

P2P based Hosting System for Scalable Replicated Databases

Mesaac MAKPANGOU

Mesaac.Makpangou@inria.fr
INRIA - REGAL TEAM, Paris, France

ABSTRACT

We propose a large-scale storage system capable to host and manage partially replicated databases. The baseline of the overall solution is threefold: the support for transparent replication and management of database replicas distributed over a peer-to-peer network; the provision of a replica control middleware that guarantees the 1-copy snapshot isolation correctness criterion for the replicas of one database distributed world-wide; and the definition of a distributed transaction processing substrate that enables transactions to access data spread out several database replicas, while still preserving the consistency of the distributed state accessed by each distributed transaction. This paper presents how the proposed system can be deployed over a P2P network and discusses our proposed certification protocol. We anticipate that this storage system will boost the performance of database-intensive application or services that are accessed by clients distributed world-wide.

1. INTRODUCTION

Large-scale replication of databases is essential to address the scalability and the performance issues that face popular web applications. Consider for instance an online transaction processing e-commerce web site that must serve a large number of clients distributed world-wide; one could benefit from edge computing facilities to distribute its workload to edge servers, closed to its clients. Such a replication has the potential to augment the service availability, its scalability, and to reduce the response time perceived by end-users. However, since web applications often rely on backend databases to generate dynamically the contents to return to clients, solutions that propose to replicate only the application logic are not suitable for these applications; this could even reveal be worse than a centralized server. What seems interesting is to replicate both the databases and the application code accessing these bases at edge servers.

A number of solutions to replicate databases in large scale settings have already been proposed. The first generation of solutions propose a primary copy approach: the database is fully replicated at edge servers and at the origin server which serves as the primary replica. All update transactions are forwarded to the primary

replica. The secondary database replicas are used to serve read-only transactions. For these solutions, the correction criterion of choice is the one copy serializability. One way to achieve the one copy serializability is to enforce the "read one, write all" strategy. To implement this strategy, one will usually lock the secondary database replicas before the modification of the primary copy. Once the primary is updated, the modifications are pushed to the secondary database replicas. One advantage of the primary copy approach is the simplicity of its consistency protocol. While the primary replica approach is simple to implement, it fails to address appropriately the scalability issue. In particular, the number of update transactions that can be supported is limited by the capacity of the primary copy. Also, the one copy serializability correctness criterion, often, makes read-only transactions wait for the termination of update transactions.

To augment the performance of replicated databases, especially in large-scale settings, one solution is to trade-off transaction isolation against performance. Recently, an increasing interest has been paid to the Snapshot Isolation (SI) concurrency control method. With the SI concurrency control method, a read transaction does not conflict with an update transaction. Each transaction is assigned a valid snapshot of the database at its begin and its accesses are performed with respect to this assigned snapshot. At the commit time, if the transaction is read-only, it is committed and the control is returned to the client. If two update transactions conflicts, the first committed wins. Thanks to the provision of the snapshot isolation, the database management system could achieve increased concurrency by relaxing the isolation requirement on transactions, while avoiding many of the anomalies avoided by the traditional serializability correctness criterion [2].

A number of authors have proposed solutions to extend the Snapshot Isolation for distributed replicated systems [5, 8, 4]. Roughly, a database management system enforcing the SI concurrency method is deployed at each edge server. Each transaction is first executed at the edge server where it was submitted; at the commit time, a read only is committed locally, while for an update transaction a certification protocol is run by each replica to determine whether the terminated transaction is allowed to commit or should be aborted. If the certification succeeds, the writeset of the terminated transaction must be reported and committed by all replicas.

Though these solutions providing SI are very attractive for read-only transactions, they do not scale for update transactions. The problem stems from the fact that an important portion of the computation resource of each replica is consumed to replay remote update transactions. To make a replicated database scales, one needs

to fragment the database and to store different fragments at different servers, such as to distribute the access load to different servers. It is essential to keep the number of replicas of each fragment small since such a condition is a pre-requisite to boost the scale-out of a replicated database for update transactions [11]. Another drawback of these solutions is the inadequacy of the symmetric writeset certification protocol used by most existing lazy replication solutions. The symmetric writeset certification approach relies on total order multicasts to all participating replicas. Total order multicasts perform well within local area networks. Hence, if the replicated database has a large number of replicas spread out a wide area network, a certification protocol that requires upon each commit request to total order multicast the writeset to all replicas will exhibit poor performance.

In addition of the concurrency control and synchronization issues that face any database replication solution, the partial replication approach adds the issue of distributed execution of transactions accessing data spread out several replicas. Existing partial database replication solution either suppose a perfect transaction placement (each transaction is submitted to a replica where all the data it requires are stored) or rely heavily on total order multicasts to provide a consistent distributed snapshot to transactions accessing data spread out multiple database replicas. Using total order broadcast to provide consistent distributed snapshot to transactions are inefficient, especially if the database is replicated at a large-scale.

We designed a large-scale storage infrastructure capable to host and manage partially replicated databases. The baseline of the overall solution is threefold. Firstly, the provision of a mean to achieve partial and adaptive database replication policy (i.e., adaptive replication of fragments of the database) in a large-scale setting, while providing the replication transparency to database end-users. Secondly, a concurrency control and replication synchronization middleware that guarantees the 1-copy snapshot isolation correctness criterion for the replicas of one database distributed world-wide. Thirdly, a distributed transaction processing substrate that enables transactions to access data spread out several database replicas, while still preserving the consistency of the distributed state accessed by each distributed transaction.

In this paper, we focus on the above two former mentioned points. The rest of the document is organized as follows. Section 2 presents the assumptions we made and introduces the concepts and the abstractions upon which we build our solution. It also presents the overall architecture of a replicated database. Section 3 discusses how we deploy the main components of the proposed replicated database architecture on top of a peer-to-peer network. Section 4 presents our concurrency control middleware. Each replicated database has its own distributed concurrency control and replica synchronization middleware. As mentioned earlier, we focus on the writeset certification protocol. Section 5 gives a brief overview of related work. Section 6 draws some concluding remarks.

2. SYSTEM ARCHITECTURE

We consider a transactional programming model. A transaction is simply a sequence of database accesses (read or write operations) preceded by a `begin()` statement and closed by a `commit()` or an `abort()` statement.

We consider partial replication of fragmented databases. Section 2.1 highlights the main assumptions we made regarding the static fragmentation policy, as well as the expected output of the frag-

mentation process. Then, Section 2.2 introduces the main components that represent a replicated database within our hosting system. These components cooperate to serve efficiently the requests of clients and to maintain the replicated database consistent. Finally, Section 2.3 discusses the main interfaces offered to database providers (respective end-users) to register (respectively access) a replicated database.

2.1 Fragmented Database Representation

Each database is statically fragmented into a set of fragments. For the sake of simplicity, each fragment is made of a set of complete database tables.

We define the type of a fragment to be the list of tables that the replicas of this fragment type should store. Two different types of fragments correspond to two distinct sets of database tables, though the intersection of these sets is not necessarily empty. We also define the type of an access request to be the list of tables accessed by this request. We assume that for each access request type, there exists at least one fragment type that stores the tables required to serve access requests of this type.

The fragmentor attaches a unique name to each database fragment type. The fragment type name distinguishes this fragment type from its counterparts. One simple way to build a unique name of a fragment type is to concatenate, in some predefined order, the names of tables that constitute this fragment type.

Overall, a fragmented database is totally characterized by the descriptions of its fragment types, plus a primitive to access the initial value of each fragment type.

2.2 Replicated Database Architecture

A replicated database is represented by several components: the origin database server, the replicated database logical manager, a set of edge replica servers, and a group of global schedulers. In this section, we focus on the description of the role of each component.

2.2.1 The replicated database logical manager

One logical manager is associated with each replicated database. The behavior of the replicated database logical manager is controlled by the database provider. With respect to other components of the system, the manager represents the origin database server and implements the provider decisions or preferences regarding the management of his replicated database.

In practice the logical manager controls the replication degree, the replicas contents, and the replicas placements. It maintains an index that permits to locate where each fragment type is stored. Finally, the logical manager associated with a replicated database is also responsible of the client redirection policy: that is, upon the connection request of a new client, it decides which edge replica server to assign to the new client.

It is worth noting that the compound component made of the origin database server and the logical manager represents completely the replicated database: at any time, this compound component must be able to serve the uptodate value of each fragment type, even if none edge replica server is running. That is, the compound component comprising the database origin server and the logical manager acts as a replica; we call this couple the origin replica.

Whether the logical manager role is served by a single or a replicated entity depends on the characteristics of the network environment on top of which the hosting system is deployed.

2.2.2 Edge Replica Server

The term edge replica server (or simply replica) designates the compound component made of a database replica and its front-end local scheduler.

A database replica stores a fragment of the database, performs local transactions, and is capable to detect and to resolve local conflicts.

A local (or replica) scheduler is a transaction processing manager placed in front of the database replica. A local scheduler plays two roles: (i) it enforces the replica control (i.e; global concurrency control and replica synchronization); (ii) it spreads out transaction access requests on replicas that are capable to serve them, while guaranteeing that each committed transaction had accessed a consistent snapshot of the database.

Transactions are first executed by edge replica servers. Edge replica servers cooperate with global schedulers to certify writesets.

2.2.3 Global Scheduler

To achieve the global concurrency control and the replica synchronization, replica schedulers cooperate with global schedulers that are the ones responsible of enforcing the global consistency of the replicated database.

Global schedulers cooperate with one another thanks to group communication primitives, mainly reliable multicast and unicast, and total order multicast.

Section 4.2 details the role the group of global schedulers and discusses how this role is achieved.

2.3 The Hosting system APIs

2.3.1 Registry API

To replicate a database, his provider has to request it, thanks to an invocation to the `registry` method. The arguments passed to the `registry` method include:

- the description of the fragmented database (i.e; fragment type descriptions);
- the origin database server related information;
- the JDBC URL of the replicated database;
- Replication policy hints (e.g.; initial number of replicas to create for each fragment type, a mapping function to help determine for each client which fragment type is suitable).

The role of the `registry` method is to instantiate: the replicated database logical manager, the group of global schedulers, and the initial set of edge replica servers.

2.3.2 Access API

Once a replicated database has been registered, the provider can pass the JDBC URL of his registered replicated database to end-users for accesses.

Hence, to access a registered replicated database, end-users rely on the standard JDBC interface. To access a replicated database, an end-user needs to know the JDBC URL of this replicated database and must link its application with our own implementation of the JDBC Interfaces.

It is worth nothing that, though we provide our own implementation of the JDBC interfaces, we do not impose the use of additional methods. In particular, our implementation generates automatic invocations of the `begin()` statement at the start of each transaction. However, if an application wants, it may benefit from the interface to express its intentions and/or preferences; these application provided information could be exploited to achieve for example some performance optimization.

3. DEPLOYMENT ON TOP OF A PEER-TO-PEER NETWORK

In this section we discuss the deployment of our hosting system on top of peer-to-peer networks. We consider a two-levels hierarchical peer-to-peer networks. We assume that top level nodes are connected with one another by low latency and large bandwidth links; also, top level nodes do not leave voluntary the system (i.e., each top level node remains in the system as long as it doesn't fail).

we assume the availability of a scalable application-level multicast support, such as Scribe [3], built on top of the peer-to-peer network we are considering. We assume also that this group communication support offers primitives for reliable anycast, reliable multicast, and total order multicasts.

The baseline of the deployment on top of such a peer-to-peer network is three-folds:

1. The managing components (i.e; the replicated database manager and the global schedulers) are deployed on top level peers, while serving components (i.e; edge replicas) are deployed on second level nodes;
2. Managers, global schedulers and edge replica servers are widely replicated. This replication is however orthogonal to the one provided by the underlying peer-to-peer network. That is, our system controls the replication components of each hosted replicated database.
3. We rely on the underlying peer-to-peer network to route messages to serving components; we rely on group communication support to anycast or to multicast requests to group members.

3.1 The group of managers

A replicated database must remain accessible as long as it is hosted by our system. To achieve such a level of availability when the system is deployed on top of a peer-to-peer network where resources may join and leave the system at anytime, possibly without any notice, we need to replicate the manager component.

To create the group of managers, we proceed as follows. Firstly, we compute the key associated with this group. One simple way to compute the key is to apply a well known hash function to the JDBC URL identifying the replicated database. Secondly, since we build on top of a system like Scribe, one simply invokes the `create group` method passing the computed key as argument. This permits

to create a rendez-vous point for the managers and clients willing to connect to this replicated database.

Now, let assume that there exists a way to determine the suitable number of managers to create, say N . We identify each manager by the string obtained by concatenating to the JDBC URL of the replicated database, the suffix "manager" follows by a sequence number between 1 and N . The key of each managing component is obtained by the hash of its name.

For each computed key identifying one manager, the registry primitive sends a request to create a manager to the peer in charge of this key. A request to create a manager carries the key identifying the manager group, plus the information that were passed by the database provider as argument of to the `registry()` primitive.

Upon the reception of this request, a peer starts a manager. Once started, the manager must join the group of managers. For that, it sends a JOIN request indicating the key of the manager group.

3.2 The group of global schedulers

The key associated with this group is obtained by applying a hash function on the JDBC URL designating the replicated database, suffixed with the term "scheduler". The computed key is then used to create the global scheduler group rendez-vous point on the peer in charge of that key.

We then construct the schedulers names in the same way we construct the ones of the managers. The name of the i^{th} global scheduler is obtained by suffixing the JDBC URL of the replicated database with the term "scheduler" concatenated with the sequence number i . The key associated with a global scheduler is obtained by applying a hash function to this global scheduler name. A number of authors have proposed solutions to extend the Snapshot Isolation for distributed replicated systems [5, 8, 4].

As for managers, once the key of a global scheduler is computed, one sends a request to create a global scheduler to the peer in charge of the computed key. This creation request carries the key of the group of schedulers.

At the reception of a global scheduler creation request, the receiving peer starts a global scheduler and then sends a JOIN message to the group of global schedulers.

3.3 Edge replica group

We create one group for all edge replicas. This group's name is the JDBC URL the replicated database, suffixed with the string ":replicas".

Like for the manager and global scheduler groups, we compute the key corresponding to the group of edge replicas; then, we request the underlying system to create the group, passing the computed key. As a consequence, a rendez-vous point will be created at the peer in charge of this key.

Now, for each fragment type, the system determines the number of edge replica to create. The name of each edge replica is obtained by concatenating the JDBC URL designating the replicated database, the fragment type name, and a number between 1 and k , where k is the number of edge replicas for this fragment type.

For each computed key, we send a request to create an edge replica

to the peer in charge of this key. This message contains the data to replicate, the keys designating: the manager group, the global scheduler group and the edge replica group of this replicated database.

Upon the reception of a request to create an edge replica, a peer creates and initializes its local fragment database. Then it starts the front-end scheduler. Finally, it sends a request to JOIN the group of edge replicas.

At the end of the creation process, this peer returns its address such as to permit direct connection to this edge replica.

3.4 Notification of edge replicas

The last action of the `registry()` primitive is to notify the replicas locations to the group of managers. Roughly, once the three groups and their initial members are created, we build the replicas location index and notifies it to the group of managers, thanks a multicast to the group of managers. The notification message carries also the keys of the global scheduler group and of the edge replica group.

Upon the reception of this notification, each manager update its fragments location index. In addition, each manager JOIN the group of edge replica, such as to be delivered, on behalf of the origin database the certified writesets.

Finally, each manager will start the monitoring of edge replicas and global schedulers. We do not address further the monitoring issue in this paper, though this is a crucial issue for the reliability of our system.

3.5 Client redirection and data location

3.5.1 Client redirection

As explained in Section 2.3, end-users access a replicated database through the standard JDBC interfaces. The peer where the request is submitted can compute both the key corresponding to the group of managers or the one corresponding to the group of edge replicas. This is possible since the former is a hash of the JDBC URL of the replicated database, while the second is the hash of the concatenation of the JDBC URL and the string ":replicas".

One can use the key designating the group of managers to obtain an explicit redirection to a suitable edge replica. For that, once the managing group key is computed, this peer anycasts a lookup request passing the group managing key to the underlying system. The first encountered replica manager selects a suitable edge replica and returns its address to the end-user peer. Once this address is received, the connection is established with the selected edge replica. Letting the manager makes the redirection might permit to exploit hints on previous end-users behaviors: which fragment type is likely to be accessed by a client? Redirecting an end-user at the first place to such a replica could improve the performance.

It is also possible to anycast the request directly to the group of edge replicas. The request will be delivered to one edge replica, depending on where the end-user is located. The edge replica will respond with its address such as to permit the connection. Upon the reception of the response, the end-user peer simply issues a connection request to the specified edge replica.

3.5.2 Data location

When a transaction requests a data which is missing at the edge replica where this transaction was first directed, a `locate` message is anycasted to the group of managers. The type of the request that causes the miss is passed within the `locate` message.

The request will be delivered to the first encountered manager. Upon the delivery of a `locate` request, a manager determines the set of locations of fragments that can serve this type of request, and returns this list to the requester.

Upon the reception of the response, the requester peer checks whether it is safe to access a remote replica. If so, it peeks one remote capable to serve the missing data, connects to it, and then forwards it the request.

4. REPLICA CONTROL MIDDLEWARE

Each transaction is first executed at one replica, the replica where this transaction was first submitted. Upon the reception of the request to commit, a certification procedure is run to determine whether this transaction can be committed or should be aborted due to conflicts with other concurrently executed transactions. If the certification procedure decision is to commit, the transaction is committed at the replica where it was executed; in addition each replica that stores a data item updated by the committed transaction will eventually run a relay transaction to report on its local database replica the updates performed by the committed remote transaction.

We rely on a two levels concurrency control approach. Conflicts between local transactions executing within the same location are handled by the local Database Management System. The conflicts between local and remote transactions, and the synchronization of database replicas are handled by the replica control middleware. The objective of our replica control middleware is to provide a *1-copy isolation* guarantee for the group of replicas of a relational database distributed over the peer-to-peer network, while offering good response time for both read-only and update transactions. In addition, we want the resulting database replication system to scale both for read-only and update transactions.

Our replica control middleware relies on global timestamps to globally schedule update transactions. Timestamps are managed by a set of cooperative schedulers associated with each replicated database. We distinguish two kinds of schedulers: *replica scheduler* and *global scheduler*.

4.1 Replica Scheduler

Each replica scheduler is associated with one database replica. A replica scheduler and its associated database replica are co-located within the same edge server. Hence, there are as many replica schedulers as the number of database replicas.

With respect to clients, a replica scheduler represents its associated database replica. The replica scheduler offers the JDBC standard interface to connect to and access its associated database replica. It also extends the JDBC standard interface to permit explicit calls to begin statements.

With respect to the concurrency control, each replica scheduler initiates and participates to the certification procedure of each update transaction that is first executed at its edge server. Each replica scheduler is also responsible of initiating the report of updates performed by committed remote update transactions. Each local replica

```

void onCertificationRequestFromReplica(rid, tid, replicaTime, ws, frags) {
    set alreadyControlledFrgs =  $\emptyset$ ;
    boolean waitResponse = true;
    string finalOutcome = "REFUSED";
    // Create an inter global scheduler certification request
    CertificationRequest certificationRequest(rid, tid, replicaTime, ws, frags);
    GlobalCertificationRequest globalCertificationRequest(certificationRequest, globalSchedulerId);
    // Multicast the inter global schedulers request to the entire group of global schedulers, then wait for responses
    total_order_multicast("GlobalCertificationRequest", globalCertificationRequest, groupOfGlobalSchedulers);
    while(waitResponse)
        globalCertificationRequest.waitForResponseNotification();
    response = certificationRequest.getNextResponse();
    if (response.outcome == "REFUSED")
        // If a negative vote is received, stop waiting
        requestTimeStamp = response.getCertificationTimeStamp();
        waitResponse = false;
    else
        alreadyControlledFrgs.add(response.getSupervisedFrgs());
        // A positive vote is received. If all updated fragments are already checked,
end waiting.
    if (frags  $\subseteq$  alreadyControlledClusters)
        finalOutcome = "GRANTED";
        order = response.getCertificationTimeStamp();
        waitResponse = false;
        CertifiedTransactionDesc ctd;

    // Prepares the certification decision, then notifies the group of global schedulers of the final decision.
    if (finalOutcome == "GRANTED")
        ctd.tid = tid; ctd.rid = rid; ctd.ws = ws; ctd.frags = frags; ctd.vts = order;
    else
        CertificationResponse response("CertificationResponse", "REFUSED", tid);
        reliable_unicast(response, rid);
        total_order_multicast("FinalOutcome", finalOutcome, order, globalSchedulerId, ctd, groupOfGlobalSchedulers);
}

```

Figure 1: Certification coordinator algorithm

scheduler is notified the descriptions of remote transactions that have committed thanks to the replica synchronization protocol.

Overall, the replica scheduler can be assimilated to a set of handlers, each corresponding either to a request that can be issued from a client, or to a notification sent by another scheduler.

4.2 Global schedulers

Each replicated database, as a whole, is associated with a group of global schedulers. The primary role of global schedulers is to certify and to globally schedule each update transaction. Global schedulers are also responsible of the replica synchronization.

Briefly, when an edge replica server wants to request the certification of a writeset that a local client has requested to commit, it simply anycasts a certify request to the group of global schedulers. This request is delivered to the first encountered global scheduler. This scheduler promotes itself the coordinator for this certification request. As the coordinator, it takes care of the request until its response is returned to requester edge replica server.

We assume that the global scheduler which receives a certification request from a replica scheduler doesn't crash during the the lifetime of this request (that is it remains alive until it returns to the requester or publishes the notification of the validation of the concerned transaction at the intention of all interested replicas.

4.2.1 Certification Coordination Algorithm

The method `onCertificationRequestFromReplica()` of Figure 1 presents the pseudocode of the certification coordination algorithm.

The coordination process starts upon the reception of the certification request by one global scheduler. The coordinator of a certification request behaves as follows.

Firstly, it total-order multicasts the corresponding global certification request to the entire group of global schedulers. This global certification request contains the initial certification request received from the replica scheduler and the identifier of this coordinator. Once the coordinator has total-order multicasted its global certification request to all global schedulers (including itself), it waits for responses from global schedulers that must participate to the computation of the global outcome for this global certification request.

Each response to a global certification request carries a positive or a negative vote. A negative vote means the global scheduler which responds has detected a conflict with an already concurrent certified transaction, while a positive vote means the global scheduler which responds doesn't detect any conflict with any concurrent certified transactions. In case of a positive vote, the voter piggybacks within the response the list of fragments that it supervises, as well as the global certification delivery order of this request.

Upon the reception of the first negative vote, the coordinator decides that the global outcome of this certification request is "REFUSED". If however none negative vote is received, once the coordinator has obtained at least one positive vote from at least one global scheduler in charge of each fragment updated by the write-set contained in the certification request, the coordinator decides that the final outcome is "GRANTED". The validation timestamps of the transaction to certify is then set to this certification request delivery order contained in all certification responses.

Once the coordinator has decided the global outcome of its global certification request, it must notify its decision to its peers as well. For that, the coordinator total-order multicasts the global outcome to the group of global schedulers.

Finally, if the global outcome is "REFUSED", the coordinator returns a response to the requester replica scheduler indicating this global outcome;

4.2.2 Conflict Detection Algorithm

The method `onCertificationRequestFromCoordinator()` of Figure 2 presents the algorithm run by each global scheduler upon the reception of a certification relayed by a coordinator. This is a distributed fragment-based transaction conflict detection: each global scheduler is in charge of detecting conflicts with respect to fragments that it supervises. We assume that, the assignment function of fragments to global schedulers guarantees that, at anytime, each fragment is supervised by at least one active and non-faulty global scheduler.

Upon the reception of a global certification request, each global scheduler first determines whether it has to participate actively to the evaluation of this request or not.

- A global scheduler participates to the evaluation of a global certification request if this global scheduler supervises at least one fragment updated by the transaction to certify.

```

void onCertificationRequestFromCoordinator(request, coordinator) {
    CertificationResponse response("CertificationResponse");
    response.replica = request.rid; response.tx = request.tid;
    response.ts = ++globalTime;
    response.outcome = "GRANTED";
    response.supervisedFragments = {}; // List of updated fragments supervised by this
    scheduler
    ∀ fid ∈ request.fragments, such that (computeCertifierId(fid) == globalSchedulerId)
        response.supervisedFragments.add(fid);

    // Is this scheduler supervising a subset of updated fragments
    if (response.supervisedFragments != {}) {

        // Adds the new request in the list of requests waiting to be proceeded.
        OutstandingGlobalRequest nr(response.ts, request.ws, response.supervisedFragments);
        outstandingRequests.append(nr); continueToWait = true;
        while (continueToWait)
            wr = nr;

        // Check for conflict with concurrent outstanding requests. If found, wait the
        global outcome.
        ∀ r ∈ outstandingRequests, if ((r.ts < wr.ts) ∧ (r.ws ∩ wr.ws ≠ {}))
            wr = r;
        if (wr == nr) continueToWait = false;
        else wr.waitGlobalOutcomeNotification();

        // Check for conflict with certified writesets.
        ∀ fid ∈ response.supervisedFragments
            ∀ t ∈ fragTable[fid].to_relay
                if ((t.vts > request.replicaTime) ∧ (t.ws ∩ ws ≠ {}))
                    { response.outcome = "REFUSED"; break; }
            if (response.outcome == "REFUSED") break;
        reliable_unicast(response, coordinator);
    }
}

```

Figure 2: Conflict detection algorithm.

- To determine if a global scheduler supervises a fragment, we compute the certifier identifier for this fragment; if the result corresponds to the identifier of this global scheduler, it is in charge of this fragment; otherwise, it is not in charge.

If the global scheduler do not have to participate to the evaluation of the global outcome of global certification request, it simply increments its `globalTime` such as to reflect the reception of this global certification request.

If however this global scheduler has to participate to the certification process, we distinguish two cases. If there exists an outstanding conflicting global certification request that requires the participation of this global scheduler, the new request is simply queued within the ordered queue, `outstandingGlobalRequests`.

If there exists none outstanding global certification request that conflicts with the newly received one, the global scheduler runs its local certification procedure. Basically, for each fragment that it supervises and which is updated by the transaction to certify, it checks whether there exists a description of a transaction which is concurrent and conflicting with the transaction to certify within `fragTable`. If such a transaction exists, it returns a negative vote to the coordinator. Otherwise, it a positive vote is casted. In case of a positive vote, the list of fragments updated by the transaction to certify and supervised by this global scheduler are piggybacked within the response.

4.2.3 Certification Request Termination Algorithm

Upon the notification of the global outcome of a global certification request, each global scheduler does the following. If it was not concerned by the computation of this global outcome (neither as

```

void onGlobalCertificationOutcome(result, deliveredOrder, coordinator, certified-
TransactionDesc) {
    found = false;
    if  $\forall r \in$  outstandingRequests
        if (r.ts == deliveryOrder)
            { found = true; wr = r; outstandingRequests.remove(r); break }
    if ((found)  $\wedge$  (result == "GRANTED"))
         $\forall$  fid  $\in$  wr.supervisedFragments
            fragTable[fid].to_relay.append(certifiedTransactionDesc);
        wr.notifyAll();
        if (coordinator == globalSchedulerId)
            publish(certifiedTransactionDesc);
}

```

Figure 3: Termination algorithm: each scheduler is informed of the decision and updates his local state accordingly.

evaluator, nor as the coordinator), it simply discards this notification.

Otherwise, if the global scheduler was an evaluator, it removes the corresponding request from `outstandingGlobalRequests`. Then, if the notified result is "GRANTED", for each fragment updated by the certified transaction and which is under its control, the global scheduler appends the description of the certified transaction within the list of certified transactions that have modified this fragment. Note that, once added, a description remains within this list until one is ensured that none certification request concerning a concurrent transaction will be received anymore. Finally, the global scheduler determines if there are awaiting requests that can now be served. If such requests exist, they are extracted from the awaiting and served as described above.

In case where a certification result is "GRANTED" and this global scheduler is the coordinator, then it must propagate the description of certified transaction to all replicas of the database that store fragments that will be updated by the certified writeset.

To achieve a selective propagation of the descriptions of certified transactions, we rely on a publish-subscribe system. In one hand, each replica scheduler subscribes with the list of database fragments that it stores. On the other hand, once the global scheduler that coordinated the certification (and hence made the certification decision) is delivered its own final outcome notification, it publishes the description of the certified transaction. We assume that the descriptions of certified transactions that are published by the group of global schedulers are delivered to replica schedulers that subscribe for them in their certification order.

5. RELATED WORK

The Snapshot Isolation (SI) concurrency control method has regained lot of interest in the recent past years. The SI concurrency control method permits to achieve increased concurrency by relaxing the isolation requirement on transactions, while avoiding many of the anomalies avoided by the traditional serializability correctness criterion.

A number of database replication solutions enforcing the Snapshot Isolation method have been recently proposed. Ganymed [10] is a middleware-based database replication system that support SI. Ganymed is based upon the Replicated Snapshot Isolation with Primary Copy (RSI-PC) scheduling algorithm. The central component of the Ganymed middleware is the RSI-PC scheduler. The RSI-PC scheduler hides to clients the set of database replicas. The role of the RSI-PC is to assign each new transaction to either the mas-

ter replica or to one of the slave replica. Roughly, upon the start of a new transaction, the RSI-PC scheduler, determines where to place the transaction based on its type: if this is an update transaction, it is directed to the master replica; if however this is a read-only transaction, the scheduler will choose one slave replica and then sends the transaction to the selected replica. Ganymed scales well for read-only transactions. However, for update transactions, the primary copy approach limits the throughput of the system to the throughput of the primary copy. Our manager is similar to the RSI-PC scheduler of Ganymed. However our manager has additional functionalities, mainly support for partition location as we considered partial replication instead of full replication considered by Ganymed. Also, according to Jim Gray [6] classification, we consider a group replication, while Ganymed proposes a primary copy approach.

In a centralized database server providing SI, each transaction accessed the latest installed snapshot at the moment of the first operation of the transaction (strong consistency). In a distributed environment, with replicas of the database deployed at different locations, providing each transaction with the latest snapshot comes with a cost. In [5], the authors propose a generalized snapshot isolation (*GSI*) abstraction. With *GSI*, a new transaction is allowed to accessed any valid snapshot. Hence, the application can control the trade-off between the snapshot freshness and the response time. Compare to *GSI*, the material described here does not address specifically the snapshot freshness issue. In our system, a read transaction accesses the latest snapshot installed on the replica where the transaction is first submitted.

Most of the existing database replication solutions [4, 5, 8] providing SI implement a symmetric certification approach: Upon the request to commit an update transaction, the replica where this transaction was executed broadcasts the writeset to certify to all replicas. Upon the delivery of each certification request, each replica checks for conflict with already certified writesets. If none such conflict is found, the writeset is certified, then is applied, and finally is committed at each replica. If however such a conflict is found, the requesting transaction is aborted. The use of atomic broadcast guarantees that all replicas are delivered and then process the flow of certification requests in the same order. Though the symmetric certification approach is well understood, it does not scale. In contrast, our solution is scheduler-based: writeset certification requests are handled by global schedulers; the number of global schedulers is too small compared to the number of edge replicas. As a consequence, our certification protocol relies on small-scale multicasts. Large-scale multicasts are only used to disseminate certified writesets to interested edge replicas.

In [9], the authors propose a sequencer-based certification protocol dedicated for large-scale replicated databases. The authors propose a hierarchical architecture exploiting the topology of the underlying networks (i.e; separate LANs interconnected by longhaul links) that are often available to large enterprises to replicate their databases. The approach consists in replicating the master server inside its own LAN, and also replicating the secondary replica servers inside their own LANs. For each LAN, one server plays the role of scheduler with respect to its peers located in the same LAN. The replicas co-located within the same LAN communicate via Group Communication Systems. A replica in the secondary LAN first validates update transactions by the local sequencer. If the validation succeeds, the local sequencer forwards it to a replica of the main server. Upon the reception of a certification from a local sequencer,

the replica of the main server, forwards this request to the main sequencer. If the certification succeeds, the main sequencer multicasts the writeset notification to its peers within the main LAN and also to the set of local sequencers. Each local sequencer will then propagate the certified writeset to its peers within the same LAN.

Our certification protocol considers the specifics of large-scale settings too, however our design choices are different. One main difference with the system presented in [9] is that we rely on lightweight schedulers: they do not hold a replica of the database; as a consequence, they can be instantiated anywhere we deem interesting. Another interesting aspect is that the number of our schedulers are not related to the number of replicas and their locations. We also differ from the described system in that we consider partial replication, whereas they consider full database replication. Total replication suffers from throughput limitations because all update queries have to be applied to every replica server, thus the throughput is limited to the threshold that is sufficient to overload one single server.

Partial replication of a database can help improve the performance of a database replication system. GlobeTP [7] employs the partial replication to improve database throughput. This system targets web applications with well known access templates. The authors propose a database tables placement algorithm and a request redirection policy that, combined altogether, minimize the load on each replica.

In [1], Armendariz-Inigo and al. propose SI-PRe, a partial database replication protocol that enforces the Generalized-SI guarantee. This protocol relies heavily on total order multicasts to capture consistent distributed snapshots to use by update transactions, and to certify update transactions. With respect to our goal, the heavy use of total-order multicasts to coordinate replicas makes this solution inappropriate for the coordination of a large number of partial database replicas distributed over a WAN.

6. CONCLUSION

This paper presented the design of a large-scale database hosting infrastructure. We discussed how such a large-scale replicated database hosting systems can be deployed on top of a P2P network, while guaranteeing some degree of robustness. We presented a replica control middleware capable to enforce SI guarantee to large-scale group of database replicas. We anticipate that, thanks to the partition of the database and our certification protocol that struggles to reduce the number of large-scale multicasts, the resulting database replication system will scale for both read and update transactions. The implementation of the first prototype is an ongoing task. It will permit to access the performance of the system.

7. REFERENCES

- [1] J.E. Armendariz-Inigo, A. Mauch-Goya, J.R. Gonzalez de Mendivil, and F.D. Munoz-Escoi. Sipre: A partial database replication protocol with si replicas. In *Annual ACM Symposium on Applied Computing*, Fortaleza, Ceara, Brazil, March 2008.
- [2] Hal Berenson, Phil Bernstein, Jim Gray, Jim Melton, Elizabeth O’Neil, and Patrick O’Neil. A critique of ansi sql isolation levels. In *SIGMOD ’95: Proceedings of the 1995 ACM SIGMOD international conference on Management of data*, pages 1–10, New York, NY, USA, 1995. ACM.
- [3] Miguel Castro, Peter Druschel, Anne marie Kermarrec, and Antony Rowstron. Scribe: A large-scale and decentralized application-level multicast infrastructure. *IEEE Journal on Selected Areas in Communications (JSAC)*, 20:2002, 2002.
- [4] Khuzaima Daudjee and Kenneth Salem. Lazy database replication with snapshot isolation. In *Proceedings of the 32nd International Conference on Very Large Data Bases.*, pages 715–726, Seoul, Korea, September 2006. VLDB.
- [5] Sameh Elnikety, Fernando Pedone, and Willy Zwaenepoel. Database replication using generalized snapshot isolation. In *Proceedings of the 2005 24th IEEE Symposium on Reliable Distributed Systems (SRDS’05)*, Orlando, Florida (USA), October 2005. IEEE.
- [6] Jim Gray, Pat Helland, Patrick O’Neil, and Dennis Shasha. The dangers of replication and a solution. In *Proceedings of the 1996 ACM SIGMOD International Conference on Management of Data*, pages 173 – 182, Montreal, Quebec, Canada, June 1996. ACM SIGMOD.
- [7] Tobias Groothuyse, Swaminathan Sivasubramanian, and Guillaume Pierre. GlobeTP: Template-based database replication for scalable web applications. In *Proceedings of the 16th International World Wide Web Conference*, Banff, Canada, May 2007.
- [8] Yi Lin, Bettina Kemme, Marta Patino-Martinez, and Ricardo Jimenez-Peris. Middleware based data replication providing snapshot isolation. In *Proceedings of the 24th ACM SIGMOD International Conference on Management of Data.*, Baltimore, Maryland (USA), 2005. ACM SIGMOD.
- [9] Yi Lin, Bettina Kemme, Marta Patino-Martinez, and Ricardo Jimenez-Peris. Enhancing edge computing with database replication. volume 0, pages 45–54, Los Alamitos, CA, USA, 2007. IEEE Computer Society.
- [10] Christian Plattner and Gustavo Alonso. Ganymed: Scalable replication for transactional web applications. In *Proceedings of 5th International Middleware Conference*, pages 155 – 174, Toronto, Ontario, Canada, October 2004. ACM/IFIP/USENIX.
- [11] D. Serrano, M. Patino-Martinez, R. Jimenez-Peris, and B. Kemme. Boosting database replication scalability through partial replication and 1-copy-snapshot-isolation. In *PRDC ’07: Proceedings of the 13th Pacific Rim International Symposium on Dependable Computing*, pages 290–297, Washington, DC, USA, 2007. IEEE Computer Society.