Report on STSM at INRIA-Regal

Valter Balegas

CITI - Dep. Informática, FCT,
Universidade Nova de Lisboa, Portugal
v.sousa@campus.fct.unl.pt

COST Action	IC1001
Reference Number	ECOST-STSM-IC1001-110912-021982
Action Description	Report on STSM at INRIA-Regal
STSM Period	from 11-09-2012 to 11-10-2012
Participant	
	Valter Balegas de Sousa
	CITI - Faculdade de Ciências e Tecnologia
	Caparica, Portugal
Host	
	Dr. Marc Shapiro
	LIP6/UPMC & INRIA
	Paris, France

1 Purpose of the STSM

Conflict free replicated data types (CRDT) have been proposed as a technique for allowing a set of replicas to be modified concurrently without coordination, and still guaranteeing the convergence of replicas. These data types have been used to provide data-replication in large scale systems. New designs of these data types provide transactions with a consistency level weaker than *snapshot isolation*. In the last few years, a large number of research projects have been pushing the limits of transaction memory (TM) in a number of directions. One interesting direction is the application of transactional memory systems in distributed settings, where a set of objects is shared by nodes (typically) in a cluster. In this setting, the latency of communication becomes a major hurdle for efficient transactional memory systems. We envision the possibility of relying on CRDTs, as a form of transactional boosting mechanism for distributed TM systems. In this mission, we intend to make the first steps in the study of the viability of such approach, and if time allow, start the initial design of such a system (or study the integration of these techniques in an existing solution).

2 Description of the work carried out during the STSM

This short term scientific mission was inserted in the ConcoRDanT research project at INRIA-Regal group. The aim of the project is investigation of Conflict-free Data Types (CRDT)[3], which guarantee (eventual) consistency[6], with no complex concurrency control and no rollback, and scale to large-scale distributed systems, such as P2P and Cloud systems.

We wanted to build a system that provides transaction in a large scale geo-replicated system with low latency to the end user.

Our system is called SwiftCloud and it has wide scope of functionalities such as different *isolation levels*, a cache layer and a notifications system to avoid contacting the data centre to update the cache.

The system uses a rich data-model based on CRDTs, which provides conflict free replication to client. Updates can be processed asynchronously removing any bottlenecks on commutation with eventual convergence and correctness.

The work carried in this scientific mission was to develop an application that would allow us to study the advantages and drawbacks of the SwiftCloud system, to study the benefits and drawbacks of different *isolation levels* and to evaluate the performance of the built system. We implemented TPC-W benchmark for that purpose and proceeded with the evaluation of the system. During the period of the mission we could not make a deeper research on the trade-offs of different *isolation levels*, due to the lack of time. We will continue that study after the STSM. However, this work is crucial to continue our research, as it provided us with a tool that will allow to study how *isolation levels* interfere in the operations semantic and performance. The evaluation carried in this mission was used in a article submitted to EuroSys'13.

2.1 Organization

This section of the report is organized as follows. In section 2.2 we will introduce CRDTs and present the SwiftCloud system. Section 2.3 presents the TPC-W benchmarks and its implementation using CRDTs. Section 2.4 presents the evaluation results and we conclude in section 3, discussing the outcome of the mission.

2.2 System Overview

The Conflict-Free Replicated Data Types (CRDTs) are a family of data types that can be updated without synchronization and still provide state convergence. Their asynchronous nature make them very suitable to provide replication in eventual consistency environments, allowing to provide scalability and fault-tolerance in large scale distributed systems.

The CAP theorem [1] states that is not possible to archive, simultaneously consistency, availability and partition-tolerance, in a distributed system. However, it is possible to pick two of those properties without huge prejudice to the latency. The eventual consistency model sacrifices consistency to provide both availability and partitioning-tolerance. However, eventual consistency poses an important drawback: executing operations without coordination between replicas, can originate conflicts that must be resolved. CRDTs tackle that problem in a systematic, theoretical proven approach, based on simple mathematical rules, by providing automatic reconciliation. Furthermore, they satisfy Strong Eventual Consistency Model [4] and can be used as building blocks of other data types that are suitable for programmer's applications.

Currently there is available a CRDT library [**?**] that provides counters, registers, sets, maps, graphs and sequences [7] that can be used to build applications. Next we will describe a Set and Counter CRDT to get the insight on CRDTs.

Counter A counter is an integer with two operations: increment and decrement. The value reflects the difference between the number of increments and decrements on it (this can be easily extended to support add/subtract operations).

The CRDT Counter is inspired by vector clocks, we call it *increment only counter*. We store the number of increments for each replica indexed by position in a vector. The query operation retrieves the sum of every vector position. When we have two replicas of the same counter that were concurrently updated and we want their state to converge, we selects the maximum value for each index in the vector for both replicas and creates a new Counter with that vector. To allow decrement operations we can combine two *increment only counters*, one for counting the increments and other to count the decrements. In this case, the value is the difference of the two counters.

Set Sets are abstract data types that are used in many applications. For example, sets can be used to store a shopping cart, to store your friends in a social

network or in many other situations. The minimal interface of a set is composed by the following operations: *add(e)*, *remove(e)*, *contains(e)* and *elements*. *add* and *remove* have the typical meaning, *contains* verify if an element e belongs to the set and *elements* retrieve the elements of the set.

The idea of OR-Sets, or Observed-Remove Set, is to control the visibility of an element according to the precedence of *add* or *remove* operations, when concurrent operations are issued for the same element. To enable it, *add* and *remove* operations associate identifiers that have the *total order relation* (e.g. Timestamps) to the elements. There are different implementations of OR-Sets [?]. For the sake of simpleness we will explain the state based Add-Wins OR-Set.

The Add-Wins OR-Set gives precedence to the *add* operation, i.e., concurrent operations over the same element e will make the element belong to the set if at least one of the concurrent operations is an *add(e)*.

To implement the Add-Wins OR-Set we need a Set S of elements and a set T of tombstones. The Set S stores (element, identifier) pairs and T is a set of identifiers. When an element is added to the set, it is stored with a new identifier. If an element e is removed, the identifiers associated to that element are moved to the tombstones and the element removed from the set. When we synchronize the state of two replicas, all elements of S in one replica that have their identifier in the Tombstones set of the other replica are removed.

The SwiftCloud System SwiftCloud is a system built on top of CRDTs that provide a transactional-based consistency model that ensures *atomicity*, *causal consistency* and *isolation*. Transactions are not serializable to avoid synchronism. We assure *isolation* by always accessing a snapshot of the database and transactions never abort. Our system does not need to abort transactions because CRDTs never originate write/write conflicts, however a transaction may be lost due to a client failure or must be handled by different node if one fails.the flavorthe flavorthe flavor

Atomicity means that if a transaction T can observe some update made by T', then it observes all updates of T'. Thus, in the social network example, any transaction that observes an arc (A, B) in the friendship graph also observes arc (B, A) in the inverse graph. Causal consistency ensures that any update includes updates observed earlier. Thus, for instance, if a transaction observes arc (A, B) and adds arc (B, C), any transaction that observes (B, C) must also observe (A, B) (unless, of course, another transaction removed it in the interval). Session guarantees ensure that application observe monotonically growing state, e.g., if a transaction observes node B and adds node C, the subsequent transactions of the application observe both.

SwiftCloud is a two tier infra-structure, as shown in figure 1. In the inner tier of the system, a small set of data centres replicate the full database, which can be geographically distributed. The outer tier is composed by a set of scouts that act like caches for the end clients. Scouts can be placed anywhere. They can be installed close to the client, providing data quickly as possible, or they can be installed in CDNs, providing data with low latency to clients near it, or even at

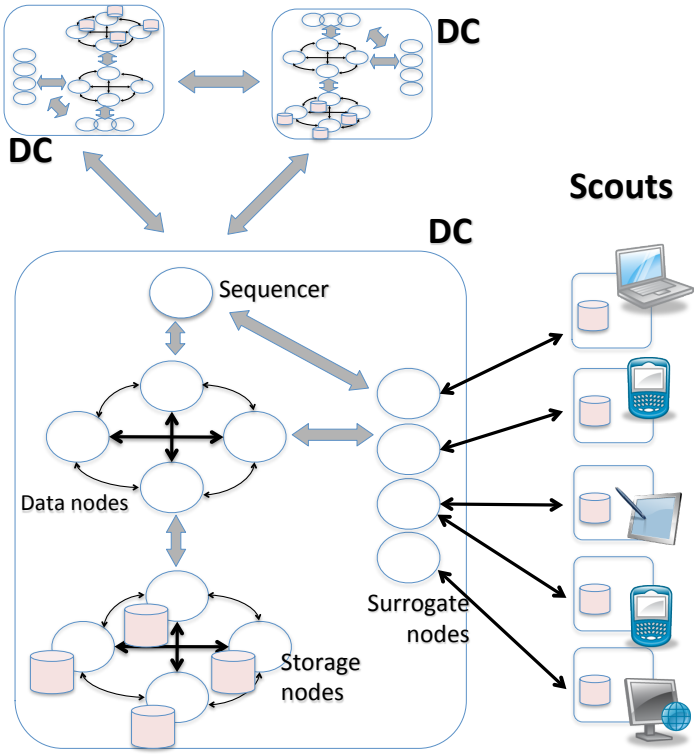


Fig. 1: SwiftCloud architecture diagram

the data centre. Scouts commit transactions locally and can start another while they send the completed ones asynchronously to the data centre.

The system uses a simple Key-Value store interface with put and get operations. Objects are versioned and uniquely identified by a key, which organize them in buckets. To execute a transactions, the client must start a new transaction specifying if it is read only, the *isolation* level and the caching policy. Currently the system supports two *isolation levels*, repeatable reads and *snapshot isolation*. Depending on the *isolation* level, the scout may serve requests locally (if they are cached) or request them from the data centre. When we cache objects in the database, we must keep them fresh in order to avoid working on stale data or fetch the more recent objects according to the versions required by the transaction. To maintain values fresh, the data centre delivers updates of objects that the scout subscribed. This mechanism allows the programmer to control the quantity of updates he wants to subscribe.

2.3 TPC-W benchmark

TPC-W simulates an online book store. The original TPC-W includes a set of operations that simulate the user's interactions through a web application with a graphical interface. In this implementation, we only simulate the data access executed from the application. The implemented operations are described in table 1. Most of these operations require access to the primary key of objects which can be done with good performance on a key-value store, as we can store data indexed by key. However, some operations require more complex queries that would benefit from the use of secondary indexes and other mechanisms normally present in DBMSs.

The TPC-W data-model is well suited for a relational database and it specifies the minimum tables the system should implement. We show the database schema in figure 2. The table Order registers the clients orders, order_line the items from a particular order and cc_xacts represents the payment of an order. The other tables store information for the customers, addresses, countries, items and authors. Some additional tables may be used to store shopping cart information or any required data.

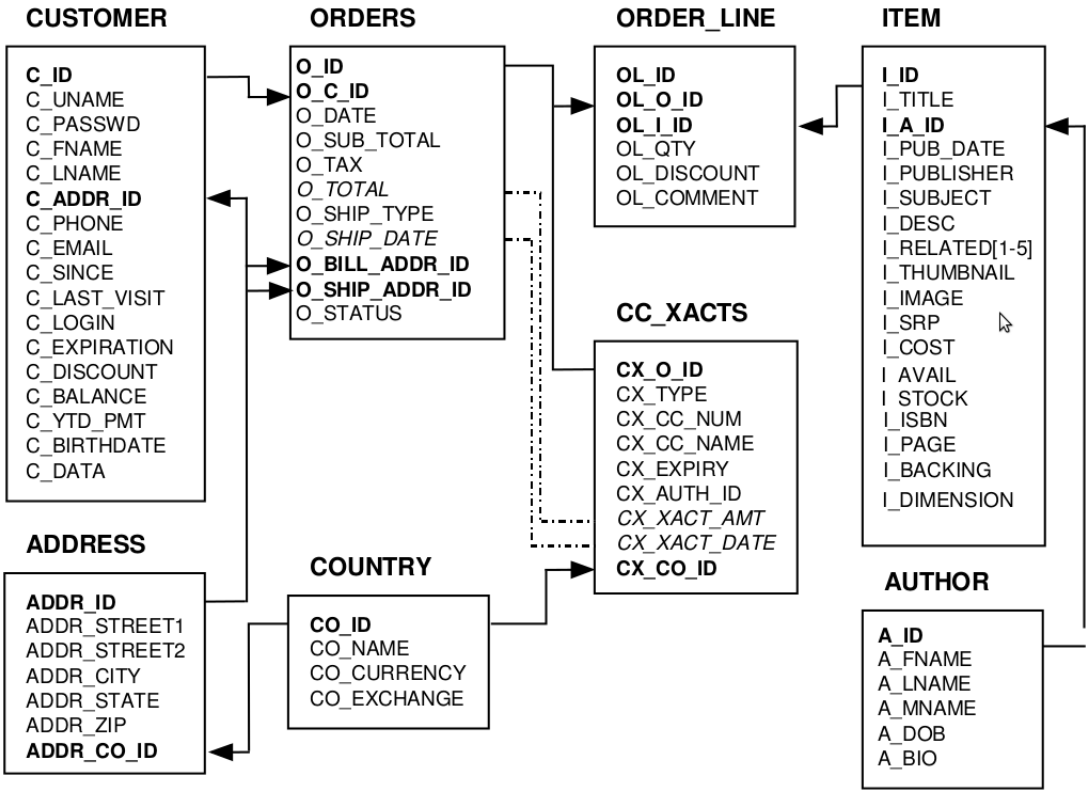
The simplified data-model of a Key-Value Store creates some challenges to develop efficient queries in the database. We address this issue in section 2.3.

There are different benchmark workloads that intend to simulate different usage patterns of the system. The workloads vary on the amount of read and write operations. The browsing workload has 95% of read-only interactions, the shopping workload has 80% and the ordering workload has only 50% of read-only operations.

Benchmark Deployment The flexibility of the SwiftCloud architecture allow us to test different deployments of the benchmark. We can also tune the consistency of the system, the size of the cache and the objects that are automatically

Operation	**Parameters**	Description
PRODUCT DETAIL	item_id	retrieves information about an item with item_id
HOME	item_id, custumer_id	retrieves information about an item with item_id and customer with customer_id
SHOPPING CART	item_id, cart_id, CREATE	adds a new item, with item_id, to an existing shopping cart with cart_id, or a new one if CREATE is set to true.
SHOPPING CART	item_id, qty	adds quantity qty of item_id items to the shopping cart
BUY REQUEST	cart_id	computes the total cost of a shopping cart and the billing information
BUY CONFIRM	customer_id, cart_id	Creates a new order and a new payment for a shopping cart that was previously processed in BUY REQUEST
ORDER INQUIRY	order_id	checks the status of an order
BEST SELLER		Computes the Best Seller information for each category of items
ADMIN ACTION	item_id	Adds the item with item_id to its subject index, adds the five most sold items to the related
CUSTOMER REGISTRATION	customer_id	Registers a new customer

Table 1: Description of TPC-W operations



Legend:

- Dotted lines represent one-to-one relationships between non-key fields related through a business rule. These fields are shown in italic.
- The arrows point in the direction of one-to-many relationships between tables.
- Bold types identify primary and foreign keys.

Fig. 2: TPC-W database schema

updated. We did not have time to make a complete evaluation of these parameters, however we plan to do that to understand the behaviour of the system and how we can improve performance.

The objective of our tests were to show that the we can handle more clients by adding more scouts to the system. Scouts commit transactions locally and they hand of the transaction to the the data centre. Read operations can be completely processed locally if the cache contains the required values. Both features combined reduce the load on data centre and, even when we find a bottleneck on the throughput of the data centre, the system scales by adding a new node to the data centre tier.

Implementation In this section we describe how we adapted the TPC-W benchmark to SwiftCloud. We ported an open-source Cassandra implementation [2] to our platform and implemented operations according to their specification [5].

To access the stored objects in SwiftCloud we must provide their identifier. This raises limitations when it is necessary to do operations where we do not know the identifiers of the elements that we want to access, e.g. range queries. We store the identifiers of the objects in ORSets to keep track of the stored elements. Another issue with range queries is that we can only fetch a value at a time. We could not avoid this and so we incur in great communication overhead to execute these operations.

The Best Seller retrieves the most sold items for a given category. A simple implementation counts the number of sold items for that category, and orders the result to identify the items that were most sold. This would be very expensive to execute in the Key-Value data-model. Our implementation, to avoid counting the results for every element, stores the amount sold items whenever an item is sold. When the Best Sellers operation is executed, the list of sold items is processed and updates the index of the most sold items per category, if any of the sold items is more sold then any other item for its category.

To implement the search operation we use a radix tree that stores in memory the items indexed by their name, author or subject. This index is fundamental to implement a search operation with efficiency. We can store the indexes in memory at the beginning of the application as none of the attributes that are used to create the index are modified during the whole experiment.

The shopping cart is stored with an ORSet to allow adding elements from different sources without loosing operations. Customers have a shopping cart per session. We store the shopping cart associated to that session in the costumer object to have direct access to it, again because it would be too costly to look for the shopping card associated to the current costumers shopping session.

We used LWW-Registers to store address, country, author, customers and cc_xacts entities. Except for the cc_xacts and customer entities, all the other entities are read-only. The LWW-Register is a CRDT with low overhead and it is very suitable to objects that is unlikely to have concurrent updates.

It is important that the available stock of an item is always consistent in the database, i.e., that we do not lose any update to the stock of an item. We used a CRDT counter to guarantee that property. The current CRDT integer operation semantic allows that two concurrent updates lead the stock to a negative value. We plan to address this issue in future work. Despite that, the benchmark prevents that situation by always adding more elements when the stock is under a certain margin.

2.4 Evaluation

The TPC-W benchmark is greatly used in the literature to evaluate systems and provides a way to compare them. We used TPC-W benchmark to evaluate the SwiftCloud system and we intend to use it extensively to find vulnerabilities that need to be improved in order to achieve better performance. We also want to use this benchmark to evaluate the system's behaviour with different *isolation levels*, locations for cache, and commit modes.

To deploy our tests we used Amazon EC2 and PlanetLab. The combination of the two infrastructures provide us an environment where we can place nodes in different locations allowing us to evaluate the effect of the latency between the client, the scout and the data centre.

The system was still under development during this evaluation which prevented us from making a comprehensive evaluation of the cost of using different *isolation levels*. Despite that we made a first approach to evaluate the impact of the size of the cache and analysed the impact of having more scouts in the performance.

The first experiment evaluates the influence of the cache size in the overall throughput of the system. For this purpose we use one data centre machine at the Amazon EC2 and compare the placement of the scout in two different locations. The first set-up consists in putting a scout with a big cache in a node closer to the clients, which resembles to a CDN service that can provide data faster to the nearby clients due to the lower latency to contact them, in our system we also provide updates with lower latency because scouts can commit transactions locally. In this set-up, the latency between the client and the scouts is 30ms, between the scout and the DC is 35ms and between the scout and the DC is 25ms. The second set-up consists in putting the scout next to the data center, in Amazon EC2, but this time with a very small cache.

We executed two different workloads of the benchmark, the shopping and the ordering workloads, and run the experiments while 30 simultaneous clients execute 100 operations each. The results of the experiment are shown in figure 3. The figure shows that in the small scout deployment the performance has high spikes in the throughput. This happens due to a limitation on our system. We only provide support to fetch one value at a time and there is an operation on the workload that requires fetching 10000 objects from the database. Since this deployment does not have a big cache, the scout must get all those values from the data centre one by one, incurring in a big latency overhead. The results for the CDN deployment do not suffer that issue because the used database fits in

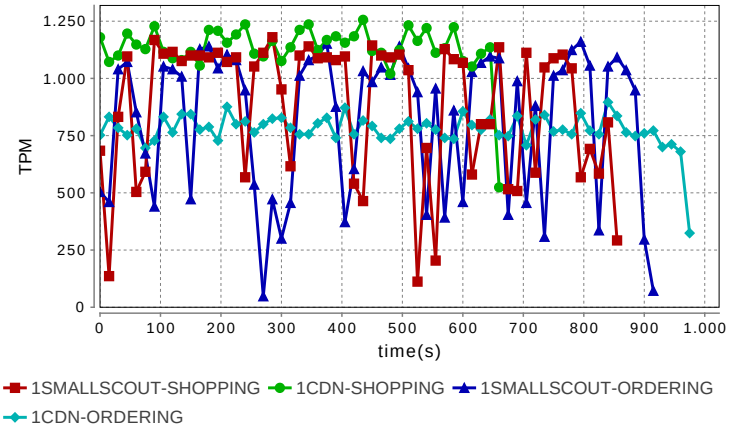


Fig. 3: TPC-W Throughput for SwiftCloud with different cache sizes.

the memory of the scout and so it can serve the requests without contacting the data centre.

In a second experiment we made an initial approach to measure the scalability of the system. We wanted to see how the system reacts by having more clients and scouts connected. To this end, we installed clients, scouts and the data centre in the same region of the Amazon EC2. We had one large machine running as data centre, and clients running in the same machine as the scouts that handle their requests. The scouts run in medium machines and we measure the throughput of the system when we add more scouts. Each scout has 300 clients connected executing operation in a close loop and the cache is big enough to store the whole database. We used a small database with 20000 objects.

The results from figure 4 show that the system has a low throughput, which show the need for improving the performance of the system, however the system scales when adding more clients which allowed us to have up to 3000 clients simultaneously. We can also observe that the performance of the system increases when there are more read only operations in the workload, which is a consequence of handling the operations locally at the scout without contacting the data center to deliver them.

3 Conclusion of the mission

The work carried on this scientific mission provided us with a tool that will help us improve our system and evaluate different usage scenarios. We found that CRDTs were very suitable to support transactional systems. They have a small overhead and support concurrent updates in different replicas. We do not give details on this reports, but our CRDT designs support multi-versioning by design with relatively low overhead. The properties of this data-types may make them suitable to develop new transactional memory algorithms. For now, we are

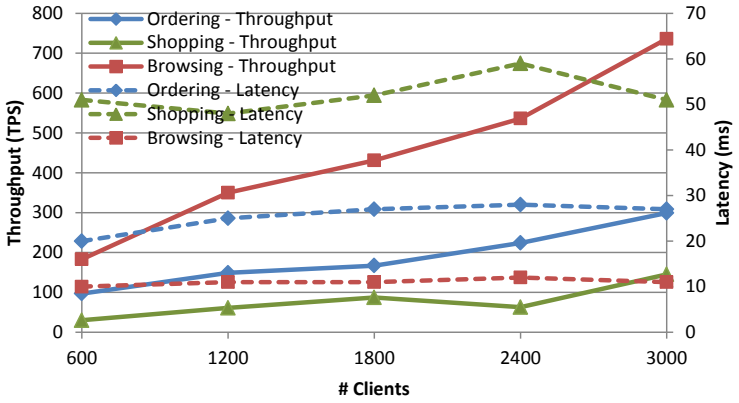


Fig. 4: TPC-W Throughput for SwiftCloud varying number of scouts and workloads

only providing geo-replicated transactional databases, but the same algorithms can be applied to provide transactional memory in a distributed memory system.

We did not have time to do a comparison on the trade-offs of different *isolation levels*, and our evaluation still needs to be improved to find the bounds of the system.

In the follow-up of this work, we will add new features to the system to address the performance issues that we found in the evaluation. Later on, we will do a more comprehensive evaluation to measure the cost of *snapshot isolation* and repeatable reads *isolation levels*. Finally we will think about stronger *isolation levels* and their feasibility with our architectures, which can be required in some transactional memory algorithms.

4 Foreseen publications resulting from the STSM

As stated before the results gathered in this mission were used in an article submitted to EuroSys'13, which is currently under review. the research group is composed by Marek Zawirski (UPMC-LIP6 & INRIA), Annette Bieniusa (U. Kaiserslautern), Valter Balegas (UNL), Nuno Preguiça (UNL), Sérgio Duarte (UNL) and Marc Shapiro (INRIA & LIP6).

5 Confirmation by the host institution of the successful execution of the STSM

The host, Marc Shapiro from INRIA & LIP6, confirms that Valter Balegas has achieved the targets set forth for this collaboration. He has developed a TPC-W benchmark for SwiftCloud and performed a set of benchmark-driven measurements. This work has helped identify some of the bottlenecks of our platform.

Valter will now be invited to extend his internship on INRIA funding, in order to continue uncovering bottlenecks and to help improve the platform accordingly.

References

1. Seth Gilbert and Nancy Lynch. Brewer's conjecture and the feasibility of consistent, available, partition-tolerant web services. *SIGACT News*, 33:51–59, June 2002.
2. Pedro Gomes. TPC-W benchmark. `https://github.com/PedroGomes/TPCw-benchmark`, Retrieved 15-oct-2012.
3. Marc Shapiro, Nuno Preguiça, Carlos Baquero, and Marek Zawirski. Conflict-free replicated data types. In Xavier Défago, Franck Petit, and V. Villain, editors, *SSS*, volume 6976 of *LNCS*, pages 386–400, Grenoble, France, October 2011. Springer-erlag.
4. Marc Shapiro, Nuno Preguiça, Carlos Baquero, and Marek Zawirski. Conflict-free replicated data types. In Xavier Défago, Franck Petit, and Vincent Villain, editors, *Stabilization, Safety, and Security of Distributed Systems*, volume 6976 of *Lecture Notes in Computer Science*, pages 386–400. Springer Berlin / Heidelberg, 2011.
5. Transaction Processing Performance Council. TPC-W benchmark specification version 1.8. `http://www.tpc.org/tpcw/spec/tpcw_V1.8.pdf`, February 2002.
6. Werner Vogels. Eventually consistent. *ACM Queue*, 6(6):14–19, October 2008.
7. Stephane Weiss, Pascal Urso, and Pascal Molli. Logoot-undo: Distributed collaborative editing system on P2P networks. *TPDS*, 21:1162–1174, 2010.