International Journal of Cooperative Information Systems
Vol. 23, No. 1 (2014) 1440002 (33 pages)
© World Scientific Publishing Company
DOI: 10.1142/S0218843014400024

Interest Aware Consistency for Cooperative Editing in Heterogeneous Environments

André Pessoa Negrão*, João Costa†, Paulo Ferreira‡ and Luís Veiga§
Distributed Systems Group, INESC-ID
Instituto Superior Técnico, Universidade de Lisboa
Rua Alves Redol 9, Lisboa, Portugal
**andre.pessoa@ist.utl.pt*
†joao.da.costa@ist.utl.pt
‡paulo.ferreira@inesc-id.pt
§luis.veiga@inesc-id.pt

Received 15 January 2013
Accepted 28 March 2014
Published 5 May 2014

Cooperative editing applications enable geographically distributed users to concurrently edit a shared document space over a computer network. These applications present several technical challenges related to the scalability of the system and the promptness with which relevant updates are disseminated to the concerned users. This paper presents Cooperative Semantic Locality Awareness (CoopSLA), a consistency model for cooperative editing applications that is scalable and efficient with regards to user needs. In CoopSLA, updates to different parts of the document have different priorities, depending on the relative interest of the user in the region in which the update is performed; updates that are considered relevant are sent to the user promptly, while less important updates are postponed. As a result, the system makes a more intelligent usage of the network resources, since (1) it saves bandwidth by merging postponed updates and (2) it issues fewer accesses to the network resources as a result of both update merging and message aggregation. We have implemented a collaborative version of the open source Tex editor *TexMaker* using the CoopSLA approach. We present evaluation results that support our claim that CoopSLA is very effective regarding network usage while fulfilling user needs (e.g. ensuring that relevant updates are disseminated in time).

Keywords: Cooperative editing; optimistic replication; data consistency; interest management; divergence bounding.

1. Introduction

Cooperative editing applications enable geographically distributed users to concurrently edit a shared document space over a computer network.[1] Recently, these applications experienced an increase in popularity as a result of the expansion of the Internet and the rapid proliferation of mobile devices, such as smartphones, PDAs and tablets.[2] These modern devices are now sophisticated enough to allow their users to execute cooperative editing applications and participate in editing sessions alongside more powerful devices (such as desktops or laptops),[3] possibly mediated by cloud infrastructures.[4]

A critical technique to support cooperative work in these new heterogeneous environments — that mix resource constrained and powerful devices, interacting over wired and wireless networks — is to replicate the application data at the users' devices and resort to optimistic protocols to manage the consistency of the shared state. Optimistic replication[5] has the potential benefit of improving performance, availability and usability by allowing faster (local) access to the data. It also makes an efficient use of the resources, since it does not require constant access to the network for synchronization purposes. Optimistic mechanisms have been extensively applied to cooperative editing, in particular the Operational Transformation paradigm[2,6–9] and, more recently, Commutative Replicated Data Types.[10–14]

While the state-of-the-art solutions provide a fair compromise between consistency and performance, we identify two aspects that can be leveraged to improve the performance and overall usability and experience of cooperative editing applications. First, we observe that the available solutions are oblivious to the variable and highly dynamic characteristics of group work, in which different users are interested (and work) on different parts of the document space: (i) a user is more interested in the zone(s) of the document that he is editing (e.g. a section, a paragraph, etc) and a few other *observation points*, rather than the whole document space equally; and (ii) the user's interest in the different parts of the document space varies over time. Second, optimistic systems based on eventual consistency[15–18] are typically prone to some level of uncertainty and disruption: while the system is ensured to converge in the future, there is limited or no support to determine how up-to-date is the data observed by the user, and to establish and enforce clear bounds or guarantees to the consistency state of each user.

In this work we argue that it is possible to make a more efficient and scalable usage of the network resources by taking the users' interest into account. To address this issue, we propose Cooperative Semantic Locality Awareness (CoopSLA), a consistency model that unifies several well-known concepts of the distributed systems field: interest management,[19] locality-awareness and divergence bounding.[20] In CoopSLA, updates are assigned a per-user priority level based on their *semantic distance* to the user's *observation point*(*s*) (interest management). Updates to document regions closer to the observation points are considered more relevant and, thus, are awarded higher priority; priority decreases as the distance to the observation point increases (locality-awareness). In each priority level, updates are managed according to a parameterizable, multidimensional consistency space that determines under which conditions updates are allowed to be postponed and when they must be propagated (bounded divergence).

With CoopSLA, we are able to make a more intelligent and semantically meaningful usage of the network resources: by postponing updates with lower priority (i.e. updates to less relevant data), the system is able to merge and aggregate them, minimizing the number of accesses to the network and reducing bandwidth.

Such an efficient network usage is of particular importance when mobile devices and wireless connections are in use. As a matter of fact, wireless networks provide low bandwidth with high latency, which has a significant impact on performance and interactivity.[21] Furthermore, a high number of accesses to the network greatly increases battery consumption.[22] Thus, minimizing the number of accesses to the network resources allows for improved performance and better device autonomy.

In addition, due to its divergence bounding approach to consistency maintenance, CoopSLA establishes clear and explicit bounds on the amount of replica deviation allowed. As such, it provides users with stronger guarantees regarding the actual consistency state of the document space.

We implemented a middleware layer that enforces the CoopSLA model. Using the CoopSLA middleware, we implemented a collaborative version of the popular Tex editor *TexMaker*. We present experimental results that support our claim that CoopSLA is able to reduce the overhead of replica synchronization while respecting the consistency needs of users for each particular text excerpt (which can be a chapter, a section, a paragraph, etc with no restriction).

The remainder of the paper is organized as follows. Section 2 introduces relevant concepts and describes the main assumptions of our work. Section 3 describes the CoopSLA consistency model in detail. Section 4 presents the main architectural aspects of the CoopSLA middleware. Section 5 overviews the implementation of our solution. Section 6 presents and discusses the experimental results. Related work is presented in Sec. 7. Section 8 concludes the paper and, finally, the pseudocode of the main algorithms is presented in Appendix A.

2. System Model

In a cooperative editing session, multiple geographically distributed users concurrently edit a shared *document space*. The document space can be, for example, a Latex document, a wiki, an XML representing a Word document or a set of such (master and input) files to be included in the final document rendering. Common to these scenarios is a hierarchical structure of *semantic regions*, which are logical sub-divisions of the document space (e.g. a file, a \section or a text paragraph of a Latex document). Also, semantic regions may have logical references to other regions (e.g. a Latex \ref or a link on a web page). When considering the interest of a user, both the structure of the document space and the references between its components must be taken into account.

We model the document space as a rooted directed graph $G = (V, E)$. Each vertex V corresponds to a *semantic region* of the document space and there is an edge (v_i,v_j) between vertices v_i and v_j *if* there is a *relation* between the two. We consider two types of relations: a *structural* edge connects two vertices that have a parent/child relation that is part of the hierarchical organization of the document (in a Latex document, for example, a \chapter has a structural relation with each of its child \section); a *semantic* edge is a non-structural edge that connects two vertices that have an application-specific reference between them (e.g. a \ref in a Latex document or a link in a web page). Structural edges define a subgraph S of G corresponding to the tree structure of the document space.

We assume a metric $d : V \times V \longrightarrow \mathbb{N}_0$ over G that captures the *semantic distance* between two vertices of the graph. The semantic distance indicates the degree of correlation between two semantic regions of a document according to some application-dependent and user-aware criteria. A lower value denotes high correlation, while a higher value denotes low correlation.

Both the composition of the graph and the semantic distance metric are application-dependent and, thus, are provided by the application designers (see Sec. 5 for more information on the integration of the application and CoopSLA). It is worth noting that, model-wise, the granularity of the semantic regions can be arbitrary. The model simply assumes the existence of a semantic region graph with a metric function, through which it can obtain the correlation between two elements of the graph. Naturally, different granularities exhibit different properties and one granularity may be better suited for one application than another. The granularity used in our particular implementation (extension to *TexMaker*) is described in Sec. 5.3; we analyze the impact of different granularities in the performance of the system in Sec. 6.7.

We denote the participants of a cooperative editing session as a *cooperation group* and the machine of each participant is termed *node*; when referring to the application used by each user to participate in the cooperation group we use the term *client*. The client at each node of the cooperation group has a local *view* consisting of a full *local replica* of the shared application state. Each replica consists of a representation of the document space graph G in which each vertex additionally holds the contents of the corresponding semantic region. In the context of replication, we refer to a vertex of the graph as an *object*.

Each object has one primary/master replica that holds its most recent value. Clients modify the state of an object by submitting *updates* to the master. From the perspective of the CoopSLA model, an update corresponds to the addition or removal of an individual character. In other words, this means that the CoopSLA model detects and considers each individual character change. However, the model does not impose any constraints on the timing at which each update is submitted to the master. This decision is left to the designers of the cooperative editor running on top of CoopSLA. For example, updates may be submitted as soon as they are generated; or they can be locally aggregated (at the editing client machine) and sent at predefined events, such as after the user presses "Enter" or saves the document. We address this particular aspect regarding our collaborative *TexMaker* implementation in Sec. 5.3.

In addition to the primary replica, each object also has one or more secondary replicas that may have a *bounded staleness* with relation to its latest state (i.e. its state at the master). We say that an object is *stale* if it has not yet received all the updates that have already been committed at the master. The CoopSLA model, explained in the next section, defines and enforces, on a per-user basis, the bounds to the staleness of each object.

3. Interest Aware Consistency Model

To capture the interest of a user in the different regions of a document, the CoopSLA model incorporates the notion of a *pivot*. A pivot is a special object that corresponds to a user's observation point and according to which the consistency requirements of the user's view are managed. Broadly speaking, the pivot determines, on a per-user basis, (i) when an update to an object is allowed to be postponed, and (ii) under which conditions the previously postponed updates are required to be propagated, and to which user(s).

Each user may have multiple pivots, each corresponding to a semantic region in which he/she is interested. Our personal experience suggests that the most intuitive approach for editing purposes is to make the pivot correspond to the semantic region in which the cursor of the user lies. However, CoopSLA does not impose any restrictions to the number and placement of pivots. It is up to the designers of the application running on top of CoopSLA to define and manage the location of pivots. For example, a programmer can define a master pivot corresponding to each user's editing position and allow the users to explicitly define additional pivots through the application's GUI.

In this section, we describe the pivot-based CoopSLA consistency model in detail. For clarity, we first describe the main concepts of the CoopSLA model considering only one pivot (Secs. 3.1 and 3.2). Then, we briefly describe the generalization of the model for multiple pivots in Sec. 3.3.

3.1. *Consistency field*

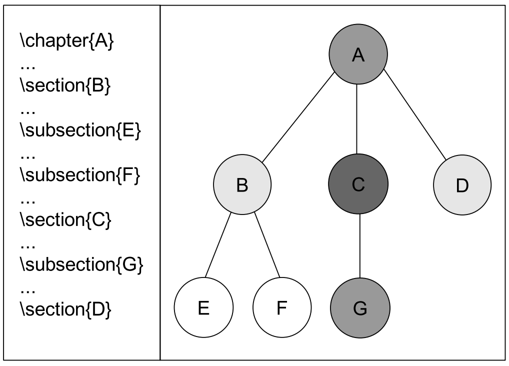
Each user's pivot p has a position in the application graph ($p \in V$). The location of the pivot can change throughout the execution of the application, mirroring the dynamic interest of the user in the document space (e.g. it can be embodied in cursor positions, current views/windows on the editor, marks left at the document, etc). The pivot generates a discrete consistency-field composed of N *consistency zones* $z_1, \ldots, z_n$. Each zone z_i has a radius r_i that defines the distance between z_i and the pivot. Radii are monotonically increasing, i.e. given two zones z_i and z_{i-1} with radius r_i and r_{i-1} (respectively), we have $r_{i-1} \leq r_i$. The programmer of the application defines default values for both the number of consistency zones and their respective radii; these values can be tuned by the user according to his particular interest on the document being edited.

At any moment, each object of the graph is assigned to a consistency zone of each pivot. The role of a consistency zone is to establish the *consistency requirements* of the objects assigned to it. The *strength* of the consistency requirements is inversely proportional to the radius of the zone: zones with smaller radius (hence, closer to the pivot) are awarded stronger consistency requirements; as the radii of the zones increase (along with the distance to the pivots), the consistency requirements become weaker.

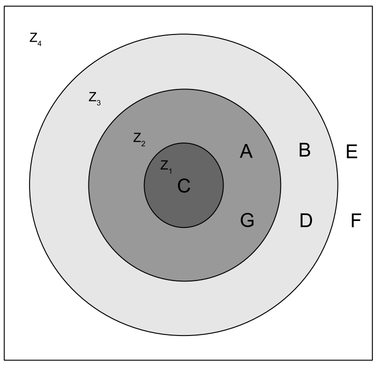
The distribution of objects across the consistency zones depends on two factors, (1) the semantic distance between the object and the pivot and (2) the radii of the consistency zones generated by the pivot. Zone z_1 is assigned the objects whose semantic distance to the pivot is smaller than the r_1, the radius of z_1; to each zone $z_i \in \{z_2, \ldots, z_{n-1}\}$ are assigned the objects with semantic distance between r_{i-1} and r_i; the outer zone z_n gets assigned every object whose semantic distance is greater than r_i. More formally, the consistency zone z_o of object o at semantic distance $d(p, o)$ from pivot p in the consistency-field is given by the following equation:

$$z_o = \begin{cases} z_1 & \text{iff } d(p,o) \leq r_1, \\ z_i, 1 < i \leq n-1 & \text{iff } r_{i-1} < d(p,o) \leq r_i, \\ z_n & \text{iff } r_{n-1} \leq d(p,o). \end{cases} \tag{1}$$

Figure 1 illustrates an intuitive example of a consistency field. In this example, the user is editing Sec. C of a Latex document (left side of Fig. 1(a)); as a result, the pivot (coinciding with the cursor) is placed in the corresponding C vertex of the application graph (right side of Fig. 1(a)). The consistency field generated (Fig. 1(b)) assigns the highest priority to updates to Sec. C, the current editing section. Updates to Secs. A and G (respectively, the parent and child Sections of C) have lower priority than C, but higher than Secs. B and D — the sections that are two graph hops away from C. Updates to Secs. E and F have the lowest priority. Still regarding this example, Table 1 describes, in the user's descending order of interest, the relation between each zone and the sections in the document. For example, zone z_1 corresponds to the section being edited, thus requiring a strong consistent view; from the user's point of view, other sections may have more relaxed consistency (being the sections in z_4 the least relevant).



(a)



(b)

Fig. 1. Example of consistency zones in a document structure. (a) Example document structure and corresponding graph representation with consistency requirements and (b) corresponding consistency zones.

Table 1. Example user interest.

Priority	Description	Zone
1	Current editing section	z_1
2	Parent and child sections of current editing section	z_2
3	Close sections — sections two graph hops away.	z_3
4	Remaining sections	z_4

3.2. *Consistency requirements*

Each consistency zone z_i has a corresponding *consistency degree* c_i that specifies the consistency requirements of the objects located within that zone. Consistency degrees respect the property $c_i > c_{i+1}$, meaning that consistency degree c_i of consistency zone z_i enforces stronger consistency than degree c_{i+1} of zone z_{i+1}. Consistency degrees are described by three-dimensional *consistency vectors* (κ) that limit the maximum divergence between the local replica of an object and the latest state of the object. Programmers, or even users in an enhanced tool, can define all or any subset of the following metrics:

- *Time* (θ): Specifies how long (in seconds) an object is allowed to remain without being refreshed with its latest value.
- *Sequence* (σ): Defines the maximum number of updates to an object that are allowed to be postponed (missing or outstanding updates).
- *Value* (ν): Specifies the maximum percentage divergence between the local contents of a replica of an object and its primary replica. *Value* is an application-dependent metric calculated by a special purpose function defined by the programmers of the application.

CoopSLA guarantees that fresh information about an object is sent to a client whenever at least one of the previous criterion is about to be violated; we refer to this event as *object timeout*. As long as these values are not reached, updates regarding that object (which would, without using CoopSLA, be sent to the client immediately) are retained, allowing for merging and compacting strategies to be applied. Remember that consistency specifications are set on a per-user and per-object basis; as a result, (i) the same object may be in different consistency zones regarding two different clients and (ii) different objects within the same consistency zone may be in different consistency states regarding a particular client.

Consider, for example, that user u_A is editing a Latex document. Further consider that $\backslash$*section*$\{B\}$ of the document is located within consistency zone z_B of user u_A, which has consistency requirements specified by consistency vector $\kappa_B = [10, 6, 20]$. CoopSLA provides the following guarantees to u_A regarding $\backslash$*section*$\{B\}$: at most 10 s have passed since the latest updates to $\backslash$*section*$\{B\}$ were sent to u_A; at most 6 updates to $\backslash$*section*$\{B\}$ have been retained at the server regarding u_A; the content of the local replica of $\backslash$*section*$\{B\}$ stored at u_A's machine differs, at most, 20% from the object's latest value (for example, 20% more characters have been written, or modified/delete, as a result of the updates).

3.3. *Model generalization*

As already mentioned in Sec. 2, CoopSLA allows the definition of multiple pivots for each user. For example, a user may be editing Chapter 3 of a book, while also being interested in the changes made to Chapter 7 (because such chapters are somehow related). Another similar situation may occur when editing multiple files: each one of a user's editing points in the different files may correspond to a different pivot. Furthermore, different pivots may have different consistency requirements, as it is natural that the current editing point is more relevant to the user. In a multi-pivot setup, an object's consistency zone is assigned with relation to its closest *pivot*.

The model also allows the definition of multiple views per user, which allows different sets of objects to be characterized with different consistency requirements regarding the same *pivot*. Consider a pivot that corresponds to the paragraph currently being edited by the user. In this scenario, a user may be more interested in sections he created than in sections created by other users, regardless of how close they are to the user. With multiple views, user created objects may be assigned more strict consistency requirements.

The model can also be generalized to encompass situations in which the document being shared spans across several individual files (e.g. in a folder, in which case all files are included during the rendering of the document). Typical examples are approaches based on master and input files, such as one main latex document that, when compiled, includes (via \input commands) a number of other documents that contain several chapters or sections each; another example is the use of master and dependent documents in Word files. The model encompasses these extended scenarios seamlessly as, regardless of the subdivision in individual files, the order by which they are referenced in the main document allows linearizing all the content as if, e.g. in a single latex file. To enforce this, implementations must track references, i.e. occurrences of file names in latex files, in order to detect dependencies (due to inclusion) on other files, and completely encompass the document space.

4. Architecture

CoopSLA is implemented by a middleware layer that aids programmers in the implementation of cooperative editing applications employing the CoopSLA approach. The middleware is responsible for managing the communication and replication aspects of the application. This way, programmers are relieved from most of the complexity of implementing a distributed application; instead, they are only required to implement a few conversion routines.

The CoopSLA middleware follows a client-server architecture, in which clients edit the shared document space and the server propagates updates to each client according to its consistency specifications. Following the replication model described in Sec. 2, the server holds a full replica of the shared application state (i.e. the application graph G). The server stores the primary replica of every object in the system and, thus, always has the most recent version of the objects. When the server receives a client update it applies it to its primary replica immediately; in contrast, it postpones propagating the received updates to the clients, as long as their consistency requirements are respected.

The main task of the server is to enforce the CoopSLA consistency model. This requires it to continuously monitor client updates and collect information about the current consistency state of each client. Periodically, and when updates are received at the server, it executes a validation algorithm that uses the collected data to verify if the consistency requirements of the clients are still met; if not, the server propagates to those clients the postponed (and possibly merged) updates needed to ensure that their consistency specifications are met.

A client consists of the editing application stacked on top of the middleware. The editing application receives the input from the user and passes it to the client-side middleware. The middleware on the client stores such input and applies it to the local replica. These updates are submitted to the server according to default configurations defined by the programmers of the application that can be tuned at runtime by the clients; Sec. 5.2 provides more information about this aspect.

Clients do not communicate directly with each other; update propagation is exclusively performed by the server. Each client holds a full replica of the data; however, unlike the server, these are secondary and, as a result, may have stale values that are managed according to each client's consistency specification.

4.1. *Software architecture*

Figure 2 shows the high-level architecture of the CoopSLA middleware, as well as its interactions with the application built on top of it and the system in which it is executed. The two main components of the CoopSLA core are the *Replica Manager* and the *Document Manager*. The Document Manager interfaces with the

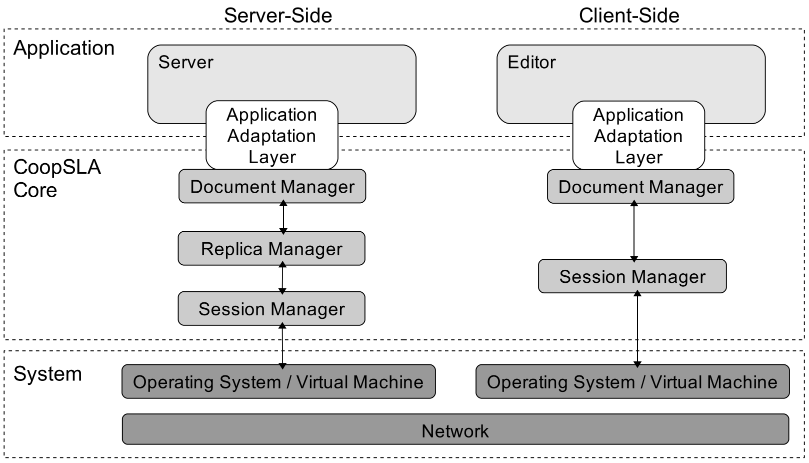


Fig. 2. High-level architecture.

application (via the *Application Adaptation Layer*) to convert the updates from its internal representation to the application's representation and *vice-versa*. Additionally, the Document Manager provides several functions necessary to verify the consistency state of the application (e.g. measuring/identifying the semantic distance between two semantic regions).

The Replica Manager (present only at the server) is responsible for storing the pending updates and managing the consistency state of each client. It maintains all the information necessary to enforce the consistency specifications of each client. It is also in charge of merging, compacting and issuing the dissemination of the postponed updates to the clients, according to their consistency specifications.

The Session Manager handles the communication part of the system. It establishes and maintains the connections between the nodes of the system and is the component in charge of sending and receiving the data, as illustrated in Fig. 2.

The Application Adaptation Layer serves as the glue that connects the middleware to the application. At the middleware side, it exports the API through which applications interact with CoopSLA. At the client side, it provides the programmer defined functions required by CoopSLA. We also assume and encourage programmers to encapsulate within this component all the modifications necessary for the application to use CoopSLA.

4.2. *Monitoring client activity*

To enforce the CoopSLA consistency model, the Replica Manager stores, for each client c_i, the client's consistency specification (pivots, zones and degrees) and *consistency state* table ψ_{c_i}. The latter stores, for each object o_i: (1) the time elapsed since c_i last received updates regarding o_i ($\psi_{c_i}[\theta, o_i]$); (2) the number of updates to o_i that have not yet been sent to c_i ($\psi_{c_i}[\sigma, o_i]$) and (3) the *value* of o_i the last time updates to it were sent to c_i ($\psi_{c_i}[\nu, o_i]$). In addition, the server maintains a *dirty table* that stores, for each object and each client, two bit flag: the *modified* flag indicates if o_i has been updated since the last time it was refreshed at c_i; the *dirty* flag will be explained later in this section.

Enforcing each client's consistency specifications requires the server to keep track of the following critical events (addressed below): content updates, structure updates and pivot movement.

4.2.1. *Content updates*

The algorithm for managing incoming updates at the server is presented in the Appendix in Algorithm A.1. When the server receives a content update, the Replica Manager first queries the Document Manager in order to identify the object o_i to which the update refers. Then, it marks the identified object as modified. Using the information retrieved, it updates the *sequence* state ($\psi_{c_i}[\sigma]$) of every client regarding o_i. Then, for each client c_i, the Replica Manager verifies if the new sequence state has reached the limit defined in c_i's consistency specification. To do so, it asks the

Document Manager to identify the semantic distance between o_i and c_i's pivot. Based on this distance, the Replica Manager retrieves the object's consistency zone (and respective consistency vector), and compares it with the current *sequence* state of o_i. If the limits of any client c_i have been reached, the Replica Manager marks o_i as *dirty* in c_i's *dirty table*. When executing the next iteration of the validation algorithm, the Replication Manager verifies that o_i is dirty regarding c_i and, as a result, propagates to c_i the updates to o_i that had been retained at the server.

4.2.2. *Structure updates*

Modifications to the structure of a document (e.g. adding or removing a section) change the distances between its regions. As a result, the placement of the objects within the consistency field of each client changes, and different consistency requirements have to be considered.

Structure updates are exchanged through special messages declaring the new semantic region, its placement within the document space and the header text that declares it (e.g. \subsubsection{*Structure updates.*}). When the server receives a structure update, the Document Manager updates its internal data structures to accommodate the new semantic region (see Algorithm A.3 in the Appendix). This includes adding a new node to the application graph and, possibly, moving text to the newly created semantic region.

After the update is performed by the Document Manager, the Replication Manager updates and re-evaluates the *sequence* state of each client for every object that moved to a *stronger* consistency zone; in addition, it marks the object as modified. Objects that are assigned to a *weaker* consistency zone as a result of the re-evaluation are not moved immediately. Instead, they are only moved after their previously postponed updates are sent to the client as a result of the enforcement of the consistency requirements specified by their current zone. Furthermore, it temporarily moves the objects modified by the structure update (the new object, its parent and its sibling regions) to the strongest consistency zone of every client, regardless of their actual relative position to the client's pivot. After these objects are updated at the client, the Replication Manager moves them to the consistency zones defined according to their distance to the client's pivot.

4.2.3. *Pivot movement*

As with structure updates, when a pivot moves to a different semantic region, the composition of its consistency zones changes. As a result, new consistency requirements have to be considered and the Replica Manager has to update its internal data structures accordingly. In particular, it must re-evaluate, for every object that moved to a different consistency zone, the *sequence* state of the client.

The process of updating the Replica Manager after a pivot movement is similar to a structure update: the consistency zone of an object is not updated immediately, unless it is stronger than its previous zone; otherwise, the object remains in the same consistency zone until its previously postponed updates are sent to the client as a result of an object timeout. The pseudocode of the algorithm is shown in the Appendix in Algorithm A.2.

4.3. *Consistency enforcement*

In each periodically executed round of the consistency validation algorithm (see Algorithm A.4 in the Appendix for more details), the Replication Manager verifies if any update received since the last round resulted in a violation of any client's consistency specification. If so, the identified updates are propagated to the clients concerned.

The validation algorithm verifies, for each client c_i and each object o_i, if the object is within the limits imposed by the consistency zone defined by the client's pivot(s). Because c_i may have multiple views and multiple pivots, the Replication Manager must first identify which pivot p_i enforces the strongest consistency requirements for o_i. This is done by identifying, for each possible pivot, the consistency zone of p_i in which o_i lies and retrieving the corresponding consistency vector κ_i. The appropriate vector is chosen by pairwise comparing the vectors and their respective metrics.

Next, each dimension of the identified κ_i is tested against the object's current consistency state. Verifying *time* (θ) and *sequence* (σ) is straightforward: for σ, the Replication Manager simply checks if the object has previously been marked as dirty (Sec. 4.2.1); for θ, it tests if the time elapsed since the last time c_i was updated with the latest version of o_i exceeds θ_{κ_i}. The latter verification requires the Replication Manager to store the last time each object was sent to a client.

To verify ν, on the other hand, the server has to compare the current value of o_i with the client's $\psi_{c_i}[\nu, o_i]$, which would require the Replication Manager to store, for every client, a copy of every object. To avoid the memory overhead of this solution, the Replication Manager asks the Document Manager to take a snapshot of an object whenever updates regarding that object are propagated to a client. Before taking a snapshot, however, it first verifies if a snapshot of the object already exists; if it does, the server uses it, avoiding an unnecessary copy and saving memory.

4.4. *Data representation and update propagation*

The CoopSLA middleware represents the contents of each object of the graph as a TreeDoc Commutative Replicated Data Type (CRDT).[10] As a result, the updates exchanged between CoopSLA nodes consist of requests to *add* or *remove* TreeDoc identifiers. By using TreeDoc, we enable replicas to converge without the need for complex conflict resolution protocols, which further enhances the relaxed synchronization properties provided by CoopSLA.

It should be noted that while the middleware uses TreeDoc as a building block, the CoopSLA model is completely independent of it. In fact, the model is compatible with any other update representation approach, including *Operational*

Transformation and state-transfer.[5] As such, it is possible to modify the implementation of the middleware so that it uses a different update representation approach. In addition, from an architectural point of view, CoopSLA is also independent of TreeDoc; as explained in Sec. 5.1, TreeDoc can easily be replaced by using a different Document Manager (Fig. 2).

Updates that have not yet been propagated to a client are stored at the Replication Manager in a per-client *update queue*. When adding a new update to a queue, the server automatically merges add/remove operations that cancel each other. In addition, when an object timeout occurs, the Replication Manager aggregates the updates stored in the corresponding client's queue into a single message. Even with these simple mechanisms, the results (Sec. 6) proved to be very encouraging. Additional (or complementary) forms of update merging and message compacting solutions are out of the scope of this paper.

5. Implementation

We implemented a prototype of the CoopSLA middleware and modified the Linux version of the Latex editor *Texmaker*[a] on top of it. In this section we describe the main implementation details of the middleware (Sec. 5.1) and the extension to *TexMaker* (Sec. 5.3). We also briefly explain how programmers interact with the middleware and specify the CoopSLA consistency settings in Sec. 5.2.

5.1. *Middleware*

The CoopSLA prototype is entirely implemented in C++ and comprises approximately 5000 lines of code. The current implementation targets Linux systems and makes use of its system libraries. For portability, we wrote a *System Abstraction Layer* that encapsulates the calls to the system libraries; as a result, to port the middleware to a different architecture it is only necessary to re-implement this library. The library provides threading and mutual exclusion support (currently built on top of Unix *pthreads*), as well as functions for data serialization and network communication.

The implementation follows the architecture shown in Fig. 2. The Document Manager is implemented in a DocManagement module that contains the classes that represent, parse and extract information from the document space. The document space is represented as a tree of semantic regions. Each semantic region is represented by a uniquely identified SRegion object. Each SRegion contains a list of children subregions and a TreeDoc with the contents of the region it represents. By having one TreeDoc per semantic region, the size of the TreeDoc identifiers exchanged in each update message is smaller than if we simply had one TreeDoc representing the whole document. As a result of this design choice, each update message also carries the identifier of the SRegion to which the updates concern.

[a]http://www.xm1math.net/texmaker/.

To facilitate the modification of the update representation strategy, the Document Manager is the only component that is aware of the specific strategy in use (TreeDoc, in the current implementation). The remaining components of the middleware are only aware of the existence of a semantic region graph and abstract updates. Information that depends on the specific update representation is obtained by consulting the Document Manager. The Document Manager provides, among others, functions to apply updates to the local replica, obtain the semantic difference between two versions of the same object and calculate the semantic distance between two objects. Examples of the interactions between the Document Manager and the remaining components can be seen in the algorithms present in the Appendix.

The Replication Manager and the Session Manager are implemented in a Client-Management module. At the server, the Replication Manager stores the consistency related data in tables and the pending updates in per-client queues. The most important tables are the consistency state tables of each client. Each table holds the current values of the consistency metrics of a client, which include a pointer to an object snapshot (as described in Sec. 4.3). Implementing the object snapshot approach requires rounds, snapshots and objects to be *versioned*. The round number r_v is an integer number that is incremented in every round. The version of a snapshot correspond to the r_v of the round in which the snapshot was taken; object versions correspond to the r_v of the round in which the object was last updated. Similarly, to save memory, the per-client queues do not store actual updates; instead, they point to the updates stored in a global update queue that stores all the updates that have not yet been sent to, at least, one client (or merged). Both snapshots and updates are garbage collected by the middleware when no reference to them exists.

5.2. *Application integration*

In our current implementation, programmers describe the consistency requirements using an XML file. When the client application (e.g. *TexMaker*) starts, it invokes an API registration function with the path to the XML file as its argument. The CoopSLA client then parses the file and sends a registration request to the server.

Programmers control the structure of the document by adding or removing semantic regions using the API functions (addSRegion/removeSRegion) exported by the *Application Adaptation Layer*. Both functions receive the parent of the new region, the region *type* and, optionally, a set of semantic links to other regions. The region type is an application-dependent data type used to aid in the identification of the consistency zone of the object regarding a particular pivot.

The input received by the editing application is passed to the CoopSLA middleware (at the client) using function *inputReceived*. Programmers can configure the timing with which the client-side of the middleware sends the updates to the server through function *setPropagationTiming*. The middleware accepts three types of propagation timings: *immediate*, in which updates are sent to the server when function inputReceived is executed; *periodic*, in which updates are stored when function inputReceived is called and sent later according to a periodicity defined with function setPropagationTiming; or *explicit*, in which case updates are only sent to the server after the application calls function *propagateUpdates*. Besides the global timing levels, programmers can control the propagation timing of individual updates through function inputReceived.

Programmers must provide two additional functions to be called by the middleware when checking a client's consistency: **semanticDifference** and **getConsistencyZone**. The **semanticDifference** function is used to verify the *value* metric. It returns the (application-specific) percentage difference between two versions of an object. **getConsistencyZone** is a function that, given a pivot and a graph object, returns the consistency zone of the object regarding the pivot. To aid in the implementation of this function, our API provides functions that interact with the Document Manager in order to extract information about the graph path between two objects, such as the number of hops or the relative tree height separating them.

5.3. *CoLaTex*

To validate our system, we have extended the Tex editor *Texmaker* to support concurrent editing between distributed clients. The extensions were introduced using *TexMaker*'s *add-ons* feature without modifying the code of the application. Our add-ons consist of simple functions that interact with the CoopSLA middleware (through the *Application Adaptation Layer*) to perform the following actions: intercept the user's modifications to the local replica of the shared documents; insert the updates received from the server into the Latex document; and keep track of the editing position of the user (i.e. the cursor). We implemented an additional add-on that generates the document space graph by parsing the Latex source files and converts the updates from (and to) the TreeDoc representation used by the Document Manager.

We chose the *subsection* construct as the minimum object granularity of our system. Our decision was based on our intuition that a subsection is (or should be) a self-contained coherent element and, as such, modifications to any of its positions are equally important. In addition, the performance results presented in Sec. 6.7 show that this granularity provides a good tradeoff between network savings, CPU consumption and memory usage. Regarding update timing, in our implementation of CoLatex, updates are sent to the server every second. This ensures a *quasi-real-time* awareness over the work performed by other users;[23] on the other hand, by not sending each update immediately, we allow updates to be batched and sent in a single message to the server.

We define one pivot for each open file, each corresponding to the semantic region being edited by the user within that file. The add-ons we developed allow the user to manually assign a region of the document to a consistency zone. Our **semanticDifference** function returns the percentage difference in number of characters between

Table 2. Consistency zones.

Zone	Time (θ)	Sequence (σ)	Value (ν)
1	1 sec.	1 update	1%
2	10 sec.	15 updates	5%
3	40 sec.	100 updates	30%
4	2 min.	750 updates	60%
5	3,5 min.	1000 updates	90%

two versions of an object. Table 2 describes the consistency zones we defined; our **getConsistencyZone** function returns the consistency zone of an object based on the following considerations:

- *Consistency Zone* 0 includes the region in which the pivot is placed and its direct children, i.e. the ones that are one graph-hop below the pivot.
- *Consistency Zone* 1 contains the direct parent — the region one graph-hop above the pivot — and indirect children — identified by traversing the graph downwards from the pivot, excluding the direct children — of the pivot.
- *Consistency Zone* 2 comprises the regions that belong to the same *chapter* as the pivot. If there is no explicit \chapter defined, we consider that the document has one implicit chapter of which every section is a part of.
- *Consistency Zone* 3 includes the top-level sections of the remaining chapters (*chapter*) of the document. If the document does not have chapters, zones two and three are merged and zone four is awarded the consistency specifications originally defined for zone three.
- *Consistency Zone* 4 contains the regions that do not belong to any of the previous zones.

As described in the next section, we used the CoLatex editor to evaluate our solution.

6. Evaluation

In CoopSLA, we claim that it is possible to make an efficient use of the available network resources by taking the individual *locality of interest* of each user into account. To validate this assertion, we conducted a series of experiments to evaluate the network performance of the system; in particular, we quantify the savings obtained by CoopSLA (when compared to a solution without the CoopSLA model) regarding the two following aspects: (1) the bandwidth required to propagate the modifications performed to the shared document space by different users (Sec. 6.2) and (2) the number of messages sent (i.e. the stress imposed on the network resources) as a result of that propagation (Sec. 6.3).

We also analyze the flexibility of the model (Sec. 6.4); thus, we studied the impact of different consistency specifications in the network savings previously mentioned. In addition, we analyzed the performance of CoopSLA in terms of CPU and memory requirements at the server to illustrate the tradeoff from network gains (Sec. 6.5). We also briefly analyze how different editing patterns influence the performance of CoopSLA, regarding network, CPU and memory usage (Sec. 6.6). Finally, in Sec. 6.7, we describe the study and results regarding the impact of the granularity of the semantic regions in the performance of CoopSLA.

In the following sections we detail the experiments conducted. We first describe the simulation environment and the parameters considered during the evaluation process. Next, we present and analyze the results obtained regarding the metrics described above.

6.1. *Simulation environment*

Clients are simulated by running a predetermined number of parametrized *editing bots*. Bots perform text insertions (write or paste), deletions (erase or cut) and browse through the document space. Table 3 shows the probability distribution that models the behavior of the bots used in our experiments.

Each simulation consisted in a series of 10-min runs during which bots executed for 7 min according to their decision tree, sending the corresponding updates to the server, as well as receiving and applying to their Document Manager the updates propagated by the server. The remaining 3 min were necessary to ensure that every update received by the server was either merged and discarded or sent to every user as a result of object timeouts.

To analyze the scalability of the system, we varied the number of bots participating in each simulation cycle. During each individual run, the server monitored inbound and outbound traffic, storing per-client and overall statistics regarding the amount of data transferred and the number of messages exchanged. At the end of each simulation cycle we computed the average values for each of the results obtained.

We compared CoopSLA with a baseline TreeDoc implementation that propagates updates to every user as soon as they arrive at the server; in the figures, and throughout this section, we refer to this version of the system as BL (short for *Baseline*). We also varied the type of document space edited; we considered three types of documents that differ mainly in size and structural complexity: a *Small* document with ten to fifteen semantic regions, each with a few text paragraphs (representative of a research article); a *Medium* size document with forty to sixty semantic regions (modeled after a short book or a company report); and a *Large* document space comprising over eighty semantic regions with large text

Table 3. Bot decision tree.

	Add		Remove		Move	
Read	Write	Paste	Erase	Cut	To sides	Up/down
60%	15%	3%	8%	3%	8%	3%

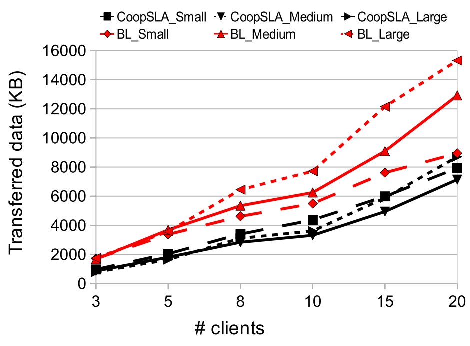
chunks each (representative of a scientific book). Unless told otherwise, the results presented were obtained using the consistency requirements described in Table 2.

The tests were conducted on Intel Core 2 Quad machines with 8GB RAM running Ubuntu Linux. The server was executed on a dedicated machine, clients were deployed on up to three machines. The computers were connected through a Gigabit Ethernet LAN.

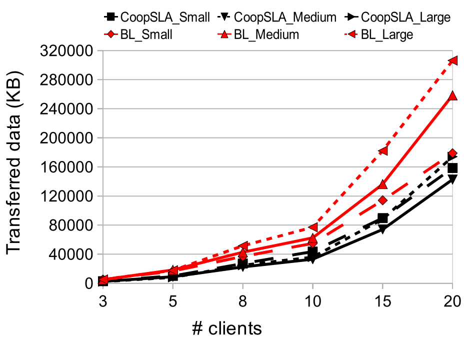
6.2. *Transferred data*

Figure 3 presents the average amount of data received by each client (Fig. 3(a)) and sent by the server (Fig. 3(b) per-simulation. We plotted the results of both CoopSLA and BL for an increasing number of users and the three types of documents we consider. The figures show that CoopSLA is able to effectively reduce the amount of data transferred over the network (when compared to BL) and, consequently, the bandwidth necessary to do so, both at the clients and the server. Moreover, we can see that, as the size of the documents increases, CoopSLA is increasingly more efficient in obtaining bandwidth savings. This behavior shows the scalability of the system, and is especially relevant considering that as an editing project grows in size, it is more likely that more users will cooperatively access it.

The main reason for these results is that in larger documents the average distance between the editing regions of each user is higher; as a result, the probability of postponing and, possibly, merging updates also increases. This fact is particularly evident if we individually compare CoopSLA with BL for each type of document space considered. With the simpler document space (CoopSLA_Small and BL_Small), the bandwidth savings obtained by CoopSLA are minimal, because the probability that an update occurs in a client's pivot region (and, consequently, is propagated immediately) is high. If the document grows in size and complexity, the savings obtained by CoopSLA increase greatly to approximately 50% at the end of the simulation.



(a)



(b)

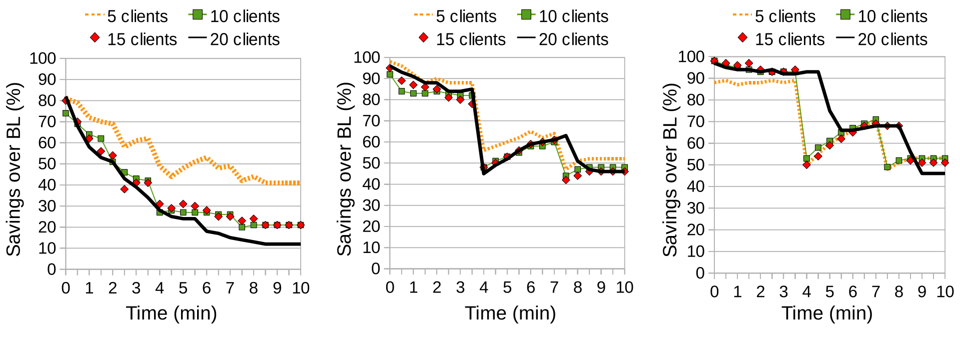
Fig. 3. Data transferred over the network. (a) Average per-client inbound traffic and (b) total server outbound traffic.

Another important conclusion that can be inferred from Fig. 3 is that CoopSLA is able to efficiently minimize one of the main drawbacks of the TreeDoc approach, i.e. the size overhead of path identifiers. When a document is represented as a TreeDoc, the size of the path identifiers increases as the document grows. Because update messages exchange path identifiers, when a document grows in size, the update messages follow the same pattern. Without CoopSLA, the larger the document is, the larger update messages are; as a result, the bandwidth required to propagate and receive updates increases. With CoopSLA, on the other hand, we take advantage of the accumulation of postponed updates at the server to merge and aggregate them before propagating them to the clients, which allows us to maintain bandwidth requirements under more acceptable levels.

The results presented so far denote the savings obtained at the end of the simulations. Figure 4 adds to this analysis by showing the evolution of savings obtained during the execution of the simulation. The figure shows the percentage of traffic saved when using CoopSLA over BL, i.e. how much less traffic was exchanged during the simulation using CoopSLA as compared to BL. The results show that, most of the time, the savings obtained by CoopSLA are higher than the final savings. In particular, during the first minutes of the simulations, CoopSLA is able to achieve over 80% savings. Around 3.5 min into the simulation, savings experience a sudden decrease as a result of the timeout of the weakest consistency zone (z_5). This decrease is exacerbated by the fact that every bot is launched at the same time and, thus, experience the timeout of their respective weakest consistency zone at approximately the same time. After this initial period, the results follow a cyclic pattern in which savings steadily increase for a few minutes, until the timeout of zone z_5 occurs again.

6.3. *Exchanged messages*

While the overall traffic generated by the clients is influenced by the specific characteristics of TreeDoc, the number of update messages issued by each client depends only on the editing pattern of the users. By measuring the number of messages



(a) (b) (c)

Fig. 4. Traffic savings over time. (a) Small, (b) medium and (c) large.

exchanged over the network, we are able to clearly isolate and analyze the individual contribution of CoopSLA. Furthermore, it provides a good indication of how CoopSLA would work with different *operation-transfer* strategies, like OT or other CRDT implementations. Figure 5 shows the results of these measurements for both CoopSLA and BL.

The results further confirm the ones regarding bandwidth. As a matter of fact, Fig. 5 shows that CoopSLA, when compared to BL, is able to reduce the number of messages received by each client and such savings increase with the size and complexity of the edited document. Again, these results are a direct consequence of CoopSLA's ability to leverage the accumulation of low-priority postponed messages by merging those that cancel each other. As a result, CoopSLA discards unnecessary messages that would, otherwise, be propagated immediately to each client. Furthermore, CoopSLA obtains additional savings by aggregating multiple updates into a single message.

The advantages of the message savings obtained are manifold. First, it means that mobile devices using CoopSLA make a more efficient and less demanding usage of the network resources, which contributes to reducing battery consumption

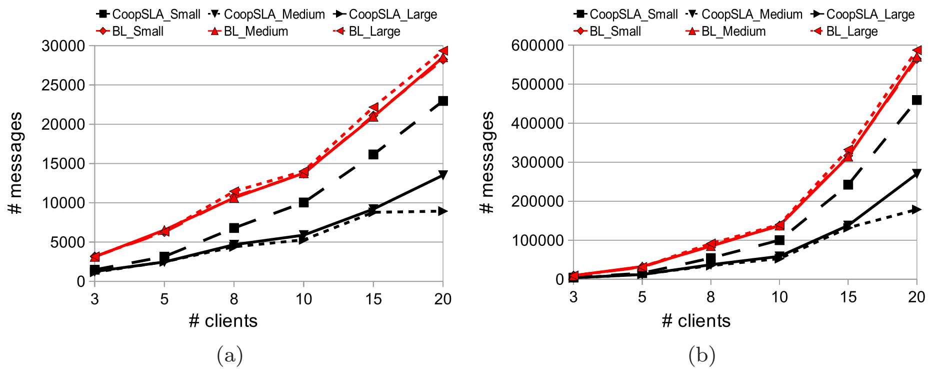


Fig. 5. Messages exchanged. (a) Average # messages received per-client and (b) total number of messages sent by the server.

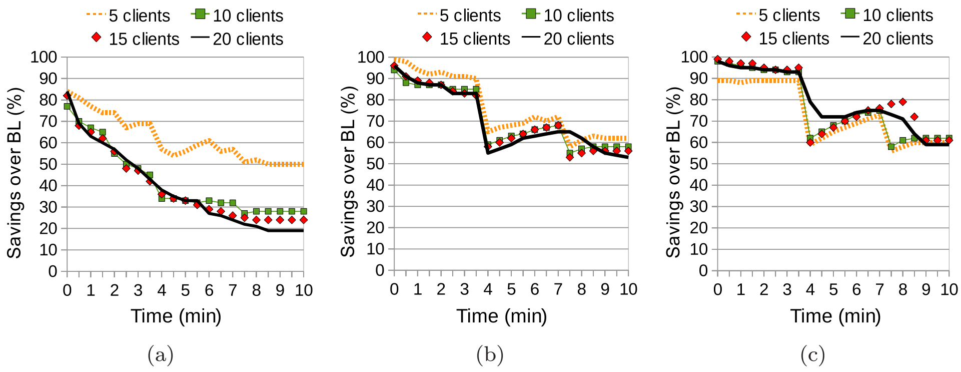


Fig. 6. Message savings over time. (a) Small, (b) medium and (c) large.

(a critical aspect in mobile devices). Second, because there is a lower number of messages and less data to process at each client, the impact on the performance of the device is less pronounced, leading to an overall higher responsiveness. Finally, it has a direct influence on the savings obtained regarding bandwidth requirements: by propagating the same amount of data using fewer messages, we minimize the overall amount of overhead data (headers and meta-data) transferred. Naturally, because such extra data comprises only a small percentage of the overall data present in the message, the bandwidth savings obtained are not as high. Specifically, we observed (in additional measurements not shown in the figures) that message aggregation accounts for roughly 5% of the bandwidth savings obtained; in comparison, around 20% of message savings are a result of aggregation.

Figure 6 shows the average message savings (as a percentage of the number of messages exchanged using BL) obtained by CoopSLA over the course of the simulations. As expected, these results follow a similar pattern to the savings obtained with the transferred data. However, CoopSLA is able to achieve around 10% higher savings in exchanged messages in comparison with exchanged traffic. This is a natural result of the higher impact that message aggregation has on message minimization, as explained in the previous paragraph.

6.4. CoopSLA *flexibility*

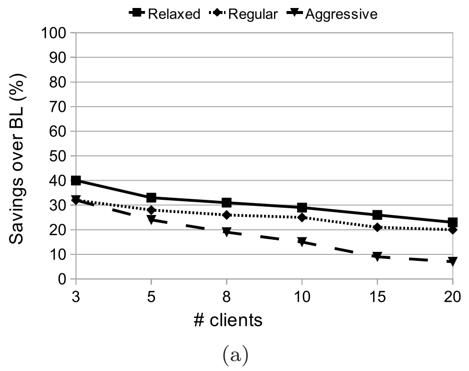
The results presented so far misleadingly indicate that when a document is small, the advantage in using CoopSLA is minimal. However, CoopSLA allows application programmers to specify consistency requirements arbitrarily, as they see fit for their applications, and users to easily configure them. As long as possible, a programmer should try to relax the consistency requirements, always ensuring the application provides the required levels of interactivity. If necessary, however, the programmer can define more demanding requirements.

To analyze the flexibility of CoopSLA, we measured the savings obtained with three CoopSLA consistency specifications (described in Table 4) that differ in how aggressively update propagation is issued. Figure 7 shows the results obtained; Figure 7(a) shows message savings, while Fig. 7(b) shows traffic savings. In both cases, the results are presented as a percentage of the values obtained with BL. The measurements were conducted with the Small document space.

As expected, the figure shows that CoopSLA is more efficient when the consistency requirements are more relaxed. These results are a consequence of the fact

Table 4. Consistency zones.

Zone	Relaxed	Regular	Aggressive
1	$\{\theta=5,\sigma=10,\nu=5\}$	$\{\theta=2,\sigma=5,\nu=5\}$	$\{\theta=1,\sigma=1, \nu=1\}$
2	$\{\theta=20,\sigma=30,\nu=10\}$	$\{\theta=10,\sigma=10,\nu=5\}$	$\{\theta=5,\sigma=5,\nu=5\}$
3	$\{\theta=40,\sigma=100,\nu=50\}$	$\{\theta=40,\sigma=100,\nu=30\}$	$\{\theta=15,\sigma=15,\nu=20\}$
4	$\{\theta=120,\sigma=750,\nu=60\}$	$\{\theta=90,\sigma=300,\nu=60\}$	$\{\theta=30,\sigma=50\ \nu=50\}$
5	$\{\theta=300,\sigma=1500,\nu=90\}$	$\{\theta=180,\sigma=750,\nu=80\}$	$\{\theta=60,\sigma=150,\nu=50\}$



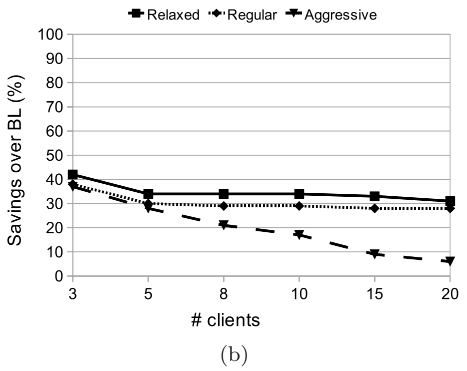


Fig. 7. Savings over BL with different consistency specifications. (a) Traffic savings and (b) message savings.

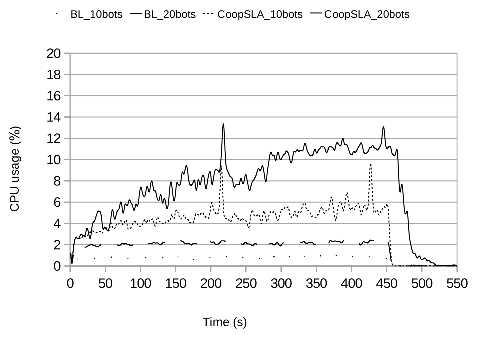
that relaxed consistency requirements allow for a larger volume of updates to be retained at the server for longer periods of time; as a result, the probability that two updates can be merged increases. Conversely, when the aggressiveness of the requirements increases, a lower volume of updates is retained at the server, which results in a lower merge efficiency. However, the results show that even with a fairly strict set of requirements, CoopSLA is able to obtain more than 20% bandwidth savings over BL.

6.5. *Resource usage*

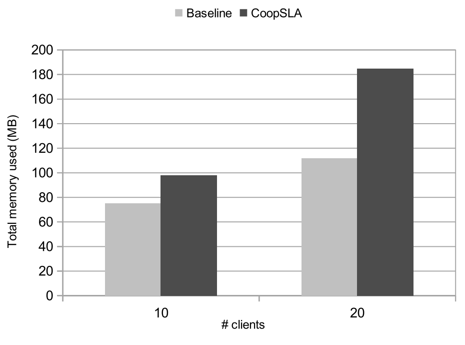
In addition to the network metrics presented so far, we analyzed the performance of CoopSLA in terms of CPU and memory requirements at the server. The data was gathered using the *top* tool available in Linux, by polling its output every second during the execution of the simulations. For clarity, we plotted only the data obtained with 10 and 20 clients and the Medium document space. The results are presented in Fig. 8.

The results show the tradeoff in CoopSLA between server load (CPU and memory) and network usage. More specifically, as expected, CoopSLA presents a computational overhead, in comparison with BL, that is due to the fact that CoopSLA must maintain and continuously process the data structures necessary to enforce each client's consistency requirements. BL, on the hand, does not store any additional information and simply propagates the updates as soon as received.

Figure 8(a), in particular the line regarding 20 bots, shows that the CPU usage of CoopSLA increases slightly for the first few minutes of the simulation and stabilizes shortly after it reaches a load spike at around 210 s. The initial increase is a natural consequence of the gradual accumulation of updates: as the number of postponed updates increases, the merging process becomes gradually heavier. This overhead reaches its peak at 210 s, which is when the timeout of the weakest consistency zones occurs. At this point, the number of postponed updates is at its highest, which requires more processing time to merge every update. This load peak, however,



(a)



(b)

Fig. 8. Resource usage at the server. (a) CPU usage and (b) memory usage.

does not compromise the performance of the system, since it does not even reach 20% CPU usage.

After this first load peak occurs, the number of updates retained at the server decreases, resulting in the corresponding reduction in server load. From that moment on, the average CPU load remains stable, because the average number of updates retained at the server, at any moment, also stabilizes. The slight deviations observed are a result of the periodic timeouts of the different consistency zones.

Regarding memory usage, the extra memory required by CoopSLA corresponds mostly to the updates retained at the server and the data structures maintained by the server to manage the consistency state of each client. In any case, the overhead introduced by CoopSLA is under an acceptable threshold. This overhead denotes the tradeoff between memory, a highly available resource (for instance, personal computers today are typically equipped at least with 4GB of RAM, and even tablets and smart phones have 1GB or more), and the network resources, which are more scarce and unreliable.

6.6. *Varying editing patterns*

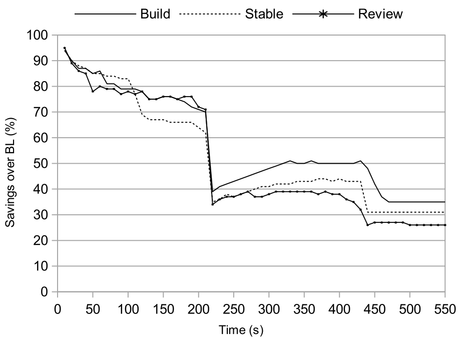
In the experiments presented so far, every client has the same editing pattern. In a real environment, however, different users have different roles and different skills, which results in different editing patterns. To analyze the influence of this aspect on the performance of CoopSLA, we designed a test case in which bots with different editing patterns are launched in the same simulation. We considered two representative patterns, a *writer* and a *reviewer*. The writer follows the same pattern as the bots used in the previous experiences. The reviewer, on the other hand, traverses the document space, changing frequently between semantic regions. It also performs content modifications, but at a much lower rate than the writer bots.

Having these two patterns defined, we varied the ratio between writers and reviewers in a set of 20 client bots. We chose three configurations: a *build* setup with

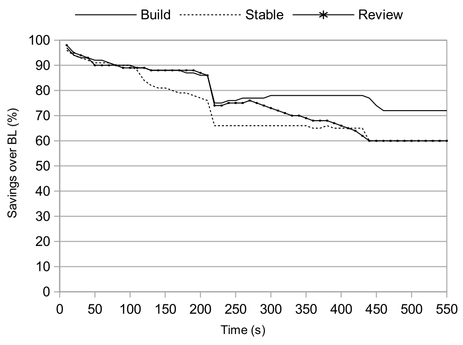
15 writers and 5 reviewer that tries to mimic the initial stages of the collaborative writing process; a *stable* configuration with 50% writers and 50% reviewers; and, finally, a *review* scenario in which 15 bots are reviewers and 5 are writers. For each scenario we measured the network savings obtained over BL, as well as CPU and memory usage. The results are shown in Figs. 9 and 10.

Figure 9 shows that network savings increase with the number of writers. This is due to the fact that writers edit more steadily and in more localized regions of the document space (in comparison with reviewers), which increases the probability that two updates cancel each other. In any case, the differences between the results obtained in the different scenarios are small. Furthermore, even in the scenario dominated by reviewers, CoopSLA is able to achieve at least 20% savings over BL.

Regarding resource usage, the *build* scenario is the one that achieves lower performance, as shown in Fig. 10. Ironically, the reason for this comparative performance loss is the same reason that makes the *build* configuration obtain better network results: because writer bots edit in a more intensive way, they generate a higher

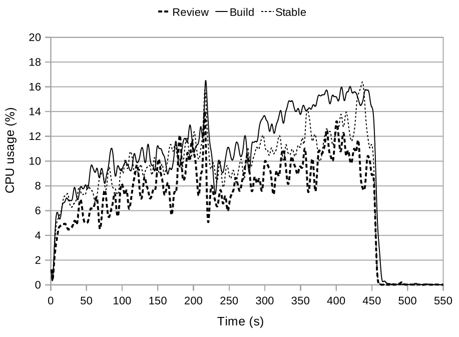


(a)

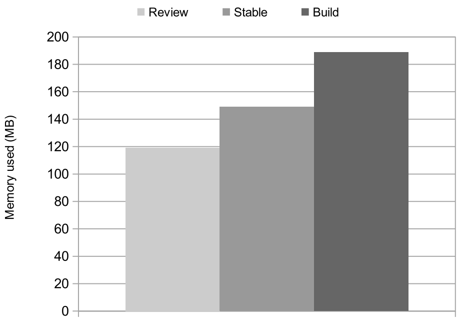


(b)

Fig. 9. Evolution of network savings with different editing patterns. (a) Data transferred and (b) exchanged messages.



(a)



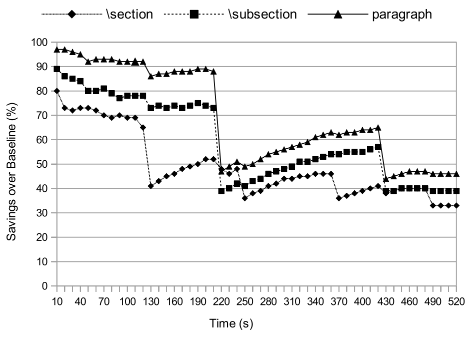
(b)

Fig. 10. Resource usage with different editing patterns. (a) CPU usage and (b) memory usage.

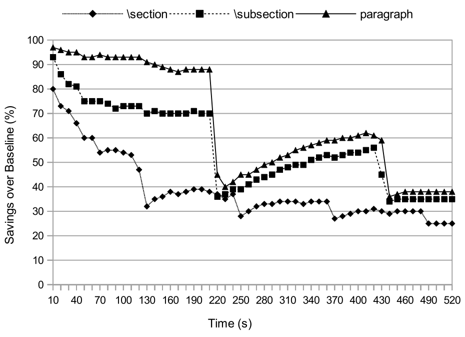
number of updates. As a result, the server ends up having to retain a larger amount of updates. In the same way, as we have already seen in the previous section, the larger number of updates retained at the server results in an increase in CPU usage, mainly due to update merging.

6.7. *Semantic region granularity*

We conducted a series of additional experiences to analyze the impact of the granularity of the semantic regions in the performance of the system. We considered three different approaches with decreasing levels of minimum granularity: \section, \subsection and paragraphs (actual text paragraphs, not Latex \paragraph). For each granularity level, we measured network, CPU and memory usage, with 10 and 20 clients and considering the Medium document space. The results are shown in Figs. 11, 12 and 13.

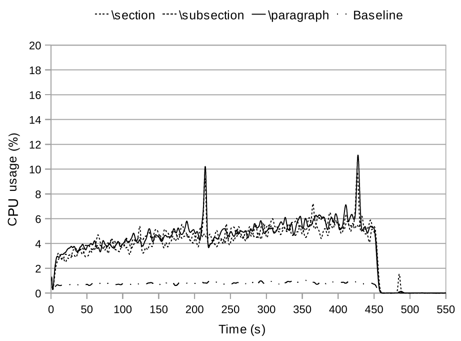


(a)

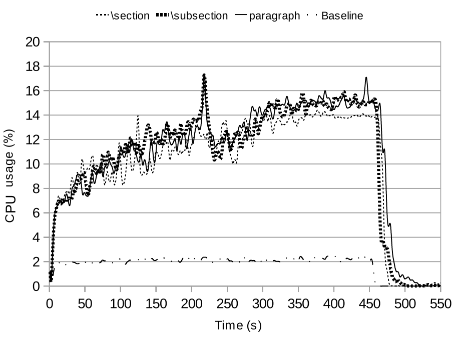


(b)

Fig. 11. Network savings with different granularities. (a) 10 clients and (b) 20 clients.

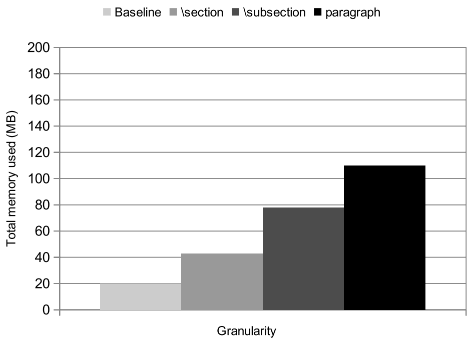


(a)

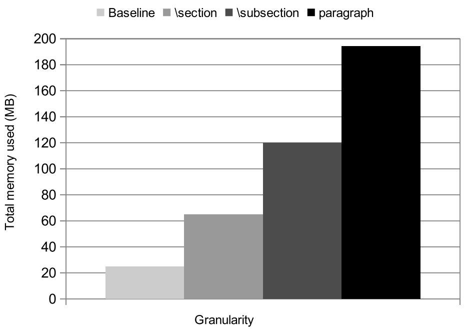


(b)

Fig. 12. CPU usage with different granularities. (a) 10 clients and (b) 20 clients.



(a)



(b)

Fig. 13. Memory usage with different granularities. (a) 10 clients and (b) 20 clients.

The results obtained confirm, once more, the expected behavior of the system: as the granularity level decreases, network savings increase and, consequently, resource usage increases accordingly. The increase in network savings occurs because with smaller granularities the number of semantic regions increases. As a result the distance between the pivots of each user also increases and, consequently, so does the average semantic distances between the pivots and the semantic regions updated during each session. This fact increases the percentage of updates assigned to weaker consistency zones, which means that a higher number of updates are retained at the server, and for longer intervals. Hence, the probability of having canceling updates at the server is higher, and merging occurs more frequently.

As shown in Fig. 13, decreasing the granularity of the semantic regions has the opposing effect of increasing memory usage. This occurs because CoopSLA's data structures (the object graph and the consistency related tables) become larger and populated with more objects. Despite this fact, CPU usage is not subject to significant differences when granularity changes. The reason is that the more time consuming operations of the server are related with update processing (e.g. update merging) and the number of updates does not change with granularity. The size of the data structures does not have a significant effect on CPU usage, because the operations over these data structures are mostly localized (e.g. direct lookups of primitive types on the consistency table or traversing short distances within the object graph).

Our conclusion is that while smaller granularities consume more memory, the values observed are not sufficient to cause a visible performance degradation at the server. Hence, the choice of granularity is ultimately a decision that depends on the goal of the application and the expectations of the users. For example, a smaller granularity may be chosen if users do not want to be disturbed by frequent remote modifications. In any case, the strength of CoopSLA is that it is very flexible, allowing any level of granularity, as well as highly customizable consistency specifications.

7. Related Work

In this section we discuss prior work on the two topics that are more closely related to our work: divergence bounding and consistency in cooperative editing.

7.1. *Divergence bounding in optimistic replication*

Designers of replicated systems typically choose between pessimistic and optimistic consistency models.[5] In many cases, however, neither the performance overheads imposed by strong consistency nor the lack of limits for inconsistency are acceptable to applications. An interesting alternative called *divergence bounding*[20] allows updates to be managed optimistically, but defines under which conditions replicas are required to converge and how to enforce that convergence. One obvious approach to divergence bounding is to force replicas to synchronize after a specified maximum time has elapsed.[24,25] Another simple, yet effective, solution is to limit the number of updates that can be applied to a local replica without synchronization.[26,27]

The TACT[28] framework proposes a multi-dimensional approach to divergence bounding that unifies in a single model three metrics: real-time guarantees, order bounding and a novel metric called *Numerical Error* that bounds the total number of updates, across all replicas, that can proceed before replicas are forced to synchronize. Our work distinguishes from TACT by embodying the notion of locality-awareness into the consistency model. This allows our system to implicitly assign different priorities to different updates that may vary throughout execution.

Vector Field Consistency[29,30] (*VFC*) is a consistency model for mobile multiplayer games that enables replicas to define their consistency requirements in a continuous consistency spectrum. The novelty of the VFC model is that it combines multi-dimensional divergence bounding with *locality-awareness* to improve the availability and user experience while effectively reducing bandwidth usage. Consistency between replicas strengthens as the distance between objects decreases. To define such mutable divergence bounds, around pivots there are several concentric ring-shaped *consistency zones* with increasing distance (radius) and decreasing consistency requirements (increasing divergence bounds). Then, in each zone, like in TACT, programmers define a three-dimensional vector: *time*, *sequence*, *value*.

None of the previous approaches tackle the issues addressed by our work to the same extent as CoopSLA. They either do not take locality in the data/object space into account, with a one-size-fits-all approach across all the data or for each data type (TACT), or they assume an orthonormal data space, typical in multiplayer game scenarios (VFC), that is not able to handle effectively data organized in graphs, hierarchies, or trees, such as documents in cooperative editing.

Regarding performance, TACT bandwidth savings are always lower bounds for the bandwidth savings attained with CoopSLA, while obeying to the same divergence bounds. By virtue of not considering locality awareness, TACT needs to enforce the strictest possible bounds globally, while CoopSLA only needs to apply them to the document section(s) where the pivot is, and more flexible bounds elsewhere.

VFC is unable to deal efficiently with a non-orthonormal data/object space, such as a graph or a document section hierarchy. Thus, while providing better savings than TACT, it can only consider a linear notion of distance between the updated data and the user pivots (e.g. distance in document file offsets). As a result, it will use as sizes for inner consistency zones the maximum length of relevant regions in the document (maximum number of characters of a paragraph, subsubsection, subsection, section), which will be an overestimate for every other region of the same level in the document. Thus, to enforce a given vector of divergence bounds associated to a document-like structure, it needs to enforce more conservative divergence bounds, which will result in more frequent update transfers, outside more relevant regions.

7.2. *Consistency in cooperative editing*

The issue of maintaining replica consistency in cooperative applications has been extensively studied in the last two decades. The most representative solutions fall into the *Operational Transformation* (OT) category.[2,6–8,31,32] In OT, each locally generated operation is associated with a timestamp and broadcast to the remaining sites. Then, each remote update received is transformed (e.g. by adjusting its insert/delete index) in order to commute with concurrent operations already applied to the shared document. As a result, transformed operations can be executed without re-ordering previously applied operations.

OT transforms updates in order to make them commute. A recently proposed alternative is to make every operation automatically commutative by representing the document as a *Commutative Replicated Data Type* (CRDT).[10–14] The CRDT approach considers that a document is composed of a sequence of immutable and uniquely identified elements that can be any non-editable component of a document, like a character or a graphics file. Commutativity is achieved by designing an identifier space that ensures that it is always possible to create a new identifier between two existing ones.[10]

The vast majority of existing work on cooperative editing (including OT and CRDTs) does not consider the dynamically changing interest of the users in the different semantic regions of a document; instead, they propagate every update with the same static priority. A different approach is taken in the design of *Docx2Go*.[33] In this work, the authors acknowledge that users, especially those using resource-constrained devices, are typically not equally interested in every part of a document. They materialize this observation by allowing users to replicate only a subset of the regions of the document at their devices. This way, *Docx2Go* is able to obtain both network and storage savings. However, unlike CoopSLA, this approach considers each of the locally replicated regions as equally important, not allowing different requirements to be set for different regions. This *all-or-nothing* approach prevents users from accompanying the evolution of the document. CoopSLA, on the other hand, provides users with, at least, an overall view of the whole state of the document; and it does so while still obtaining considerable savings regarding the usage of the network resources.

Unlike CoopSLA, the systems described in the previous paragraphs follow an *eventual consistency* approach to consistency maintenance. As a result, they cannot provide clear guarantees about the consistency state of the data at any particular moment. Most importantly, our consistency model is compatible with any of these systems and can, thus, be used to improve their performance. In fact, as mentioned in Sec. 4.4, CoopSLA uses the TreeDoc CRTD as a building block.

8. Conclusion

In this paper we presented a semantic and locality aware consistency model for cooperative editing applications. Our model, named CoopSLA, explores the heterogeneous and dynamic interest of users in different regions of a document space in order to minimize network communications between the participants of an editing session. CoopSLA assigns, on a per-user basis, different priorities to different updates, based on the semantic distance between the place in the document in which the update is performed and the places in which the user is more interested. Updates with high priority are sent promptly to the user, while low priority updates are postponed and, when possible, merged. In between high and low priority, our system allows the definition of an arbitrary number priority levels. Each priority level is characterized by a multidimensional consistency degree that defines how many and for how long updates to a particular object are allowed to be postponed.

We implemented a middleware layer enforcing CoopSLA and extended the popular Tex editor *TexMaker* with cooperative features using it. We conducted a series of tests to experimentally evaluate the performance of CoopSLA. The results presented in this paper support our claim that CoopSLA is very effective in reducing the overhead of replica synchronization without constraining application models and respecting users consistency needs.

Acknowledgments

This work was partially supported by national funds through FCT – Fundação para a Ciência e Tecnologia, under projects PEst-OE/EEI/LA0021/2013, PTDC/EIA-EIA/113613/2009 and PTDC/EIA-EIA/113993/2009.

Appendix A

Algorithm Appendix A.1 Processing updates received

```
1: function UpdateReceived($u, o$)
2:     DocumentManager.ApplyUpdate($u, o$)
3:     for $c \in CLIENTS$ do
4:         ReplicaManager.MarkModified($o, c$)
5:         $\psi_c[\sigma, o] \leftarrow \psi_c[\sigma, o] + 1$
6:         $z \leftarrow c_z[o]$
7:         if $\psi_c[\sigma, o] = z[\sigma]$ then
8:             ReplicaManager.MarkDirty($o, c$)
9:         end if
10:    end for
11: end function
```

Algorithm Appendix A.2 Processing pivot movement

```
1: function PivotMovement(p,c)
2:     for $o \in OBJECTS$ do
3:         $z_{old} \leftarrow c_z[o]$
4:         $z_{new} \leftarrow$ ReplicaManager.ConsistencyZone($o, p$)
5:         if $z_{new} < z_{old}$ then
6:             $c_{z,o} \leftarrow z_{new}$
7:             if $\psi_c[\sigma, o] \geq z[\sigma]$ then
8:                 ReplicaManager.MarkDirty($o, c$)
9:             else
10:                ReplicaManager.MarkModified($o, c$)
11:            end if
12:        else
13:            ReplicaManager.MarkOutdatedZone($o, c$)
14:        end if
15:    end for
16: end function
```

Algorithm Appendix A.3 Handling modifications to the graph

```
1: function StructureChanged
2:     foreach $c \in CLIENTS$ do
3:         foreach $p \in$ ReplicaManager.Pivots($c$) do
4:             PivotMovement($p$,$c$)
5:         end foreach
6:     end foreach
7: end function
```

Algorithm Appendix A.4 Consistency validation algorithm

```
function ValidationLoop
    foreach $c \in CLIENTS$ do
        foreach $o \in ModifiedObjects(c)$ do
            $O_c \leftarrow \{\}$
            if ReplicaManager.IsDirty($o, c$) then
                $z \leftarrow c_z[o]$
                if $z[\theta] \geq t - \psi_c[\theta, o]$ then
                    $\nu_t \leftarrow$ DocumentManager.SemanticDifference($o, \psi_c[\nu, o]$)
                    if $\nu_t \geq z[\nu]$ then
                        $O_c \leftarrow O_c \cup o$
                        ReplicaManager.ClearFlags($o, c$)
                        $c_z[o] \leftarrow$ ReplicaManager.ConsistencyZone($o, c$)
                    end if
                end if
            end if
        end foreach
        SessionManager.Propagate($O_c, c$)
    end foreach
end function
```

Algorithm Appendix A.5 Retrieve consistency zone

```
function GetConsistencyZone($o, c$)
    $z_{c,o} \leftarrow \oslash$
    for $p \in$ ReplicaManager.Pivots($c$) do
        $s_d \leftarrow$ DocumentManager.SemanticDistance($o, p$)
        $z \leftarrow$ ReplicaManager.ConsistencyZone($s_d, p$)
        $z_{c,o} \leftarrow$ Strongest($z_{c,o}, z$)
    end foreach
    return $z_{c,o}$
end function
```

References

1. C. A. Ellis, S. J. Gibbs and G. Rein, Groupware: Some issues and experiences, *Commun. ACM* **34** (1991) 39–58.
2. B. Shao, D. Li and N. Gu, A fast operational transformation algorithm for mobile and asynchronous collaboration, *Parallel Distributed Syst., IEEE Trans.* **21**(12) (2010) 1707–1720.
3. K. Veeraraghavan, V. Ramasubramanian, T. L. Rodeheffer, D. B. Terry and T. Wobber, Fidelity-aware replication for mobile devices, in *Proc. 7th Int. Conf. Mobile systems, Applications and Services, MobiSys '09* (ACM, New York, NY, USA, 2009), pp. 83–94.
4. A. J. Feldman, W. P. Zeller, M. J. Freedman and E. W. Felten, Sporc: Group collaboration using untrusted cloud resources, in *Proc. 9th USENIX Conf. Operating*

Systems Design and Implementation, OSDI'10 (USENIX Association, Berkeley, CA, USA, 2010), p. 1.

5. Y. Saito and M. Shapiro, Optimistic replication, *ACM Comput. Surv.* **37**(1) (2005) 42–81.
6. C. A. Ellis and S. J. Gibbs, Concurrency control in groupware systems, in *SIGMOD '89: Proc. 1989 ACM SIGMOD Int. Conf. Management of Data* (ACM, New York, NY, USA, 1989), pp. 399–407.
7. C. Sun and C. Ellis, Operational transformation in real-time group editors: Issues, algorithms and achievements, in *CSCW '98: Proc. 1998 ACM Conf. Computer Supported Cooperative Work* (ACM, New York, NY, USA, 1998), pp. 59–68.
8. R. Li and D. Li, A new operational transformation framework for real-time group editors, *Parallel Distributed Syst. IEEE Trans.* **18**(3) (2007) 307–319.
9. D. Sun, S. Xia, C. Sun and D. Chen, Operational transformation for collaborative word processing, in *CSCW '04: Proc. 2004 ACM Conf. Computer Supported Cooperative Work* (ACM, New York, NY, USA, 2004), pp. 437–446.
10. N. Preguiça, J. Manuel Marques, M. Shapiro and M. Letia, A commutative replicated data type for cooperative editing, in *ICDCS '09: Proc. 2009 29th IEEE Int. Conf. Distributed Computing Systems* (IEEE Computer Society, Washington, DC, USA, 2009), pp. 395–403.
11. G. Oster, P. Urso, P. Molli and A. Imine, Data consistency for p2p collaborative editing, in *CSCW '06: Proc. 2006 20th Anniversary Conf. Computer Supported Cooperative Work* (ACM, New York, NY, USA, 2006), pp. 259–268.
12. S. Weiss, P. Urso and P. Molli, Logoot: A scalable optimistic replication algorithm for collaborative editing on p2p networks, in *ICDCS '09: Proc. 2009 29th IEEE Int. Conf. Distributed Computing Systems* (IEEE Computer Society, Washington, DC, USA, 2009), pp. 404–412.
13. H.-G. Roh, M. Jeon, J.-S. Kim and J. Lee, Replicated abstract data types: Building blocks for collaborative applications, *J. Parallel Distrib. Comput.* **71** (3) (2011) 354–368.
14. Q. Wu, C. Pu and J. E. Ferreira, A partial persistent data structure to support consistency in real-time collaborative editing, in *Data Engineering* (*ICDE*), *2010 IEEE 26th Int. Conf.* (2010), pp. 1707–1720.
15. W. Vogels, Eventually consistent, *Commun. ACM* **52**(1) (2009) 40–44.
16. D. B. Terry, M. M. Theimer, K. Petersen, A. J. Demers, M. J. Spreitzer and C. H. Hauser, Managing update conflicts in bayou, a weakly connected replicated storage system, in *Proc. fifteenth ACM Symp. Operating Systems Principles, SOSP '95* (ACM, New York, NY, USA, 1995), pp. 172–182.
17. M. Satyanarayanan, J. J. Kistler, P. Kumar, M. E. Okasaki, E. H. Siegel and D. C. Steere, Coda: A highly available file system for a distributed workstation environment, *IEEE Trans. Comput.* **39**(4) (1990) 447–459.
18. R. Ladin, B. Liskov, L. Shrira and S. Ghemawat, Providing high availability using lazy replication, *ACM Trans. Comput. Syst.* **10**(4) (1992) 360–391.
19. H. Abrams, K. Watsen and M. Zyda, Three-tiered interest management for large-scale virtual environments, in *Proc. ACM Symp. Virtual Reality Software and Technology, VRST '98* (ACM, New York, NY, USA, 1998), pp. 125–129.
20. C. Pu and A. Leff, Replica control in distributed systems: As asynchronous approach, in *Proc. 1991 ACM SIGMOD Int. Conf Management of Data, SIGMOD '91* (ACM, New York, NY, USA, 1991), pp. 377–386.
21. D. Li and M. Anand, Majab: Improving resource management for web-based applications on mobile devices, in *Proc. 7th Int. Conf. Mobile Systems, Applications and Services*, *MobiSys '09* (ACM, New York, NY, USA, 2009), pp. 95–108.

22. T. Imielinski and B. R. Badrinath, Mobile wireless computing: Challenges in data management, *Commun. ACM* **37**(10) (1994) 18–28.
23. P. Dourish and V. Bellotti, Awareness and coordination in shared workspaces, in *Proc. 1992 ACM Conf. Computer-Supported Cooperative Work*, CSCW '92 (ACM, New York, NY, USA, 1992), pp. 107–114.
24. R. Alonso, D. Barbara and H. Garcia-Molina, Data caching issues in an information retrieval system, *ACM Trans. Database Syst.* **15**(3) (1990) 359–384.
25. F. J. Torres-Rojas, M. Ahamad and M. Raynal, Timed consistency for shared distributed objects, in *Proc. Eighteenth Annual ACM Symp. Principles of Distributed Computing, PODC '99* (ACM, New York, NY, USA, 1999), pp. 163–172.
26. C. Zhang and Z. Zhang, Trading replication consistency for performance and availability: An adaptive approach, in *Proc. 23rd Int. Conf. Distributed Computing Systems, ICDCS '03* (IEEE Computer Society Washington, DC, USA, 2003), pp. 687.
27. N. Krishnakumar and A. J. Bernstein, Bounded ignorance: A technique for increasing concurrency in a replicated system, *ACM Trans. Database Syst.* **19**(4) (1994) 586–625.
28. H. Yu and A. Vahdat, Design and evaluation of a conit-based continuous consistency model for replicated services, *ACM, Trans. Comput. Syst.* **20**(3) (2002) 239–282.
29. N. Santos, L. Veiga and P. Ferreira, Vector-field consistency for ad-hoc gaming, in Middleware '07: *Proc. ACM/IFIP/USENIX 2007 Int. Conf. Middleware* (Springer-Verlag, New York, NY, USA, 2007), pp. 80–100.
30. L. Veiga, A. Negrão, N. Santos and P. Ferreira, Unifying divergence bounding and locality awareness in replicated systems with vector-field consistency, *J. Internet Serv. Appl.* **1**(2) (2010) 95–115.
31. D. Sun and C. Sun, Context-based operational transformation in distributed collaborative editing systems, *IEEE Trans. Parallel Distrib. Syst.* **20**(10) (2009) 1454–1470.
32. S. Xia, D. Sun, C. Sun, D. Chen and H. Shen, Leveraging single-user applications for multi-user collaboration: The coword approach, in *CSCW '04: Proc. 2004 ACM Conf. Computer Supported Cooperative Work* (ACM, New York, NY, USA, 2004), pp. 162–171.
33. K. P. N. Puttaswamy, C. C. Marshall, V. Ramasubramanian, P. Stuedi, D. B. Terry and T. Wobber, Docx2go: Collaborative editing of fidelity reduced documents on mobile devices, in *Proc. 8th Int. Conf. Mobile Systems, Applications and Services, MobiSys '10* (ACM, New York, NY, USA, 2010), pp. 345–356.