Distributed Transactions in Practice

Prabhu Ram, Lyman Do, and Pamela Drew
Boeing Phantom Works, Mathematics and Computing Technology,
The Boeing Company,
P.O. Box 3307, M/S 7L-40,
Seattle, WA 98124-2207.
{prabhu.ram, lyman.s.do, pamela.a.drew}@boeing.com

Abstract

The concept of transactions and its application has found wide and often indiscriminate usage. In large enterprises, the model for distributed database applications has moved away from the client-server model to a multi-tier model with large database application software forming the middle tier. The software philosophy of "buy and not build" in large enterprises has had a major influence by extending functional requirements such as transactions and data consistency throughout the multiple tiers. In this article, we will discuss the effects of applying traditional transaction management techniques to multi-tier architectures in distributed environments. We will show the performance costs associated with distributed transactions and discuss ways by which enterprises really manage their distributed data to circumvent this performance hit. Our intent is to share our experience as an industrial customer with the database research and vendor community to create more usable and scalable designs.

1 Introduction

In today's competitive business world, enterprises prefer to focus on their core business. They spend minimum effort on IT development projects and build their business systems by integrating commercial off the shelf (COTS) software [COR98, MIC98]. The enterprises' management have deemed this approach to be the most cost effective way of developing their information management systems. Each COTS software meets a functional need of the business system. Selected COTS software products are integrated using distributed communication technology such as CORBA, DCE, and DCOM [OMG98, OSF98, MIC98]. Examples of such COTS software include enterprise resource planners, product data managers, enterprise process modelers, financial management software, etc.

Many COTS software products are vertical database applications that store their business and internal data in a database system. If the enterprise is geographically distributed, as most large enterprises are, the COTS software products and their underlying databases are often distributed too for locality, autonomy, and performance reasons. Most of these COTS software products use the DBMS as a data archive that provides access to their contents (business data). Application and business logic are embedded in the COTS software.

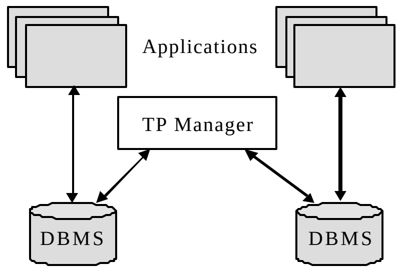


Figure 1: Two Tier Architecture

In the classical client/server model (Figure 1) of system deployment most of the logic pertaining to the application is embedded in the server. "Thin" clients request services of the servers and the servers execute the application logic on the client's behalf. The function of the transaction manager was to coordinate transactions between databases. The proliferation of generic, pre-built vertical applications - the COTS software products - has introduced additional tiers between the client and the application servers. These multi-tiered architectures (Figure 2) have had two major impacts on distributed transactions. First, the global transaction manager is no longer directly interacting with the databases but the coordination is through the COTS software products. As we will describe in Section 2, the COTS software has to provide additional functionalities to facilitate such coordination. Second, the multi-tier architectures need a heterogeneous commit protocol which, as shown in Section 3, imposes a significant performance penalty on distributed transactions.

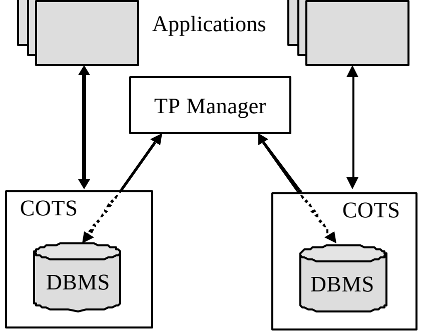


Figure 2: Multi-Tier Architecture

2 Effect of COTS Software

In this section, we introduce the effects COTS software has on distributed systems. The implications on transaction management and how enterprises manage the consistency of their data are also discussed.

2.1 COTS Software based Environments

Users interact with COTS software by means of "business transactions", which are atomic units of work from the users' perspective. A business transaction may constitute multiple interactions between the applications and the users. These interactions may also cause the COTS software to initiate multiple database transactions to the underlying databases. If only one COTS software is involved, it may be possible to use the DBMS' native distributed transaction management facility if the underlying databases are homogeneous, or to use a transaction processing monitor to provide transaction management support. However, a typical integrated business system consists of more than one COTS software and usually these COTS software products and their embedded (heterogeneous and autonomous) databases are integrated in a multi-tiered architecture. In addition, most COTS software products are developed by independent software vendors and the integration with other products occurs almost always much later in its lifecycle and in some cases, in an ad hoc manner. Even when products interoperate, it is most likely they only interact at the interface level and not at the semantic level. For example, the definition of an atomic unit of work in one product may not match that of another. The heterogeneity and autonomy of the encapsulated databases coupled with the COTS software not being designed for large scale integration make the task of providing transaction management much more difficult in an integrated system. In order to provide transaction management in this environment, each COTS software has to function as a resource manager. It serves as a wrapper to its encapsulated database (as shown in Figure 2) and bridges the communication between a global transaction coordinator and its encapsulated database(s).

2.2 Consistency Management Approaches

There are three possible scenarios as to how transaction management can be implemented with COTS software - a) the COTS software can use the native transaction management facility of its embedded DBMSs (with some limitations that we will discuss), b) the COTS software interacts with an external transaction management system to coordinate distributed transactions, and c) the software provides no transaction management at all and lets the end user assume the responsibility for maintaining data consistency via business processes. In The Boeing Company, we have production systems which are examples of each of the above scenarios.

In the first case, there can be multiple homogeneous DBMSs controlled by the same COTS software (due to distribution) and the COTS software can choose to use the distributed transaction management facility of the DBMS product by bracketing the business transaction with transaction `begin` and `commit`. The assumption clearly is that the databases are now aware of each other and function as one distributed database. This approach may not allow multiple COTS software products to interact, due to the potential heterogeneity of the embedded databases and the encapsulation of the databases by the COTS software products. Hence, using the native transaction management function has limited applicability in integrated multi-COTS software business systems.

If transaction management is required and the COTS software encapsulates the database, most employ middleware technology for the purpose. As a result of encapsulation, the DBMSs participating in distributed transactions must now coordinate their actions through the COTS software products. Transaction management components can either be embedded within the COTS software or can be provided by an external software as a service. Middleware, such as transaction processing monitors [BEA96, TRA97], object transaction monitors [ION97, RDPZ99], and transaction servers [MTS98], provide the two phase commit mechanism by allowing distributed transactions to be bracketed using proprietary transaction `begin` and `commit`.

Regardless of whether the transaction manager is embedded or is external to the COTS software, numerous modifications must be made to the COTS software to support the transaction manager. If a COTS software encapsulates the underlying database and is expected to interact transactionally with other software, it must be able to behave as a resource manager to interact with a global transaction coordinator and to re-direct the global coordination to its encapsulated databases. This involves mapping a global transaction to one or more local transactions and exporting transaction states such as PREPARED to allow coordination with externally originating transactions. It must be noted that the concept of heterogeneous two phase commit (2PC) and support for it through operations such as PREPARE took nearly a decade to get implemented by DBMS vendors. Breitbart [BRE90] references several early research efforts that mentioned the need for an operation such as PREPARE but it was not until the last four to five years that most popular DBMS vendors support such operations. Because COTS software products encapsulate the databases, transaction coordination happens at the COTS level instead of at the database level, and the requirement to support 2PC has been passed on to COTS software as well.

Given the technical complexities associated with integrating external transaction managers into the architecture and given that this requires modification of the COTS behavior, several COTS software products simply do not provide global transaction management and leave it up to the end-users to mitigate the risks to their data through their own business processes. This is fairly common in large scale integration projects. This approach is aided by the fact that most transaction management software is another software system for the end-user to manage and the maintenance cost becomes prohibitive in a long run. Hence, there has not been enough of a push by end-users to demand transaction management compliance of COTS software products. Other factors which detract from transaction management usage, such as performance degradation, will be discussed in Section 3.2.

The business data is still valuable to the enterprise. So even if the COTS software products do not provide transaction management, the end-users have business and management processes to maintain the consistency of data in their business systems. An example of a business process is that a user may not be allowed to execute a particular application when another application is executing. Business processes often depend on the end-users' training and discipline and hence are prone to errors. Other processes may be used to detect and correct inconsistent data across COTS software. For instance, end-users may monitor message traffic across COTS software by using message entry/exit points in the distributed communication layer and capture the message traffic as it flows through the system. The captured messages may be written to a database and be analyzed later for errors. For example, if CORBA is the integration technology used, the messages are captured each time an inter-ORB communication (ORB of one COTS software to another) occurs using marshaling exits. This essentially captures four message entry/exit points (as shown in Figure 3) in a distributed business transaction. This kind of reactive approach can detect failed transactions but can never detect an incorrect interleaving between two sub-transactions. As a result, data errors can potentially proliferate to other applications due to the time lag between the failed transaction and the manual correction effort.

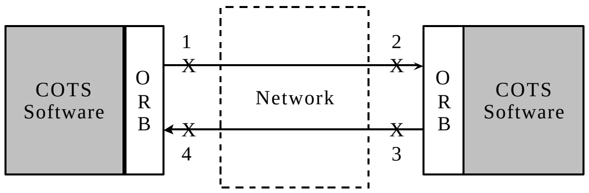


Figure 3: Auditing of Distributed Message

The risk of not detecting all the faults is assumed by the end-user organization but this could be mitigated by the COTS software characteristics and application semantics. For example, if the COTS software versions its data, the potential for conflict does not exist any more and only failed non-atomic business transactions need to be detected. Given the technical, management, and performance complexities associated with incorporating distributed transaction management into COTS software products (as shown in Section 3.2), end users feel justified by following processes such as these to improve the quality of their data without using transaction management.

Given the three approaches to maintaining data consistency, by native DBMS transaction management functions, by middleware transaction managers, and by leveraging business processes without any global transaction coordination, an enterprise may choose any of the above approaches (based on what their COTS vendors supply) to maintain the consistency of its business data and buttress it with their own business and data management processes. However, many enterprises lean towards the last approach of not using transaction management at all and use their own business and management processes to keep their data consistent. One dominant reason for this is the performance overhead associated with transaction management in distributed integrated systems.

3 The Performance Impact

Besides delivering on functionality, performance in large distributed system is often the gauge of how successful an implementation is. Transactions come with a performance cost for reasons including rollback segments maintenance, additional logging, etc. When transactions are distributed and heterogeneous, other factors such as database connections management and commit protocols also contribute to the overhead.

When distributed databases are involved, presumed abort 2PC is by far the dominant mechanism used in commercial products to achieve atomicity in distributed transactions. Coupled with strictness [BHG87], they provide serializability to global distributed transactions. 2PC is very often the only mechanism available to maintain data consistency in many distributed systems. Additionally, most of the 2PC implementations are coupled with synchronous communication mechanisms (such as RPC) causing each step of the transaction to be blocked until the step preceding it has completed resulting in further degradation of response time.

One alternative to synchronous 2PC for improving transaction response time is asynchronous mechanisms such as persistent queues. It allows a client to detach from a server prior to the completion of the server execution, resulting in better response time for the client. If the asynchronous mechanism also guarantees transactional semantics (once and only once execution) it can be very attractive to applications that do not require synchronous (and blocking) behavior.

Interestingly, asynchronous mechanisms (which are implemented as resource managers) also use the 2PC protocol in their execution. They essentially break a distributed transaction into three parts - one to deliver the data to the asynchronous mechanism, the second to reliably deliver the data to a remote computer (by the asynchronous mechanisms), and the third to remove the data from the asynchronous mechanisms. If databases are involved in steps one and three, a heterogeneous 2PC protocol is executed between the asynchronous mechanism and the databases to give atomicity to those individual transactions. Since the asynchronous mechanism guarantees once and only once execution of its messages, the three steps together behave as one atomic global transaction. Besides giving a better response time for clients, by splitting an atomic global transaction into three atomic transactions, asynchronous transactional mechanisms have the potential to improve transaction throughput.

An additional component of heterogeneous, distributed transactions is the use of the XA interface [XA91]. If using heterogeneous or encapsulated databases, the XA interface is the only available commercial way of coordinating commits of distributed transactions. XA is an X/Open standard that is implemented by the resource manager vendors and allows the resource manager to expose transaction states for global commitment purposes. Both synchronous and asynchronous protocols use the XA interface to coordinate a transaction between database systems and other resource managers. The use of the XA interface also comes with a performance cost which we will discuss in Section 3.2.

Since distributed transactions introduce overhead, we performed several experiments to quantify the costs associated with them. For various workloads we will show the cost associated with performing a global commit and the cost of using the XA interface to facilitate commit coordination. We will also compare response time and throughput of 2PC coordinated transactions against an asynchronous transaction mechanism implemented using transactional queues for various workloads and system configurations.

3.1 Experimental Setup

Our experiments are designed to isolate the costs associated with commit coordination and related components of distributed heterogeneous transactions. To monitor the behavior of the transaction management components, the DBMS was required to perform uniformly at all times. We achieved this by making all our database operations *Inserts* – which causes a true write to the disk system (as against an update for which the DBMS may go to a cache and the commercial DBMS we used in our experiments allows dirty reads). Our workloads varied from single record insert to 1400 record inserts per transaction characterizing a spectrum that can map on to OLTP to Decision Support Systems type of workloads. The experiments were also run in isolation without any non-essential applications running. Records in our databases were 100 bytes long each.

Our experiments were performed on three 300 MHz NT servers with 128M RAM each. Two copies of a commercial DBMS product were hosted on two servers and functioned as autonomous DBMSs unaware of the other's existence. The DBMSs were tuned identically and the database and log size were set large enough to handle our workloads without causing extent allocation during the execution of our tests. All the results reported are from warm runs - the applications were run for a few minutes, followed by an interval in which results were recorded, followed by the application running for few more minutes. We controlled the experiments using a commercial transaction processing monitor that provided a heterogeneous 2PC mechanism and a transactional queue which we used as the asynchronous mechanism. The queue (which is also a resource manager) was hosted in the third machine, so that the queue and the DBMSs do not compete on I/O.

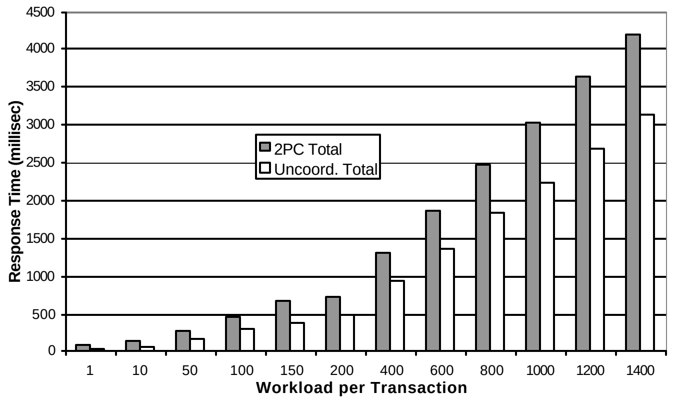


Figure 4: 2PC vs. Uncoordinated Transactions

3.2 Results and Discussions

Figure 4 shows the overall response times of 2PC coordinated synchronous transactions against the equivalent uncoordinated distributed "transactions". The 2PC coordinated transactions were implemented through the XA interface to obtain connections and handles to the DBMSs and used the 2PC mechanism provided by the TP monitor to coordinate global transactions. The uncoordinated operations were implemented by having the client call remote servers that were implemented using the DBMS' native APIs and without blanketing with the global transaction `begin` and `commit`. The risk here is that global atomicity is not guaranteed and the intent of this experiment is to highlight the cost associated with running a coordinated transaction. The X-axis represents a wide range of workloads. The response time is reported in milliseconds on the Y-axis. From the figure, it can be seen that 2PC coordination adds at least 25% more to the response time across the breadth of the workloads. We will analyze the components that cause this performance overhead in Figure 5.

Figure 5 shows the time taken by the individual components of the 2PC coordinated transactions shown in Figure 4. The components of a distributed transaction are the time taken by the services to modify the databases and

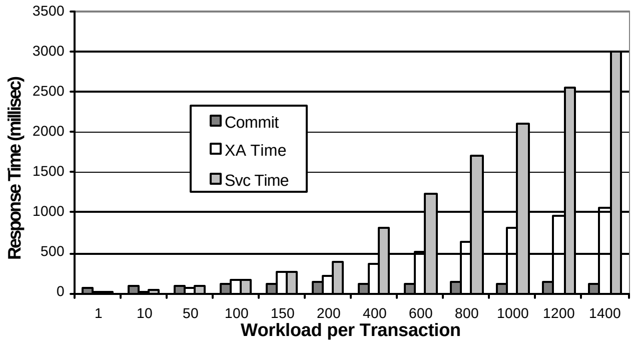


Figure 5: Distributed Transaction Components

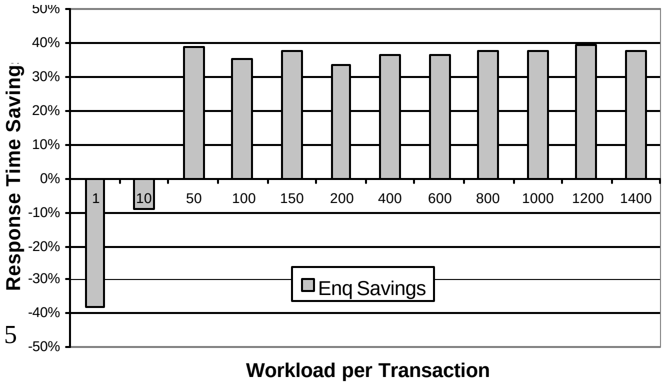
the total time to coordinate the two sub-transactions. The coordination component can be further broken down into the time taken associated with XA and the time taken to execute the 2PC algorithm. The 2PC execution time was obtained by instrumenting before and after the *commit* call of the distributed transaction and includes the local commit costs at the two DBMSs.

While it is impossible to isolate the XA and 2PC components, we used the following method to get to a very close approximation. The time taken by a coordinated transaction includes the total service time, two local commit times, the extra time taken due to the usage of the XA interfaces by the services, and the 2PC algorithm execution time. Additionally, we instrumented the services to get the time associated with sub-transactions' service times at each DBMS when the uncoordinated "transactions" of Figure 4 were executed. We obtained the XA time by using the following formula: Total time of 2PC coordinated transaction - Total service time - 2PC time.

From Figure 5, as one would expect the time associated with the 2PC algorithm execution is a constant regardless of the workload. The XA times on the other hand range all the way from 10% of the total transaction time for very small workloads to over 30% of the total transaction time for larger workloads. For a 10 insert transaction, the service times total up to approximately 21%, the XA time approximately 14%, and, the 2PC time approximately 65% of the total response time of the transaction. For a 100 insert transaction, the percentages are 37%, 37% and 26% for service times, XA time, and 2PC time, respectively. The percentages for a 1000 insert transaction are 69%, 27% and 4%, respectively. It can also be observed that the XA costs increase as the number of SQL statements increase (shown by the magnitude of the XA times in Figure 5 for larger workloads).

Figures 4 and 5 can explain why many COTS software vendors are reluctant to include transaction management components in their products. While the transaction management components are required for maintaining data consistency in a multi-COTS software integrated system, it makes individual COTS software performance look bad. Additionally, the COTS vendors try to follow SQL standards (SQL92 for now) when interacting with a database so that their product can function with most popular commercial databases for a wider market appeal. The effect though is that there is now more SQL traffic between the COTS software and the DBMS (through the XA interface) and the performance degradation is further amplified (as shown in Figure 5). A further detrimental effect of following current SQL standards is that the COTS software is unable to take advantage of optimizations that may be available from individual DBMS products such as special indices, SQL hints, etc.

If the application can tolerate a time lag between their sub-transactions and conflicting transactions are minimal, asynchronous transaction mechanisms are a legitimate alternative to 2PC coordinated transactions across COTS software [DRD99]. Figure 6 shows the percentage comparison of the response times between 2PC coordinated transactions between the two databases *vs.* the enqueue time into a persistent queue based asynchronous transactions for various workloads. The enqueue time is the perceived user response time. It includes the time to insert records into the DBMS and to put the data in the queue as a single 2PC transaction between one database and the queue. The messages in the queue include an overhead for message header. For smaller workloads (both single insert and 10 inserts fit on one disk page), the overhead associated with queue management makes 2PC coordinated transactions a better performance option. The percentage overhead for a 10 insert transaction is less than that for single insert transactions, because the message header overhead becomes a smaller percentage of the total message. For all larger loads, the response time associated with the queue is approximately 35-40% better than the equivalent 2PC transactions. It is our observation from



experience that due to the nature of "business transactions" and due to multiple interactions between the COTS software and the databases in the context of one "business transaction", the workloads associated with transactions in COTS based environments tend to be on the larger side. As mentioned before we hosted the queue and the databases on different machines so that they do not compete on I/O in the NT servers. A side effect of this is that all the enqueue operations between the queue and the database now use the network to execute the 2PC mechanism. In larger production strength machines, the queues and the databases will mostly likely be in the same computer though they are likely to be located on different I/O sub-systems. So the enqueue savings shown in Figure 6 are likely to be higher in production systems than our results would indicate.

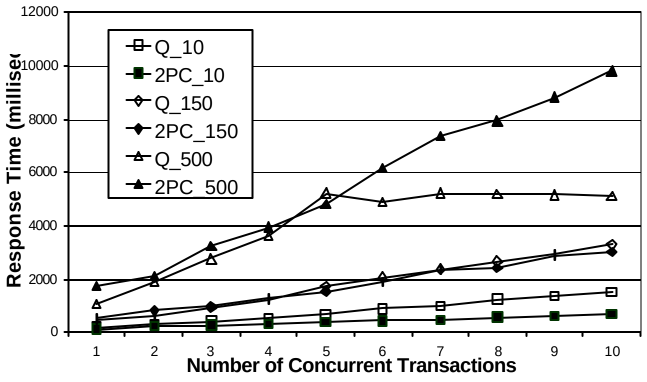


Figure 7: Response Time - 2PC Vs. Queue

Figure 7 compares the response time of 2PC coordinated transactions vs. queue based asynchronous transactions similar to that described in the previous paragraph. Unlike Figure 6, Figure 7 compares the two approaches when concurrent transactions (varied from 1-10) are executed. We ran experiments for several workloads but only show the results from representative of small (10 inserts/transaction), medium (150 inserts/transaction) and large (500 inserts/transaction) workloads. As the number of concurrent transactions increases, the response time of each transaction degrades as anticipated for both approaches. In general for smaller sized workloads, 2PC coordinated transactions returned a better response time. It may also be observed that the queue message overhead is quite significant for lower workloads and make them unattractive in these workload ranges. For the largest workload in Figure 7, the queue response time flattens after 5 concurrent transactions. This is because the queue gets saturated with the larger messages and the service that performs database inserts becomes the bottleneck. We anticipate that for small and medium size workloads, if we further increase the number of concurrent transactions, the queue response time will flatten out but the 2PC response time will continue to grow linearly. The response time of the queue based transactions are also negatively impacted by the experiment configuration issue (remote database and queue) that was discussed earlier and the response time will be better on production strength machines.

Figure 8 compares the throughputs between 2PC coordinated transactions and queue based transactions for 1-10 concurrent clients. We measured throughput as the number of successful transactions executed within a fixed time interval. For the queue based transactions, we initiated the enqueue operations (1-10 processes) and after a brief time lag, we initiated the dequeue operations (1-10 processes) to avoid starvation of the dequeue processes. Similar to the enqueue operations of Figure 6 and 7, the dequeue operations consists of dequeuing a message from the queue, inserting the workload number of records into a database, and then performing a distributed 2PC between the queue and the local database. For the same workloads of Figure 7, as anticipated the throughputs were inversely proportional to the size of the workloads. In general, the throughputs associated with the queue was better for all workloads except for very small workloads. The throughputs of the queue based transactions are flat in Figure 8 because the queue is kept busy and close to saturation. One would expect the throughputs to drop off after the process scheduler on the machine becomes overwhelmed with the larger number of concurrent clients. We did not reach this limit even when we ran 15 concurrent clients (not shown in figures) and began to reach memory and other machine limitations for larger number of concurrent clients.

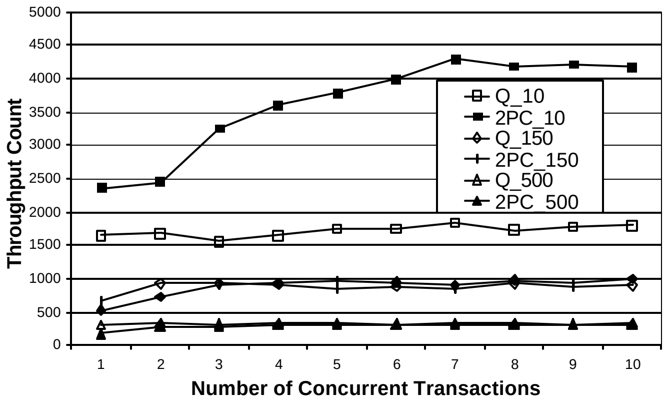


Figure 8: Throughput - 2PC vs Q

4 Discussions

This paper examines the impacts of a multi-tiered software architecture on transaction management. Specifically, in any large scale enterprise system that consists of multiple COTS software products, the responsibility of providing interfaces to support global transaction coordination is assumed by COTS software products, instead of by the DBMSs. This requirement, however, is not realized since supporting heterogeneous global transaction coordination is prohibitive in both cost and performance to enterprises and to COTS software vendors alike. Enterprises may choose not to use any global transaction management mechanisms to enforce data consistency, but rather to employ business processes that promote disciplined use of the system by its users through process restrictions. Our experience working on one of the largest scale distributed computing environments [CSTG96] combined with empirical study described in this paper support this argument.

Asynchronous mechanisms are quite promising as an alternative to synchronous mechanism to improve performance. However, there are several open issues [DRD99]:

- It is trivial to use asynchronous mechanisms if there is a one to one uni-directional interaction, for instance, an ATM machine and the application server in a bank. However, many global transactions involve multiple COTS software products. It is not trivial to resolve conflicts if there is bi-directional interaction which may result in non-serializable execution schedules.
- In addition, there can be multiple asynchronous mechanisms linking to a server, for instance, multiple persistent queues. It is not trivial to order the messages in these queues if these messages are inter-related. Using a single queue may not be a solution since it will be easily overloaded.

This study also led to the observation that the database vendor community is moving in the opposite direction of what is required by industrial multi-tiered distributed computing architectures. Most COTS software that provide packaged enterprise computing services opt only to use standard database functions, such as in SQL92, in order to become generic and flexible in the way they use the databases they encapsulate. However, there is a trend among database vendors to pack more proprietary functionality, such as intelligent query optimizers and vertical functionalities in the form of cartridges, blades, etc., into their products, many of which go unused in multi-tiered architectures. Instead, the end-user needs various functions, such as an external global transaction manager and COTS-level resource management, to be implemented again at different layers of the architecture. The resulting generic COTS software products, the middleware which implements the integration, and value-added databases give a resource intensive system, with a large foot-print and redundant functions, some of which are either not used or not needed at the DBMS layer. It would appear the directions taken by the COTS software vendors and the database vendors are at odds with the needs and requirements of enterprises, such as ours, which use these systems.

References

[BEA96] BEA Systems., Tuxedo Whitepapers, 1996, www.beasys.com/products/tuxedo/wps/index.htm

[BHG87] Bernstein, P., Hadzilacos, V., and Goodman, N., Concurrency Control and Recovery in Databases, Addison-Wesley, 1987.

[BRE90] Breitbart, Y., *Multidatabase Interoperability*, Sigmod Record, September, 1990.

[COR98] CORBA Success Stories, 1998. www.corba.org/

[CSTG96] Cleland, V., Sholberg, J., Tockey, S., and Gordon, J.P., *Boeing Commercial Airplane Group's Application Integration Strategy*, www-irl.iona.com/clientpart/clients/boeing2.html

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

[ION97] Iona Technologies, *The Object Transaction Service*, Whitepaper, January 1997.

[DRD99] Do, L., Ram, P., and Drew, P., *The Need for Distributed Asynchronous Transactions*, Proc. of SIGMOD 99. Also in Boeing Tech. Report SSG-TECH 98-028.

[MIC98] Microsoft Distributed Component Object Model, 1998, www.microsoft.com/com/dcom.asp

[MTS98] Microsoft Transaction Server on-line articles, 1998. www.microsoft.com/com/mts.asp

[OMG98] OMG CORBA 2.2 Specification, February, 1998. www.omg.org/library/c2indx.html

[OSF98] The Open Group publications, 1998, www.osf.org/publications/catalog/web.htm

[RDPZ99] Ram, P., Do, L., Drew, P., and Zhou, T., *Building Distributed Transactions using the Object Transaction Service,* submitted for publication. Also in Boeing Tech. Report SSG-TECH 98-017.

[TRA97] Transarc Corporation, Encina 2.5 Doc., 1997. www.transarc.com/Library/documentation/encina_doc.html

[XA91] Distributed Transaction Processing: The XA Specification, X/Open Company Ltd., 1991.