Later Can Be Better

Increasing Concurrency of Distributed Transactions

Weihai Yu

Department of Computer Science
University of Tromsø
Norway
weihai@cs.uit.no

Abstract

Distributed transactions have poor scalability, both in terms of the amount of concurrent transactions, and the number of sites a distributed transaction may involve. One of the major factors that limit distributed transaction scalability is that every site that a transaction involves has to hold data resources until the termination of the entire transaction, i.e., every sub-transaction's lifetime of resource consumption is bounded to the slowest sub-transaction. There are several solutions to this problem. The basic idea of these solutions is to release data resources as early as possible before the termination of the entire distributed transaction. These solutions either suffer from cascading aborts or require knowledge of application business details such as the use of compensation. In this paper, we present a totally different approach that delays altruistically the execution of fast sub-transactions. We will show that this approach increases concurrency of the entire distributed system without sacrificing transaction response time. Furthermore, this approach assumes no more than strict two-phase locking and requires very little knowledge of specific application business process.

1 Introduction

Distributed transactions are essential for large-scale distributed applications like e-commerce, enterprise applications, and so on. However, distributed transactions are limited by their scalability, mainly due to sharing of data resources. When a transaction is updating a piece of data, other transactions are excluded from accessing the same data. Typically, the exclusion of access will last until the updating transaction terminates. In fact, nearly all commercial transaction processing systems and transaction processing standards use strict two-phase locking [8][11][15], i.e., locks acquired by a transaction are held until its termination. When transactions get long or involve many sites, they tend to hold data resources exclusively for long duration.

However, holding data resources such long may be unnecessary. A lot of research has focused on how data updated by a transaction can be released early such that other transactions will not be unnecessarily blocked (many examples can be found in [6]). Releasing resources early, however, is not without problems. One difficult problem is that a transaction that releases its updated data might be aborted later. Concurrency control mechanisms that allow (somewhat) early release of data resources, like non-strict two-phase locking [3] and altruistic locking [13], suffer from the problem of cascading aborts. Moreover, non-strict two-phase locking requires notification of the

start of the shrinking phase, and is thus not particularly suitable in a distributed environment. Another commonly applied approach to remedy the inconsistency due to early release is the use of compensation [7]. Compensation is not a silver bullet, either. Some operations cannot be compensated. Even when all operations can be compensated, applications must provide compensation transactions, making the development of the application more complicated.

We present a totally different approach to reducing blocking without incurring the problem of early release of data resources. More specifically, our approach

- reduces unnecessary resource holding, and therefore increases system concurrency,
- does not release data resources before transaction termination, and therefore does not lead to cascading aborts and does not require compensation,
- ensures transaction response time at least as good as with strict two-phase locking, and
- is likely to be supported by middleware without application involvement.

This paper is organized as follows. Section 2 presents the intuition behind our approach. The key idea is to delay altruistically the execution of fast sub-transactions such that no sub-transaction finishes local processing unnecessarily earlier than other sub-transactions. Section 3 presents the execution model of distributed transactions for further discussions. Section 4 discusses how altruistic delays can be determined in the transaction execution model. Section 5 argues that altruistic delays can also be determined in a generalized transaction execution model. Section 6 discusses how to control altruistic delays in practice. Section 7 presents our plan for future research. Section 8 discusses some related work. Finally Section 9 summarizes the contributions of this work.

2 Approach Overview

Figure 1 illustrates the timing of a typical execution of a transaction distributed at processes 1, 2, 3 and 4 as sub-transactions. The execution of each sub-transaction consists of the following stages:

1. Start of the sub-transaction. A request is delivered to the process on behalf of the transaction. The necessary initiation and enlistment is done at this stage.
2. Acquisition of data resources. Locks are acquired on the data the sub-transaction is going to access.
3. Data processing. The data on which the sub-transaction has acquired locks are read and updated.
4. Unnecessary holding of data resources. After processing the data, the sub-transaction has to hold the locks until the termination of the transaction, i.e., until it is notified of the outcome of the transaction.
5. Release of data resources. Finally the locks are released upon termination of the transaction.

As mentioned in the introduction, one major factor that limits the scalability of distributed transactions is stage 4 above. That is, sub-transactions have to hold data resources unnecessary after having accessed them. The duration of this unnecessary

holding can be long and is generally not decidable locally at the sites of the sub-transactions.

In our approach, we introduce *altruistic delays* to the executions of sub-transactions. A sub-transaction acquires locks on data resources only after the altruistic delay. Ideally, by executing local data processing after the altruistic delays, the sub-transactions do not hold data resources unnecessarily (Figure 2). In practice, however, it is impossible to totally eliminate unnecessary holding of data resources. Our goal is therefore to minimize, or at least significantly reduce, unnecessary holding of data resources.

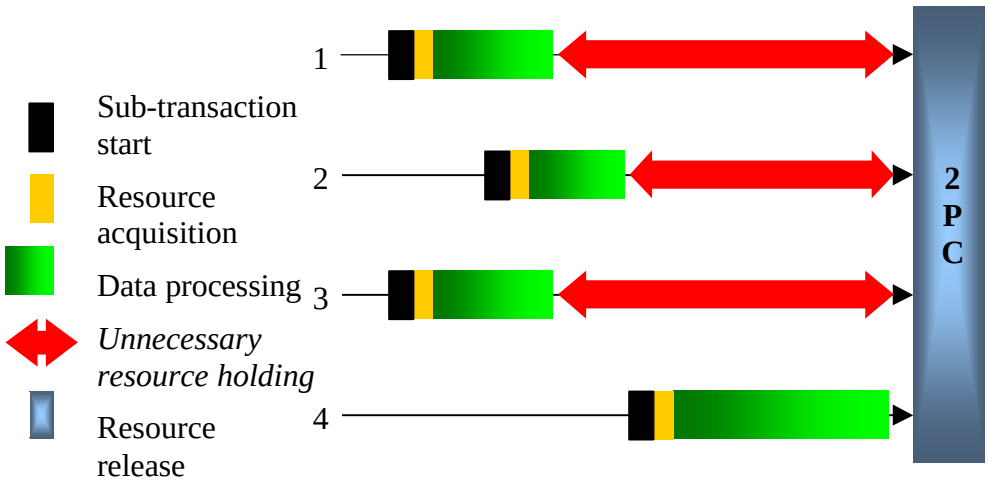

Figure 1. A typical execution of a distributed transaction with strict two-phase locking

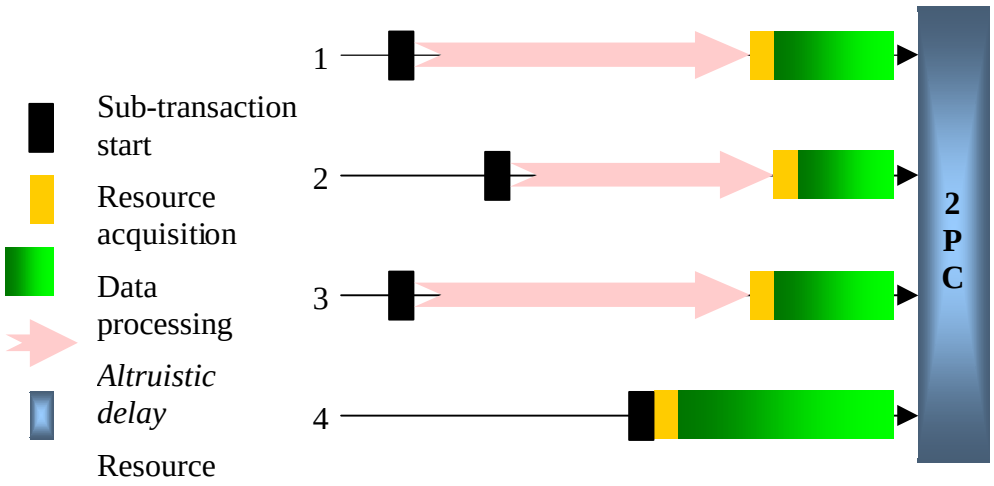

Figure 2. An ideal execution of a distributed transaction with strict two-phase locking

The remaining question is: how can we determine the altruistic delays? We address this issue in three steps:

- We start with a primitive transaction execution model where the necessary timing of transaction invocation and execution is known. This execution model

is defined in Section 3. We will show in Section 4 that in this model altruistic delays of sub-transactions can be determined locally at their parent nodes.

- Next, we will show in Section 5 that the result in the primitive transaction execution model applies also to a generalized execution model where the necessary timing is still known.
- Our ultimate goal is to improve performance of distributed transaction processing in practice where the timing is generally unknown in advance. This will be our future work. In Section 6, we will discuss some possible approaches to achieving this goal.

3 An execution model of distributed transactions

We model execution of distributed transactions in a multi-tier client-server architecture, which typically consists of a presentation tier, a business process tier, a data access tier and a database tier. A transaction is typically started at a process of some tier, say near or at the presentation tier. A transaction at a process may invoke one or more sub-transactions at other processes, possibly at the next tier. Here a process is loosely defined as an address space. The only way for processes to share information is via message sending. Processes may run on different machines. The invocations of sub-transactions form a tree (transaction invocation tree). In the tree, the nodes are sub-transactions at different processes; an invoking process is the parent of the invoked processes; the root is the process where the transaction is started. A distributed transaction is terminated via an atomic commitment protocol, the most widely used of which is the two-phase commitment protocol. In an atomic commitment protocol, a process is also called a participant. Usually, the root of the transaction invocation tree is also the coordinator of the commitment protocol, and the voting and the decision notifications (commit or abort) are communicated between the participants and the coordinator through the invocation tree.

The execution of a distributed transaction T is modeled as a tree:

$T = (N, E, exec, inv, res)$, where

- $N = \{0, 1, 2, \ldots, n\}$ is the set of nodes and 0 is the root,
- $E \subseteq N \times N$ is the set of edges,
- $exec$: $N \rightarrow$ Reals, is the time needed to execute sub-transactions locally at nodes,
- inv: $E \rightarrow$ Reals, is the time needed to invoke sub-transactions, and
- res: $E \rightarrow$ Reals, is the time needed to send replies.

Figure 3 illustrates a transaction as a tree. Every node of the tree is a sub-transaction at a process. Suppose there is no re-entrant invocation at the same process by the same transaction. Thus node i denotes both process i and the sub-transaction at the process. Assume also that there is no message other than those for requests, replies and atomic commitment processing, so executions at different nodes are not synchronized beyond these. An edge from parent node i to child node j indicates an invocation from process i to process j. $exec(i)$ is the time needed to execute sub-transaction locally at process i. $inv(i, j)$ is the time needed to invoke from process i to process j. $res(i, j)$ is the corresponding reply from j to i. Note usually $inv(i, j) \neq res(i, j)$, because the management operations associated with invocations and replies are quite different.

Associated with invocations are operations like authentication, authorization, transaction enlistment, initiation, etc., whereas with replies (implicit yes-votes, explained shortly in this sub-section), force-writing of prepare log records etc..

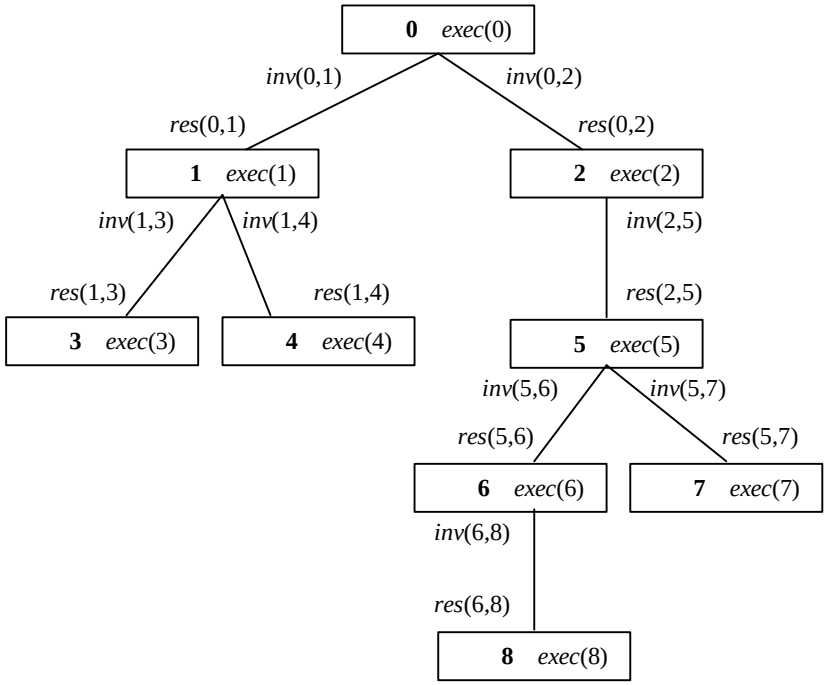

Figure 3. Distributed transaction as a tree

For an intermediate node *i*, the local sub-transaction is executed in parallel with the child sub-transactions. Note that this does not exclude the cases where the local sub-transaction is executed *before* or *after* the child sub-transactions. In those case, the local execution time is included in *inv*(*p*,*i*) (the *before* case) or *res*(*p*,*i*) (the *after* case) where *p* is the parent of *i*, and *exec*(*i*) is simply 0.

The "non re-entrant invocation" assumption is not unrealistic. After all, distributed applications should be designed to avoid a component invocating the same remote component multiple times (see for example [2]). An exception can occur at the data access tier where a component invokes a database multiple times in the same transaction via the same database connection (stored procedures are a way of avoiding multiple invocations of this kind). In this case, we can model the entire data access tier as a single node and our non re-entrance assumption still holds. In Section 5, we will see that even the non re-entrance assumption is not necessary.

In our transaction execution model, we assume strict two-phase locking and the two-phase commitment protocol [3], which are used in nearly all transaction processing products and standards. The root of the transaction tree is the coordinator of the two-phase commitment protocol. We assume the *implicit yes-vote* optimization of the two-phase commitment protocol [14], because it can be applied more directly in the subsequent discussions. A leaf node generates an implicit yes-vote right after its execution of all the local operations of the transaction. Note in our model there is no re-entrant invocation to the same process. In general, however, the last successful return from the requests of the same transaction is regarded as an implicit yes-vote. A non-leaf node generates an implicit yes-vote when it has received implicit yes-votes from all its children and when it has finished its local execution. So the implicit yes-votes are propagated from the child nodes to their parents and recursively upward to the root.

Ideally, every implicit yes-vote should arrive at the root at the same time, so no node is unnecessarily holding data resources while the root is waiting for the implicit yes-vote yet to be generated or delivered from some other node.

4 Determining altruistic delays

First, we define the *height* h_i of node *i* as the time interval between the process *i* receives a request and it is ready to reply with an implicit yes-vote:

$$h_i = \max\ (exec(i), \max_{(j \text{ is a child of } i)}\ (inv(i, j) + h_j + res(i, j)))$$

That is, after receiving the request, it concurrently starts local execution and sends requests to all subsequent processes. It is ready for replying with implicit yes-vote when it finishes local execution and receives implicit yes-votes from all child sub-transactions.

Ideally for every non-leaf node, *i*, the implicit yes-votes from all its children arrive exactly at the moment the local execution finishes. The delay *d*(*j*) of its child *j* and delay of its own local execution *d_local*(*i*) can be assigned such that

$$inv(i, j) + d(j) + h_j + res(i, j) = d\_local(i) + exec(i) = h_i$$

Note that the altruistic delay at node, *i*, consists of two parts:

- *d*(*i*), delay all activities, including local execution and invocations to the children, and
- *d_local*(*i*), delay of local execution in addition to *d*(*i*).

Now the altruistic delays of all nodes can be obtained, starting from the root.

```
main()
{
     d[0] ← 0;
     assign_delay_for_children(0);
}
assign_delay_for_children(i)
{
     d_local[i] = h_i – exec[i];
     if "there is no child"
          return;
     for every child j
     {
          d[j] ← h_i – (inv[i,j] + h_j + res[i,j]);
          assign_delay_for_children(j);
     }
}
```

Figure 4 shows the executions of the distributed transaction in Figure 3, with and without altruistic delays, given some hypothetical values of *inv()*, *exec()*, and *res()*. Notice that no altruistic delay is introduces at node 8, due to the slowest path 0→2→5→6→8. At all other nodes (with the exception of node 2), the time lengths for resource blocking are significantly reduced with the introduction of altruistic delays. At node 2, *exec*(2) is 0, whereas *inv*(1,2) is quite long, a case in which local sub-transaction

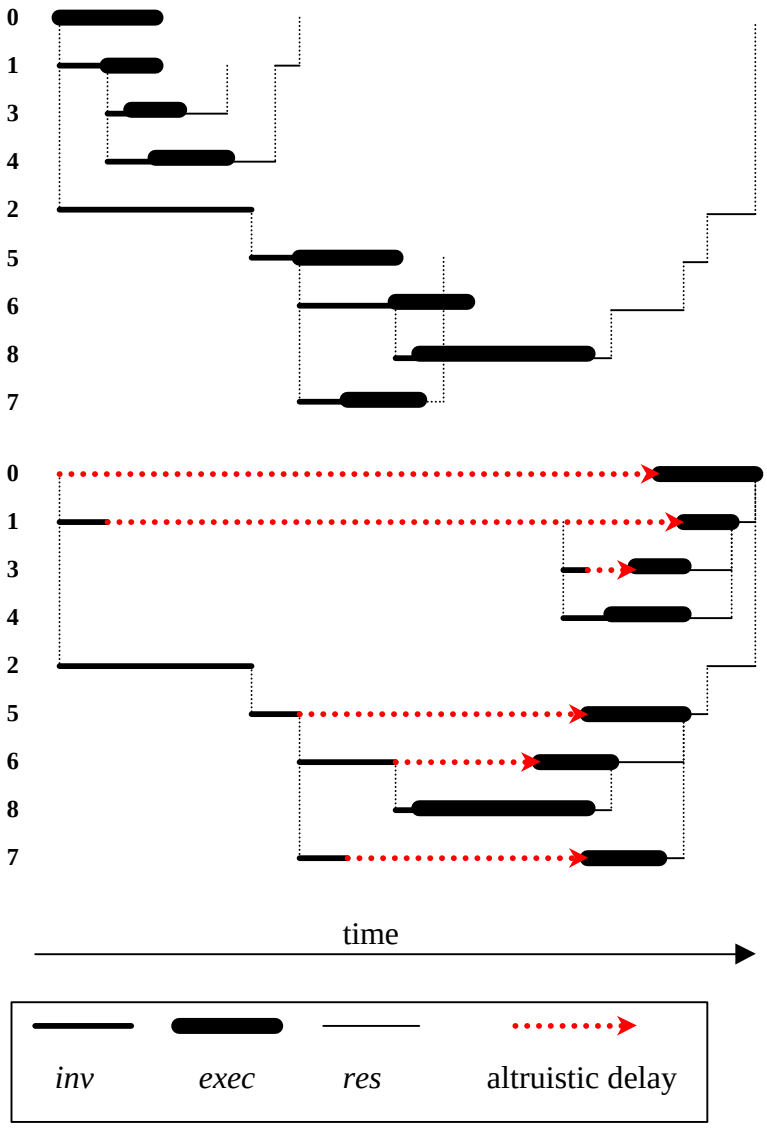

Figure 4. Executions with and without altruistic delays

is executed before the child sub-transactions. Like at other nodes, at node 2, locks acquired during local execution must be held until the termination of the entire distributed transaction.

5 Generalization of the execution model

Our transaction execution model presented in Section 3 can be generalized. In the generalized model, a node can be re-entrant and the execution of some children may be dependent on the execution of some other children of the same parent.

In Figure 5, execution of *a* consists of a_1, a_2, a_3 and a_4. Execution of *b* is independent of its parent *a* and other children of *a*. Execution of *c* is dependent on a_1. Execution of *d* is dependent on both a_2 (which is dependent on *c*) and sibling *c*. *a* is re-entrant with a_3 executing the invocation from *d*. Finally, the execution a_4 is dependent on the execution of *d*.

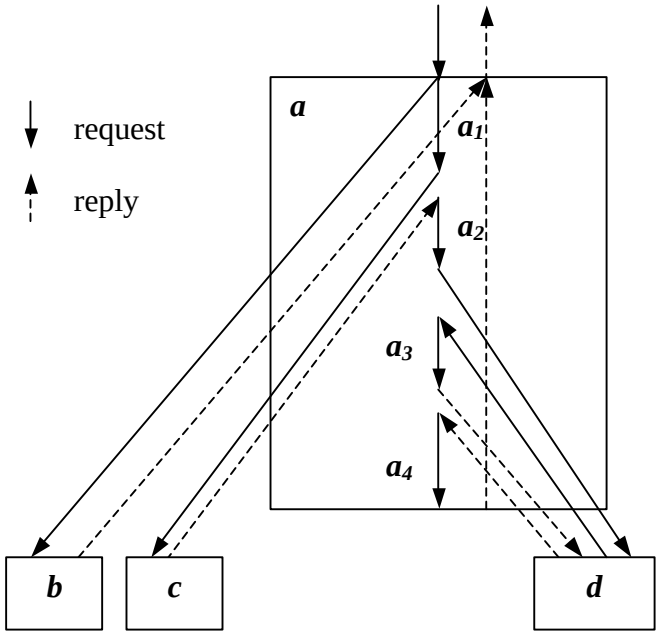

Figure 5. Node in the generalized execution model

Execution dependencies among child transactions introduce some complexity in determining the altruistic delays. For example, delaying at *c* will postpone the execution of *d*, which will incur even more unnecessary delay of the entire transaction. One way to model this is to regard the entire $A = (a_1 \rightarrow c \rightarrow a_2 \rightarrow d_1 \rightarrow a_3 \rightarrow d_2 \rightarrow a_4)$ as a single sequential execution in parallel with *b*. Altruistic delays of *b* and *A* are thus decidable at *a* using the algorithm in Section 4.

In essence, altruistic delays should only be considered at the forks of concurrent execution branches. Sequential executions, albeit distributed over different nodes, will not benefit from the introduction of altruistic delays.

6 Controlling altruistic delays in practice

In reality, local execution time and message delays are non-deterministic and unknown in advance. Non-determinism is a combined effect of factors like workload variation, race conditions, run-time resource scheduling, network protocol queue lengths, paging, caching, query spaces, program branching, etc. Controlling any of these factors is a major issue of research in areas like QoS management.

We propose here two possible approaches to control altruistic delays in practice:

- Control based on static analysis. Here timing of different components is obtained based on a white-box analysis of the application. This approach is particularly suitable for tuning benchmarks or applications whose business process structure is pretty static.
- Control based on dynamic analysis. Here altruistic delays are dynamically determined based on analysis on statistics of previous executions of transactions. This can be done either by the application or by the middleware.

Next, we briefly discuss these different approaches.

6.1 Control based on static analysis

In this approach, the application execution without altruistic delay is first profiled. Profiling at components that create concurrent sub-transactions is particularly important, because as shown in Section 4, altruistic delays can be determined locally at parent nodes. If the differences of response time of sub-transactions are stable and considerable, the slowest sub-transaction is the basis for determining the altruistic delays of the other sibling sub-transactions. Care must be taken of whether the slowest sub-transaction is dependent on some other sibling sub-transaction (as in the generalized execution model).

6.2 Control based on dynamic analysis

This approach can be implemented either directly by the application or by the supporting middleware.

Implementation by the application is similar to the approach based on static analysis. Statistics on response time is taken at components that create concurrent sub-transactions. Again, if the differences of response time of sub-transactions are stable and considerable, executions of sub-transactions are aligned with the slowest sub-transaction.

Ultimately, control of altruistic delays should be supported by middleware, whose main task is to support large-scale distributed applications.

Basically, analysis can be done by interceptors on a per remote invocation basis. In transactional component middleware such as EJB and COM+, the run-time such as a container intercepts every incoming and outgoing request.

For example, the middleware run-time records in the thread-specific transaction contexts (such as the context objects in COM+) the timestamp of the sending of every request and the reception of every reply. In this way, the recording is thread-specific and need not be synchronized with other threads. The recorded data is flushed to a shared data structure during component life-cycle operations (such as returning to object pool, deactivation etc.). A low-priority thread periodically reads the shared data structure and adjusts altruistic delays using certain forecasting methods.

One of the difficulties with middleware support is the lack of knowledge of the structure of application business process. Particularly useful is the knowledge of execution dependencies. Hopefully, detailed statistic analysis will uncover the dependencies. For example, a clear indication of some dependency between two child sub-transactions of the same parent is that a request message to one child is always sent after the parent receives a reply from the other child. This is one of the issues we will work on in the future.

7 Future Work

We are now evaluating the benefits of our approach with static analysis of some typical distributed applications.

Our ultimate goal is middleware support for altruistic delays of distributed transactions in large-scale applications. This work is part of the Arctic Beans project [1]. The primary aim of Arctic Beans is to provide a more open and flexible enterprise component technology, with intrinsic support for configurability, re-configurability and evolution, based on the principle of reflection [4]. Enterprise components are

components at server sites running within so called *containers* which provide support for complicated mechanisms like resource management (threads, memory, sessions etc.), security and transactions. The project focuses on the security and transaction services and how to adapt these services according to the application needs and its environment. We believe a container is the right place to support control of altruistic delays.

We will also study if altruistic delays can be applied in adaptive applications. For example, if one sub-transaction is often dominantly long, due to weak connections or even occasional disconnections, executions of other sub-transactions should be aligned with this long sub-transaction.

8 Related Work

Most researches that aim at reducing unnecessary resource holding is based on early release of data resources. The key problem is to maintain database consistency when the transaction that has already released some of its updates finally aborts.

With non-strict two-phase locking, locks can be released during the shrinking phase [3]. Altruistic locking allows even earlier release of locks if more knowledge of the data access patterns is available [13]. Serializability of transaction executions is ensured. The well-known problem with these approaches is cascading aborts, which can be even more costly to deal with than unnecessary resource holding, particularly in large-scale systems. They are generally unsuitable for distributed transactions because of the extra notification, such as of the termination of the expanding phase, required among involved sites. Moreover, independence of data accesses among different sites is not fully explored.

With the use of compensation [7], the early released effects of a transaction can be logically undone if the transaction finally aborts. One problem with this approach is that not every operation is compensatable. Some compensatable operations still have some effects on end-users, such as charge of cancellation (of reservations) fee. In general, implementing a compensation transaction for every (sub-) transaction is not an easy task.

With dynamic re-structuring [9], a long transaction can be split into multiple smaller ones. Some of them can commit early. There may or may not be cascading aborts, depending on the dependencies among the new split transactions. Again, detailed knowledge of application business process is required.

Both compensation and dynamic re-structuring can be used to deal with transactions with long sequential executions, whereas our approach cannot. All these methods can be used jointly in particular applications.

We use the implicit yes-vote optimization [14] of the two-phase commitment protocol to reason about our approach. There are several improvements to the implicit yes-vote optimization (some of them can be found in [5]). One noticeable improvement is called the dynamic two-phase commitment protocol (D2PC) [12]. Actually, D2PC is not limited to implicit yes-votes, though the gain of D2PC in implicit yes-votes is more significant because of the difference in time yes-votes are generated. The basic idea is that propagation of yes-votes does not stop at the root of the tree while some slow sub-transaction is still busy processing. The propagation will go on toward the slowest node (which will become the new coordinator). With D2PC, the total response time is improved, therefore also entire system concurrency. While D2PC focuses on

"smoothing out" the effect of the slowest sub-transaction during commitment processing, we consider the combined effect of execution and message delays. With D2PC, when one sub-transaction is considerably longer than the others (say all other yes-votes have already "piled up" at this slowest node), the others still have to hold data resources unnecessarily (possibly considerably long). There is also an additional cost associated with transfer of coordinator in D2PC. It would be interesting to compare quantitatively the D2PC with our approach.

As a final remark, the problem our work seeks to address is different from that of scheduling workflow tasks on a set of resources (such as the job-shop scheduling [9]), where the set of workflows with (more complex) inter-task dependencies is known in advance. The goal with task scheduling is either to minimize total execution time of workflows or to meet deadlines of workflows. In our work, although the inter-transaction dependencies could be simple, the total set of transactions is not known in advance. Nor is our goal to meet deadlines or to minimize total execution time.

9 Conclusion

One major factor that limits the scalability of distributed transactions is that all sub-transactions have to hold data resources until the slowest sub-transaction finishes. Holding of data resources this long can be unnecessary, because some sub-transaction may have already finished processing its local data. In contrast with the approaches that release data resources before the termination of the entire distributed transaction, we introduce altruistic delays to the executions of sub-transactions. We show in transaction execution models where the necessary timing is known in advance, with the introduction of proper altruistic delays, no sub-transaction holds data resources unnecessarily after local processing. Furthermore, we show some nice properties of our approach: concurrency of the system is enhanced without sacrificing transaction response time; altruistic delays can be determined locally at parent sub-transactions. We also discussed how altruistic delays could be controlled in practice. It seems that altruistic delays can be supported by middleware without specific knowledge about application business structure. How this can be realized will be part of our future research work.

10 References

[1] Anders A., G. S. Blair, V. Goebel, R. Karlsen, T. Stabell-Kulø and W. Yu, "Arctic Beans: Configurable and Reconfigurable Enterprise Component Architectures", Middleware 2001 Work-in-Progress paper, *IEEE Distributed Systems Online*, 2(7), On-line at: http://dsonline.computer.org/0107/features/and0107.htm, 2001.

[2] Alur, D., J. Crupi and D. Malks, *Core J2EE Patterns, Best Practice and Design Strategies*, Upper Saddle River, NJ: Printice Hall PTR, 2001.

[3] Bernstein, P. A., V. Hadzilacos and N. Goodman, *Concurrency Control and Recovery in Database Systems*, Reading, MA: Addison-Wesley, 1987.

[4] Blair, G., G. Coulson, P. Robin and M. Papathomas, "An Architecture for Next Generation Middleware", In *Proceedings of Middleware '98*, 1998.

[5] Chrysanthis, P. K., G. Samaras and Y. J. Al-Houmaily, "Recovery and Performance of Atomic Commit Processing in Distributed Database Systems", In Kumar V. and M. Hsu (Eds.), *Recovery Mechanisms in Database Systems*. Upper Saddle River, NJ: Printice Hall, pp. 370-416, 1998.

[6] Elmagarmid, A. (ed.), *Database Transaction Models for Advanced Applications*, Morgan Kaufman Publishers, 1992.

[7] Garcia-Molina, H. and K. Salem, "SAGAS", In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, 1987.

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

[9] Jain, A. S., and S. Meeran "Deterministic Job Shop Scheduling: Past, Present, Future", *European Journal of Operation Research*, Elsevier Science, Vol 113, pp 390-434, 1999.

[10] Kaiser, G. E., C. Pu, "Dynamic Restructuring of Transactions", In [6], pp. 265-295, 1992.

[11] Object Management Group, *CORBA Services Specification*, Chapter 10, Transaction Service Specification, December, 1998.

[12] Raz, Y., "The Dynamic Two Phase Commitment (D2PC) Protocol", In Gottlob G. and M. Y. Vardi (Eds.), *Proceedings of 5th International Conference on Database Theory*, LNCS 893, Springer, 1995

[13] Salem, K., H. Garcia-Molina and J. Shands, "Altruistic Locking", ACM Transactions on Database Systems, 19(1), pp. 117-165, 1994.

[14] Stonebraker, M., "Concurrency Control and Consistency of Multiple Copies of Data in Distributed INGRES", *IEEE Transactions on Software Engineering*, 5(3), pp 188-194, 1979.

[15] X/Open Company Ltd., *Distributed Transaction Processing: The XA Specification.* Document number XO/CAE/91/300, 1991.