Replicated Convergent Data Containers

Tobias Herb and Odej Kao

Technical University Berlin, Germany
{tobias.herb,odej.kao}@tu-berlin.de

Abstract. Managing replicated data in distributed systems that is concurrently accessed by multiple sites is a complex task, because consistency must be ensured. In this paper, we present the Replicated Convergent Data Containers (RCDCs) - a set of distributed data structures that coordinate replicated data and allow for optimistic inserts, updates and deletes in a lock-free, non-blocking fashion. It is crucial that continuous data harmonization among containers takes place over time. This is achieved by a synchronization mechanism that is based on a technique called Operational Transformation (OT) which continously reconciles diverging containers. A generic architecture is placed on top of this underlying synchronization mechanism that allows to realize a multitude of different RCDCs. Two container specializations are presented: (a) the linear container that organizes data in an ordered sequence, (b) the hierarchical container that organizes the data in an n-ary tree.

1 Introduction

Managing data in distributed systems that is concurrently accessed and modified by multiple processes is a complex task, e.g. in a groupware software where multiple clients try to edit the same document. The main problem consists in maintaining the consistency and integrity of the data without loss of performance. A common solution to that challange is to serialize all concurrent write accesses, so that at any time only one process perform its modifications. In order to improve the overall throughput, there exist advanced transactional techniques that allow simultaneous access for non-conflicting modifications. The disadvantages of this approach are despite well-developed techniques the need for continuous access to the remote data storage and the resultant high access latency. We present in this paper an approach for optimistic data synchronization where we pursue the idea of embeddeding and organizing replicated data in abstractions similar to collections that are well known from programming. We call these structures Replicated Convergent Data Containers (RCDCs). The container abstractions offer a common interface where data can be inserted, retrieved, updated and deleted in a lock-free, non-blocking fashion. It is crucial that consistency among the replicas is established and a continuous harmonization takes place over time. This is achieved by a synchronization mechanism that is based on a technique called Operational Transformation (OT). This mechanism continously reconciles diverging container states in background by exchanging and transforming container modification operations. The key requirement that enforces the use of OT

Y. Ait Ameur et al. (Eds.): MEDI 2014, LNCS 8748, pp. 238–249, 2014.

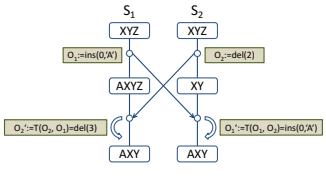

Fig. 1. simple OT scenario

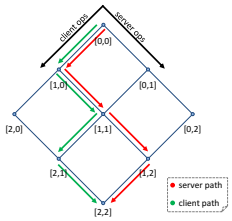

Fig. 2. client/server paths in state-space

is to preserve the internal arrangement of data, which means that not only data itself but also the structure within the container is synchronized. This is relevant in cases where not only the data elements themselves but also the arrangement of the data within the container is important. A generic architecture is placed on top of this underlying synchronization mechanism that allows to realize a multitude of different RCDCs. Two container specializations are presented: (a) the linear container that organizes data in an ordered sequence, (b) the hierarchical container that organizes the data in an n-ary tree.

1.1 Contributions

The contributions of this paper are the following:

- We propose the novel concept of syncable data containers that enable lock-free, non-blocking modifications.
- We provide a simple "topological" classification of different container types. A abstract architecture is derived on the basis of this classification.
- We present the linear container type and its embedding in our overall architecture. In addition, we introduce a stable iterator allowing container traversals while updated.
- Finally we present the hierarchical container organizing items in a tree structure. We show the associated transformations are designed and how to safely delete in hierarchical structures.

2 Preliminaries

This section gives a basic introduction to the theoretical framework of operational transformation (OT) that solve the challenge of consistency maintenance of distributed replicas.

2.1 Operational Transformation

Operational Transformation (OT) is a theoretical framework for concurrency control that enables consistency maintenance of replicated data objects in a

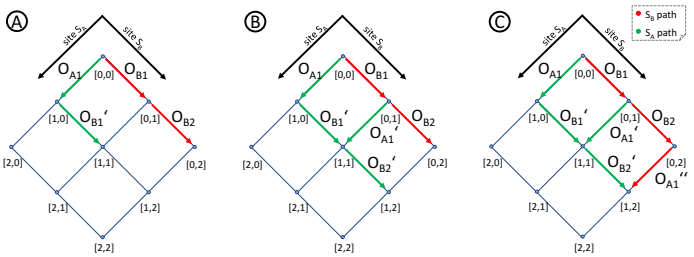

Fig. 3. multistep divergence in client/server processing

distributed environment [13]. Each site is allowed to modify any part at any time of the replica by applying operations on it. The operations are instantaneously executed without being blocked or delayed and stored in a history buffer (HB). After local execution the operation is propagated to the remote sites. Operations arriving from remote sites must be transformed along all concurrent operations that reside in the HB. Concurrent operations in the HB are determined via Lamport- or vector clocks [5,8] attached to the operations. The transformation "includes" the effects of the HB operations in the remote operation. This can be well illustrated by a simple text editing scenario (see figure 1). A character sequence XYZ is replicated on two sites. Site S_1 inserts character A at position 1 (O_1), resulting in local state $AXYZ$. Site S_2 deletes the character Y at position 2 (O_2), resulting in state XZ. The operations are locally executed, stored in the HB and then exchanged, respectively. By applying OT the position-parameter of the incoming operation is adjusted according to the (concurrent) operations in HB. On site S_1 the position of the incoming delete operation is incremented in respect to the insert operation on the lower position. On site S_2 the local delete operation has no influence on the incoming insert operation, because the deletion takes place behind the insert. The essence of keeping the distributed replicas coherent with OT is the application of the transformation function on pairs of concurrent operations that are processed in different order on different sites. This kind of transformation is called *inclusion transformation* (IT) in OT theory [13]. It owns the general signature: $T(O_x, O_y) = \{O'_x, O'_y\}$. The function produces adapted pairs of the input operations, in a sense that the resulting operations take the "effect" of each other into account. It must be noted that transformation function can only be applied on pairs of concurrent operations which originate from the same state. The general interaction between two sites can be well visualized with a so-called two-dimensional state space graph (see figure 2). As operations are processed, the two sites walk down the state space. If they process the operations in the same order, then they take the same path through the state space. If client and server process different operations, then their paths begin to diverge. In the illustrated scenario both sites move to state [1,1] together since both processes first executes the operation of site 1 and thereafter operation of site 2. At state [1,1] they execute different operations, moving to [2,1] and [1,2], respectively. To reach again a common state, both

sites transform the remote operation against the buffered operation and executes it. The transformation can be applied because both originate from state [1,1]. To guarantee sound transformations, must convergence property of the function between all possible combinations of available operations must be ensured, the so called TP1-property [3]: $O_x \circ O'_y \equiv O_y \circ O'_x$. If the two sites ($S_A$ and S_B) diverge more than one step, we can not directly apply the transformation function, because these operations have no common origin. The state space diagram in figure 3 depicts such a situation: S_A and S_B begin to diverge at state [0,0], where S_A executes the local operation O_{A1} and S_B executes subsequently O_{B1} and O_{B2} (A). S_A receives the operation O_{B1} and is able to transform it against its local operation O_{A1} to obtain the adapted server operation O'_{B1}. The first transformation step works, because O_{A1} and O_{B1} were generated at same state [0,0]. When S_A receives the second server operation, the transformation of O_{B2} against O_{A1} does not work, because they were generated in different states. At this point the transformed local operation O'_{A1} of the first transformation process $T(O_{A1}, O_{B1}) = \{O'_{A1}, O'_{B1}\}$ is required, because this operation bridges the gap between client state and server state. The remote operation O_{B2} must be transformed against O'_{A1} to retrieve a correct adapted operation O'_{B2} (B). To achieve consistency between S_A and S_B, the site S_B only needs to transform successively the incoming operation O_{A1} against O_{B1}, O_{B2} (resulting O''_{A1}) and execute it (C).

2.2 Control Algorithm

We use a client/server based OT control algorithm that controls transformation (including multistep divergence). This algorithm has both client and server parts which drive the transformation of operations. The presented algorithm is derived from the Jupiter algorithm [9] and follows the descriptions of [14]. The server maintains a unique operation history for all clients. This implies the restriction on incoming client operations, which have to be parented at some point on the server path (in state space). The server only needs to transform all incoming client operations from this point against the concurrent operations lying on the server path. In return, the client needs to keep track of the server state and is responsible for the transformation of the local operations into the server space (insofar known to the client). The tracing of the servers path on the client site is called inferred server path [14]. To keep the inferred server path valid during interaction, every transmitted client operation have to be acknowledged by the server.

3 Replicated Data Containers

This work presents a new kind of data structures which automates data synchronization among multiple processes in a distributed system. We call our data structures syncable data containers. The containers are replicated on all sites in the system. Each site can (optimistically) modify any part at any time of the

container. The underlying OT framework hereby ensures convergence among the container instances. In this approach the data synchronization is achieved by a two-party synchronization protocol. Each site synchronizes only with the server. The server serializes and transforms all changes and propagates them further to the other replicas (see control algorithm). The containers with their lock-free, non-blocking property allow highly interactive application scenarios. For example is it relatively easy to build realtime collaborative application on top of these data structures. A key aspect that must be observed is the convergence behavior of these containers. Due to the optimistic modifications and the continous background reconcilation, cannot be guaranteed that all container instances have a consistent view on the data at any time. Our set of data containers provide an eventual consistency model [11] and are therefore not suitable for consistency critical applications. The core abstraction of these container types is similar to the data collections in general purpose programming languages (e.g. Java Collections Framework [1]). The containers manage a group of data objects. These objects are called data items and represent the basic unit that is controlled by the container. A data item itself can be atomic (i.e. primtive type) or any arbitrary composite types. Our data containers stores the data items in an organized way that is amenable to the underlying synchronization mechanism. Two different containers types are presented in this work, the linear container that implements a finite ordered sequence of data items, similar to an abstract list data type [7]. The second type is the hierarchical container that organizes the contained data items in a hierarchical structure.

3.1 Topological Classes

Data collections in programming languages are always specialized according to the way how contained elments are organized, e.g. lists, trees and graphs. The notion of so called topological classes is introduced to make this distinction of the organizational structure more general and abstract. We define three general topological classes. The classification is based on restrictions of the linkage among data items. **(1) Unrestricted:** The unrestricted topology has no limitations on the item linkage and can represent arbitrary graphs. **(2) Hierarchical:** Every data item can have only one predecessor (or parent), but any number of successors (or childs). **(3) Linear:** Data items can have only one predecessor and one successor. The idea behind this classification scheme is to build for each topological class a basic container type that is able to embed and organize the contained data items according to the given topology.

3.2 Architecture

We derive on the basis of the topological classification an abstract architecture that defines the common implementation structure for different data container types. All types are built up of three main components:

Data Model: The data model defines the internal representation of a data container and defines how operations provided by the *Operation Model* interact with this internal representations. There are no specific assumptions or restrictions how a data model has to be designed, i.e. it can contain arbitrary complex logic and representations of the contained data items.

Operation Model: All container types have a common set of basic operations. This set defines the so called **CRUD** operations: A **C**reate operation to create (or add) new data items, a **R**etrieve operation to read/get data items, an **U**pdate operation to update existing data items and a **D**elete operation to remove data items. These basic operations, except Retrieve, are used by the underlying OT system for synchronization. Retrieve is purely local operation and does not change any state of the container or of the contained elements. That is why read operations in general must not be propagated over the network. The signatures of *create, update, delete* operations are container type independent defined as:

$$create(C) : \Gamma \times \Theta \times N \to \Gamma$$

$$update(U) : \Gamma \times \Theta \times \Delta \to \Gamma$$

$$delete(D) : \Gamma \times \Theta \to \Gamma$$

Γ is the type of the container instance on operations are applied. Θ is a container type dependent parameter that determines access to data elements. This access parameter selects the data item on which or next to it (that depends on the operation type) the operation is performed. Containers of different topological classes require therefore different representations for the access parameter type. For example elements in a linear container can be simply accessed by an positional index. For more complex topologies must the access parameter describe a path (see hierarchical container). N is the new data items that is inserted in the container. The parameter Δ occuring in update operation is a map that contains the (attributes) updates that are performed on the selected data item.

Transformation Model: The transformation model contains the set of transformation functions for all possible operations combinations in the operation model. This results in a 3×3 matrix where each element represents a transformation function of two concurrent operations:

$$\begin{bmatrix} T(C_L, C_R) & T(C_L, U_R) & T(C_L, D_R) \\ T(U_L, C_R) & T(U_L, U_R) & T(U_L, D_R) \\ T(D_L, C_R) & T(D_L, U_R) & T(D_L, D_R) \end{bmatrix}$$

L - Local, R - Remote

The transformation of operations is specific to the underlying topology, because the transformation functions adapt the access parameters Θ of the concurrent local and remote operations. The exact properties of such a transformation are discussed in the OT section.

3.3 Conflicting Operations

In some transformation cases occur conflicts, that must be explicitly resolved. A conflicting transformation is a transformation where the effect of one operation loose its original intention or the operation gets completely lost. This situation occurs either if two concurrent operations modify the same element (or rather the operations relate to the same position) or if one operation modifies an element and the other operation deletes the element. For example the concurrent insertion of two data items at the same position lead to a conflict, because it must determined if either the local or the remote insertion have precedence and takes place at the specified position. The matrix below identifies all possible conflict cases for our **CRUD** operation model:

$$\begin{array}{c} \\ \mathbf{C} \\ \mathbf{U} \\ \mathbf{D} \end{array} \begin{array}{c} \begin{array}{ccc} \mathbf{C} & \mathbf{U} & \mathbf{D} \end{array} \\ \begin{pmatrix} X & - & - \\ - & X & X \\ - & X & - \end{pmatrix} \end{array}$$

The solving of conflicts is in general application dependent [11]. In systems where such conflicts ($\Leftrightarrow$ lost updates) rarely happen or the data is not critical predefined mechanisms can be applied. In other scenarios the decision should be passed to an external decider that has more knowledge to manually resolve the conflict. As a result of these considerations the conflict-solver logic must be separated from the transformation model to preserve the freedom to hook in domain specific behavior. All conflict cases of the transformation function are delegated to an external handler where the application specific behavior can be defined. If no domain specific conflict handler is provided, then default strategies are applied. The default strategy combines two simple tactics. First, an operation precedence for the different conflict generating operations is defined. Operations with higher precedence win against operations with lower precedence in conflict situations. The conflict solver replaces the operation with lower priority with no-op (no-operation), for example delete-operations win over update-operations. If two operations of the same precedence are in conflict, a last-writer win strategy (LWW) is applied. LWW means that in the client case the local operation always wins against the incoming remote operation. For the server side in turn means that the incoming operation always wins against 'local' operation. In the case of concurrent update operations targeting the same data item does only exist a conflict if the delta sets (attribute changes) of the update operations are not disjoint: $\Delta(U_L) \cap \Delta(U_R) \neq \varnothing$.

4 Linear Container

The linear container organizes the contained data items in an ordered sequence. It can be regarded as a distributed list collection. The contained data items are managed by positional access and can be created, retrieved, updated and deleted (see **CRUD** operation model) at any time. Beyond this basic functionality a

stable iterator [4] exists that allows to iterate over the content while the container is concurrently modified. These stable iterators will be discussed in detail in the next section. Due to the way the container organizes the containing data items by their numerical position, the access parameter Θ of the corresponding operation model is an simple integer index. This index determines the position in the sequence where the operation is applied. The associated linear transformations for this type are well known from collaborative text editing and can be taken for example from [9]. A common way to implement the underlying data model that supports the positional access would be to use internally a linked list or a growable array. This whole attachment of operation model, transformation model and OT algorithm can be seen as a kind of proxy on top of a normal list collection that extends the data structure to distributed, lock-free, non-blocking synchronization.

4.1 Iterators

Another important aspect is the traversal over the container content. Data items are continously created, updated and deleted by the user or control algorithm process which usually leads to an invalidation of an concurrent iteration process. We propose the idea of so-called transformable cursors that allows iterations over the original container while being modified. A transformable cursor is pointer to a concrete data item consisting in the case of the linear container of a simple index. This index participates in the OT transformations and is transposed according to the local and remote changes. It keeps always the positional reference to the assigned data item as long as this data item is not deleted. If the referenced item is deleted, the cursor points to the up moving successor item. The cursor additionally provides an *advance* function that let the cursor manually move on to the successor item. It must be ensured that transformation call for the cursor and the *advance*-call are well synchronized to keep the cursor position consistent.

5 Hierarchical Container

This section presents the concept of an hierarchical container that organizes data items in a n-ary tree arrangement. Arbitrary many child items can be attached to each item. Items with a linked subtree are called nodes else they are called leafs. The deletion of a node thus deletes all linked sub-items, comparable to the deletion of a folder in a filesystem. The synchronization of delete operations is more complex in the case of the hierarchical topology because concurrent operations may exist that refer to items belonging to the attached subtree of the deleted node. We later introduce the exact deletion algorithm that is used to keep the containers synchronized. The container organizes the hierarchical structure by nested lists. For each item, a ordered collection is maintained that keeps references to the associated child items. Access to a data item is carried out via its root path (path from root to node). Paths are realized as so-called **access vectors**, where each element is an index in the associated child-list reffering to

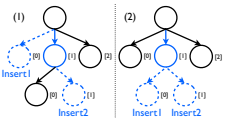

Fig. 4. create-create

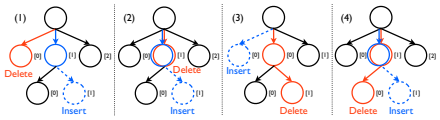

Fig. 5. create-delete

the subjacent item. Thereby corresponds the number of elements to the tree-level of the selected item. The access parameter Θ of the operation is thus bound to an access vector of type $\mathbb{N}_0^X$ where X is the tree-level of the referenced item. The *create*-operation handles the access vector slightly differently compared to the update- and delete-operation. The elements $a_0 ... a_{n-1}$ of the access vector $\begin{bmatrix} a_0 \ ... \ a_{n-1} \ a_n \end{bmatrix}^{\mathrm{T}}$ are interpreted as the path from the root to the target item and the last element a_n as position in the target child-list where the new item is inserted. Update and delete interpret all elements $a_0 ... a_n$ as path to the target item. In the hierarchical container all concurrent operations do not necessarily have to influence each other. Operations become in general **effect-dependent** and need to be transformed, if either an operation has an impact along the access path of a concurrent operation or the operations refer to the same target node. The first case requires the adaption of the relevant index in the access vector of the affected operation. The second case equates to the linear transformation. To capture the relationship of concurrent hierarchical operations formally is the following relation introduced:

Definition 1 ***Effect Dependence***
Operations O_X *and* O_Y *are effect dependent* $O_X \rightsquigarrow O_Y$ *iff:*

(i) The access path of O_X *is part of the access path of* O_Y *or vice versa or they are equal:*

(a) $\Theta(O_X) \subset \Theta(O_Y)$
(b) $\Theta(O_X) \supset \Theta(O_Y)$
(c) $\Theta(O_X) \equiv \Theta(O_Y)$

(ii) If (a) or (b) and the positional reference of the shallow access path is less than or equal to the corresponding component of the deeper access path of the concurrent operation.

On basis of the effect dependence relation, the possible transformation cases of the underlying operation model are derived. For lack of space we do focus only on the transformations related to the *create*- and *delete*-operations.

The *update*-operation has no impact on access paths and for the accompanying conflict cases the presented conflict handling can be applied. The **create-create** transformation consists of two essential cases, presuming the conditions of effect dependence are fulfilled (see figure 4). *(1)* If a create-operation targets an item at deeper level in the hierarchy than another concurrent create-operation and the target index (the components of the access vector) of the second operation is less than or equal to the index of the first operation, then an index transposition

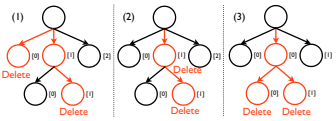

Fig. 6. delete-delete

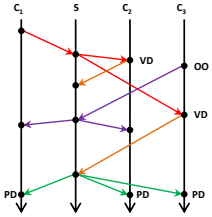

Fig. 7. bulk-delete scenario

at the corresponding component of first operation must take place to get obtain the correct access vector. *(2)* If both operations target the same node then both indices have to be adapted. For the **create-delete** transformation four different cases are considered (see figure 5). *(1)* The delete-operation removes an item along the access path of the create-operation. *(2)* The opposite case is a create-operation that inserts a new item along the access path of a concurrent delete-operation. *(3)* If a (local) delete operation removes a node that lies on the access path of a (remote) create-operation then the create-operation has no effect and can be transformed to no-operation. *(4)* If both operations target the same node then both indices have to be adapted as in the linear transformation cases. The *delete-create* transformation is symmetric and must not considered further. The **delete-delete** transformation is the last combination that is considered here (see figure 6). *(1)* The delete-operation removes an item along the access path of the second delete-operation. *(2)* If a (local) delete operation removes a node that lies on the access path of a (remote) delete-operation then the delete-operation has no effect and can be transformed to no-operation, because the effect is already included in the upper delete-operation. *(3)* If both operations target the same node then both indices have to be adapted.

5.1 Bulk Deletes

The deletion of a data item in the hierarchical container as well leads to the deletion of all child data items that are contained in the connected subtree, we call this a *bulk delete*. This results in additional problems, because concurrent operations (generated by other processes) may still buzz around in the system that target these deleted items. The problem is solved by lifting the local deletion to a system-wide coordinated process which ensures that no more concurrent operations are in the system that operate on these deleted items. The actual removing of the data items is deffered until all concurrent operations were carried out. Our algorithm handles the *bulk delete* by dividing the process into two phases, the so-called *virtual delete phase* and *physical delete phase*. In the virtual delete phase all data items (the delete target + linked subtree) are marked as deleted and removed form the *index*-mapping, preventing the generation of new operations that target deleted items. All deleted marked items are stored under a unique

deletion-key in an additional index structure. The delete-operation is propagated, together with the deletion-key, to all replicated containers. On receipt of the delete-operation are also all data items marked as deleted and removed from the *index*-mapping. Additionally is a vdel-operation (virtual-delete) containing the deletion-key appended to the local transmit buffer of that container. This vdel-operation marks the boundary between all possible concurrent operations that still may target the deleted items and the subsequent operations (following the vdel-operation) that respect the deletion. Concurrent operations that target deleted items do lead to a conflict. Conflicts can be either explicitly handled by the process according to specific application semantics or default strategies can applied (see Conflicts), e.g. operations on deleted items are transformed to *no-operation*. The OT server collects all vdel-operations and acknowledges them all at once. On receipt of the vdel-acknowledge are all as deleted marked items for that deletion-key physically deleted and removed from the delete-index.

6 Related Work

Modern web technologies for data management [2] do support automatic and manual synchronization of data on multiple clients and server-side. But these approaches do not allow optimistic insertions and deletions on ordered sequence of data items. An alternative to the OT approach is *Operation Commutativity (OC)*. In OC does all operations commute when they are concurrent. This approach does not need a explicit concurrency control compared to OT [6]. Shapiro et al. introduced the Commutative Replicated Data Types (CRDTs) [12]. They developed a shared sequential data structure, where concurrent operations can be executed in an arbitrary order on different sites. A further approach presented by Hyun-Gul Roh et al. in [10] is also based on operation commutativity. They developed the so-called Replicated Abstract Data Types (RADTs), a set of replicated linear data structures: replicated fixed-size array (RFAs), replicated hash tables (RHTs) and replicated growable array (RGAs). They exploit the commutativity of the operations with a principle called precedence transitivity (PT). PT allows to derive a unique order of concurrent operations.

7 Conclusion and Outlook

We presented an approach for distributed data containers that coordinate replicated data and allow for optimistic inserts, updates and deletes in a lock-free, non-blocking fashion. The required synchronization mechanism responsible for reconcilation of diverging containers is based on Operational Transformation. We developed a generalized architecture on top of this synchronization mechanism to support structures with different topologies like linear and hierarchical containers. In general can these sort of convergent data containers be used to efficiently build a wide range of applications with realtime collaborative features. The major advantage of our approach is the unified architecture consisting of common components which massively simplifies the development of multiple

RCDCs with different properties and behaviour. An important next step is to extend the containers for offline capability. Changes should be made locally without beeing conntected to the system, when a connection is restored are the local changes reintegrated in the global state. This enables the data to be (asynchronously) available and workable at anytime and at anyplace, which makes it especially interesting for mobile environments.

References

1. The collections framework @MISC, http://docs.oracle.com/javase/7/docs/technotes/guides/collections/
2. Granite data services @MISC, https://www.granitedataservices.com/
3. Ellis, C.A., Gibbs, S.J.: Concurrency control in groupware systems. SIGMOD Rec. 18(2), 399–407 (1989), http://doi.acm.org/10.1145/66926.66963
4. Gamma, E., Helm, R., Johnson, R., Vlissides, J.: Design Patterns: Elements of Reusable Object-oriented Software. Addison-Wesley Longman Publishing Co., Inc., Boston (1995)
5. Lamport, L.: Time, clocks, and the ordering of events in a distributed system. Commun. ACM 21(7), 558–565 (1978), http://doi.acm.org/10.1145/359545.359563
6. Letia, M., Preguiça, N., Shapiro, M.: Consistency without concurrency control in large, dynamic systems. SIGOPS Oper. Syst. Rev. 44(2), 29–34 (2010), http://doi.acm.org/10.1145/1773912.1773921
7. Liskov, B., Zilles, S.: Programming with abstract data types. In: Proceedings of the ACM SIGPLAN Symposium on Very High Level Languages, pp. 50–59. ACM, New York (1974), http://doi.acm.org/10.1145/800233.807045
8. Mattern, F.: Virtual time and global states of distributed systems. In: Parallel and Distributed Algorithms, pp. 215–226. North-Holland (1988)
9. Nichols, D.A., Curtis, P., Dixon, M., Lamping, J.: High-latency, low-bandwidth windowing in the jupiter collaboration system. In: Proceedings of the 8th Annual ACM Symposium on User Interface and Software Technology, UIST 1995, pp. 111–120. ACM, New York (1995), http://doi.acm.org/10.1145/215585.215706
10. Roh, H.G., Jeon, M., Kim, J.S., Lee, J.: Replicated abstract data types: Building blocks for collaborative applications. J. Parallel Distrib. Comput. 71(3), 354–368 (2011), http://dx.doi.org/10.1016/j.jpdc.2010.12.006
11. Saito, Y., Shapiro, M.: Optimistic replication. ACM Comput. Surv. 37(1), 42–81 (2005), http://doi.acm.org/10.1145/1057977.1057980
12. Shapiro, M., Preguiça, N., Baquero, C., Zawirski, M.: Conflict-free replicated data types. In: Défago, X., Petit, F., Villain, V. (eds.) SSS 2011. LNCS, vol. 6976, pp. 386–400. Springer, Heidelberg (2011)
13. Sun, C., Ellis, C.: Operational transformation in real-time group editors: Issues, algorithms, and achievements. In: Proceedings of the 1998 ACM Conference on Computer Supported Cooperative Work, CSCW 1998, pp. 59–68. ACM, New York (1998), http://doi.acm.org/10.1145/289444.289469
14. Wang, D., Mah, A., Lassen, S.: Google wave operational transformation type @MISC (2010), http://wave-protocol.googlecode.com/hg/whitepapers/operational-transform/operational-transform.html