Project no. 609551
Project acronym: SyncFree
Project title: *Large-scale computation without synchronisation*

European Seventh Framework Programme

ICT call 10

Deliverable reference number and title:	D.2.2.2 CRDTs and CRDT composition in partial-replication setting
Due date of deliverable:	April 1, 2015
Actual submission date:	April 1, 2015
Start date of project:	October 1, 2013
Duration:	36 months
Name and organisation of lead editor for this deliverable:	Technische Universität Kaiserslautern
Revision:	0.1
Dissemination level:	PU

Contents

1 Executive summary

This report, Deliverable D.2.2.2, accompanies the software deliverable including the code for the project's reference platform. In this report, we give an overview of the software deliverables. We describe the general architecture and the interaction between the components. We also give hints on how to install, deploy and run the software.

The software for this deliverable has been developed as part of our common research platform, Antidote. The initial version of Antidote, delivered at M12 in Deliverable D 2.1, comprised a geo-replicated causally-consistent data store with transactional support for CRDTs. The underlying protocol is an extension of the ClockSI protocol for multi-DC support and partitioning [3]. During the last reporting period, we have developed and implemented the following extensions to the Antidote platform.

Reference protocols To study the advantages and usability of different protocols for supporting causally consistent data stores, we have implemented two other protocols for geo-replicated data stores from the literature in the Antidote platform: Gentlerain [4] and Eiger [6]. Their implementation required only few changes in the interface and logging layer; the transaction and replication layer have been re-implemented for each reference protocol.

Having implementations of these protocols on the same platform allows us to run benchmarks against the same interface, and to evaluate their differences using the same programming language and runtime environment.

Partial replication protocol The protocols described in the previous paragraph assume that each DC instance replicates all entries in the data store. Antidote also runs now a new protocol which we designed especially for this deliverable, named Charcoal, supporting partial replication. This means that a DC replicates only (potentially overlapping) subsets of the universe of data. The Charcoal protocol builds on a number of components from the first Antidote prototype, such as logging and materialization of CRDT objects, and extends the framework with its specific consistency, transaction and replication components.

Bounded Counter CRDT Antidote supports besides the PNCounter CRDT allowing arbitrary numbers of increments and decrements now also a Bounded-Counter CRDT. This bounded CRDT is able to maintain a non-negative value by explicitly exchanging permissions between DCs to execute decrement operations.

Protocol Buffer interface Similarly to riak, Antidote now provides a Protocol Buffer interface [1] to support the development of data store clients. Using this standardized message serialization format, clients can interact with the data store in a simple way. The interface has been implemented so far on the server side with

[1] Protocol buffers are a language-neutral, platform-neutral, extensible mechanism for serializing structured data, developed by Google. Since its initial release in 2008, protocol buffers have been widely adopted for communication in distributed systems.

an Erlang client library. Supporting other client programming languages, such as Java, will be added in the future.

Applications and Benchmarks To evaluate the Antidote platform, we implemented a selection of benchmarks and applications. Besides synthetic read and write workloads on random selections of objects, we implemented the wallet application selected in Work Package 1 (WP1) as use case, and a prototypical social network application. For the evaluation, we employ tools that were developed the WP5 deliverable D.5.1., such as `basho_bench` or deployment scripts for Amazon Web Services (AWS EC2).

Adaptive replication Building on components from the Antidote platform, we have also developed a prototype of Adaptive Replication, described in **(author?)** [1]. It provides a directory service, which maintains information about the distribution of replicas across the DCs. At each DC, a simple KV store serves as backend for the adaptive replication module. A replica manager forwards read requests and updates to either the local data store or to a remote DC. Our prototype version currently does not support transactions, but allows us to focus on the replication strategies, role of parameters and the different consistency requirements of the directory component as we described in the WP2 deliverable D.2.2.1.

All software can be found on the project's open source repository at

https://github.com/SyncFree/.

2 Milestones and Tasks

WP2 has reached the following milestones:

Mil. no	Milestone name	WP	Date due	Actual date
S1	CRDT consolidation in a static environment	WP2	M12	M12
S2	Extended guarantees and composition in a dynamic environment	WP2	M24	M18

The corresponding tasks are:

Task no	Task name	Date due	Actual date	Leader
D.2.1.1	Protocols for CRDTs in small-scale full replication	M6	M12	KL
D.2.1.2	Platform for CRDTs in small-scale full replication	M6	M12	KL
D.2.2.1	Protocols for CRDTs and CRDT composition in partial-replication setting	M12	M18	KL
D.2.2.2	Platform for CRDTs and CRDT composition in partial-replication setting	M12	M18	KL

Shifting of target dates Several of the main developers on WP2 could only be recruited and employed in February 2014. This caused a delay of several months. Familiarization with the project led to another delay of some weeks. Thus, the design and development of Antidote started effectively in March/April 2014. The SyncFree EB decided to shift the M6 and M12 deliverables for WP2 by 6 months.

3 Contractors contributing to the Deliverable

The following contractors contributed to the deliverables

3.1 KL

Annette Bieniusa, Deepthi Akkoorath.

3.2 INRIA

Alejandro Tomsic, Tyler Crain, Marc Shapiro.

3.3 Louvain

Manuel Bravo, Zhongmiao Li.

3.4 Nova

Diogo Serra, Nuno Preguica.

3.5 Trifork

Amadeo Ascó.

4 Results

In the following, we give a short introduction to the individual software deliverables in D2.2.1.

4.1 Antidote

Antidote is a geo-replicated CRDT data store which features scalable, conflict-free implementations of transactions, by providing consistent, stable snapshots and atomic multi-CRDT updates. It provides Transactional Causal+ Consistency, offering the following guarantees:

- A transaction reads a causally consistent snapshot;
- Updates of a transaction are atomic (all-or-nothing) and durable (later transactions observe this transaction's updates); and
- Concurrently committed updates do not conflict (CRDT property).

Under Transactional Causal+ Consistency, clients can observe the same set of concurrent updates, but applied in different orders. To avoid divergence, we assume that concurrent updates are mergeable (CRDT approach). Mergeable transactions commute with each other and with non-mergeable transactions, which allows to execute them immediately in the cache, commit asynchronously in the background, and render the system available even in failure scenarios. For the Antidote data store, we offer mergeable transaction in the form of read-only transaction or update transactions that modify CRDTs. Future work (in collaboration with WP3) will extends mergeable transactions with some form of synchronization to maintain invariants.

The code for Antidote is available at

`http://github.com/SyncFree/antidote`,

including test cases, example applications and benchmarks.

Architecture One of the design goals of Antidote is a layered architecture with a clear separation of concerns. This will allow to experiment with different approaches for specific tasks. For instance, the Transaction Layer is in charge of implementing protocols for retrieval and commit of objects and their updates according to the respective transaction semantics. Examples for adaptations of Antidote are the reference protocols and the Charcoal protocol, which we describe in detail in later sections.

Figure 1 shows the layered architecture of Antidote.

Log Layer Antidote uses a log-based backend to provide fast and fault-tolerant write access and efficient management of multi-versioning for CRDT objects. The log layer immediately accepts all operations it receives. At any DC, each partition maintains its own log.

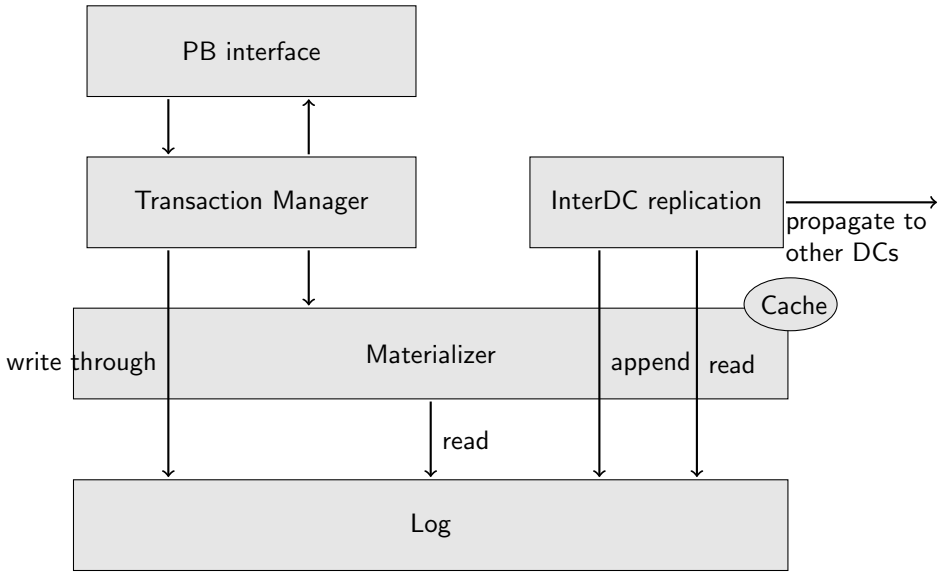


Figure 1: Architecture of Antidote.

Materializer The responsibility of the materializer is to generate snapshots of objects by applying operations and to improve performance by caching operations. To achieve these goals, each partition runs a materializer process with an in-memory cache, represented in the form of a key-value store mapping the key to recent versions of a CRDT object.

Transaction Manager The transaction layer coordinates read and update operations involving multiple partitions. The current standard implementation provides Transactional Causal+ Consistency semantics with mergeable transactions.

InterDC Replication Layer This communication layer is responsible for delivering every successfully committed transaction to other DCs as well as receiving and applying transactions from remote DCs at the local DC. Since applying updates requires consistency checking, which depends on the transactional semantics, this module is coupled with the transaction manager. We currently provide several implementations for the InterDC layer (and corresponding transactional manager) to support ClockSI, GentleRain and Charcoal.

PB Interface Clients interact with Antidote using a Protocol Buffer interface. This layer exposes single object read and write as well as multi-object read/write with transactional semantics, as described above, to the clients. A client library in Erlang is also provided. Section 4.1.2 contains an overview of the interface and presents an example of how an Erlang client program can access Antidote.

4.1.1 How to install and test Antidote

- Prerequisites

    1. An Unix-based OS.

create_crdt	(*key,type*)	→ *ok*
get_crdt	(*key, type*)	→ *crdt*
store_crdt	(*crdt*)	→ *ok*
snapshot_get_crdts	([{*key,type*}], *clock*)	→ [*crdt*]
atomic_store_crdts	([*crdt*])	→ {*ok, clock*}
begin_txn	(*clock*)	→ *txnId*
get_crdt_txn	(*key, type, txnId*)	→ *crdt*
store_crdt_txn	(*crdt, txnId*)	→ *ok*
commit_txn	(*txnId*)	→ {*ok,clock*}

Table 1: Client API of Antidote

2. Erlang R16B02

- Getting Antidote
  From your shell, run: *git clone http://github.com/SyncFree/antidote*
- Building Antidote
  1. Go to the antidote directory.
  2. Run *make stagedevrel*.
- Setup `riak_test` for Antidote
  1. Clone `https://github.com/SyncFree/riak_test` into another directory, called here RIAK_TEST.
  2. Switch to the features/csm/antidote branch:
     git checkout features/csm/antidote
  3. Change to the RIAK_TEST directory and run *make*.
- Running tests To execute all tests, go to the Antidote directory and run *make riak-test*. Alternatively:
  1. Run to set-up the test environment *./riak_test/bin/antidote-setup.sh* (Only for the first time)
  2. Run *./riak_test/bin/antidote-current.sh* to start the data store
  3. Go to the RIAK_TEST directory
  4. Run an invidiual test with *./riak_test -v -c antidote -t "TEST_TO_RUN"*
     TEST_TO_RUN is any test module in antidote/riak_test/
     eg:- ./riak_test -v -c antidote -t clocksi_test

4.1.2 Interface

Antidote allows clients to access and update objects in the data store using the API shown in Table 1. The API is exposed using the Protocol Buffer interface.

- *get_crdt*: reads an object stored under the *key* provided. The *type* parameter denotes the CRDT type of the object.

- *store_crdt*: stores the updates to the CRDT object on the Antidote server.
- *snapshot_get_crdts*: reads multiple objects from a consistent snapshot satisfying *transactional causal+ consistency*. The parameter is list of key-type tuples for the respective objects. The optional parameter *clock* denotes the last timestamp known to the client. It is used to guarantee *monotonic reads* and *read your own writes* even when the client connects to different data centers.
- *atomic_store_crdts*: stores updates on several CRDTs atomically at the data store. It guarantees that the updates become visible together. It returns the committed timestamp of the transaction.

In addition to the above interface, Antidote also provides an API for interactive transactions where clients can start a transaction, execute multiple read and update operations and then commit. All reads and writes apply to the same snapshot.

Example We have implemented a client library written in Erlang for client programs to access Antidote based on the Protocol Buffer Interface (`https://github.com/SyncFree/antidote_pb`). The following code snippet shows an example of how an Erlang client can use the library to access Antidote:

```
{ok, Pid} = antidote_pb_socket:start(?ADDRESS, ?PORT),
{ok, C} = antidote_pb_socket:get_crdt(Key1, riak_dt_orset, Pid),
{ok, S} = antidote_pb_socket:get_crdt(Key2, riak_dt_pncounter, Pid),
C1 = antidote_set:add(1, C),
S1 = antidote_counter:increment(2, S),
{ok,_} = antidote_pb_socket:atomic_store_crdts([C1,S1],Pid),
Result = antidote_pb_socket:snapshot_get_crdts([{Key1,
        riak_dt_orset}, {Key2,riak_dt_pncounter}], Pid),
{ok, _, [Set1, Counter2]} = Result,
antidote_pb_socket:stop(Pid)
```

After opening a local Protocol Buffer socket, an ORSet CRDT and a PNCounter CRDT are read. Next, an element is added to the ORSet object, while the PNCounter is incremented by 2. Then, both updates are submitted atomically to the data store server. The following snapshot read should retrieve both updates together. Finally, the socket is closed and the listening process is stopped.

4.1.3 Additional Features

Bounded Counters In addition to standard CRDT objects, Antidote also support invariant preserving CRDTs, namely bounded counters. Bounded counters preserve the invariant that the value of the counter will never be negative, even with concurrent asynchronous decrements. To this end, bounded counters utilize the reservation technique developed in WP3 (see D3.1 and [2]). In addition to standard counter operations such as *increment* and *decrement*, the bounded counter exposes an *transfer* operation which transfers the reservation to other replicas. A client that tries to decrement the counter value without having sufficient reservations will receive an error response, upon which it can request a reservation transfer operation. If the transfer operation is successful, it can re-issue the decrement operation.

Reference protocols The base version of Antidote, delivered at M12 in D2.1, is built on a protocol for interDC replication which preserves Transactional Causal+ Consistency for mergeable transactions on CRDTs.

In order to compare with state-of-the-art, we have now implemented other similar protocols from the literature in the Antidote framework. With few small changes in the interface and logging layer, Antidote now offers alternative transaction and replication protocols. We have implemented two recent causal consistency protocols for a geo-replicated partitioned database: GentleRain [4] and Eiger[5]. We adapted the GentleRain protocol slightly to support causal delivery of updates and support the operation *atomic_store_crdts*, because the original protocol only supported snapshot reads and single writes. In addition, we have implemented a variant of Antidote that provides only eventual consistency guarantees, to evaluate the overhead in latency and meta-data size incurred by causal consistency. The code for the three reference protocols is available at:

- `https://github.com/SyncFree/antidote/tree/ec_antidote`
- `https://github.com/SyncFree/antidote/tree/gentlerain`
- `https://github.com/SyncFree/antidote/tree/eiger`

4.1.4 Benchmarking

In collaboration with WP5, we implemented and adapted a number of tools for benchmarking the Antidote platform. These tools include scripts for deploying Antidote on the Amazon Webservice EC2, generating generic and application specific workloads, as well as collecting, processing and visualizing the results with `basho_bench`. This allows for exploration of the throughput, latency and scalability in different scenarios, varying the number of clients, DCs, read and write operations, etc.

The tools are described in the WP5 deliverable D5.1 in more detail. Here, we will only show an example how they can be applied.

Experimental setup We use the benchmark framework `basho_bench`[2]. The benchmark is configured to use 100 simulated clients, evenly distributed among Antidote instances. The benchmark accesses 2000 keys with Pareto distribution. `basho_bench` issues transactional operations to Antidote. In our use case here, each read or write transaction is indeed a transaction reading or updating a single key. The read/write ratio is 10:1. We run all our experiments with Amazon EC2 instances. Each Antidote replica (which can be either a partition of a data center or can be a whole data center, which will be explained later) are hosted on an EC2 m3.large instance, while `basho_bench` runs on an EC2 c4.xlarge instance.

Throughput and latency of a single Antidote instance In the following, we show the throughput and latency of a single Antidote instance (i.e., a data center). Figure 2 shows that the throughput of Antidote under the described workload is

[2]https://github.com/basho/basho_bench

about 4000 transactions per second and it is stable over a three-minute duration. Figure 3 gives the mean, median and 95th percentile latency for append (i.e. update) and read, respectively. The latency is also stable over the duration of the benchmark.

Figure 2: Throughput of an Antidote instance.

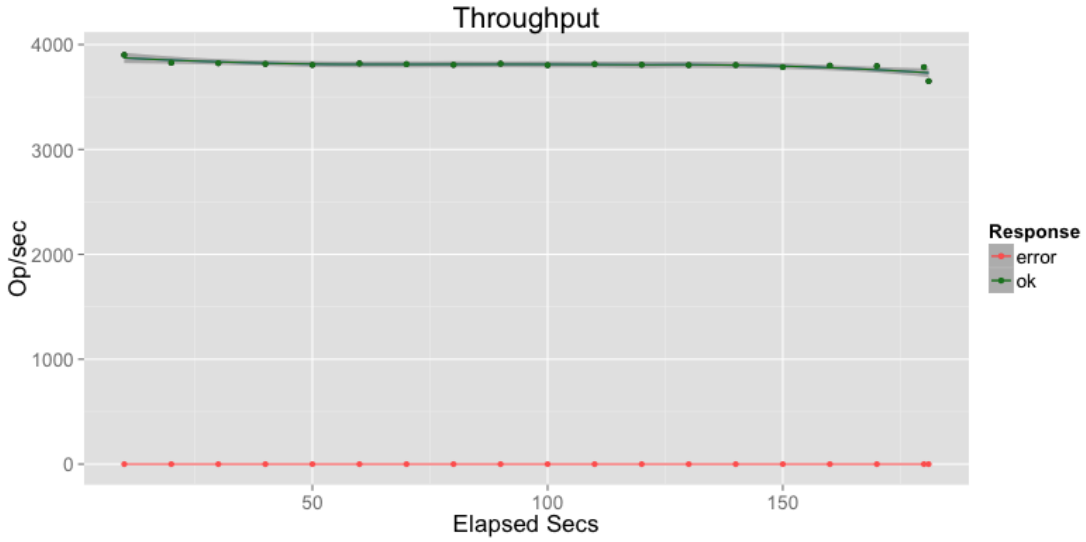
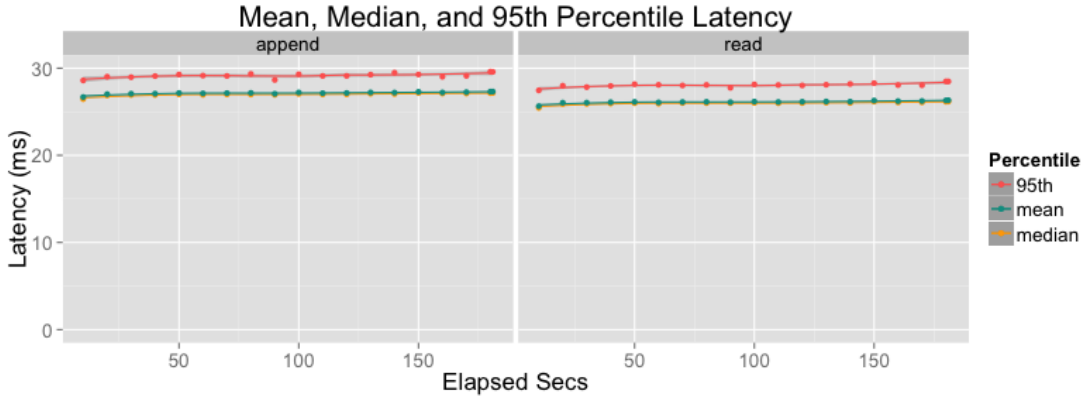


Figure 3: Latency of append and read operations.



Scalability We also evaluated the scalability of Antidote w.r.t. the number of partitions of a single data center and the number of Antidote instance. The following table gives the throughput of a single Antidote data center with different numbers of partitions.

Number of partitions	1	2	4	8
Throughput(txn/s)	3812	6115	11005	14922

The throughput of Antidote scales linearly with the number of partitions up to 4 partitions. However, scalability is reduced when adding more partitions. A careful investigation of the log files showed that in this specific run this problem happened due to a high clock skew on a single machine that forces the ClockSI transaction protocol to wait. A transaction started in a partition with a fast clock waits for another partition with a slower clock. In order to overcome this issue, a solution would be to force the transaction to read from an older snapshot by assigning older timestamp as its snapshot time.

The next table shows the overall throughput of multiple connected Antidote data centers, each consisting of a single partition/machine.

Number of datacenters	1	2	4	8
Throughput(txn/s)	3812	7651	14831	23636

Here, Antidote scales almost linearly with the number of data centers. This is expected, because our inter-DC causal consistency protocol is asynchronous, therefore it allows DCs to process requests individually without interaction with other DCs. Updates are propagated asynchronously in the background.

4.2 Partial Replication

Antidote's base protocol from D2.1. does not support partial replication. For this deliverable, we developed a new protocol, named Charcoal, in the Antidote framework. To support partial replication, some components had to be redesigned. This section describes how the Antidote code base was modified to support partial replication. The source code can be found on the SyncFree GitHub repository at

`https://github.com/SyncFree/antidote/tree/partial_replication/`.

The data store with partial replication can be built and tested using the same interface and tools as the version with Antidote's base protocol.

Unmodified components The log and materializer were left unmodified. Although, not all objects are logged and materialized at any given DC, this distinction is handled at the higher layers, thus allowing the log and materializer code to remain the same.

Modified components Inside a DC, the transaction manager ensures that a transaction accesses and update consistent view of the data. The transaction layer consists of two main components, one for transaction execution, through which a client interacts with the system, and one for an ordering component, which assigns timestamps and vector clocks to maintain an ordering locally and to keep track of dependencies between transactions.

Since the transactions under partial replication provide the same semantics as under the full replication protocol, major parts of the code for transaction execution remained kept the same. The main modification is to allow the transaction to read and update objects not replicated at the DC where the transaction is being executed. When performing a read operation on a locally non-replicated key, the

read request is forwarded to another DC that does replicate the key, The returned result is cached at the local DC. Updates to non-replicated keys are also stored locally to ensure durability under network partitions, but are flagged to ensure they are included in future reads and are materialized correctly. Like in full replication, a transaction receives a scalar timestamp assigned at the local DC and use a vector clock to track dependencies. However, this dependency vector clock is now assigned by the safe-time collector (a new component described below).

The interDC replication component also needed modification to support partial replication. At the sending DC before propagating an update, the sender checks which DCs replicate the update, using the replication check component (a new component described below). The receiving DC may partition keys differently than at the sender, therefore, the updates of the transaction are forwarded to their correct partition. Whereas, Antidote's base protocol uses hearbeats to keep track of when data is safe to read, in Charcoal, the safe-time collector is used.

New components The replication check component maps a key to the set of DCs where the key is replicated. In the Charcoal protocol, the replication scheme can be defined specifically for each key, or can use a pre-defined function.

At each DC, a new process listens for incoming read requests from other DCs. These DCs may request materialized objects for keys that they do not replicate themselves. These read request are different from normal reads performed directly by clients, as the reply includes specific meta-data for replica management that should not be sent to the client.

Finally, a safe-time collector runs at each DC. At fixed intervals, it asks every server in its DC to return the maximum time for which this server has sent all completed updates to the other DCs. It then computes and sends safe time notifications to each external DC, allowing new transactions to safely read these updates.

4.3 Adaptive Replication

We implemented a prototype for our work on adaptive replication, we developed a prototype for supporting adaptive replication in geo-distributed key-value stores. Re-using major parts of the Antidote platform, we designed several components and interfaces which together manage the placement of replicas among DCs. Figure 4.3 sketches the organization of the system for a single DC.

The *client interface* accepts and answers requests from clients. It is responsible for the session coordination. Depending on the internal DC-local structure of the data store (e.g. partitioning), it forwards the incoming requests to an internal server, here named *replication coordinator*.

The replication coordinator deals with all information regarding replication of objects. It is parametrized by a *strategy* for installing or removing local object replicas, based on the frequency of (local) read and (local and global) write requests. Accordingly, it either processes the request locally, or it forwards the request to a DC that replicates the object.

Information about replica placement is maintained by a global directory service (not represented in the figure).

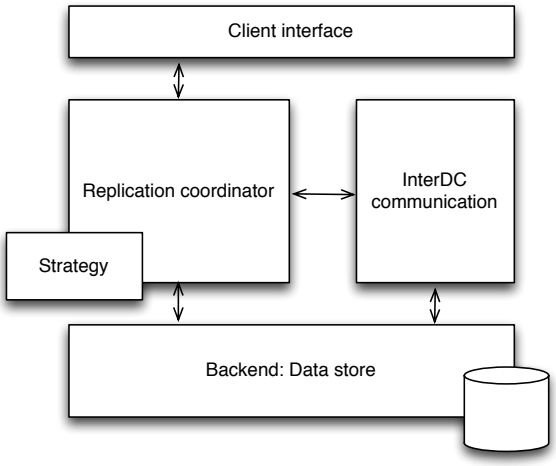


Figure 4: Architecture of the Adaptive Replication component.

The *interDC communication* forwards requests to other DCs and processes the response, possibly returning information to the replication manager. As in Antidote's base version, it also listens for updates to local replicas that have been issued at other DCs and applies them to the local data store. Similarly, it also forwards local updates to the other DCs.

Status This proof-of-concept prototype is intended to investigate which components needed to be modified and adapted to support adaptive replication in a data store.

Our prototype platform can be found at

`https://github.com/SyncFree/adpreplic`

The code base currently contains only the basic functionality and is currently still under development. In particular, it is missing important features to make it applicable in practice. For example, the directory containing the placement information for the objects is needs to be implemented in partition- and fault-tolerant way.

References

[1] Amadeo Ascó and Annette Bieniusa. Adaptive strength geo-replication strategy. In *Proceedings of the Workshop on Principles and Practice of Consistency for Distributed Data*, PaPoC '15, New York, NY, USA, 2015. ACM.

[2] Valter Balegas, Diogo Serra, Sérgio Duarte, Carla Ferreira, Rodrigo Rodrigues, and Nuno M. Preguiça. Technical report.

[3] Jiaqing Du, Sameh Elnikety, and Willy Zwaenepoel. Clock-si: Snapshot isolation for partitioned data stores using loosely synchronized clocks. In *Proceedings of the 2013 IEEE 32Nd International Symposium on Reliable Distributed Systems*, SRDS '13, pages 173–184, Washington, DC, USA, 2013. IEEE Computer Society.

[4] Jiaqing Du, Calin Iorgulescu, Amitabha Roy, and Willy Zwaenepoel. Gentlerain: Cheap and scalable causal consistency with physical clocks. In Ed Lazowska, Doug Terry, Remzi H. Arpaci-Dusseau, and Johannes Gehrke, editors, *Proceedings of the ACM Symposium on Cloud Computing, Seattle, WA, USA, November 03 - 05, 2014*, pages 1–13. ACM, 2014.

[5] Cheng Li, Daniel Porto, Allen Clement, Johannes Gehrke, Nuno M Preguiça, and Rodrigo Rodrigues. Making geo-replicated systems fast as possible, consistent when necessary. In *OSDI*, pages 265–278, 2012.

[6] Wyatt Lloyd, Michael J. Freedman, Michael Kaminsky, and David G. Andersen. Stronger semantics for low-latency geo-replicated storage. In Nick Feamster and Jeffrey C. Mogul, editors, *Proceedings of the 10th USENIX Symposium on Networked Systems Design and Implementation, NSDI 2013, Lombard, IL, USA, April 2-5, 2013*, pages 313–328. USENIX Association, 2013.