	240.0%	302.8%
WebFrame	N/A	86.4%	163.1%	236.4%	298.8%

We can infer the following conclusions from Table 4:

1. The 2PC protocol is well implemented, which guarantees that ISTX can correctly handle transaction requests when the quantity of the resource managers varies.
2. In the simulated wide network environment, ISTX's efficiency can be greatly improved as the quantity of resource managers involved in transactions grows.

Suppose the quantity of resource managers involved in a transaction is *n*. Since all the operations on resource managers can be performed entirely concurrently in *asynchonous operation dispatching* model, the execution time of the transaction 2PC protocol will be only 1/*n* compared with that under the condition of disabling the optimization. However, *asynchonous operation dispatching* inevitably adds extra system loads, and ISTX's performance isn't solely determined by the efficiency of transaction 2PC protocol, so it is hard to reach the theoretic optimum.

5. Related work

Some research work has been explored in extensible transaction processing framework. [11] presents an object-oriented transaction processing framework, which supplies common patterns and necessary operations in building transaction systems. [12] builds a reflective framework on traditional TP (transaction processing) monitors, which give supports to a number of extended transaction models. Nevertheless, this framework is built on traditional TP monitors, and cannot be portable to J2EE application servers easily.

As for transaction processing performance, [13] presents a transaction committing protocol OPT, which can efficiently improve the transaction throughput by "optimistically" borrowing uncommitted data. Aimed at the main memory database systems, [14] also presents a transaction committing protocol that can make the committing procedure shorter by utilizing some features of update operations and logs of main memory database systems. However, these approaches focus on local transactions, so they are unfit for J2EE application servers. Some work is intended for particular transaction. For example, [15] provides a new way of statically analyzing transaction procedures during the process of compiling applications. A sequential execution procedure is presented by combining the semantic information from static analysis and the runtime information from dynamic execution. But this idea is also unsuitable for J2EE environment because JTS doesn't support nested transactions which is exactly the premise of Peter's analysis way. Moreover, it depends on highly structured object bases, which are quite different from widely used relational database systems.

6. Conclusions

As a key component of J2EE application servers, distributed transaction manager following JTS and JTA specifications plays an important role in J2EE environment. Specific transaction manager designed and implemented by using traditional software engineering methods is hard to be extensible and reusable, and its performance is one of the most important factors that limit the transaction throughput in J2EE application servers. In this paper, we present our design which takes into account the factors of service extensibility for a distributed transaction manager ISTX. Then we explore optimization strategies and implementation for ISTX to gain a better performance, and the later tests confirm our efforts. The design and performance optimization strategies of ISTX can also be applied to a broader middleware development.

References

[1] Sun Microsystems Inc., *Java 2 Platform Enterprise Edition Specification v 1.3*, July 27, 2001.

[2] Sun Microsystems Inc., *Java Transaction Service (JTS)*, December 1, 1999.

[3] Sun Microsystems Inc., *Java Transaction API (JTA)*, April 29, 1999.

[4] C. Mohan, "Tutorial: Advanced Transaction Models Survey and Critique", In *ACM SIGMOD International Conference on Management of Data*, Minneapolis, May 1994.

[5] J. E. B. Moss, *Nested Transactions: An Approach to Reliable Distributed Computing*, In MIT Press, 1985.

[6] Hans Bergsten, "Improved Performance with a Connection Pool", http://webdeveloperjournal.com/columns/connection_pool.html.

[7] James Hu, Irfan Pyarali, Douglas C. Schmidt, "Applying the Proactor Pattern to High-Performance Web Servers", In *Proceedings of the 10th International Conference on*

Parallel and Distributed Computing and Systems, IASTED, October 1998.

[8] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides, *Design Patterns: Elements of Reusable Object-Oriented Software*, Addison Wesley, 1995.

[9] R. G. Lavender, and D. C. Schmidt, “Active Object: An Object Behavioral Pattern for Concurrent Programming”, In *Pattern Languages of Program Design* (J. M. Vlissides, J. O. Coplien, and N. L. Kerth eds.), Addison-Wesley Publishing Company, 1996, pp.483-499.

[10] Transaction Processing Performance Council, *TPC BENCHMARK^{TM} C, Standard Specification Revision 5.0*, February 26, 2001.

[11] Tekinerdogan, B. “An Application Framework for Building Dynamically Configurable Transaction Systems”, In *OOPSLA '96, Development of Object-Oriented Frameworks Workshop*, 1996, San Jose, US

[12] R. Barga and C. Pu. “Reflection on a legacy transaction processing monitor”. In *Proceedings Reflection '96, San Francisco*, CA, USA, April 1996.

[13] Ramesh Gupta, Jayant Haritsa, Krithi Ramamritham, “Revisiting Commit Processing in Distributed Database Systems”, In *Proceedings of the 1997 ACM International Conference on Management of Data*, Tucson, Arizona, USA, May 1997.

[14] Inseon Lee, Heon Y. Yeom, “A Fast Commit Protocol for Distributed Main Memory Database Systems”, *International Conference on Information Networking Wireless Communications Technologies and Network Applications 2002*, Lecture Notes in Computer Science, Volume 2344, 2002.

[15] P.C.J. Graham and K.E. Barker, “Improved Scheduling in Object Bases Using Statically Derived Information”, In *The International Journal of Microcomputer Applications (IJMA)*, 1995, Vol. 14, No. 3, pp. 114-122.