Project no.004758

GORDA

Open Replication of Databases

Specific targeted Research Project

Software and Services

GORDA Wide-area protocols report

GORDA Deliverable D3.1

Due date of deliverable: 2006/03/31

Actual submission date: 2006/10/01

Start date of the project: 1 October 2004 Duration: 36 Months

University of Lugano

Contents

1 Scope

This document describes GORDA database replication protocols for wide area networks. Since the amount of information shared among users worldwide grows, replication protocols in wide area networks are more and more demanded. Unfortunately, scaling existing cluster based replication protocols to wide area network is troublesome. Some key points to obtain acceptable performance across wide area networks are to reduce communication overhead as much as possible, to overlap execution with message delay, and to relax consistency requirements.

Hereafter we present two protocols: in Section 2 we describe the Database State Machine [11] replication with weaker consistency criterion – the Snapshot Isolated Database State Machine; Section 3 presents WICE [7], a replication protocol targeted at multiple clusters interconnected by wide area network.

2 Snapshot Isolated Database State Machine

Of particular relevance for the performance of the Database State Machine (DBSM) replication [11] is its dependency on transaction readsets needed for certification. First, if the considered readset is larger than the set of tuples actually read by the transaction, spurious aborts may arise; if the readset does not contain the tuples actually read, then serializability may be compromised. Second, in the DBSM the size of readset may have a serious impact on the network bandwidth.

If weaker consistency criterion is considered (snapshot isolation), the protocol could be simplified. For example, to satisfy snapshot isolation, certification does not need to check for read-write conflicts and thus the transaction readsets are not required. Therefore the protocol can be simplified by not propagating the readsets and using a simpler certification test. Such an alternative has also a benign impact on the performance of DBSM, since it reduces the number of aborted transactions [8].

2.1 Readsets-free certification

The basic idea of the DBSM remains the same: read-only transactions are processed locally at some database replica; update transactions do not require any synchronization between replicas until commit time. During commit the outcome of update transactions execution is broadcast to all the replicas for certification. To ensure that each replica converges to the same state, each server has to reach the same decision when certifying transactions and guarantee that conflicting transactions are applied to the database in the same order. These requirements are enforced by an atomic broadcast primitive and a deterministic certification test. In contrast to the original DBSM, transactions are executed locally under the multi-version concurrency control (e.g., snapshot isolation) and when an update transaction requests a commit, only its updates and writesets are broadcast to the other sites. Certification checks whether the writesets of concurrent transactions intersect; if they do, the transaction is aborted. Transaction t_i passes certification at the replica site s_i if the following condition holds:

$$\left[\begin{array}{l} \forall t_j \text{ committed at } s_i : \\ t_j \rightarrow t_i \vee (writesets(t_j) \cap writesets(t_i) = \emptyset) \end{array}\right]$$

where $\rightarrow$ is a precedence relation between transactions t_i and t_j. If t_i and t_j execute on the same replica s_i, then $t_j \rightarrow t_i$ only if t_j enters the committing state at s_i before t_i enters the committing state at s_i. If t_i and t_j execute on different replicas s_i and s_j, respectivelly, then $t_j \rightarrow t_i$ only if t_j is committed at s_i before t_i enters the committing state at s_i [11].

2.2 Snapshot Isolated DBSM

Snapshot isolation (SI) is a multi-version concurrency control algorithm introduced in [6]. SI does not provide serializability, but is still attractive and used in commercial and open-source database engines, such as Oracle and PostgreSQL. Under SI a transaction t_i sees the database state produced by all the transactions that committed before t_i started. Thus, if t_i and t_j are concurrent, neither will see the effects of the other. According to the *first-committer-wins* rule, t_i will successfully commit only if no other concurrent transaction t_j that has already committed writes to data items that t_i intends to write.

Although specific workloads will not be serializable under SI, such cases seem to be rare in practice. Fairly complex transaction mixes, such as the TPC-C benchmark, are serializable under SI. Moreover, there are different ways to achieve serializability from SI [1, 2].

In Snapshot Isolated DBSM transactions executing in the same database replica are snapshot isolated. Are global transactions also snapshot isolated? It turns out that the answer to this question is yes. First, notice that any two concurrent transactions executing at different replicas are isolated from one another in the Snapshot Isolated DBSM: one transaction does not see any changes performed by the other (before commit). Second, the Snapshot Isolated DBSM's certification test provides the first-committer-wins behavior of SI since the first transaction to be delivered for certification commits and later transactions abort.

A similar approach to Snapshot Isolated DBSM is taken in [9]. The authors propose SI-Rep - a hybrid, update everywhere replication protocol. Transactions are first executed at a local database replica providing SI. At the end of the execution the updated records are extracted. After retrieving the writesets, SI-Rep performs a validation to check for

write/write conflicts with transactions that executed at other replicas and have already validated. If validation succeeds, the transaction commits at the local replica and the writeset is applied at the remote replicas in a lazy fashion. The proposed protocol is evaluated in a local area network.

3 The WICE protocol

WICE is a database replication protocol based on group communication that targets interconnected clusters [7]. In contrast with previous proposals, it uses a separate multicast group for each cluster and thus does not impose any additional requirements on group communication, easing implementation and deployment in a real setting. Nonetheless, the protocol ensures one-copy equivalence while allowing all sites to execute update transactions.

3.1 System model

The database is a collection of named data items which have values [10]. The combined values of the data items at any given moment is the database state. No assumptions are made on the granularity of data items.

A database site is modeled as a sequential process. In detail, the execution of each site is modeled as a sequence of steps that may change the site's state. Namely, the database state is manipulated by executing READ(x) and WRITE(x) steps, where x represents a database tuple. A transaction is a sequence of read and write operations followed by a COMMIT(t) or ABORT(t) operation. Each site contains a complete copy of the database and is responsible for ensuring local concurrency control.

The database sites communicate through a fully connected network. Both computation and communication are asynchronous. Sites may fail only by crashing and do not recover, thus stopping to execute database operations, or send or deliver further messages.

Database sites are organized in clusters. Within a cluster we assume a primary component group membership service that provides current and consistent views of the sites believed to be up [4]. The failure of an entire cluster is reliably detected at the other sites. Among sites within the same cluster, a group communication toolkit is available providing reliable point-to-point communication and FIFO uniform view-synchronous multicast [4]. Among clusters, messages are exchanged through a point-to-point FIFO reliable channel.

3.2 The protocol

The WICE protocol adopts an optimistic concurrency control policy. Transactions are executed optimistically at any site and then, just before commit, certified against concurrent transactions. WICE borrows from protocols such as Postgres-R [3] and DBSM [11] often called *certification based* protocols. These protocols share two fundamental characteristics: (1) each database site is assumed to store the whole database and transactions can be executed at any site, and (2) all update transactions are certified and, if valid, committed in the same order at all sites.

This total order allows the use of a simple optimistic certification procedure to provide one-copy serializability (1-SR). As discussed in [5] and [12], we can gain scalability in multi-version database systems if we instead apply one-copy snapshot isolation (1-SI). The certification procedure is the same, but read-operations are never considered as each transaction T is assumed to read from a *snapshot* defined by all committed transaction when T entered execution[6].

The fundamental difference between the three protocols is when and where the certification is performed. Both Postgres-R and DBSM use a total order broadcast primitive and certify each transaction once the totally ordered message is delivered. In Postgres-R, each transaction is certified at the site that executed it and the outcome of the certification is then sent to all the other sites. In the DBSM, the read and write sets of the transaction are sent to all sites allowing the certification to be carried at all sites avoiding the last communication step of Postgres-R.

WICE does not make use of a total order primitive, instead ordering is explicitly implemented by the protocol. In WICE, one of the sites plays the role of certifier, it totally orders and certifies all transactions. Each valid transaction is then broadcast together with its commit order and updates. This allows to leverage the knowledge about the system's topology and make optimizations that would not be possible otherwise.

In a nutshell, the protocol handles transactions as follows. Consider a system consisting of two clusters A and B. Each cluster has a designated delegate responsible for handling the communication with the other cluster. The delegate of cluster A, site s_2 is also responsible for certifying all executed transactions. When an update transaction T is submitted to site

s_1 (T's initiator), it is readily executed and sent to the certifier. If it succeeds, then the certifier propagates T's updates and commit order, both locally and to cluster's B delegate. The latter, in turn, propagates T locally. Once a delegate is certain that all sites in its cluster delivered T's data it acknowledges the fact to the other cluster's delegate. This acknowledgement is multicast locally by each delegate. Once a database site knows T's data has been delivered everywhere and all previous transactions had been committed, then it commits T. The initiator of T can then reply to its client.

The detailed protocol algorithm is presented in Figure 1. It is composed of a set of handlers that deal with events triggered by the database engine ("Events at the initiator" and "Transaction commit") and with message delivery. We assume that every database site knows the current system's certifier through a function certifier(). The local concurrency control strategy of a given site, which is either snapshot isolation (SI) or strict two-phase locking (S2PL), is given by the function localCC(). Each cluster delegate can find the other participating clusters through a function remoteClusters() as well as identifying some delegate's cluster through function cluster(). Further, the function delegate() is used to determine whether the current site is the delegate of its cluster or not.

Global site variables Each database site manages four sets containing transactions known to be certified, locally updated, locally commited and remotely stable. It keeps track of the number of locally executed transactions in variable lts. The certifier keeps track of the number of certified transactions in variable gts.

Events at the initiator Before a transaction tid executes its first operation, the onExecuting handler is invoked. The version of the database seen by tid is required for the validation procedure, and for sites running snapshot isolation, this is equal to the number of committed transactions when tid begins execution. For sites using two-phase locking, the version must instead be recorded at the end of the execution, i.e. in the onCommitting-handler.

If the transaction at any time aborts locally, onAborting() is invoked and the transaction is simply forgotten by the protocol. On the contrary, if tid succeeds execution then onCommitting() is invoked. If local consistency is S2PL, the database version is recorded here. Then, tid's read set, write set and written values (rs, ws and wv) provided by the

database are reliably sent to the certifier along with the version of the database on which the transaction executed. The transaction's execution is left suspended until it is certified and its outcome known. If tid ends up committing then continueCommitting(tid) will be called, otherwise the initiator receives an (ABORT, tid) message from the certifier and forces the transaction to abort locally.

Certification Upon delivering an update transaction to certify — (CERTIFY, tid, ts, rs, ws, wv) — from some initiator site the certifier performs the certification of tid against its concurrent transactions. For every certified transaction (but not necessarily committed yet) ctid with timestamp equal or greater than tid's, a certification function is called with ctid's write set and tid's read and write sets. When preserving 1-SR the certification function checks tid's read and write sets against ctid's write set. If 1-SI is the adopted consistency criterion then only the write sets of both transactions are compared. In both cases, if there is a non empty intersection then the certification fails and an abort message is sent back to tid's initiator.

When tid's passes the certification test then the certifier's sequence number is incremented and tid added to its set of certified transactions. The transaction's id, commit order, write set and written values are then sent to all other replicas. Locally, tid is sent using the FIFO uniform view-synchronous multicast primitive as a (UPDATE_LOC, tid, gts, ws, wv) message. Remotely, it is sent using the FIFO reliable point-to-point primitive to each remote cluster as a (UPDATE_REM, tid, gts, ws, wv) message.

Remote delivery of updates Once a cluster delegate delivers a transaction from the certifier it simply forwards the message to the local replicas using the FIFO uniform view-synchronous multicast primitive.

Local delivery of updates When a replica delivers a transaction tid it signals the fact adding it to its set of updated transactions. The use of a uniform primitive ensures that once the transaction is delivered at the current replica it is eventually delivered at all non faulty replicas in the cluster. Therefore, if the replica is a cluster delegate it acknowledges the fact that tid became stable at the cluster to all clusters. The just delivered updates are

applied. If the replica is the tid's initiator then it just needs to proceed with continueCommitting(tid). Although tid does not hold high priority locks at the initiator, the fact that it passed certification means that between its execution and the given commit order, no other certified transaction conflicted with it, and consequently, tid will not be aborted by another transaction requesting high-priority locks at tid's initiator. For all other sites, db_update is invoked.

Delivery of remote acks Each time a delegate delivers a stability acknowledgment for transaction tid from some cluster, the pair (tid, cluster) is added to its acks set. When tid has been acknowledged by all remote clusters, then the delegate locally declares the transaction remotely stable using the (non- uniform) view-synchronous multicast primitive — (STABLE_REM, tid). When this message is delivered each replica adds tid to its remotestable set.

Transaction commit Here, each site handles the onCommitted callback. When onCommitted (tid) is invoked the site just increments its local database version lts and adds tid to its committed set. Since all tid locks have been released then any new transaction can read from tid and therefore from a more recent version of the database. When tid is known to be commited locally and stable everywhere the database is then allowed to reply to the client, which happens after continueCommitted(tid).

3.3 Failure handling

The WICE algorithm tolerates both the failure of single database sites as well as the failure of whole clusters. Locally, each cluster is governed by a group membership service and local communication rests on view-synchronous multicast primitives. This definitely eases failure handling locally. In the event of a site been expelled from the group (because it was taken down, has fail, became unreachable, etc.) every other site in the group eventually becomes aware of the fact by installing a new view of the group. This allows each site to deterministically determine the cluster's delegate should the former failed. Moreover, view-synchrony ensures that all sites surviving the previous view delivered the same set of

```
Global site variables
1  local = ts = []
2  certified = updated = ()
3  commited = remotestable = acks =
   {}
4  gts = lts = 0

Events at the initator
5  upon onExecuting(tid)
6       if localCC() == SI then
7            local[tid]=lts
8       continueExecuting(tid)
9  end

10 upon onComitting(tid, rs, ws, wv,
   type)
11      if localCC() == S2PL then
12           local[tid]=lts
13      rsend(CERTIFY, tid, local[tid],
        rs, ws, wv) to certifier()
14 end

15 upon onAborting(tid)
16      continueAborting(tid)
17 end

18 upon rdeliver(ABORT, tid) from i
19      db_abort(tid)
20 end

(1) Certification
21 upon rdeliver(CERTIFY, tid, ts, rs,
   ws, wv) from initiator
22      foreach (ctid, cts, cws, cwv) in
        certified do
23           if cts ≥ ts and
             !certification(cws, rs, ws)
             then
24                r_send(ABORT, tid)
                  to initiator
25                return
26      gts = gts + 1
27      enqueue (tid, gts, ws, wv) to
        certified
28      fifo_u_vscast(UPDATE_LOC,
        tid, gts, ws, wv)
29      foreach cluster in
        remoteClusters() do
30           fifo_r_send(UPDATE_REM,
             tid, gts, ws, wv) to cluster
31 end

(2) Remote delivery of updates
32 upon fifo_r_deliver(UPDATE_REM,
   tid, ts, ws, wv) from certifier
33      fifo_u_vscast(UPDATE_LOC,
        tid, ts, ws, wv)
34 end

(3) Local delivery of updates
35 upon fifo_u_vsdeliver(UPDATE_LOC,
   tid, ts, ws, wv)
36      ts[tid] = ts
37      enqueue (tid, ts, ws, wv) to
        updated
38      if delegate() then
39           foreach cluster in
             remoteClusters() do
40                r_send(ACK_REM,
                  tid) to cluster
41      if local[tid] then
42           continueCommitting(tid)
43      else
44           db_update(tid, ws, wv)
45 end

(4 and 5) Delivery of remote acks
46 upon r_deliver(ACK_REM, tid) from
   cluster
47      acked = {}
48      add (tid, cluster) to acks
49      foreach (tid, c) in acks do
50           add c to acked
51      if remoteClusters() ⊆ acked
        then
52           u_vscast(STABLE_REM,
             tid)
53 end

54 upon vsdeliver(STABLE_REM, tid)
55      add (tid) to remotestable
56 end

Transaction commit
57 upon onCommitted(tid) and ts[tid] =
   lts + 1
58      lts = lts + 1
59      add tid to commited
60 end

61 upon (tid) in commited and (tid) in
   remotestable
62      continueCommitted(tid)
63 end
```

Figure 1: WICE protocol

messages, thus not requiring any synchronization among them. As a result, no particular procedure is required on the failure or an ordinary site.

The detailed algorithms for the cluster's delegate and certifier failover are presented in [7]. In short, on a view change when the delegate fails, site d becomes aware it is the new cluster's delegate. To ensure that no transactions are blocked, d must rerun all transaction updates and acknowledgements received from remote clusters that may have been incom-

pletely processed by the previous delegate. The most serious single server failure is when the current system's certifier becomes unavailable. When initialized, the new certifier advertises itself to all delegates. There may be previously certified transactions not yet known to new certifier so a state synchronization is due.

The WICE protocol shall tolerate situations where multiple servers or entire clusters can fail abruptly. Most failure scenarios can be handled using a combination of the procedure for single servers. To avoid blocking during synchronization, we assume that all running synchronization routines are restarted if a delegate fails. The only scenario which requires special treatment is the loss of an entire cluster. In that case, the other clusters must be informed as soon as possible to allow blocking current and future transactions to become stable.

References

[1] A.Fekete. Serialisability and snapshot isolation. In *Proceedings of the Australian Database Conference, Auckland, New Zealand*, January 1999.

[2] A.Fekete, D.Liarokapis, E.O'Neil, P.O'Neil, and D.Shasha. Making snapshot isolation serializable. *ACM Transactions on Database Systems*, 2005.

[3] B.Kemme and G. Alonso. Don't be lazy, be consistent: Postgres-R, a new way to implement database replication. In *Proceedings of the 26th International Conference on Very Large Data Bases*, 2000.

[4] G. V. Chockler, I. Keidar, and R. Vitenberg. Group communication specifications: a comprehensive study. *ACM Computing Surveys*, 2001.

[5] S. Elnikety, F. Pedone, and W. Zwaenepoel. Database replication using generalized snapshot isolation. In *Proceedings of the 24th IEEE Symposium on Reliable Distributed Systems*, 2005.

[6] H.Berenson, P.Bernstein, J.Gray, J.Melton, E. O'Neil, and P.O'Neil. A critique of ANSI SQL isolation levels. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, 1995.

[7] J.Grov, L.Soares, A.Correia Jr., J.Pereira, R. Oliveira, and F. Pedone. A pragmatic protocol for database replication in interconnected clusters. In *Proceedings of IEEE International Symposium on Pacific Rim Dependable Computing*, 2006.

[8] A. Correia Jr., A. Sousa, L. Soares, J. Pereira, R. Oliveira, and F. Moura. Group-based replication of on-line transaction processing servers. In *Proceedings of Dependable Computing: Second Latin-American Symposium (LADC'05)*, 2005.

[9] Y. Lin, B. Kemme, M. Patino-Martínez, and R. Jiménez-Peris. Middleware based data replication providing snapshot isolation. In *Proceedings of the 2005 ACM SIGMOD International Conference on Management of Data*, 2005.

[10] P.Bernstein, V.Hadzilacos, and N.Goodman. *Concurrency Control and Recovery in Database Systems*. Addison-Wesley, 1987.

[11] F. Pedone, R. Guerraoui, and A. Schiper. The database state machine approach. *Journal of Distributed and Parallel Databases and Technology*, 14:71–98, 2002.

[12] S. Wu and B. Kemme. Postgres-R(SI):combining replica control with concurrency control based on snapshot isolation. In *Proceedings of the IEEE International Conference on Data Engineering*, 2005.