1 Protocol Description

A large amount of work has been dedicated to distributed data stores with simple interfaces allowing users to read or write one key-value pair at a time. However, application developers using such data stores could benefit from more advanced services such as serializable transactions: regions of code consisting of multiple accesses to the distributed data store which appear to execute atomically.

Systems that do provide distributed transactions either offer weak semantics (e.g. snapshot isolation), or offer strong guarantees, but depend on additional constraints such as synchronized clocks (e.g., Spanner []).

We present a generic framework providing serializable distributed transactions. Our framework can easily be adapted to work with a large variety of data models and interfaces. At the heart of our framework is `new_protocol_name` , an optimistic distributed multi-version concurrency control protocol. Similarly to Lomet et al. [], we use timestamp intervals to find potential serialization points and minimize the number of conflicts. However, we extend previous work by applying and optimizing it in the context of a distributed data store rather than focusing on a single machine. This allows us to achieve a higher degree of transaction concurrency than previous work on distributed data stores []. It is important to note that while our protocol does provide serializability, it does not provide strict serializability: a transaction may be serialized even in the past.

Essentially, `new_protocol_name` adds a transaction management layer on top of both the client and the storage server of a traditional, non-transactional data store (see Figure **??**). We go into more details concerning the precise architecture of our system in the next section.

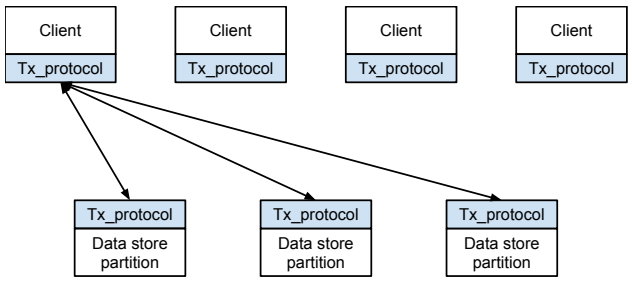

Figure 1: System overview

2 System Architecture

The main components our protocol uses are shown in Figure **??**. The components we are adding to the default data store architecture are highlighted in blue.

2.1 The Client

Transactional interface. We provide the clients with a transactional interface: special instructions are provided to start and end a transaction. All the normal data store operations that are within a transaction are executed according to the ACID properties. It is important to note that we allow dynamic transactions: the read and write sets may depend on previously read values. The transactional service provided is thus more than simple atomic multi-get/multi-put, and beyond the mini transactions of Sinfonia [].

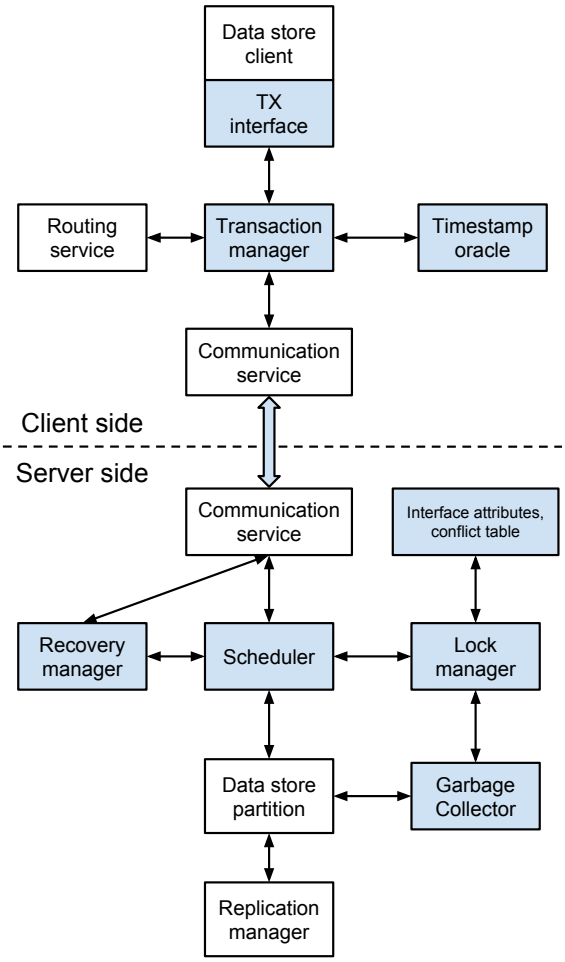

Figure 2: System architecture. The components we need to provide are highlighted.

Timestamp oracle. In addition, the client also uses a timestamp oracle. In essence, the task of the timestamp oracle is to provide serialization intervals for new transactions based on the outcomes of past transactions. In the simplest case, it is simply a component local to each client that does not communicate with other oracles. It can also be a distinct remote service, gathering transaction information from all clients. It is important to note that regardless of the output of the timestamp oracle, the safety of the system is never jeopardized: the only danger is an increase in the number of unnecessary aborts.

Transaction manager. In our protocol, the client is responsible for coordinating its transactions. This means keeping track of the current state of the transaction and contacting the appropriate storage servers in order to commit (or abort) one's operations.

Routing service. The client also needs a routing service, which is responsible for providing the address of the storage server responsible for a particular key. This component is independent of our protocol, and, if provided by the underlying data store, can be used unmodified.

Communication service. This component is responsible for serializing and sending data from clients to storage servers.

2.2 The Server

Communication service. Similarly to the analogous service on the client, this component is responsible for exchanging data between a server and other nodes in the protocol.

Scheduler. The scheduler is responsible for implementing the concurrency control policy. This component handles the client requests, and applies, delays or aborts them.

Lock manager. The lock manager holds the lock metadata. The metadata that needs to be kept for each version is the timestamp at which a version was created, the state of the version, as well as the timestamp it was last read from. In the default variant, it is located on the storage servers, with the data it is responsible for. However, this component can be stored on a separate metadata server as well.

Recovery manager. The recovery manager is responsible for recovering the system to a consistent state when there are transient failures, such as unresponsive transaction coordinators or server reboots.

Interface attributes, conflict table. This component encodes the semantics of the operations provided in the client interface. More specifically, it defines whether an operation returns a function of the current state of an object to the client, and what other operations it conflicts with.

Data store. This layer is responsible for storing the data that this server is responsible for. The partitioning strategy employed by the data store to spread data between the servers is largely orthogonal to our protocol. We assume a function mapping keys to servers is provided by the underlying data store (and used by the clients).

Replication manager. The replication manager is responsible for keeping replicas of the data store and the lock manager log consistent. We also defer replication-related concerns to the underlying layer.

Garbage collector. The garbage collector is responsible for removing old versions of the data. To do this, it must be periodically given a bound below which timestamps will not be issued in the future.

3 The Protocol

In this section, we describe the functionality of our protocol. We mostly focus on the server components, as the client is somewhat simpler.

3.1 General Approach

We now present the general structure and mechanism of our protocol. We present a number of details and policies in the following subsection.

The server. We first look at the server's operations. In particular, we focus on the server scheduler, which handles read, write, commit and abort messages from the clients.

When receiving a read, the server fetches the appropriate data version given the current serialization interval of the transaction. In case there are multiple versions which might be chosen, the decision is made according to a policy we describe in the next subsection. In case the version is marked as *pending*, the server waits for this version to become available or for a timeout before it responds to the client. We use a timeout in the case of RW transactions in order to avoid deadlock. In the case of read-only transactions, the timeout is set to infinity. The server also applies a read lock for an appropriate interval, and returns the updated serialization interval to the client (along with the data, and possibly other locking-related information - see Section **??**) through a *rl_info* entry. In case no version was ever created for the searched key, we create a version with a minimum timestamp and an empty marker as a value. This is because we want to avoid future inserts with timestamps smaller than our failed read (we essentially want to avoid the phantom anomaly). The pseudocode for this operation is shown in Algorithm 1.

Algorithm 1 Reception of a read message at a storage server

```
1:  function RECEIVE_READ(client, t_id, t_interval, key)
2:      wait until timeout() or ((version = get_version(key,t_interval,READ_FLAG))
    and ((version == NULL) or ((version.state == COMMITTED) and (rl_info =
    get_read_lock(version,t_interval) != NULL)))
3:      if timeout() then
4:          send(client, t_id, READ_TIMEOUT);
5:          return ;
6:      end if
7:      if version == NULL then
8:          data = NULL;
9:      else if
10:         thendata = data_store.read(version.key:version.timestamp);
11:     end if
12:     send(client, t_id, data, rl_info);
13: end function
```

When receiving a write request, the server first tries to obtain (create) a data version intersecting with the serialization interval sent along with the write. In order to be able to write to a version, the maximum timestamp of a read lock taken for this version must not be larger than the right limit of the serialization interval. This is because the write should occur later than any read on a particular version; otherwise the serializability of the reading transaction would be jeopardized. In case no such version exists, the transaction is aborted[1]. If an appropriate version is found, a write lock is taken (by essentially creating a new version with an interval as a timestamp and marked as

[1]As we show in Section **??**, we can in fact take advantage of the knowledge regarding the versioning of the write set and restart the transaction with an updated serialization interval

pending). An acknowledgement, as well as possibly additional versioning information (for example, hints regarding other potential serialization points) is returned to the client. The pseudocode for the write operation is provided in Algorithm 2. In this algorithm, the `get_write_lock` function actually has to create a new version. The information regarding this new version is not persisted at this point, but other transactions accessing the concerned key are able to see it. It is important to note that we do not specify the moment when these write messages are received at the storage servers. These messages can be issued by the clients as the writes occur in the transaction, or they can be sent when the transaction is to be committed.

Algorithm 2 Reception of a write message at a storage server

```
1: function RECEIVE_WRITE(client, t_id, t_interval, key, value v)
2:     prev_version = get_version(key,t_interval,WRITE_FLAG);
3:     if (prev_version.max_read_lock ≥ t_interval.end) or (wl_info =
   get_write_lock(prev_version,t_interval) == NULL) then
4:         abort_transaction(t_id);
5:         send(client, t_id, ABORT);
6:         return ;
7:     end if
8:     t_id.write_set.add(wl_info.version);
9:     wl_info.version.value = v;
10:    send(client, t_id, wl_info);
11: end function
```

Upon receiving a commit message, the server must update the timestamps and state for the write-set of the concerned transaction to *committed*. The commit request should contain the final serialization timestamp of the transaction. The pseudocode for the commit operation is provided in Algorithm 3.

Algorithm 3 Reception of a commit message at a storage server

```
1: function RECEIVE_COMMIT(client, t_id, ts)
2:     for version V in t_id.write_set do
3:         V.timestamp = ts;
4:         V.max_read_from = ts;
5:         V.state = COMMITTED;
6:         lock_manager.persist(V);
7:         data_store.write(V.key:ts,V.value);
8:     end for
9:     delete(t_id.write_set);
10:    send(client, t_id, ACK);
11: end function
```

In case of an abort, the server needs to release the locks it has taken for the concerned transaction, and clean up any pending versions. Undoing uncommitted writes is straight-forward: the server simply removes the versions marked as "pending" corresponding to the concerned transaction. We note that a *prepare* message in this scenario is not necessary: the prepare phase essentially occurs as the transactions obtain write locks during the execution of the transaction. In the case

of reads, we have the option of reverting the `max_read_from` tag of the read versions to reflect a state where the aborting transaction did not occur. However, this is just an optimization, and, if the lock intervals are fairly small, it is not very useful.

The client. We now briefly look at the client operations. The client sends read requests to the appropriate server as soon as they appear in the transaction under execution. As mentioned in the previous section, the client obtains the id of this server from a routing service, and the timestamp interval with which it begins its transaction from a timestamp oracle. The client can choose the moment when the write requests are sent to the storage server, starting from the moment they appear in the transaction, and until commit time. The sooner a client notifies the server of the writes, the smaller the probability of being aborted by another transaction is. If the current transaction aborts however, by eagerly sending writes to the servers and acquiring write locks, we risk aborting other transactions unnecessarily. Most importantly, the client needs to keep track of the current serialization interval: this may change after each interaction with a data server, depending on when the client's operations can be serialized on the server. If this interval becomes empty, the client aborts the transaction.

Beyond simple reads and writes. We have so far focused on simple read and write operations. However, `new_protocol_name` supports efficient implementations of other operations, such as:

- Multi-gets: read multiple keys-value pairs at the same time;
- Multi-sets: write multiple key-value pairs at the same time;
- Read-modify-write operations;
- Arithmetic and logic operations;
- Iterations;

The main reason why some of these operations (e.g. increment) can be implemented efficiently is due to the fact that they can be executed entirely at the storage server, without having to ship intermediate results to the client. This also grants us more freedom in serializing such operations. For example, the order of two commutative operations that do not return data to the client can be modified without affecting the external view of the system.

In general, in order to extend the interface, one has to specify whether a new operation has read semantics (i.e., returns a value to the client), write semantics, or both, and provide a conflict table for the entire interface.

The code for a more generic operation is similar to the read and write algorithms presented above. The main difference is that the checks an operation will have to do will depend on its conflict table and its read semantics. **[TODO: write the code for this as well at some point.]**

Iterations (tentative). There are a number of challenges in providing iterations in this multi-version protocol. In particular, we have to prevent any anomalies which can appear (e.g. the phantom anomaly, where we don't apply read locks to non-existent keys when iterating). In brief, we employ a standard solution, with a number of modifications. We use a B-tree like structure. When iterating, we also mark the internal nodes with the serialization point corresponding to the

iteration (each internal node has a `max_read_from` tag). When doing a write to a key that is not currently present in the data store, we take the maximum of the `max_read_from` seen on the path to the node, and if that exceeds the write timestamp, we abort the latter. In addition, we also need to provide efficient *prev* and *next* operations. This is partly dependent on the number of versions we generally need to keep for each data item. **[TODO: update this paragraph when we decide on the iterations approach.]**

Data model. In its simplest variant, our protocol handles data which has a unique (variable length) key, and an opaque value. However, the protocol can easily be extended to cover multi-column tables as well. In this version, instead of using a single timestamp per row, the protocol can associate a timestamp for each row and column (this granularity of locking is configurable). This allows us to perform operations at the granularity of a single column, as well as at row granularity.

3.2 Protocol Policies

The previous section presented the general structure and operation of our protocol. We now discuss the specific policies we employ within this general framework.

One of our main goals was minimising the number of accesses to the read-set servers. We assume a deployment with a large number of storage servers, and a workload where long-running read-mostly transactions are common**[TODO: experimentally check the impact of the number of trips to the read set on the performance]**. As such, most of the message exchanges would occur between the coordinator and the storage servers containing the read-set of the transaction. Thus, by reducing the number of communication rounds between the coordinator and the read set, we would obtain a significant reduction in the total number of messages as well as match the best known concurrency control schemes with a single RTT latency per commit.

In this section, we present the details of the protocol version we use, which requires only one access to the read set in the common, conflict-free case. It is worth noting that different policies can be used, which we discuss at the end of the section.

Timestamp oracle. The timestamp oracle provides an initial serialization interval for transactions. This can be an entirely local component that simply provides an interval starting from the current local clock value and ending after a fixed initial interval. However, in our instantiation of the protocol, we use a remote oracle service that periodically gathers transaction statistics from clients, and uses this information to increase the quality of timestamp interval assignments. An ideal oracle of this type (which had knowledge of all past and future transactions) could prevent all aborts. However, not even an ideal oracle could remove all waiting. For instance, two read-write transactions operating on the same data cannot execute concurrently: one must wait to read the data until the other has finished writing it.

A practical oracle cannot of course operate under perfect knowledge of past and future transactions. It can however gather statistics on previous executions. In particular, it can estimate how far in the past (compared to the highest timestamp it has issued) it should schedule read-only transactions. Also, if a write-intensive transaction needs to be restarted, based on the intervals issued so far, it can estimate the timestamp interval with which it should be scheduled such that the chances of it failing again are low.

Locking optimizations. We start our transactions with an optimistic request for a small serialization interval. The advantage of a short interval is increased concurrency. The potential disadvantage is that we may fail to find a serialization point for some transactions. It is likely the case that for many operations, a longer interval is available to be locked than the one we actually lock. Therefore, along with the lock, the server also returns this maximum interval to the client. Besides keeping track of the serialization interval for which it has locks, the client also keeps track of this "maximum serialization interval". If at the end of the transaction the serialization interval is empty, but the maximum interval is not, we can re-try the transaction with an interval contained in this final maximum serialization interval. Such a re-try will re-acquire locks for the operations up to the previous abort point according to the new interval (if not possible, the transaction is aborted), and then continue normally.

We want to support long-running transactions with a large number of reads, which then write one or a small number of values at the end. However, it is likely that by the time we process the writes, the serialization interval has converged to a set of values that may not allow the write to proceed. Therefore, we allow the user to "pre-declare" write keys: in such cases, we adjust the initial timestamp of the transaction such that it corresponds to possible serializations of the final write. It is important to note that we do not grab any locks for the pre-declared writes, but merely obtain a hint as to when we could serialize the transaction.

A potential issue is the starvation of write-intensive transactions. We tackle this issue by allowing such transactions to acquire write locks as early as possible.

Choosing intervals and versions for locking. We now discuss the problem of choosing an appropriate interval to lock given the current serialization interval of the transaction. We impose a limit on the size of the timestamp interval rows can be locked for: `max_lock_interval`. The interval a read or write lock is applied to should not span multiple versions, in order to limit unnecessary aborts. If the serialization interval intersects multiple versions, we have to choose for which one we should apply the lock. We can apply the following criteria in order to choose an interval to lock:

- for reads, try to lock a version which is not in *pending* state; this way, we would not have to do any waiting before we serve the read;
- in the case of writes, we cannot create a new version if the `max_read_from` timestamp of the previous version is greater than the right limit of the serialization interval; if we have multiple locking intervals to choose from, we should of course choose one where a new version can be created;
- try to lock the version that maximizes the updated serialization interval; the updated serialization interval after an operation will be necessarily smaller or equal to the serialization interval before the operation; allowing subsequent operations to shrink the serialization interval too much will likely result in an increase in the abort rate of transactions;

Algorithm **??** describes how the version to apply a lock to is chosen.

Read locks. The acquisition of a read lock proceeds as shown in Algorihtm **??**. The *version.next()* function returns the next version of an object. If no such version exists, it returns a special entry whose timestamp is infinity. It is worth noting that the function expects versions that are no loner in progress. Algorithm **??** ensures that a read lock is taken only on versions that are in committed state.

Algorithm 4 Obtaining a version to lock

```
1: function GET_VERSION(key, t_interval, flag)
2:     if flag == READ then
3:         v_set = {ver_key | [t_interval.start, t_interval.end] ∩ [ver_key.ts, ver_key.next().ts] ≠ ∅};
4:         if v_set == NULL then
5:             mark_read_not_found(key,t_interval.end);
6:             return NULL;
7:         end if
8:         sort v_set by (state, length of intersection with t_interval);
9:         v = v_set.pop_first();
10:        if v.state != COMMITTED then
11:            return NULL;
12:        end if
13:        return v;
14:    else if flag == WRITE then
15:        v_set = {ver_key | [t_interval.start, t_interval.end] ∩
    [ver_key.right_ts, ver_key.next().left_ts] ≠ ∅};
16:        if v_set == NULL then
17:            return get_read_not_found_mark(key);
18:        end if
19:        sort v_set by (state, intersection of [max_read_from, interval_end] with t_interval);
20:        v = v_set.pop_first();
21:        return v;
22:    end if
23: end function
```

Algorithm 5 Taking a read lock

```
1: function GET_READ_LOCK(version, t_interval)
2:     left_ts = version.timestamp;
3:     next_timestamp = version.next().timestamp;
4:     right_ts = min(next_timestamp, t_interval.end);
5:     rl_info.locked = [left_ts, right_ts]
6:     rl_info.potential = [version.timestamp,next_timestamp];
7:     if version.max_read_from < right_ts then
8:         version.max_read_from = right_ts;
9:     end if
10:    return rl_info;
11: end function
```

Obtaining write locks. Write locks are taken as shown in Algorihtm **??**. We note that the function *version.next().timestamp* also takes into account pending versions. In this case, it returns the left timestamp of the interval of the pending version. This timestamp should always be greater than the `max_read_from` of the current version. In case no suitable interval is found, we return an entry with a NULL locked interval, which will prompt the client to restart the transaction. We store the newly created versions (essentially the write locks) in volatile storage until they are committed.

Algorithm 6 Taking a write lock

```
1:  function GET_WRITE_LOCK(version, t_interval)
2:      left_ts = max(version.max_read_from,t_interval.start);
3:      next_timestamp = version.next().timestamp;
4:      right_ts = min(next_timestamp, t_interval.end);
5:      if left_ts > right_ts then
6:          wl_info.locked = NULL;
7:      else
8:          wl_info.locked = [left_ts, right_ts];
9:      end if
10:     wl_info.potential = [version.max_read_from,next_timestamp];
11:     wl_info.the_version = new_version(wl_info.locked);
12:     wl_info.the_version.state = PENDING;
13:     add_to_volatile_storage(wl_info);
14:     return wl_info;
15: end function
```

Clock skew. (tentative) We assume that most of the time the machines responsible for the timestamp oracle guarantee a maximum clock skew S between them (perhaps using NTP). In essence, if a remote machine has been issued a timestamp T from the timestamp oracle, it can be sure that no other timestamp oracle will issue a timestamp smaller than $T - S$. In our context, S need not be very fine grain. Such a bound is nevertheless useful in tasks such as garbage collection of versions. It is important to note that this bound is not needed for the safety of the system, but only for performance.

Protocol Variations We have so far focused on a version of the protocol that only accesses the read set once in the common case. It is possible to use variants of the protocol that access the read set twice, or three times. The advantage of these alternatives is that they are able to reduce the interval the read set is locked for to reflect the serialization point which was decided for a transaction, as well as some additional potential optimizations. However, we decided that the savings offered by reducing the number of message exchanges with the read set outweigh a slight reduction in transaction aborts**[TODO: verify this experimentally]**.

We briefly mention these alternatives. In a version with two accesses to the read set, we would go to the servers holding the read locks once the serialization point has been decided, and adjust the read lock to only cover an interval up to the serialization point. Additionally, we can also store all the metadata on separate servers. This would allow us to modify transaction intervals

as a function of other concurrent transactions, but would result in additional communication with these metadata servers. In a version with three read set access, we would run a classical two phase commit for both the read and write sets. In this scenario, the clients would not have to keep track of the state of the current transaction; this information would be kept at the servers.

3.3 Recovery

Coordinator failure. In case of a coordinator failure, we use a protocol similar to the one proposed in Sinfonia []. However, since we allow write locks to be taken during the transaction execution and when committing only send the serialization timestamp (hence not requiring a prepare phase), some changes to the original protocol are necessary (in case writes are buffered and only applied through a 2PC, we can use the unmodified protocol). If a coordinator crash is detected (through a transaction timeout mechanism), an abort coordinator is selected. This abort coordinator attempts to execute a 2PC, where it proposes an abort for the concerned transactions to each storage node. If the node has not committed the transaction yet (recall that the coordinator need only send commit messages to the write set at the end of the transaction), it replies with an "abort ok" message, along with the potential serialization interval. Otherwise, it replies with "already committed" and the serialization point chosen. If the abort coordinator receives at least one "already committed" message, it means that the transaction has already executed its writes and was in the process of broadcasting the commit decision and the serialization point. Thus, the abort coordinator broadcasts a commit together with the serialization timestamp received with an "already committed" message. If no "already committed" is received, the abort manager broadcasts an abort. Once a node has responded with an "abort ok" or "already committed" to the abort coordinator, it ignores messages from the original coordinator regarding the transaction. We use the same mechanism to detect an abort coordinator failure. A new abort coordinator can try to continue the abort operation once it identifies the nodes that are part of the transaction.

Storage node failure. In case a storage node fails, pending transactions (for which data is stored in volatile memory) are aborted. If the underlying systems (the data store and lock manager) support replication, the failed node can be replaced by one of the replicas. If stable storage is not affected, the failed node can be recovered using persistent data stored in the data store partition and the lock manager.

4 Component Interfaces

In this section, we discuss the interfaces between the various components of our system.

We use largely functions already employed in the algorithms presented in Section **??**. We begin by describing the interfaces between the various client components, after which we focus on the server interfaces.

4.1 Client Interfaces

The interfaces between the client-side components are illustrated in Figure **??**. It is worth noting that the Transaction Manager component can in fact handle transactions from multiple concurrent data store clients. In order to support transactions, we provide the data store client with additional

operations to start, end, and abort a transaction, as well an operation notifying the client of the result of a recently finished transaction.

The timestamp oracle is used to obtain serialization intervals, and is also notified of the outcome of transactions.

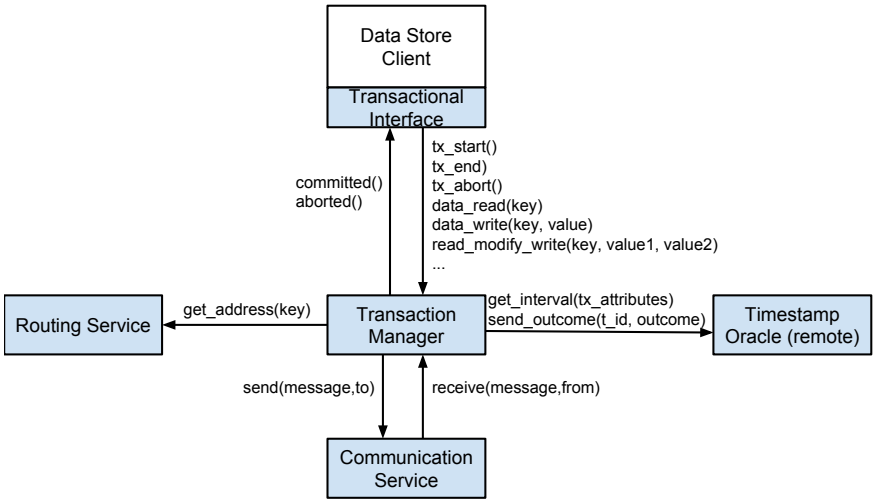

Figure 3: Interfaces between client components

4.2 Server Interfaces

The server interfaces are depicted in Figure **??**. The scheduler is the component orchestrating the operations and handling messages from clients. It is worth noting that we do not require a transaction mechanism from the existing data store. The operations we are able to provide the client within a transaction do however depend on the data store interface. We also note that in order to be able to garbage collect old versions, we need to be able to iterate over the data versions. While this could be provided by either the lock manager or the data store partitions, it is more efficient if the former would provide iterations (since we don't need to iterate over the values as well). The garbage collector also needs to periodically obtain a bound on the minimum timestamp that can be issued from the timestamp oracle.

5 Implementation

5.1 Main data structure

5.2 Handling concurrent connections on the server

5.3 Handling concurrent connections on the client

5.4 Interacting with the underlying data stores

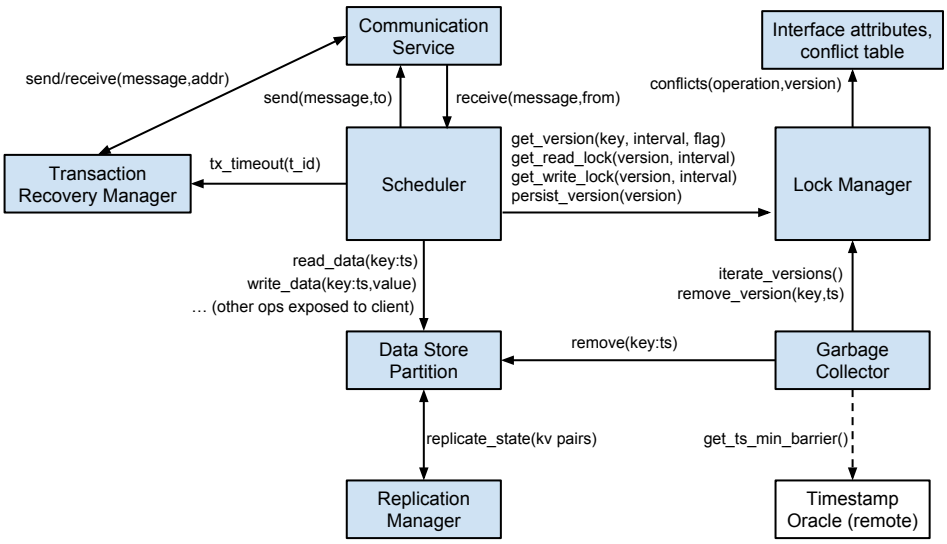

Figure 4: Interfaces between client components