Self-Deadlocks in Disparate Scientific Data Management Systems

Fragkiskos Pentaris and Yannis Ioannidis
Department of Informatics and Telecommunications,
University of Athens,
Panepistimioupolis, 15771, Athens, Greece
{frank,yannis}@di.uoa.gr

Abstract

In large statistical and scientific data management environments, where mediation architectures are used to integrate disparate and autonomous systems, a new problem — self-deadlock — may cause global transaction failures. In this short paper we briefly examine the reasons causing this problem and identify some algorithms for resolving it.

1. Introduction

Recently, the idea of using Wide Area Network(WAN) distributed architectures for constructing scientific data management systems (SDMSs) has gain a lot of supporters (e.g., the Grid). Figure 1 shows an example of such a system, consisting of a few network nodes. Each of them is an autonomous "black box" that may export/import a subset of its local data and resources to/from remote nodes. Using such a system, scientists can access any available distant information held at remote nodes, which, in turn, may *transparently* import/export this information from/to other ones.

We expect that data-consumers will not always be aware of the actual origin of the data they use or update. For instance, in Figure 1 a scientist at node A asks for some pieces of information from node B while updating some data at node C. Both of these nodes are actually importing their data from a single DBMS. Thus, node A is actually querying and updating information stored in the same DBMS node. Such kind of multi-path accesses, may exist for performance, security, policy, distributed design, or user-preference reasons, or even simply by mistake. However, local concurrency control (CC) mechanisms of remote nodes (e.g., the DBMS) will most likely disallow the use of two different, yet semantically correct, access paths, if they conflict with each other (e.g., they both access the same information and one of them leads to an update). We call this type of failure, caused by CC conflicts between local transactions participating in the same global one, a *self-deadlock*.

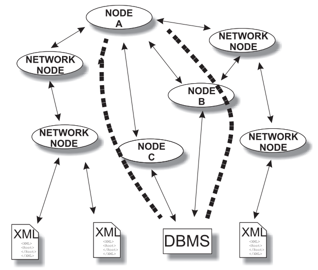

Figure 1. Distributed SDM.

Proposed distributed transaction models and CC algorithms (e.g., [2, 3]) cannot be used to solve the problem of self-deadlocks [4] and usually cause global transaction failures (see for example the recommendations in the IBM DB2 DataJoiner manual [1]) or lead to a livelock condition where distributed transactions are restarted and killed indefinitely [4]. In this short paper, we briefly examine the problem of self-deadlocks, present the conditions that must be satisfied for the problem to occur, identify some methods to resolve it and discuss some experimental results showing that under certain conditions, self-deadlock resolution is a feasible and beneficial task. Further details on self-deadlocks can be found in the long version of the paper [4].

2. Execution Environment

In this short paper we consider Internet-based, distributed SDMSs composed of many disparate nodes. The nodes persistently store information (*storage nodes*), and/or act as intelligent *read-write* "gateways" to information held by other nodes. Multi-path accesses to the same data on the same node are allowed, though, we do not require from nodes to be aware of the actual origin of the source data.

Local transactions (LT) are started either implicitly, when a remote node retrieves or updates data held in a storage node for the first time, or explicitly, when this is requested by the remote node. The initiating node is always the owner of the transaction, i.e. the only one that can commit its work. Global transactions (GT) are started whenever a query requires information residing in more than one remote storage nodes. We do not assume that a two-phase commit protocol is used, as Internet-exposed nodes and proprietary systems are usually not able or willing to participate in such protocols [5].

3. General Deadlocks

The lack of information on the locks that have been obtained by remote nodes participating in a GT leads to a self-deadlock condition. Generally, *a self-deadlock occurs iff an (update) query submitted to a node requires locks that are already granted to a different local transaction that is part of the same global transaction*. This is in contrast to traditional distributed deadlocks, which occur iff there is cyclical waiting of local transactions for locks granted to local transactions of different global transactions.

Since traditional deadlocks and self-deadlocks have many similarities, we have combined these two problems into a more general concept named *general deadlock* [4] using a hybrid wait-for-locks (WFL) graph. The nodes of this graph are the GTs and their respective LTs. A directed arc may connect either two LTs nodes that belong to different GTs, or a GT node to one of its LTs nodes. In the first case, the arc indicates that the source LT is blocked waiting for the target LT to release certain locks. The second type of arcs exists only on self-deadlock prone systems and indicates that the source GT is waiting for the specific target LT to finish its work.

4. Detection and Resolution of Self-Deadlocks

The identification of general deadlocks, is especially useful for ensuring that in the environments that we consider, all global transactions manage to successfully complete their processing. This is only possible when systems run avoidance or detection algorithms capable of discovering both types of deadlocks. Unfortunately, avoidance is not always possible unless someone knows before-hand all queries and data manipulation statements participating in GTs. Thus, we have limited ourselves in detecting and resolving self-deadlocks.

Both traditional deadlocks and self-deadlocks are detectable using a slightly modified nodes-chasing algorithm. However, we cannot modify any of the existing traditional deadlock resolution algorithms to make them capable of also resolving self-deadlocks. Therefore, in [4], we propose three new methods to resolve them, named *locks release*, *locks sharing* and *proxy execution*.

5. Overhead of Resolving Self-Deadlocks

Systems designers must carefully select the proper method for detecting and recovering from general-deadlocks. A pessimistic method will assume that all GT failures are due to self-deadlocks and immediately runs a self-deadlock detection and resolution algorithm. A more optimistic approach will simply ignore self-deadlocks (i.e., this is what existing systems currently do) and assume that GT failures are due to traditional deadlocks. The offending GTs will be retried for a number of times and if all retries fail, then they will be aborted and application-supplied alternative GTs will run. Finally, a hybrid approach may assume that all GT failures are due to self-deadlocks and yet, only run the detection algorithm. If this algorithm indicates that a self-deadlock has occurred, then the offending GT will be aborted and an application-supplied alternative GT will run (i.e., no self-deadlock resolution algorithm is run).

In [4], we provide analytical functions estimating the cost of each of the above approaches. Furthermore, we provide the results of a set of experiments proving our analytical results. These experiments show that the overheads of resolving self-deadlocks may be substantial and depend on the number of nodes, their interconnection topology, the network latency and the frequency of self-deadlocks. The experiments further prove that depending on the probability of self-deadlocks, the use of a hybrid or pessimistic approach for handing self-deadlocks may be beneficial to simply ignoring the problem of self-deadlocks (optimistic approach).

References

[1] I. Corp. *DB2 DataJoiner Version 2, Administration Supplement*, chapter 6, Distributed Unit of Work (DUOW) Transactions, pages 59–78. IBM Corp., 1998.

[2] V. Gligor and R. Popescu-Zeletin. *Distributed Data Sharing Systems*, chapter Concurrency control issues in distributed heterogeneous database management systems, pages 43–56. North Holland Publishing Company, 1985.

[3] G. Kappel, S. Rausch-Schott, W. Retschitzegger, and M. Sakkinen. Multi-parent subtransactions, covering the transactional needs of composite events. In *Proc. of Int. Workshop on (ATMA)*, Coa (India), Sept. 1996.

[4] F. Pentaris and Y. Ioannidis. Self-Deadlocks in Disparate Scientific Data Management Systems. Available at, `http://www.di.uoa.gr/~frank/ssdbml04.pdf`, 2004.

[5] A. Zhang, M. Nodine, and B. Bhargava. Global scheduling for flexible transactions in heteogenious distributed database systems. *IEEE TKDE*, 13:439–450, 2001.